Indexing XML Data Stored in a Relational Database

Shankar Pal, Istvan Cseri, Oliver Seeliger, Gideon Schaller, Leo Giakoumakis, Vasili Zolotov

Microsoft Corporation
One Microsoft Way
Redmond WA 98052
USA
{shankarp, istvanc, oliverse, gideons, leogia, vasilizo}@microsoft.com

Abstract

As XML usage grows for both data-centric and document-centric applications, introducing native support for XML data in relational databases brings significant benefits. It provides a more mature platform for the XML data model and serves as the basis for interoperability between relational and XML data. Whereas query processing on XML data shredded into one or more relational tables is well understood, it provides limited support for the XML data model. XML data can be persisted as a byte sequence (BLOB) in columns of tables to support the XML model more faithfully. This introduces new challenges for query processing such as the ability to index the XML blob for good query performance. This paper reports novel techniques for indexing XML data in the upcoming version of Microsoft® SQL Server™, and how it ties into the relational framework for query processing.

1. Introduction

Introducing XML [3] support in relational databases has been of keen interest in the industry in the past few years. One solution is to generate XML from a set of tables based on an XML schema definition and to decompose XML instances into such tables [2][5][11] [16][20]. Once shredded into tables, the full power of the relational engine, such as indexing using B^+trees and query capabilities, can be used to manage and query the data.

**Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004**

The shredding approach is suitable for XML data with a well-defined structure. It depends on the existence of a schema describing the XML data and a mapping of XML data between the relational and XML forms.

The XML data model, however, has characteristics that make it very hard if not practically impossible to map to the relational data model in the general case. XML data is hierarchical and may have a recursive structure; relational databases provide weak support for hierarchical data (modeled as foreign key relationships). Document order is an inherent property of XML instances and must be preserved in query results. This is in contrast with relational data, which is unordered, and order must be enforced with additional ordering columns. On the query front, a large number of joins are required to re-assemble the result for realistic schemas. Even with co-located indexes, the reassembly cost of an XML subtree can be prohibitively expensive.

XML is being increasingly used in enterprise applications for modeling semi-structured and unstructured data, and for data whose structure is highly variable or not known *a priori*. This has motivated the need for native XML support within relational databases.

Microsoft SQL Server 2005 introduces a native data type called XML [12]. A user can create a table T with one or more columns of type XML besides relational columns. XML values are stored in the XML column as large binary objects (BLOB). This preserves the XML data model faithfully, and the query processor enforces XML semantics during query execution. The underlying relational infrastructure is used extensively for this purpose. This approach supports interoperability between relational and XML data within the same database making way for more widespread adoption of the XML features.

XQuery expressions [19] embedded within SQL statements are used to query into XML data type values. Query execution processes each XML instance at runtime; this becomes expensive whenever the instance is large in size or the query is evaluated on a large number of rows in the table. Consequently, an indexing mechanism is required to speed up queries on XML blobs.

B^+tree index has been used extensively in relational databases and is a natural choice for indexing XML blobs as well. The B^+tree index must provide efficient evaluation of queries on XML blobs. Query execution may need to reassemble the XML result from the B^+tree index (*XML serialization*) while preserving document order and document structure. Some operators in XPath 2.0 [18] — most notably the descendant-or-self axis // — navigate down an XML tree recursively. Thus, B^+tree lookups can be recursive.

In this paper, we discuss the techniques used in Microsoft SQL Server 2005 for indexing XML blobs. A shredded representation conforming to Infoset items [4] of nodes is stored in a B^+tree. This is referred to as the *primary XML index*. A novel node labeling scheme called ORDPATH [13] allows us to capture document order and document hierarchy within a single column of the primary XML index. This index is clustered on the ORDPATH value for each XML instance and provides very efficient access to subtrees using a simple range scan. The ORDPATH column is used extensively to determine relative order of nodes within a document and the parent-child and ancestor-descendant relationships between two nodes. The ancestor-descendant relationship check eliminates the need for recursive traversal down the XML tree and is a significant optimization.

Materialization of the Infoset speeds up query processing on XML columns by eliminating runtime shredding costs. Further performance gains can be obtained by creating secondary indexes on the primary XML index for different classes of queries. We identify three important classes of queries (path-based queries, property bag scenarios and value-based queries) that commonly occur in practice and investigate three secondary indexes — PATH, PROPERTY and VALUE — to optimize those classes of queries. Content indexing of XML instances based on the structural information stored in primary XML index is also discussed.

The performance gains using the XML indexes for the well-known XMark benchmark [15] are presented in the paper.

The reminder of the paper is organized as follows. Section 2 gives a background of native XML support in Microsoft SQL Server 2005 and describes the concept of ORDPATH. Section 3 introduces the techniques for indexing XML data, Section 4 provides experimental results, and Section 5 discusses related work. The paper concludes with a summary in Section 6.

2. XML Support in Microsoft SQL Server 2005

This section provides a brief overview of XML support in Microsoft SQL Server 2005.

2.1 XML Data Type

Native support for the XML data model is introduced using a new, first-class data type called “xml”. It can be used as the type of a column in a table or view, a variable and a parameter in a function or stored procedure. Thus, a table can be created with an integer column and an XML column as follows:

```
Create  table DOCS (ID int primary key, XDOC xml)
```

XML values saved in the XDOC column can be trees (“XML document”) or fragments (“XML content”). They are stored in an internal, binary representation that is streamable and optimized for query processing. Some compaction occurs, which is incidental rather than the goal of the binary representation.

