

# Performance Optimizations for XML Data Type in SQL Server 2005

SQL Server Technical Article

Author: Shankar Pal, Babu Krishnaswamy, Vasili Zolotov, and Leo Giakoumakis,   
Microsoft Corporation

**Published:** October 2005

Applies to: Microsoft® SQL Server™ 2005

Summary: This paper explores several ideas to improve the query and data modification performance of the XML data type in Microsoft® SQL Server™ 2005. To get the most value from this paper, you need to be familiar with the XML features in SQL Server 2005. For background material, see XML Support in Microsoft SQL Server 2005 and XML Best Practices for Microsoft SQL Server 2005 on the Microsoft Development Network (MSDN).

## Copyright

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

 2005 Microsoft Corporation. All rights reserved.

Microsoft, ActiveX, Visual Basic, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

## Table of Contents

[Introduction 1](#_Toc118733414)

[Data Modeling with the XML data type 1](#_Toc118733415)

[Structure of XML Data 2](#_Toc118733416)

[Using Specific Markups 2](#_Toc118733417)

[Attribute-centric Markups 3](#_Toc118733418)

[Typed or Untyped XML 3](#_Toc118733419)

[Property Promotion 4](#_Toc118733420)

[Using a Computed Column 4](#_Toc118733421)

[Using a Property Table 6](#_Toc118733422)

[Bulk Loading XML Data 6](#_Toc118733423)

[Using OpenRowset 6](#_Toc118733424)

[Storage Space Consideration 7](#_Toc118733425)

[Indexing XML Data 8](#_Toc118733426)

[XML Indexes 8](#_Toc118733427)

[Partial XML Update 9](#_Toc118733428)

[Optimizations Prevented by Union of Types 10](#_Toc118733429)

[Disabling XML Index Selection 10](#_Toc118733430)

[Full-text Index on the XML Column 11](#_Toc118733431)

[Snapshot Isolation and XML Indexes 12](#_Toc118733432)

[Query and Data Modification 13](#_Toc118733433)

[Merging Multiple value() Method Executions for Indexed XML 13](#_Toc118733434)

[Using the exist() Method for Checking Existence 15](#_Toc118733435)

[Using the nodes()-value() Combination 16](#_Toc118733436)

[Optimizations for XML blobs 17](#_Toc118733437)

[Multiple tempDB Files for Better Scalability of XML Variables and Parameters 17](#_Toc118733438)

[Eliminating Extra Casts to XML data type 18](#_Toc118733439)

[Specifying Singleton Elements 19](#_Toc118733440)

[Eliminating Multiple Executions of XML data type Methods 20](#_Toc118733441)

[Data(), text(), and string() Accessors 21](#_Toc118733442)

[Text Aggregation in Untyped XML 21](#_Toc118733443)

[Parameterize Your XQuery and XML DML Expression 22](#_Toc118733444)

[Optimizations for Predicates and Ordinals 24](#_Toc118733445)

[Moving Ordinals to the End of Paths 25](#_Toc118733446)

[Evaluating Predicates by Using the Context Node 25](#_Toc118733447)

[Range Conditions 26](#_Toc118733448)

[Parent Axis 27](#_Toc118733449)

[Dynamic Querying 27](#_Toc118733450)

[Rowset Generation from XML Data 29](#_Toc118733451)

[Conclusion 30](#_Toc118733452)

#### Introduction

Enterprise applications are increasingly using XML for modeling semi-structured and unstructured data. Microsoft® SQL Server™ 2005 provides extensive support for XML data processing to help develop such applications. XML data can be stored natively in an XML data type column, which can be typed according to a collection of XML schemas, or left untyped. Fine-grained data manipulation is supported using XQuery, an emerging W3C recommendation currently in Last Call, and an XML data modification language. The XML column can be indexed to improve query performance. Enterprise applications are increasingly using XML for modeling semi-structured and unstructured data and will benefit from the XML support in SQL Server 2005.

This paper provides suggestions for optimizing the storage, queries, and data modification of applications that use the XML data type. The ideas are illustrated with code samples. For a discussion of best practices for XML data modeling and usage, see the related white paper, [XML Best Practices for Microsoft SQL Server 2005](http://msdn.microsoft.com/library/en-us/dnsql90/html/sql25xmlbp.asp?frame=true) (http://msdn.microsoft.com/library/en-us/dnsql90/html/sql25xmlbp.asp?frame=true)!href(http://msdn.microsoft.com/xml/default.aspx?pull=/library/en-us/dnsql90/html/sql25xmlbp.asp), on the Microsoft Developer Network (MSDN). For information about optimizations for XML view technology using mapping, see [Optimizing SQLXML Performance](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql2k/html/sqlxml_optimperformance.asp) (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql2k/html/sqlxml\_optimperformance.asp)!href(http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql2k/html/sqlxml\_optimperformance.asp) in the MSDN Library.

In this paper, first considered are the data modeling guidelines using XML, including database design principles, and then query and data modification guidelines for optimizing the performance of applications are provided.

#### Data Modeling with the XML data type

The XML data type provides data modeling capabilities for semi-structured and unstructured data within an enterprise. The performance of XML storage and query processing depends on the database schema design and includes factors such as the structure and granularity of the XML data and property promotion from XML columns.

The first decision to make is whether an application needs the features of the XML data model. Structured data is best modeled as relational and stored in tables with relational columns. Your interests are best served with the XML data model if you have semi-structured or markup data that needs to preserve the document order and containment hierarchy, and perhaps has a recursive structure.

Sometimes, it is beneficial to store structured data in an XML data type column such as when the data has a flexible structure or the structure is not known a priori.

This scenario occurs in property management where metadata information about objects is modeled as XML and stored in an XML data type column. Properties of different types of objects, even with different structures and content models, can be stored in the same XML column and queried across. Properties queried most often are promoted into columns of the same table or a different table. The promoted properties can be indexed and queried, and query plans are simpler than querying the XML column.

Alternatively, the incoming XML data can be decomposed into tables and queried by using the SQL language. If XML generation is a significant part of the query workload, it may be beneficial to store a redundant copy of the XML data in an XML data type column. The redundant copy avoids the run-time cost of XML generation.

There are no absolute rules for modeling data with the XML data type and the pros and cons must be carefully weighed in each modeling situation. Equally important is the choice between typed and untyped XML columns, and even the way XML markup is introduced into the data. These and some other considerations are discussed in the remainder of this section.

##### Structure of XML Data

The same data can be marked up in different ways (element-centric, attribute-centric, and combined). This choice is dictated by the perception of what constitutes content (element value) and what constitutes meta information (attribute value), and the cardinality of the markup (multi-occurrence of elements). Introducing XML markup into semi-structured and unstructured data in one way can be more efficient than some other way for storage and query processing.