The supplied XML values are checked for well-formedness and conformity to the XML data model (e.g. end tags match start tags) for storage in the XML column.

The XML column can optionally be typed by a collection of XML schemas that may be related (e.g. by <xs:import>) or unrelated to one another. Each XML instance specifies the XML namespace from the schema collection it conforms to. The database engine validates the instance according to the XML schema before storing it in the XML column.

XML type information is stored in the database’s meta-data. It contains the XML schema collections (and their contained XML schemas) and mapping between the primitive XSD and relational type systems. Typed XML instances contain XSD type information in the internal, binary representation. This enables efficient processing for typed XML and allows building domain based value indexes for efficient lookups.

2.2 Node Labeling Using OrdPath

ORDPATH [13] is a mechanism for labelling nodes in an XML tree, which preserves structural fidelity. It allows insertion of nodes anywhere in the XML tree without the need for re-labelling existing nodes. It is independent of XML schemas typing XML instances.

ORDPATH encodes the parent-child relationship by extending the parent’s ORDPATH with a labelling component for the child. In the following, we use a string representation for the ORDPATH to illustrate the idea while the internal representation is based a compressed binary form. For example, children of a parent node labelled with the ORDPATH "1.5.3.9" may have the labels "1.5.3.9.1" and "1.5.3.9.7", where the ending "1"and "7" are labelling components for the children. A byte comparison of two ORDPATH labels yields the relative order of the nodes in the XML tree. Thus, the child "1.5.3.9.1" precedes "1.5.3.9.7" in document order.

For the XML instance shown in Figure 1, sample ORDPATH labels are shown for the corresponding XML tree in Figure 2.

```
<BOOK ISBN="1-55860-438-3">
    <SECTION>
        <TITLE>Bad Bugs</TITLE>
        Nobody loves bad bugs.
        <FIGURE CAPTION="Sample bug"/>
    </SECTION>
    <SECTION>
        <TITLE>Tree Frogs</TITLE>
        All right-thinking people
        <BOLD> love </BOLD> tree frogs.
    </SECTION>
</BOOK>
```

Figure 1. Sample XML data

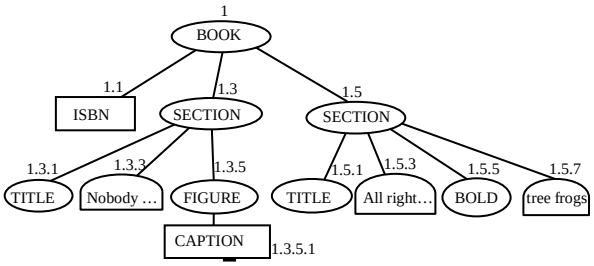

Figure 2. ORDPATH Node Label

In the ORDPATH values shown in Figure 2 (such as "1.3.5.1"), each dot separated component value ("1", "3", "5", "1") reflects a numbered tree edge at successive levels on the path from the root to the labelled node. Only positive odd integers are assigned during an initial load; even-numbered and negative integer component values are reserved for later insertions into an existing tree.

A new node N (possibly the root node of a subtree) can be inserted under any node in an existing tree. It is assigned a label component in between those of its left and right siblings using an even numbered auxiliary position that introduces a new level for N. This preserves the relative order between the siblings and avoids re-labelling the left or right siblings of N. Leftmost and rightmost insertion is supported equally efficiently by extending the range of label components on both ends. Leftmost insertions may generate label components that are negative numbers.

2.3 XML Query Processing

XQuery [19] embedded in SQL is the language supported for querying XML data type. XQuery is a W3C standards-based language in development. It is a very powerful functional language for querying XML data. In particular, it includes XPath 2.0 [18].

Methods are provided on XML data type for querying into XML values. These methods accept XQuery expressions as arguments. The methods are:

- query(): returns XML data type
- value(): extracts scalar values
- exist(): checks conditions on XML nodes
- nodes(): returns a rowset of XML nodes that the XQuery expression evaluates to

As an example, consider the following query that retrieves section titles in the book with a specified ISBN:

```
SELECT ID, XDOC.query('
  for $s in
      /BOOK[@ISBN= "1-55860-438-3"]//SECTION
  return <topic>{data($s/TITLE)} </topic>')
FROM DOCS
```

Query execution is tuple-oriented as in the rest of the relational framework. The SELECT list is evaluated on each row of table DOCS and produces a two-column result. Query compilation proceeds by producing a single query plan for both the relational and the XML parts of the query, and the overall query tree is optimized by the cost-based query optimizer.

The XML data type methods process the XML instances on which they are invoked. Each XML instance can be up to 2GB in storage, so that the runtime shredding cost can be significant for large XML instances.

In the next section, we consider techniques for indexing XML instances to speed up queries.

3. Indexing XML Data

For an XPath expression such as /BOOK[@ISBN = "1-55860-438-3"]//SECTION shown in Section 2.3 and executed on the XDOC column of DOCS table, the XPath expression is evaluated on all rows in the table. This is costly for the following reasons:

- The XDOC column value in each row must be shredded at runtime to evaluate the query.
- We cannot determine which of the XML instances satisfies @ISBN = "1-55860-438-3" without processing the XDOC values in all rows.