###### Using Specific Markups

Sometimes it is convenient to use generic element names and distinguish between different types of elements by using additional attributes. This does not perform well in queries because it does not allow XML index lookups to be done efficiently. For more information about XML indexing, see Indexing XML Data!href(#IndexingXMLData) later in this paper.

Specific, semantically rich element names on the one hand yield markups that are more humanly readable and help to generate more efficient query plans. On the other hand, very verbose markups increase storage cost. The following example illustrates these points.

Example: Generic vs. Specific Markups

Suppose you want to use XML markup for book and DVD information. One choice is to have a generic element called <item> with an attribute @type that has one of the two values, “book” and “DVD” to distinguish between the two types of items. A book and a DVD can be represented as follows:

<item type=”book”><title>Writing Secure Code</title></item>

<item type=”DVD”><title>The Godfather</title></item>

Path expressions for book and DVD, respectively, can be written as /item[@type = “book”] and /item[@type = “DVD”].

On the other hand, <book> and <DVD> are more direct XML markups:

<book><title>Writing Secure Code</title></book>

<DVD><title>The Godfather</title></DVD>

This representation yields simpler path expressions /book and /DVD. Query plans are also simpler and more efficient, because the predicate on the attribute @type is eliminated.

Furthermore, it cuts down the number of rows in the primary XML index for untyped XML from 4 (1 row for <item>, 1 for @type and its value, 1 for <title>, and 1 for the title’s value) to 3 (1 row for <book> or <DVD>, 1 for <title>, and 1 for the title’s value). For more information about indexing XML data, see Indexing XML Data!href(#IndexingXMLData) later in this paper.

For typed XML in which <title> is a simple-valued element, the value of the <title> element is stored in the same row as the element itself. This reduces the storage overhead from 3 to 2 rows and is a significant savings.

###### Attribute-centric Markups

An attribute’s value is stored with the attribute markup in a single row of the primary XML index for both typed and untyped XML. By comparison, the value of a simple-valued element in untyped XML is stored in a separate row from the element markup. Thus, less storage is required by using attribute values within untyped XML.

Furthermore, evaluation of predicates is more efficient because the attribute’s value is obtained from the same row as its markup in the primary XML index. This eliminates the need to access another row for the value. This is illustrated in the following example.

Example: Attribute-centric Markup

In the previous example, title can be modeled as an attribute instead of as an element as follows:

<book title="Writing Secure Code"/>

<DVD title="The Godfather"/>

For untyped XML, this cuts down the number of rows in the primary XML index from 3 (a row each for <book>, <title> and the title’s value) to 2 (one row for <book> and a second one for the attribute @title). The case is similar for DVD.

The path expression /DVD[title = "The Godfather"] finds the DVD whose title is “The Godfather” with element-centric markup for title. With attribute-centric markup, the same query is written as /DVD[@title = "The Godfather"] and requires one less JOIN.

##### Typed or Untyped XML

Element and attribute values in untyped XML (XML data not described by XML schemas) are stored internally as Unicode strings. Operations on them require data conversion to the appropriate type. For example, when the path expression (/book/price)[1] > 19.99 is evaluated, the string value of <price> is converted to decimal for the numeric comparison. A large number of such comparisons can become costly.

Type information provided by XML schemas is used by the database engine in several ways. Inserted and updated XML data are verified for conformance with the XML schemas and stored in a binary representation (XML blob). Element and attribute values are stored as typed values within XML instances. This allows XML blobs to be parsed more efficiently than the corresponding textual form. Typed values are stored in XML indexes and allow index usage whenever data conversions are eliminated. Query compilation uses type information to check the static type correctness of XQuery expressions and data modification statements. Type mismatch errors are detected at compilation time and can be avoided by using explicit type casts.

Query optimizations based on type inference are also performed (if <price> of <book> is of type xs:decimal, conversion of (/book/price)[1] to xs:decimal is eliminated). This can have a positive effect on XML index lookup. A range predicate such as (/book/price)[1] < 19.99 performs a range scan on secondary XML index of type VALUE. For more information about indexing XML data, see Indexing XML Data!href(#IndexingXMLData) in this paper.

The data conversion required for untyped XML prevents such range scans. Furthermore, the ordinal [1] in (/book/price)[1] is unnecessary if the XML schema specifies a single <price> element and only allows a single <book> element in each XML instance.

Typed XML requires validation during XML data insertion and modification. The cost of validation may be non-trivial and depends on factors such as the complexity of the schema definitions and the number of tags occurring in the XML data.

##### Property Promotion

During query processing, structural information, such as document order and containment hierarchy, are preserved in XML instances. Consequently, the query plans tend to be complex. The plan can be simplified for some queries by promoting scalar values from an XML column into relational columns of the same or different table, and writing queries directly against these columns. The promoted properties may be indexed. Materializing and indexing the property values yields better performance than using XQuery on the XML column in the same way that pre-computed values speed up query performance.

Property promotion improves performance when the property value is retrieved or the property value is used as a filter to retrieve the corresponding XML blob. In the latter case, the selectivity of the property value is an important factor.

Single-valued properties can be promoted into columns of the same table as computed columns. Both single-valued and multi-valued properties can be promoted into columns of a different table and maintained by using triggers. These two ways of promoting properties are considered in the following section.

###### Using a Computed Column

A Transact-SQL user-defined function is first created to extract a scalar value using XML data type methods. A computed column defined by the user-defined function is then appended to the table. These two steps are repeated for each promoted property, and relational indexes are created on those columns as needed.

The XQuery expression on the XML column must be rewritten as an SQL statement that uses the computed columns, and the XML instances are retrieved from the matching rows. Indexes on computed columns are selected by the query optimizer based on the costing of the query. Promoted properties yield faster performance than querying the XML column directly because the computed columns are pre-computed.

Indexing the computed column can be avoided when the column is used only in SELECT lists and not for evaluating predicates. In such cases, persistence of the computed column is sufficient for performance benefits. When the computed column is indexed, it needs to be persisted if the computed column expression is imprecise or non-deterministic.

The following example illustrates the use of computed column for property promotion.

Example: Using a Computed Column for Property Promotion

Suppose your workload typically looks up books given their ISBN number so that promoting the ISBN number into a computed column is worthwhile. Define a user-defined function to retrieve the ISBN number as follows:

CREATE FUNCTION udf\_get\_book\_ISBN (@xData xml) RETURNS varchar(20)

WITH SCHEMABINDING

BEGIN

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

END

Add a computed column to the table docs for ISBN:

CREATE TABLE docs (id int PRIMARY KEY, xCol XML)

ALTER TABLE docs ADD ISBN AS dbo.udf\_get\_book\_ISBN(xCol)

Create a non-clustered index on the ISBN column:

CREATE INDEX COMPUTED\_IDX ON docs (ISBN)

Rewrite your query as:

SELECT xCol

FROM docs

WHERE xCol.exist ('/book/@ISBN[. = "0-2016-3361-2"]') = 1

to use the computed column as follows:

SELECT xCol

FROM docs

WHERE ISBN = '0-2016-3361-2'

The rewritten query generates a simpler query plan, because the extraction of the ISBN value is pre-computed.