We can speed up query processing by saving the parsing cost at runtime. This is achieved by materializing the shredded form of the XML instances in a B^+tree that retains structural fidelity of the XML instances in the XDOC column. The query processor decides whether to process rows of the base table before those in the XML index (*top-down execution*) or use targeted seeks or scans on the XML index first followed by a back join with the base table (*bottom-up execution*). (The table in which an XML column is defined is referred to as the *base table*.) Additional secondary XML indexes provide another degree of freedom for the optimizer to choose the execution plan.

This section introduces the notion of a *primary XML index* on an XML column. It is a B^+tree that materializes the Infoset content of each XML instance in the XML

column. Indexing the Infoset content in additional ways is discussed as *secondary XML indexes*.

In the following discussions, we use table DOCS of Section 2.1 for illustrative purposes.

3.1 Primary XML Indexes

This subsection describes the structure of the primary XML index and discusses query execution using it.

3.1.1 Structure of Primary XML Index

The B^+tree containing the shredded form of the XML instances in a column is called the *primary XML index* or the "Infoset" table.

We generate a subset of the fields in the Infoset items of the XML nodes by shredding an XML instance. This is stored in a B^+tree in the system. The Infoset contains information such as the tag, value and parent of each node; we add the path from the root of the tree to the node to allow path-based lookups. The B^+tree has the following columns amongst others:

ORDPATH	TAG	NODE_ TYPE	VALUE	PATH_ ID
1	1 (BOOK)	1 (Element)	Null	#1
1.1	2 (ISBN)	2 (Attribute)	'1-55860-438-3'	#2#1
1.3	3 (SECTION)	1 (Element)	Null	#3#1
1.3.1	4 (TITLE)	1 (Element)	'Bad Bugs'	#4#3#1
1.3.3	10 (TEXT)	4 (Value)	'Nobody loves Bad bugs.'	#10#3#1
1.3.5	5 (FIGURE)	1 (Element)	Null	#5#3#1
1.3.5.1	6 (CAPTION)	2 (Attribute)	'Sample bug'	#6#3#1
1.5	3 (SECTION)	1 (Element)	Null	#3#1
1.5.1	4 (TITLE)	1 (Element)	'Tree frogs'	#4#3#1
1.5.3	10 (TEXT)	4 (Value)	'All right-thinking people'	#10#3#1
1.5.5	7 (BOLD)	1 (Element)	'love '	#7#3#1
1.5.7	10 (TEXT)	4 (Value)	'tree frogs'	#10#3#1

Figure 3. XML "Shredded" into relational Infoset table

Figure 3 shows the rows corresponding to the XML tree in Figure 2. The ORDPATH column preserves structural fidelity within a single XML instance; the Infoset table also contains the primary key column ID of the base table (not shown) for *back join*. The primary key of the Infoset table is the combination of the primary key ID of the base table and the ORDPATH column.

The TAG column shows the markups found in the XML instance; it is used here for illustrative purposes only. Instead of storing string values, each markup is mapped to an integer value and the mapped values are used in storage. This mapping is referred to as *tokenization* and yields significant compression.

The NODE_TYPE column stores the type of the node in the Infoset content. For typed XML column, it stores a tokenized type value corresponding to the XSD type of the node.

The VALUE column stores the node's value, if one exists, otherwise it is NULL. It stores typed XML values as SQL Server's native type within a generic variant type.

The PATH_ID column contains a tokenized path value from the root to the node. This column represents all the paths in the tree similar to the dataguide computation [7]. Whereas each node within an XML instance has a distinct ORDPATH value, the PATH_ID value is the same for multiple nodes with the same path. Thus, nodes 1.3.1 and 1.5.1 refer to two different TITLE nodes but the paths leading to these nodes are both expressed as /BOOK/SECTION/TITLE. As such, they have the same PATH_ID value #4#3#1, where #1, #3 and #4 are for BOOK, SECTION and TITLE, respectively.

Nodes of the XML tree are traversed in XML document order and ORDPATH labels are generated during the population of the primary XML index.

The primary XML index contains some redundancy and is larger in size than the textual form of the XML instance; the primary key column of the base table, ID, for example is repeated in all rows for an XML instance. The increased I/O cost, added to the serialization cost of converting shredded rows in the Infoset table to XML form, makes retrieval of the XML blob cheaper from the base table when the whole XML instance is required.

Primary XML index stores values using the SQL type system. Most of the SQL types are compatible with XQuery type system (e.g. integer), and value comparisons on XML index columns suffice. A handful of types (e.g. xs:datetime) are stored in an internal format and processed specially to preserve compatibility with the XQuery type system.

The primary XML index can be optimized in various ways, such as by generating a single row for simple-valued elements (instead of two rows). This in practice significantly reduces on-disk size. Prefix compression [1] reduces the size of the primary XML index significantly. Another optimization is to point back from the VALUE column for large-sized values to the XML blob to avoid redundancy. A more detailed discussion of these and other optimizations are beyond the scope of this paper.

3.1.2 Query Compilation and Execution

An XQuery expression is translated into relational operations on the Infoset table. The result is a set of rows from the Infoset table that must be re-assembled into an XML result.