###### Using a Property Table

A separate property table requires setting up insert, delete, and update triggers for its maintenance. It is suitable for multi-valued properties where each row in the property table contains a property value (*unpivoted* representation). An example illustrating the creation and maintenance of property tables can be found in [XML Best Practices for Microsoft SQL Server 2005](http://msdn.microsoft.com/library/en-us/dnsql90/html/sql25xmlbp.asp?frame=true) (http://msdn.microsoft.com/library/en-us/dnsql90/html/sql25xmlbp.asp?frame=true)!href(http://msdn.microsoft.com/xml/default.aspx?pull=/library/en-us/dnsql90/html/sql25xmlbp.asp).

A sequence number column is desirable in the property table if the relative order of siblings is important for the application. This, however, complicates property table maintenance for XML subtree insertion and deletion.

Single-valued property columns can be added to the table for convenience. It introduces redundancy in the column, but eliminates a JOIN when both properties are required.

If the maximum cardinality N of the promoted property is small and known ahead of time, it may be convenient to create N computed columns instead of a separate property table and have the query processor maintain those column.

#### Bulk Loading XML Data

XML data can be bulk loaded into XML data type columns by using the bulk load capabilities of SQL Server. This includes the BCP IN, BULK INSERT, and OPENROWSET methods.

BCP input has been optimized to avoid intermediate copies of XML data wherever possible. Thus, if no (row or column) constraint exists on the XML column, BCP has the best performance among the three alternatives.

##### Using OpenRowset

OPENROWSET is a convenient way of loading XML data from files into XML columns, variables, and parameters. Querying the XML data in variables or parameters multiple times may retrieve the data that many times from the file. It is better to read the XML data once into an XML variable and to query it multiple times, as shown in the following example.

Example: Querying the Output of OPENROWSET

In the following query, the XML data is read from the file into the column [Contents] of the table expression XmlFile. The **nodes()** method finds the <author> elements in the XML instance. Each **value()** method evaluates a path expression relative to an <author> element, which loads the XML data from the file each time.

WITH XmlFile ([Contents]) AS (

SELECT CONVERT (XML, [BulkColumn])

FROM OPENROWSET (BULK N'C:\temp\Filedata.xml', SINGLE\_BLOB) AS [XmlData]

)

SELECT nref.value('first-name[1]', 'nvarchar(32)') FirstName,

nref.value('last-name[1]', 'nvarchar(32)') LastName

FROM [XmlFile] CROSS APPLY [Contents].nodes('//author') AS p(nref)

The file data can be loaded once as shown in the following rewrite for better performance. The file content is read only once into the XML variable @xmlData and reused in the SELECT statement:

DECLARE @xmlData XML;

SELECT @xmlData = CONVERT (XML, [BulkColumn])

FROM OPENROWSET (BULK N'C:\temp\Filedata.xml', SINGLE\_BLOB) AS [XmlData];

SELECT nref.value('first-name[1]', 'nvarchar(32)') FirstName,

nref.value('last-name[1]', 'nvarchar(32)') LastName

FROM @xmlData.nodes ('//author') AS p(nref)

##### Storage Space Consideration

In the presence of primary XML index, bulk loading data into an XML column and inserting very large XML instances takes up a large amount of transaction log space. This issue can be avoided by delaying the creation of the XML indexes on the XML column and using the SIMPLE recovery model, as shown in the following:

1. Run the following command where <database\_name> is the name of the database into which the XML data will be loaded:

ALTER DATABASE <database\_name> SET RECOVERY SIMPLE

1. Create the XML column in a new or existing table, but not the XML indexes.
2. Insert the XML data into the XML column.
3. Create the XML indexes on the XML column.
4. Run the following command to reset the recovery model to full:

ALTER DATABASE <database\_name> SET RECOVERY FULL

Alternatively, you can use BCP to load the XML data into the database and use the BULK\_LOGGED recovery model instead of SIMPLE. For more information about the SIMPLE and BULK LOGGED recovery models, see SQL Server 2005 Books Online.

In these recovery models, if the data file has been damaged since the last backup, the database must be restored from the backup and the operations redone.

In either case, pre-allocating the database file before inserting the XML data is faster than dynamically growing the database file.

#### Indexing XML Data

##### XML Indexes

­For fine-grained queries into an XML column, it is advisable to create the primary XML index on the XML column. A primary XML index can be created on both untyped and typed XML columns, and indexes all paths and values within the entire XML column. Primary XML indexes create a B+tree based on a shredded representation of the XML instances in the XML column. This B+tree is created in addition to the XML blobs in the XML column and is larger in size than the combined size of the XML blobs in the XML column. The B+tree is used for querying the XML data using XML data type methods. The XML blob is used to optimize the cases in which the entire XML blob is retrieved from the base table, such as in SELECT \* FROM docs. This is faster than serializing the XML content from the primary XML index owing to its smaller size and the serialization cost.

Secondary XML indexes provide further options for the query optimizer to come up with a better plan. Your application can get a further boost by using secondary XML indexes of type PATH, PROPERTY, and VALUE.

* The PATH index is useful whenever path expressions such as /book[@ISBN = "0-2016-3361-2"] occur on an XML data type. The benefits are greater for longer path expressions. The PATH index provides good, overall speed-up.
* The PROPERTY index is useful when multiple properties of an XML instance are retrieved within a SELECT statement. Clustering the properties of each XML instance together can yield better performance.
* The VALUE index is useful for path expressions containing the descendant axes (the //-operator) and wildcards (/book[@\* = "novel"]).

Analysis of the query workload is required to determine whether one or more of the secondary XML indexes are helpful. The index maintenance cost should also be taken into account in measuring the overall benefit of indexing the XML data.

Many applications know the expected query workload and will benefit by indexing only the paths occurring in the queries. Those paths can be promoted as properties as discussed in Property promotion!href(#PropertyPromotion) later in this paper.

##### Partial XML Update

In-place update of the XML data type yields significant performance improvement during fine-grained data modification. The difference between the new (after the data modification) and the old (before the data modification) states is computed and applied to the XML column storage as well as the primary XML index. Changes in the primary XML index are propagated to the secondary XML indexes as well. The performance benefits come from the smaller amounts of data updated in storage and the corresponding savings in the transaction log. These savings offset the cost of comparing the new and the old states in most cases.

The best case scenario is the modification of the value of an attribute or an element using the “replace value of” statement in XML DML. This requires updating a single row in each of the primary and secondary XML indexes on the XML column. The update is also local to the on-disk page of the XML blob containing the updated attribute or the element. Of course, replacing the old value with a large value causes new disk pages to be written. The following is an example where the update is very efficient.

Example: Updating the Value of an Attribute

Modifying the <price> of a <book> as shown in this example performs in-place update of the XML instance and the XML indexes:

UPDATE docs

SET xCol.modify ('replace value of (/book/price/text())[1] with 29.99')

For insertion of an attribute, an element, or a subtree, the newly inserted node and the siblings following it, together with their subtrees, are updated or inserted. Similar changes occur in the XML blob. The case is similar for node deletions and the sibling beyond the point of deletion are updated.

The worst case scenario occurs during the insertion of a node as the leftmost fragment of an XML data type instance or the insertion of the leftmost child of the root element. This updates the entire XML instance. This situation can be avoided by inserting the node as the rightmost fragment in the XML instance or the rightmost child of the root element.

Deletion of the leftmost fragment or the leftmost child of the root element has similar costs. If an element is inserted and deleted often, it is better to insert it as the rightmost fragment or the rightmost child of the root element. The following example illustrates an expensive case.

Example: Costly Update

The <publisher> element is inserted as the leftmost child of the <book> element and causes updating all the sub-elements of <book>.

UPDATE docs

SET xCol.modify ('

insert <publisher>Microsoft Press</publisher>

before (/book/title)[1]')

Inserting the <publisher> as the rightmost child of <book> is more efficient:

UPDATE docs

SET xCol.modify ('

insert <publisher>Microsoft Press</publisher> into (/book)[1]')

XML schema constraints may determine the insertion point, and inserting the new node in the rightmost permissible position yields the best performance.

##### Optimizations Prevented by Union of Types

A value of a union type requiring an implicit cast prevents lookup of secondary XML indexes for the value, although secondary XML indexes may be used for matching paths. Thus, it prevents range scans from occurring on the VALUE secondary XML index. For more information, see Range Conditions!href(#RangeConditions) in this white paper. The same reasoning applies to <xs:anyAttribute> as well.

Model groups (<xs:choice> and <xs:all>), substitution groups, and wildcard sections (xs:any) have as a content model a union of types. When the exact type is not known during query compilation and optimization, run-time type casts may be needed for operating on their values. This slows down the query. Thus, such XML schema structures and data types should be avoided, if possible, for performance reasons.

Using schema structures that indicate singleton occurrence of elements helps with query optimization. For this reason, a <xs:choice> structure is preferred over a <sequence> with optional elements.

##### Disabling XML Index Selection

XML index selection is disabled in check constraints because the query optimizer does not guarantee that the XML index is modified before evaluating the constraint,or vice versa. Sufficient care must be exercised to ensure that the constraint can be evaluated efficiently on XML blobs by following the performance guidelines in this paper. Furthermore, XML index selection is disabled in views with CHECK OPTION.

##### Full-text Index on the XML Column

A full-text index can be created on an XML column independently of XML indexes on the column. It indexes element content, ignores the XML markup tags and attribute values, and uses the markup tags as token boundaries.

The XQuery function **fn:contains**() has the semantics of a literal, substring match that is case-sensitive in the implementation. On the other hand, a full-text search using **CONTAINS**() uses a token match with stemming. Thus, their semantics are different. To illustrate the differences, a search for the word “data” matches the word “database” in Xquery, but not with full-text semantics. On the other hand, a search for the word “drove” matches the word “driving” with full-text semantics, but not in XQuery. Furthermore, full-text search cannot be used for searching over attribute values, while XQuery expressions need to use the aggregate function **fn:string()** to search over mixed content.

When full-text index exists on an XML column, it is advisable to do the following:

* Filter the XML values of interest using full-text search.
* Query the selected XML instances using XML data type methods. XML indexes on the XML column get used during this step.

This leads to the use of both full-text and XML indexes. The high selectivity of the search word or phrase in a full-text search narrows down further processing for the XQuery search to a relatively small number of rows in the base table. This can significantly speed up the query. This approach can be used when the search phrase consists of word stems that match keyword boundaries.

The XQuery search specifies the context (the node set) for the search. A part of the XML data can be promoted to a computed XML column XC on which a full-text index is created. This defines XC as the full-text search context.

Example: Combining Full-text with XQuery Matches

The following query performs a full-text search for the keyword, data, and verifies that the word “data” occurs in the context of the <title> element of a <book>. It uses the full-text **contains()** method to locate the XML instances containing the search word. The XML data type method **exist()** verifies that the XML instances contain the substring in the correct context.

select \*

from docs

where contains(xCol, 'data')

AND xCol.exist('/book/title/text()[contains(.,"data")]') = 1

Example: Using Prefix Search in Full-text

It is possible to perform prefix searches in full-text indexes. For a query that will match any keyword that starts with data, such as database, the previous query can be rewritten as follows. The XQuery search matches database as well.

select \*

from docs

where contains(xCol, '"data\*"')

and xCol.exist('/book/title/text()[contains(.,"data")]') = 1

Note the use of the double quotes in the full-text **contains()** method.

##### Snapshot Isolation and XML Indexes

XML data modification updates old XML instances with the new ones. These changes are propagated to the primary and secondary XML indexes. Modified rows in the base table and the XML indexes are locked, and the row and page locks may escalate to table locks at the discretion of the query optimizer. Concurrency suffers owing to lock escalation, especially when modification is common in the workload.

In SQL Server 2005, snapshot-based isolations introduce a new isolation level called snapshot and a new implementation of read-committed isolation level. More information about these can be found in SQL Server Books Online. These are based on an internal versioning mechanism that eliminates lock contention between readers and writers when the database is enabled for snapshot isolation. The reduced lock contention may yield higher throughput.

A read operation under snapshot-based isolations can access the versioned data without getting blocked on a concurrent update. This reduced blocking can potentially improve transaction throughput in concurrent workloads.

With snapshot isolation, XML column values and the corresponding primary and secondary XML index rows are versioned upon update. This avoids unnecessary versioning of the XML columns when modifications in non-XML columns cause the containing row to be versioned. This optimization makes snapshot isolation very useful for XML processing.

#### Query and Data Modification

##### Merging Multiple value() Method Executions for Indexed XML

In the indexed case, the execution of multiple **value()** methods on the same typed XML column in a SELECT list may be combined for faster execution. The decision to combine the executions is made by the query optimizer based on its costing of the query. This can yield significant speed-up. Following is an example.