Consider the evaluation of the path expression /BOOK[@ISBN = "1-55860-438-3"]/SECTION on an XML instance. The following SQL statement expresses the execution logic. PATH_ID (*path*) yields the tokenized path value for the specified *path*. SerializeXML (ID, ORDPATH) assembles the XML subtree rooted at the node (ID, ORDPATH) from the Infoset table. Parent (C-

ORDPATH) returns the parent's ORDPATH as the prefix of C-ORDPATH without the last component for the child.

```
SELECT SerializeXML (N2.ID, N2.ORDPATH)
FROM   infosettab N1
       JOIN infosettab N2 ON (N1.ID = N2.ID)
WHERE N1.PATH_ID = PATH_ID(/BOOK/@ISBN)
AND    N1.VALUE = '1-55860-438-3'
AND    N2.PATH_ID = PATH_ID(
                          BOOK/SECTION)
AND    Parent (N1.ORDPATH) =
       Parent (N2.ORDPATH)
```

When the path expression /BOOK[@ISBN = "1-55860-438-3"]/SECTION is evaluated on the XDOC column of a row in DOCS table, the primary key value ID is used to seek into the Infoset table (N1). Rows for the XML instance in N1 are scanned to locate the ones having the values /BOOK@ISBN and "1-55860-438-3" in the PATH_ID and the VALUE columns, respectively. Using the same primary key value, the execution seeks into the Infoset table a second time (N2), finds rows containing the PATH_ID value for /BOOK/SECTION and determines whether the BOOK elements found in N1 is the parent of the SECTION elements found in N2. The XML fragments corresponding to the qualifying SECTION element are serialized from the Infoset table.

The cost of reassembly may be non-trivial. For queries that retrieve the whole XML instance, it is cheaper to retrieve the XML blob. Similarly, a query containing a simple path expression that must be evaluated on all rows of the base table may be more efficient on the XML blob than on the primary XML index if the re-assembly cost outweighs the cost of parsing the XML blobs. A cost-based decision must be made whether to execute the query by shredding XML blobs at runtime or to operate on XML indexes.

Insertion, deletion and modification of XML values require primary XML index maintenance as is to be expected.

3.2 Secondary XML Indexes

The primary XML index is clustered in document order and each path expression is evaluated by scanning all rows in the primary XML index for a given XML instance. Performance slows down for large XML values.

Secondary indexes can be created on the primary XML index to speed up different classes of queries. While a secondary index can be created on any of the columns in the primary XML index, it is interesting to study the specific indexes that benefit common classes of queries. We introduce four such index types: PATH (and its variation PATH_VALUE), PROPERTY, VALUE and content indexing in the following subsections.

Secondary XML indexes help with bottom-up evaluation. After the qualifying XML nodes have been found in the secondary XML indexes, a back join with the primary XML index enables continuation of query execution with those nodes. This yields significant performance gains.

3.2.1 PATH and PATH_VALUE Indexes

Going back to the SQL rewrite in Section 3.1.2, evaluation of path expressions over an entire XML column benefits from a secondary index built on the PATH_ID column. The path expression is compiled into the tokenized form (e.g. /BOOK/@ISBN $\Rightarrow$ #2#1 in the example of Figure 3). An index with PATH_ID as the leading key column helps such queries.

The PATH index is built on the columns PATH_ID, ID and ORDPATH, where ID is the primary key of the base table. During query evaluation, the tokenized path value PATH_ID and ID are used to seek into the PATH index and find the corresponding ORDPATH values, thereby saving the cost of primary XML index scans. The index seek is what brings the performance gain, and the cost is relatively independent of the path length. A back join with the primary XML index on ID and ORDPATH pair continues with query execution to check conditions such as the specified value of ISBN, and re-assemble the resulting XML fragments (e.g. the subtrees rooted at the SECTION nodes in our example).

The PATH_ID column stores a "reversed" representation of the path. When a full path such as /BOOK/SECTION/TITLE is specified, it is mapped into the value #4#3#1 for PATH index lookup; the full PATH_ID value is known in this case. However, a wildcard or the descendant-or-self (//) or the descendant axis requires careful handling.

For a path expression containing the //-axis, such as //SECTION/TITLE, only the last two steps in the path expression are known. Storing the forward path in the PATH_ID column is not very useful in this case; the entire PATH index would have to be scanned. With the reverse path, however, prefix match of the PATH_ID column for the value #4#3 yields faster execution. The situation is similar for path expressions containing a wildcard or //-axis in the middle of the path expression, such as /BOOK/*/TITLE or /BOOK/SECTION//TITLE. In the latter case, the exact match for the PATH_ID value for /BOOK/SECTION (i.e. #3#1) and prefix match for TITLE (i.e. #4) yield two sets of nodes. The ancestor-descendant relationship between node pairs from these sets is verified using their ORDPATH values.

For path expressions such as /BOOK/SECTION[TITLE ="Tree Frogs"] that fit the pattern "path=value", a variation of the PATH index is more useful. If the PATH index is built only on the PATH_ID column, this type of query requires a back join with the primary XML index to check the node's value. This back join can be avoided by including the VALUE column in the index to yield a PATH_VALUE index, which is built on the columns (PATH_ID, VALUE, ID

and ORDPATH). The path /BOOK/SECTION/TITLE is compiled to the tokenized value #4#3#1 and an index seek is performed on the PATH_VALUE index with the key values (#4#3#1, "Tree Frogs"). For the qualifying TITLE nodes, the parent's key value (ID, Parent (ORDPATH)) is then used to seek into the primary XML index to obtain and re-assemble the SECTION subtrees in the result.