Example: Combining Multiple value() Method Executions

Suppose that the content model for <book> element is defined by the XML schema namespace "http://www.microsoft.com/book" in an XML schema collection bookCollection. Furthermore, a table TypedBooks is created with an XML column xBook typed using bookCollection, and the primary XML index is created on this column. The XML schema definition is shown in the following:

CREATE XML SCHEMA COLLECTION bookCollection AS

'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"

xmlns="http://www.microsoft.com/book"

targetNamespace="http://www.microsoft.com/book">

<xsd:element name="book" type="bookType" />

<xsd:complexType name="bookType">

<xsd:sequence>

<xsd:element name="title" type="xsd:string" />

<xsd:element name="author" type="authorName"

maxOccurs="unbounded"/>

<xsd:element name="price" type="xsd:decimal" />

</xsd:sequence>

<xsd:attribute name="subject" type="xsd:string" />

<xsd:attribute name="releasedate" type="xsd:integer" />

<xsd:attribute name="ISBN" type="xsd:string" />

</xsd:complexType>

<xsd:complexType name="authorName">

<xsd:sequence>

<xsd:element name="first-name" type="xsd:string" />

<xsd:element name="last-name" type="xsd:string" />

</xsd:sequence>

</xsd:complexType>

</xsd:schema>'

GO

CREATE TABLE TypedBooks

(id int PRIMARY KEY, xBook XML(DOCUMENT bookCollection))

CREATE PRIMARY XML INDEX idx\_priXML\_xBook ON TypedBooks (xBook)

In the following query, the execution of the **value()** methods are combined because they are invoked on the same XML column, and the singleton cardinality of the <title> and <price> elements are statically inferred from the XML schema:

WITH XMLNAMESPACES ('http://www.microsoft.com/book' AS "bk")

SELECT xBook.value ('/bk:book/title', 'nvarchar(128)') Title,

xBook.value ('/bk:book/price', 'decimal(5,2)') Price

FROM TypedBooks

The following conditions must hold for the optimization to occur:

* The XML column must be typed so that singleton cardinality of nodes can be inferred from the XML schema collection typing the column. Wherever appropriate, the XML column should be declared with the column option DOCUMENT. Otherwise, the **nodes()** method must be used to generate single node references. The optimization works also for untyped XML when the **nodes()** method is used, thus ensuring singleton elements, and the **value()** method extracts attribute values from those elements.
* Full paths must be specified for the optimization to occur. Paths containing wildcards (\*), descendant axes (//-operator), ancestor axis (..), XPath functions, and node tests (**node()**) prevent this optimization.
* Path expressions in the **value()** methods may not contain predicates or ordinals.
* The optimization occurs for path expressions relative to the context items obtained from the **nodes()** method. In this case, the first argument of the **value()** methods must be the full relative path with the previously stated restrictions.
* The **value()** method calls must appear contiguously in a SELECT list to be merged. The execution of non-contiguous **value()** methods are not combined.

The optimization may also occur with **value()** methods in the Transact-SQL WHERE clause for predicates such as xCol.value(…) = xCol.value(…). It does not occur in predicates of the type xCol.value(…) = constant.

##### Using the exist() Method for Checking Existence

For better performance, use the **exist()** method on the XML data type whenever possible, instead of the **value()** method. The **exist()** method is most helpful when used in the SQL WHERE clause and utilizes XML indexes more effectively than the **value()** method. This is true even when you use **sql:variable()** and **sql:column()** in XQuery expressions.

For example, consider the following query that retrieves books having the title "Writing Secure Code" using the **exist()** method:

SELECT \*

FROM docs

WHERE xCol.exist('(/book/title/text())[.="Writing Secure Code"]') = 1

The PATH or VALUE secondary XML indexes are used to evaluate the path expression ((/book/title/text())[.="Writing Secure Code"]), including value lookup ("Writing Secure Code" in this example) in those indexes, to yield the XML instances to return. If the path and the search value are highly selective, the resulting execution can be much faster than evaluating the path expression over all the XML blobs in the column. The search value can be supplied by using **sql:variable()** or **sql:column()**. For more information, see Parameterize Your XQuery and XML DML Expression!href(ParameterizeExpressions) in this paper.

Writing the query using the **value()** method as shown in the following evaluates all book titles first and then applies the filter "Writing Secure Code".:

SELECT \*

FROM docs

WHERE xCol.value('(/book/title)[1]', 'varchar(50)') = 'Writing Secure Code'

This yields a less efficient query execution, because the filter value "Writing Secure Code" is not used in XML index lookup. A filter value specified by using a SQL variable or another **value()** method exhibits similar behavior.

Example: Use of sql:column()

The following query finds books written by more authors than the id of the book:

SELECT \*

FROM docs

WHERE xCol.exist('/book [count(author) > sql:column("id")]') = 1

##### Using the nodes()-value() Combination

The **nodes()** method generates a rowset of internal node references that can be used within a **value()** method to extract scalar values from those nodes. These methods together can be used to represent XML data in a relational form.

Each row in the output of the **nodes()** method represents a single reference so that the ordinal predicate used for selecting attributes of the context node in the **value()** method can be eliminated, as shown in the following example. Furthermore, if the **nodes()** method yields exactly one reference, removing the **nodes()** method altogether makes the query perform faster. These optimizations are most useful for XML variables and parameters.

Example: Eliminating Ordinal Predicate with the nodes() Method

This query extracts the ISBN attribute from each book instance in column xCol of table docs. The **nodes()** method emits a separate reference to each distinct <book> element (the context node), and there can be at most one @ISBN attribute on the context node.

SELECT ref.value('@ISBN', 'nvarchar(32)')

FROM   docs CROSS APPLY xCol.nodes('/book') AS node(ref)

If no more than one <book> element occurs in each XML instance, the following rewrite is faster:

SELECT xCol.value('(/book/@ISBN)[1]', 'nvarchar(32)')

FROM docs

##### Optimizations for XML blobs

###### Multiple tempDB Files for Better Scalability of XML Variables and Parameters

XML variables and parameters use main memory as storage as long as their values are small. Large values, however, are backed by tempdb storage. In a multi-user scenario, if many large XML blobs occur, tempdb contention may become a bottleneck for good scalability. Creating multiple tempdb files reduces the storage contention and yields significantly better scalability. The next example illustrates how multiple tempdb files can be created.

Example: Creating Multiple tempdb Files

This example creates two additional data files for tempdb, each with an initial size of 8 MB, and two log files that each have an initial size of 1 MB.