3.2.2 PROPERTY Index

A useful application of XML is to represent an object's properties with the help of XML markup, especially when the number and type of the properties are not known *a priori*, or properties are multi-valued or complex. This allows properties of different types of objects to be stored in the same XML column. The XML schema (if one exists) for this scenario is typically non-recursive.

Common queries have the form "find properties X, Y, Z of object P", where X, Y and Z are path expressions. In our model, this means the ID value is known for the object and the PATH_ID values are know for X, Y and Z. Evaluating this query on the primary XML index requires scanning all rows corresponding to the given ID value.

On the other hand, the rows for each of the paths X, Y and Z from all objects are clustered together in the PATH_VALUE index. Thus, the execution becomes a seek into the PATH_VALUE index for each of the paths, scan of all rows with the same PATH_ID value and a match for the specified ID value.

Clustering all properties of each object together into a PROPERTY index significantly speeds up property lookup for objects. The columns in the PROPERTY index are (ID, PATH_ID, VALUE and ORDPATH). This organization helps retrieve multi-valued properties for an object (same ID and PATH_ID values). Retrieving all properties of an object requires scanning the same number of rows in the primary XML index and the PROPERTY index. However, the higher record density of the PROPERTY index yields faster result, especially when no back join with the primary XML index is required.

To illustrate the point with an example, consider the extractions of the ISBN (i.e. /BOOK/@ISBN) and the title of the first section (i.e. (/BOOK/SECTION/TITLE)[1]) from the XDOC column of table DOCS. The execution logic can be expressed in the following SQL statement:

```
SELECT (SELECT TOP 1 N1.VALUE,
          FROM    infosettab N1
          WHERE DOCS.ID = N1.ID
          AND      N1.PATH_ID =
          PATH_ID (/BOOK/@ISBN)),
           (SELECT TOP 1 N2.VALUE,
           FROM    infosettab N2
           WHERE DOCS.ID = N2.ID
           AND      N2.PATH_ID =
           PATH_ID(/BOOK/ SECTION/TITLE))
FROM    DOCS
```

The primary key ID and the PATH_ID values are known, so that seeking into the PROPERTY index permits efficient retrieval of the ISBN and TITLE values.

To retrieve a single property of an object, the PROPERTY index is more suitable than the PATH_VALUE index, since the latter clusters the same path from all objects together. When N properties are to be retrieved, the cost-based optimizer must decide between N seeks into the PROPERTY index (same ID, N different PATH_ID values) or a scan in the PROPERTY index for the N property values of the object.

3.2.3 VALUE Index

Value-based queries of the type /BOOK/SECTION[FIGURE/@* = "Sample Bug"] specify a value and have a wildcard for the path. It requires scanning the primary XML or PROPERTY index for each XML instance while trying to match the specified portion of the path. Using the PATH_VALUE index is worse and a larger part of the index is usually scanned.

For efficiency, an index that locates the specified value first can induce a bottom-up query plan and perform much better. Such an index is the VALUE index built on the columns (VALUE, PATH_ID, ID and ORDPATH). An index lookup occurs using the value "Sample Bug" and, for the qualifying rows, the specified part of the PATH_ID is matched. A back join with the primary XML index is generally needed to re-assemble the result (the ancestor node SECTION in this example). As noted above, the ORDPATH of a parent or ancestor can be computed as a prefix of a descendant's ORDPATH.

If the XML column is typed, then values stored in the index receive appropriate typing. If the XML column is untyped, then values are indexed as strings. Untyped XML is more beneficial for document scenarios than data scenarios.

As an example, consider the evaluation of the path expression /BOOK/SECTION[FIGURE/@* = "Sample Bug"] on an XML instance. The following SQL statement expresses the execution logic:

```
SELECT SerializeXML (N1.ID,
                     Parent (N1.ORDPATH))
FROM    infosettab N1 JOIN infosettab N2 ON
        (N1.ID = N2.ID AND
         N1.ORDPATH = Parent(N2.ORDPATH))
WHERE N1.PATH_ID =
        PATH_ID(/BOOK/SECTION/FIGURE)
AND     N2.NODE_TYPE = Attribute
AND     N2.VALUE = 'Sample Bug'
```

An index seek into the VALUE index with the search value 'Sample Bug' yields (ID, ORDPATH) pairs that are joined with the primary XML index. Each such (ID, ORDPATH) node is checked for attribute type and child relationship to the nodes found for the path

/BOOK/SECTION/FIGURE. The resulting SECTION elements are serialized in the result.

3.2.4 Content Indexing

The origin of the XML standard is in the document community where the most important part of an XML instance is the text (the "content") in the document marked up by the tag structure. Accordingly there has been increasing amount of focus on information retrieval (IR) techniques in the XML space. These range from simply discarding the markup and using traditional inverted word list techniques augmented with tag/path information to include the markup in the full text index and so leverage the IR search even for element and attribute names.

We support two solutions in this space. We can leverage the IR capabilities of the engine by creating a full text index over an XML data type column. The filter in the text indexer discards the markup and creates an inverted word index with full support of our SQL text search sublanguage over the XML data type instances. The text search expressions now can be combined with XQuery expressions in the same SQL statement and the optimizer leverages all existing indexes (relational, XML and full text) in order to evaluate the query efficiently.

This solution works well for traditional IR queries but it is not optimal if we want to combine searching for a certain word within a specific context, for example, in a particular XML element. Here we want to take advantage of the XML indexes we build over the XML infoset but we want to have finer granularity than text nodes since the VALUE index does not help us locate individual words efficiently. In order to achieve this we can extend the full text inverted word index with information from the infoset or we can extend our infoset table with word information. Here we choose the later solution by building what we call the *word break index*.

The word break index has the same structure as the infoset table except that we break up the text nodes into words according to XML whitespace. Now we can take advantage of all the information present in this table and we can do efficient fine granularity searches on XML whitespace boundaries and tag boundaries. This does not replace a fully annotated full text index since it does not have weighting, ranking and relevance-oriented information [9] but it provides a very efficient index structure for most of the full text like searches.

3.3 Evaluating Complex Path Expressions

A complex path expression may require multiple lookups of one or more XML indexes. Rows found in different lookups are joined (on the primary key ID and ORDPATH in the most common cases) as required for evaluating the path expression. (Section 4 discusses several examples.) This is executed using the proper JOIN type (nested loop join, merge join or hash join [17]). Thus, the overall execution consists of relational operations with special optimizations for ORDPATH properties (order and hierarchy).

A complex path expression is rewritten to use the primary XML index as shown in the previous sections. The choice of PATH, PROPERTY and VALUE indexes are done by the cost-based optimizer using such information as the distributions of PATH_ID, VALUE, primary key and ORDPATH. The query rewrites in the above sections also indicate that the query optimizer may choose to use multiple XML indexes, and evaluate parts of the XPath expression using a post-filter on the output of the index lookups.

The next section presents experimental data on the gain in query performance using various XML indexes.

4. Experimental Results using XMark Benchmark

XMark [15] is an XML query benchmark that models an auction scenario. It specifies 20 queries for exact match, ordered access, regular path expressions, following references, construction of complex results, join on values, search for missing elements, and so on.

This section reports the performance improvements we found with different XML indexes. We explain the reasons for the performance gain for several queries.

4.1 Workload

Sample XML data conforming to the XMark schema was produced using the document generator XMLGEN provided by the authors of XMark. Instead of storing the entire data as a single, large XML instance, it is more natural in a relational database to store the data in tables representing the different entities in the data model. This yields five tables for people, open auctions, closed auctions, items and categories.

Information about bidders is stored in the table PEOPLE, while those about ongoing and closed auctions are stored in the tables OPEN_AUCTIONS and CLOSED_AUCTIONS, respectively. The table ITEMS contains data about the auction items. Lastly, the CATEGORIES table contains information on the classification scheme of items.

Each of these tables contains two columns: an integer id column and an untyped XML column containing the data. The table schema is shown in the appendix. XML indexes of the same type are created on all the XML columns to measure the usefulness of that index type.

Cross references among XML instances is maintained as ordinary attributes instead of IDREF since the reference is across XML instances with our five tables. For example, the bidder of an open auction is stored as a "person" attribute with the person's id as the value in the open auction XML instance.

We manually rewrote the original XMark queries to use joins among our five tables. Some of the query rewrites are shown in the appendix.

We generated data only for the North America region and changed Q9 accordingly to avoid returning an empty result for Europe. Q13 (reconstruction query) does not have an auction item that satisfies the path /site/regions/australia/item used in the query. An optimization in the relational engine knows upfront that no rows will be returned and the path expression is not executed in the indexed case. We changed the query slightly to use "africa" instead of "australia" to return a non-null result.

4.2 Experimental Setup and Results

The XMark database is created for scale factors 0.5 and 30, the latter having sixty times as many rows in each table as the former. The size of the XML data type instances are the same in both cases.

XMLGEN generates a single XML instance whose size is 60 MB for scale 0.5 and 3.35 GB for scale 30. The number of rows in the PEOPLE, OPEN_AUCTIONS, CLOSED_AUCTIONS, ITEMS and CATEGORIES tables are 12750, 6000, 4875, 10875 and 500, respectively, for scale 0.5, and 765000, 360000, 292500, 652500 and 30000, respectively, for scale 30.

The disk space consumption for scale factor 0.5 is 142 MB for the five tables and 345 MB for the primary XML indexes. The secondary XML indexes of each type (PATH, PROPERTY and VALUE) took up another 101 MB. The corresponding sizes for scale factor 30 are 8.3GB, 20GB and 5.9GB, respectively.

The workload is run in single user mode on a 4-way 700 MHz Pentium III machine running Windows Server 2003. It has 2GB RAM and a 3-disk array of 36GB each. The database is a pre-release build of Microsoft SQL Server 2005. The query execution time is measured at the client.

QUERY	PRIMARY	PATH_ VALUE	PROPERTY	VALUE
Q1	5.8	28.8	6.7	28.8
Q2	2.8	2.6	3.5	2.0
Q3	2.2	1.8	2.3	2.4
Q4	8.3	8.0	7.8	7.7
Q5	2.9	2.9	2.7	2.9
Q6	1.0	1.1	1.2	1.1
Q7	7.9	43.6	14.7	12.8
Q8	1.7	1.8	1.7	1.7
Q9	0.6	0.6	0.6	0.6
Q10	6.3	6.3	19.7	5.9
Q11	3.7	3.8	3.8	3.7
Q12	2.9	3.0	3.0	1.5
Q13	2.8	3.4	5.4	2.6
Q14	7.0	8.3	7.6	7.3
Q15	7.7	7.5	7.5	6.4
Q16	7.4	19.1	9.6	10.2
Q17	3.0	2.0	1.9	2.0
Q18	6.0	1.0	2.5	0.8
Q19	2.3	5.7	5.5	2.4
Q20	0.8	1.0	0.8	0.8

Table 1 Gain in using XML index for XMark queries (i.e. execution time using XML blob/execution time using XML index) for scale factor 0.5.