USE TEMPDB

GO

ALTER DATABASE tempdb ADD FILE

(NAME = 'Tempdb\_Data1',

FILENAME = 'C:\temp\Tempdb\_Data1.MDF', SIZE = 8 MB),

(NAME = 'Tempdb\_Data2',

FILENAME = 'C:\temp\Tempdb\_Data2.MDF', SIZE = 8 MB)

GO

ALTER DATABASE tempdb ADD log FILE

(NAME = 'Tempdb\_Log1',

FILENAME = 'C:\temp\Tempdb\_Log1.LDF', SIZE = 1 MB),

(NAME = 'Tempdb\_Log2',

FILENAME = 'C:\temp\Tempdb\_Log2.LDF', SIZE = 1 MB)

GO

These files can be removed by using the ALTER DATABASE tempdb REMOVE FILE command. For more information, see SQL Server Books Online.

###### Eliminating Extra Casts to XML data type

In an inlinable function with an input argument of type XML, the caller can supply a text or binary value that is implicitly converted to XML data type. Each use of the XML argument in the body of the callee casts the input value to XML data type. This cost can be avoided by copying the argument into an XML data type variable, which causes one conversion of the argument’s value to the XML data type, and the XML variable is used multiple times in the body of the function or the stored procedure. The following example illustrates this point.

Example: Removing Conversions

Consider the following function GetTitleAndIsbnOfBook() that returns the title and the ISBN of a book:

CREATE FUNCTION GetTitleAndIsbnOfBook (@book XML)

RETURNS TABLE AS

RETURN

SELECT @book.value ('(/book/@ISBN)[1]', 'nvarchar(32)') ISBN,

@book.value ('(/book/title)[1]', 'nvarchar(128)') title

If the function is invoked with a string value, a conversion to XML data type occurs for each **value()** method invocation. The function can be rewritten as follows to have only one conversion of its argument to XML data type. However, the table variable @retTab required for a multi-statement, table-valued function introduces additional cost. This can be offset by a sufficient number of accesses to the XML variable when the XML data size is large.

CREATE FUNCTION GetTitleAndIsbnOfBookOpt (@book varbinary(max))

RETURNS @retTab TABLE (ISBN nvarchar(32), title nvarchar(128)) AS

BEGIN

DECLARE @xbook XML

SET @xbook = @book

INSERT INTO @retTab

SELECT @xbook.value ('(/book/@ISBN)[1]', 'nvarchar(32)'),

@xbook.value ('(/book/title)[1]', 'nvarchar(128)')

RETURN

END

##### Specifying Singleton Elements

A singleton cardinality estimate removes the need to specify ordinals in queries and data modification statements. This simplifies the query plan and produces efficient JOIN operations. It typically involves making the proper choices for the inner and outer loops in nested loop joins.

In typed XML, elements by default have singleton cardinality in XML schema structures unless overridden by using the values of minOccurs and maxOccurs. In addition, the DOCUMENT constraint on a typed XML column, variable, and parameter guarantees exactly one top-level element in the XML data type instance.

For untyped data or when multiple sibling elements are allowed in a schema, singleton cardinality of nodes can be indicated in path expressions using an ordinal value that chooses exactly one node satisfying the path expression, as shown in the following example. The ordinal [1] is evaluated using the Transact-SQL TOP 1 ascending while the ordinal **last()** is evaluated as TOP 1 descending. The **nodes()** method also sets a singleton context item for each of the resulting XML instances.

If the choice of a single node is omitted, the query optimizer uses a default cardinality estimate which can be much too high. This can, for example, lead to suboptimal choices for the inner and outer loops in nested loop joins when predicates have to be computed. The effect is more pronounced in the case of XML blobs where no XML indexes exist and no statistical information is available for better estimation of the cardinality.

Example: Specifying Singleton Cardinality for Untyped XML

Suppose each XML instance in the xCol column contains a single top-level <book> element that has a single <title> sub-element. Consider the query:

SELECT xCol.query ('/book/title')

FROM docs

The query optimizer uses a default cardinality estimate for <title> element. Each <book> has a single title so that <title> is a singleton element, but the optimizer’s estimate is much higher than that. The reformulated query conveys the correct cardinality to the optimizer:

SELECT xCol.query ('(/book/title)[1]')

FROM docs

The semantic difference between the similar-looking path expressions (/a/b)[1] and /a/b [1] is discussed in the paper [XML Best Practices for Microsoft SQL Server 2005](http://msdn.microsoft.com/library/en-us/dnsql90/html/sql25xmlbp.asp?frame=true) (http://msdn.microsoft.com/library/en-us/dnsql90/html/sql25xmlbp.asp?frame=true) on MSDN.

##### Eliminating Multiple Executions of XML data type Methods

A query such as:

SELECT case isnumeric (xCol.value ('(/book/price)[1]', 'nvarchar(32)'))

when 1 then xCol.value ('(/book/price)[1]', 'decimal(5,2)')

else 0

end

FROM docs

computes the <price> of a boo and converts the price to decimal (5, 2) if it is a numeric type. This logic is useful when the application might encounter non-numeric price values.

This query computes the **value()** method twice (this can be avoided by using a subquery as shown in the following) in which the **value()** method is computed in a subquery and reused in an outer SELECT:

SELECT case isnumeric(Price)

when 1 then CAST (Price AS decimal(5,2))

else 0

end

FROM (SELECT xCol.value ('(/book/price)[1]', 'nvarchar(32)') Price

FROM docs) T

The same optimization can be used in other places as well such as in NULLIF:

SELECT NULLIF (Title, '')

FROM (SELECT xCol.value ('(/book/title)[1]', 'nvarchar(64)') Title

FROM docs) T

Using the **value()** method in NULLIF() would compute the **value()** method twice when it returns a non-empty string.

##### Data(), text(), and string() Accessors

XQuery provides a function **fn:data()** to extract atomic, typed values from nodes, a node test **text()** to return text nodes, and the function **fn:string()** to return the string value of a node. However, their usage can be confusing. Guidelines for their proper use in SQL Server 2005 are illustrated in the following by using the XML instance <age>12</age>:

* **Untyped XML**: The path expression /age/text() returns the text node under <age>, whose value is “12”. The function fn:data(/age) returns the string value “12” as does fn:string(/age).
* **Typed XML**: The expression /age/text() returns static error for any simple typed <age> element in SQL Server 2005. On the other hand, if <age> has a simple integer content, fn:data(/age) returns the integer 12, while fn:string(/age[1]) yields the string “12”.

These functions have different performance characteristics. The **fn:string()** function recursively aggregates all text nodes under the context node. This is overkill when the context node is single-valued. Thus, **fn:data()** and **text()** not only suffice, but also are more efficient.

For untyped XML, when the value of a node is desired, returning a text node using **text()** is faster than **fn:data()**. The path expression /book/text() returns the text node children of a <book> element. Within the **query()** method, these text nodes are serialized out and appear to be a concatenation of the values of the text nodes. On the other hand, **fn:data()** aggregates all values in the subtree of the <book> element. This aggregation makes the computation of **fn:data()** more expensive than **text()** even for elements with simple content.

###### Text Aggregation in Untyped XML

According to XQuery semantics, a query such as the following on untyped XML:

SELECT xCol.value ('(/book/title[.="Writing Secure Code"])[1]',

'nvarchar(64)')

FROM docs

or

SELECT xCol.value ('(/book/title

[fn:string()="Writing Secure Code"])[1]'), 'nvarchar(64)')

FROM docs

requires all text nodes under the <title> element to be aggregated to evaluate the predicate. This inhibits XML index lookup for the search string.

If the <title> element has only one text node, a more efficient way of writing the query is to evaluate the predicate on the text node, as shown in the following:

SELECT xCol.value ('(/book/title/text())[1]

[. = "Writing Secure Code"]', 'nvarchar(64)')

FROM docs

XML index lookup for the value "Writing Secure Code" can occur in this case.

##### Parameterize Your XQuery and XML DML Expression

XQuery and XML DML expressions are not auto-parameterized. Thus, it is preferable to use **sql:column()** or **sql:variable()** to supply parameter values to your XQuery or XML DML expressions, if two XQuery expressions differ only in the values of parameters, instead of using dynamic SQL statements. Using these functions auto-parameterizes the query.

The following example shows a stored procedure execution. The technique can be applied to parameterization of any query, function or method call, or data modification statement.

For example, the following stored procedure finds books with a lower price than the input argument:

CREATE PROC sp\_myProc

@Price decimal

AS

SELECT \*

FROM docs

WHERE 1 = xCol.exist('(/book/price)[. < sql:variable("@Price")]')

In ADO.NET and OLEDB, bind the input value of @Price to a parameter. This avoids query recompilation when the parameter is bound to a different value. Using **sql:column()** yields similar benefits.

The following Microsoft® Visual Basic® .NET code shows parameter binding in the stored procedure invocation:

'myConn is the connection string

SqlCommand cmd = New SqlCommand("sp\_myProc", myConn)

cmd.CommandType = CommandType.StoredProcedure

'Parameter binding

Dim myParm As SqlParameter = cmd.Parameters.Add("@Price", \_

SqlDbType.Decimal)

myParm.Direction = ParameterDirection.Input

myParm.value = 2

'Invoke the stored procedure

SqlDataReader myReader = cmd.ExecuteReader()

'Invoke the stored procedure a second time

myParm.value = 49.99

SqlDataReader myReader = cmd.ExecuteReader()

For more information, see the Microsoft® Visual Studio® .NET documentation.

Example: Using sql:variable() in Data Modification

Suppose the <price> of a <book> whose ISBN is "0-2016-3361-2" is discounted by 10 percent. Both the discount and the ISBN can be passed into the XML data modification statement as parameters, and the statement remains the same for a different book or a different discount.

DECLARE @discountFactor float, @sqlisbn nvarchar(32)

SET @discountFactor = 0.9

SET @sqlisbn = N'0-7356-1588-2'

UPDATE docs

SET xCol.modify('replace value of (/book/price/text())[1] with

sql:variable("@discountFactor")\*(/book/price/text())[1]')

WHERE xCol.exist('/book[@ISBN = sql:variable("@sqlisbn")]') = 1

Example: Using sql:variable() in Element Construction

The **modify()** method shown in the following illustrates the use of **sql:variable()** for supplying a value within a constructed element:

DECLARE @name nvarchar(64)

SET @name = 'Microsoft Press'

UPDATE docs

SET xCol.modify ('

insert <publisher Name = "{sql:variable("@name")}"></publisher>

into (/book/title)[1]')

##### Optimizations for Predicates and Ordinals

Full paths (absolute location paths from the root node to selected nodes containing only child and self axes) without node tests or branching (without predicates or ordinals on intermediate nodes in the path) can be evaluated more efficiently than path expressions with branching. In the indexed case, full paths can be used in index seeks. For XML blobs, parsing is faster for such paths than for paths with branching or wildcards (\*).

Node tests and predicates at the end of a full path are used as filters on the selected nodes. Indexes are used. For XML blobs, parsing is efficient. Following is an example.

Example: Full-path Evaluation

Consider the path expression that selects books written by authors whose first name is Davis:

SELECT xCol.query ('/book[author/first-name = "Davis"]')

FROM docs

Although the predicate is not directly on the <book> element, the <first-name> nodes located using the collapsed path /book/author/first-name are filtered by the value “Davis”. The returned <book> elements are those satisfying the given predicate.

Path-based lookup is efficient even for partially specified paths without predicates or ordinals, such as /book//first-name. The query compiler uses the LIKE operator to match such paths in the XML indexes. Thus, specifying as much of the path as possible contributes to more efficient processing.

Branching (node tests and predicates in the middle of a path expression) as in /book[@ISBN = “1-8610-0157-6”]/author/first-name evaluates the path expressions /book[@ISBN = “1-8610-0157-6”] and /book/author/first-name, and takes the intersection between the two sets of <book> elements. Consequently, execution is slower than path expressions without branching. The use of node tests and predicates in the middle of path expressions should be avoided as much as practical. This is sometimes possible with careful data modeling, as discussed in the example, Generic vs. Specific Markups!href(GenericMarkupsExample).

###### Moving Ordinals to the End of Paths

Ordinals used in path expressions for static type correctness are good candidates for placement at the end of path expressions. The path expression /book[1]/title[1] is equivalent to (/book/title)[1] if every <book> element has <title> children. The latter can be evaluated faster for both the XML indexed case and the XML blob case by determining the first <title> element under a <book> element in document order. Similarly, the path expression (/book/@ISBN)[1] yields faster execution than /book[1]/@ISBN.

###### Evaluating Predicates by Using the Context Node

In addition to moving predicates, ordinals, and node tests to the end of path expressions, evaluating these conditions by using the context node yields still better performance. Following is an example of this rewrite.

Example: Predicate Evaluation Using the Context Node

The following query finds books on the subject of “security”. It requires the evaluation of two path expressions, namely /book and /book/@subject, and a check for the value “security” for the latter path.

SELECT \*

FROM docs

WHERE xCol.exist ('/book[@subject = "security"]') = 1

The following rewrite evaluates a single path /book/@subject and checks whether this path has the value “security”. This yields a simpler query plan than the previous one and is much faster.

SELECT \*

FROM docs

WHERE xCol.exist ('/book/@subject[. = "security"]') = 1

##### Range Conditions

Range conditions benefit from the use of typed XML. The data stored in XML columns and XML indexes are typed according to the type definitions specified in XML schemas. Value comparisons avoid run-time conversion of data and permit range scans on the VALUE secondary XML index. This also requires specifying the context node (.) in the range condition for efficient access, as the following example illustrates.