We compare the benefits of using the various XML indexes with the blob case. Table 1 shows the "gain" in using XML indexes as measured by the ratio of the execution times using XML blobs (i.e. without any XML indexes) and the execution times with different XML index configurations for scale factor 0.5. For example, the PROPERTY configuration creates the primary and PROPERTY XML indexes on each XML column since a secondary XML index is created on the Infoset table. These measurements are taken with no parallelism in query execution. Parallel plans make the gain higher in some cases. Owing to space limitations, we discuss the measurements for scale factor 30 briefly in Section 4.7.

Execution on XML blobs evaluates simple path expressions without predicates and produces an Infoset work table with rows for the qualifying nodes and their subtrees. The PATH_ID column is not present in this work table. Predicates are applied as a post-filter step. The rest of query execution proceeds as in the indexed case described in Section 3.

Looking at the gains in Table 1 — which gives the factor by which the choice of an XML index speeds up queries relative to the blob case — it is evident that XML indexes benefit the workload significantly. We consider a few of the queries below.

4.3 Primary XML Index

The performance gains are mainly related to parsing XML blob multiple times to evaluate the path expressions in the blob case. For primary XML index, not only is the parsing cost saved but also path expressions of the form "path=value" can be evaluated faster using the PATH_ID and VALUE columns. A case in point is Q4 (ordered access query), where the path expressions /site/open_auctions/open_auction/bidder/personref [@person="person18829"] and (/site/open_auctions/ open_auction/bidder/personref [@person = "person10487"] are evaluated using the primary XML index and yields nodes whose relative positions can be determined by comparing their ORDPATH labels.

Q6 (regular path expression query) performance is the same with and without XML indexes since the query counts the number of rows in the ITEMS table and no XML processing occurs.

One of the queries — Q9 (reference chasing query) — is slower than the execution on XML blob. It scans all rows of the primary XML index and evaluates two joins on values within XML instances. Owing to the larger size of the primary XML index compared to the XML blobs,

the index scan cost outweighs the cost of parsing and slows down the query. Query Q20 (aggregation query) has about the same performance as blobs.

4.4 PATH_VALUE Index

The PATH_VALUE index is very effective in speeding up some of the XMark queries, as shown in the PATH_VALUE column in Table 1.

Consider query Q1 (exact match query), which evaluates the two path expressions PE_1 = (/site/people/person/name/text())[1] and PE_2 = /site/people/person/@id[.= "person0"], as shown in the appendix. The path expression /site/people/person/@id is compiled into a PATH_ID value, and "person0" is the required VALUE, which is unique in the XML column in the PEOPLE table. The combination (PATH_ID, VALUE) yields a very selective seek into the PATH_VALUE index. The other path expression PE1 yields a PATH_ID value. Lookup of the PATH_VALUE index with only this value would cause a large number of rows in the index to be scanned. Instead, a primary XML index seek occurs with the ORDPATH of the "person" node (and the same ID value). Scanning down the primary XML index, the rest of the path expression is evaluated using the PATH_ID column. Evaluation of the query on the XML blob is much slower since PE2 is evaluated on all rows in the PEOPLE table. For the qualifying rows, the XML blob is parsed a second time to evaluate PE_1.

The performance gain with Q7 (regular path expression query) is large. The XML blob query has to scan all rows in four of the five tables and evaluate the three path expressions //description, //annotation and //email. On the other hand, these path expressions locate the "description", "annotation" and "email" node clusters within the PATH_VALUE index on each XML column, and eliminate duplicate ID values for each cluster. This yields very efficient evaluation of the query.

Other queries also benefit from the PATH_VALUE index to varying degrees, such as Q16, which evaluates long path expressions.

4.5 PROPERTY Index

Q2 (ordered access query) evaluates the path expression /site/open_auctions/open_auction/bidder[1]/increase/text() on all rows of the OPEN_AUCTIONS table. The primary key value ID is known from this table. Using ID and the PATH_ID value for the path /site/open_auctions/open_auction/bidder (ignoring the ordinal [1]), an index seek into the PROPERTY index finds the first bidder node within the XML instance. A back join with the primary XML index on the (ID, ORDPATH) value for the bidder node and a subtree scan for the remaining part of the path expression (increase/text()) yields the result. As a matter of fact, performing the tree scan on the primary XML index for a given ID value also performs quite well for the given data.

Q10 (construction of complex result query) finds persons with interest (the path expression PE is /site/people/person[profile/interest/@category]) and for each such person retrieves personal attributes. The primary key ID of the PEOPLE table and the compiled PATH_ID value is known. Consequently, PE can be evaluated very efficiently using an index seek on the PROPERTY index. For these persons (ID and ORDPATH values are known), various properties (e.g. gender and age) are retrieved efficiently from the PROPERTY index using ID and PATH_ID values for the different properties (identified by appropriate path expressions). The gain is pronounced compared to the other XML index types. An index seek into the PROPERTY index occurs for each property. In the other indexed cases, an index scan of the rows for each person occurs on the primary XML index to retrieve the properties.