Example: Context Node in Range Conditions

Consider the query to find books in the typed XML column xBook of table TypedBooks whose price is in the range of $9.99 to $49.99:

SELECT xBook

FROM TypedBooks

WHERE xBook.exist ('

declare default element namespace "http://www.microsoft.com/book";

/book[price > 9.99 and price < 49.99]') = 1

The path expressions /book/price > 9.99 and /book/price < 49.99 are evaluated separately. The query optimizer does not know that the <price> elements are the same, because multiple <price> elements can exist under the <book> element. This inhibits range scan on the VALUE secondary XML index. The following rewrite ensures that the same context node for <price> is used, and range scan of the VALUE secondary XML index occurs for values between 9.99 and 49.99. This yields better performance:

SELECT xBook

FROM TypedBooks

WHERE xBook.exist ('

declare default element namespace "http://www.microsoft.com/book";

/book/price[. > 9.99 and . < 49.99]') = 1

##### Parent Axis

Use of the parent axis in path expressions blocks certain optimizations. The XQuery compiler concatenates segments of path expressions without branching into a longer path, which can be evaluated more efficiently than the segments separately. This optimization is referred to as path collapsing.

This technique, however, does not work for the parent axis. Furthermore, XML indexes cannot be used to evaluate parent axes. Instead, using paths for forward traversal only yields better performance. For example, instead of writing /book/title[../@ISBN = "0-7356-1588-2"], it is better to write the path expression as /book[@ISBN = "0-7356-1588-2"]/title.

Navigating to the parent of a node in typed XML loses type information and returns an element of the most general type xs:anyType. Further operations on the node may need explicit cast and slows down query processing. Also, the cast may prevent XML index use. It is advantageous to navigate down from the parent node instead of climbing up to the parent.

##### Dynamic Querying

XQuery expressions are specified as literals within XML data type methods. Their evaluation uses XML indexes as available and as chosen by the query optimizer.

Application development is convenient when XQuery expressions can be dynamically specified instead of literals. This is possible in the following ways:

* Query Construction

Create the query as a string and use **sp\_executesql** for its execution. Unlike **EXEC**, this caches the compiled query plan and the optimizer may reuse the compiled plan. The query can be parameterized, because it is formed as a string and may contain embedded parameters. Adequate care should be taken to avoid SQL injection attacks.

* Use XPath Functions

Replace each location step in an XPath expression with the **name()** function, or **local-name()** and **namespace-URI()** functions. This yields a query to which you can pass in node names and search values. You can parameterize further as described in the Parameterize Your XQuery or XML DML Expression!href(ParameterizeExpressions) example. Such parameterized queries are convenient for developing applications. However, the query plan generated for it ignores XML indexes, because specific paths are not known at compilation time.

Although the query construction approach performs better than parameterizing the path expressions, it includes the cost of run-time query compilation, which makes it slower than specifying the full query as a literal. The actual query passed in by the user must be validated to avoid SQL injection attacks. Otherwise, this approach should be avoided in favor of parameterization of the query. For more information, see the Parameterize Your XQuery and XML DML Expression!href(ParameterizeExpressions) example. The following example illustrates this approach.

The second approach specifying node tests using node names avoids the SQL injection problem. However, the query plan becomes complex and performs much worse than the original query. This is shown in the second example that follows.

Example: Query Using sp\_executesql

Suppose you want to create the following query dynamically and pass in the search value of @subject using a parameter:

SELECT \*

FROM docs

WHERE xCol.exist('/book/@subject [. = "security"]') = 1

The dynamic query can be created and executed as shown in the following. The query string is created in the variable @SQLString and contains an embedded variable @bksubj used in the **exist()** method. The variable @subj supplies the run-time value of the parameter. The dynamic query passed in using @SQLString should be validated (not shown) to avoid SQL injection attacks.

DECLARE @SQLString NVARCHAR(500)

DECLARE @subj NVARCHAR(64)

DECLARE @ParmDefinition NVARCHAR(500)

--- Build the SQL string once

SET @SQLString =

N'SELECT \*

FROM docs

WHERE xCol.exist(''/book/@subject[. =sql:variable("@bksubj")]'')=1'

SET @ParmDefinition = N'@bksubj NVARCHAR(64)'

--- Execute the string with the first parameter value

SET @subj = 'security'

EXECUTE sp\_executesql @SQLString, @ParmDefinition,

@bksubj = @subj

Example: Query Using local-name()

The previous query can be rewritten to use tag names as literals as follows:

DECLARE @elemName nvarchar(4000), @attrName nvarchar(4000)

DECLARE @subjValue nvarchar(4000)

SET @elemName = N**'**book**'**

SET @attrName = N**'**subject**'**

SET @subjValue = N**'**security**'**

SELECT \*

FROM docs

WHERE xCol.exist(**'**/\*[local-name() = sql:variable("@elemName") and

@\*[local-name() = sql:variable("@attrName") and

. = sql:variable("@subjValue")]]**'**) = 1

The rewritten query contains wildcards (\*) and node tests using node names and is hard to optimize well. Consequently, it performs much worse than the original query and the query construction approach.

##### Rowset Generation from XML Data

Some applications need to generate a rowset from XML data by promoting one or more properties into columns of the rowset. For example, an application may query for the authors of books and display the result as a table containing two columns for the first and last names. Such rowset generation can be done both at the server and at the client with different performance characteristics:

* At the server, use one of the following mechanisms:
* Combination of **nodes()** and **value()** methods on XML data type
* OpenXML
* Streaming table-valued function in common language runtime (CLR)
* Alternatively, the XML result is returned to the client, which uses client-side programming (DataSet) to convert the data to a rowset.

Client-side rowset generation offloads the server and is useful when almost the entire data sent from the server to the client is mapped into the rowset. Otherwise, the cost of shipping the data may outweigh the benefits of client-side processing.

Server-side rowset generation is useful for rowset generation from incoming XML data at the server. It is generally preferable when a small fraction of XML data stored at the server is promoted into columns of the rowset. More discussions regarding the relative merits and demerits among the server-side approaches can be found in [XML Best Practices for Microsoft SQL Server 2005](http://msdn.microsoft.com/library/en-us/dnsql90/html/sql25xmlbp.asp?frame=true) (http://msdn.microsoft.com/library/en-us/dnsql90/html/sql25xmlbp.asp?frame=true) on MSDN.

#### Conclusion

This paper discusses several ideas for optimizing the performance of applications using XML data type. These ideas range from data management aspects and XML schema design to the style for writing queries on XML data type. Employing these techniques can yield significant performance boost. It is helpful to study the Showplan output to see how the queries benefit from the XML indexes and to experiment with query rewrites to see how the query plans change.

For more information:

<http://msdn.microsoft.com/sql/>