4.6 VALUE Index

Q1 (exact match query) performs very well with the VALUE index. Two path expressions PE_1 = (/site/people/person/name/text())[1] and PE_2 = /site/people/person/@id[.= "person0"] occur in the query, as shown in the appendix. The value "person0" is unique in the XML column of the PEOPLE table, and the PATH_ID value is known at compilation time. Consequently, PE_2 is very selective on the VALUE index. Other queries benefit to different extents. Q9 does not use the VALUE index and uses the primary XML index.

4.7 Results for Scale Factor 30

The gains for scale factor 30 generally are more subdued than scale factor 0.5 since the processing becomes I/O bound. We present only a few of the measurements in Table 2 owing to space limitations.

QUERY	PRIMARY	PATH_ VALUE	PROPERTY	VALUE
Q1	2.8	595.3	5.2	602.2
Q5	1.2	1.1	0.8	1.1
Q15	1.8	18.3	6.2	5.9
Q16	1.4	48.2	4.5	5.0

Table 2 Gain in using XML index for XMark queries (i.e. execution time using XML blob/execution time using XML index) for scale factor 30.

Q1 performs extremely well with PATH_VALUE and VALUE indexes since the search predicate is highly selective. Bottom-up evaluation leads to improved gain in Q15 and Q16 as well using the PATH_VALUE index.

In the case of primary XML index, many more rows in the Infoset table are scanned for Q1 to evaluate the predicate, for which the gain is smaller than in the case of scale factor 0.5. Similar effects are seen in the other queries as well, such as Q5.

The PROPERTY index is a little slower in Q1 because a larger number of rows in the PEOPLE table are scanned to find their primary key values that are then used in PROPERTY index lookup.

5. Related Work

Several ideas have been proposed in the literature for decomposing XML data into a fixed database schema. Document order and structure is efficiently captured using a single ORDPATH in our approach as opposed to the EDGE table [6], Monet system [14], XRel [21], XParent [10] and accelerator table [8].

The EDGE table and XParent both use an Ordinal column to store the relative order of siblings in XML instances. They also store parent-child relationships, so that determining ancestor-descendant relationship and serializing XML require transitive closure computation. The XParent approach suggests materializing the ancestor-descendant relationship in an ANCESTOR table with a Level column that can be used for parent-child checks as well, but requires more space than ours.

In both EDGE table and XParent, insertion of subtrees requires incrementing the Ordinal value of the "following-siblings" [18]. The ANCESTOR table requires more maintenance. ORDPATH avoids such relabelling.

The Monet system partitions the XML data into a set of tables corresponding to the different paths. This distributes the children of a node into different tables, and determining the children of a node requires a number of joins. The Monet and XRel systems store the byte range of each XML subtree in the original XML. Serialization of XML is straightforward: the byte range is used to retrieve the corresponding XML fragments, and avoids scanning rows from the primary XML index in our approach. Document order is determined by comparing the starting byte of each node. Ancestor-descendant relationship requires checking for byte range inclusion, and a check for the minimal containing range is needed for parent-child relationship; for ORDPATH, both result in matching prefixes. The byte ranges of the "following" nodes [18] must be changed when a subtree is inserted or deleted, which is an expensive operation. ORDPATH is very flexible for subtree insertion and deletion.

The accelerator table labels XML nodes with their pre-order and post-order ranks in the XML tree, and is otherwise an edge table. Its properties are similar to the byte range approaches. For example, ancestor-descendant relationship requires checking for inclusion of pre- and post-order rank pairs, and subtree insertion updates the pre- and post-order ranks of a large number of nodes.

Path-value based queries require multiple joins to match the path in EDGE and accelerator tables. The Monet system looks up the value in the table corresponding to the path. For wildcard and //-axis queries, it potentially requires a large number of table look ups. The XRel and XParent schemes look up the data table using a mapped value for the path stored in a path directory. Property look ups have similar characteristics.

Value-based lookups benefit from a separate VALUE table in the EDGE table approach, which is similar in spirit to our VALUE index. The Monet system has to search a number of CDATA tables for imprecisely specified path. The specified value is used as a filter on the data table in XRel and XParent, and the accelerator table.

Our notion of secondary XML indexes can be applied to each of these approaches to speed up different query classes. On the other hand, we could introduce a path directory to save space in XML indexes, although it adds a JOIN in case of wildcard and //-axis queries.

6. Conclusions

This paper introduces techniques for indexing XML instances stored in a relational database in an undecomposed form. It introduces a B^+tree called primary XML index that encodes the Infoset items of XML nodes. We have avoided the approach of decomposition of XML instances based on their schema since our goal is uniform data representation and query processing with or without XML schemas. Secondary XML indexes improve the performance of common classes of queries: (a) PATH (or PATH_VALUE) index for path-based queries, (b) PROPERTY index for property bag scenarios (c) VALUE index for value-based queries, and (d) work break index for content indexing with structural information. Performance measurements using the XMark benchmark show that these indexing ideas are highly effective for a wide class of queries.

The above indexing ideas can be extended in several ways. Many applications know the expected query workload and will benefit by indexing only the paths occurring in the queries. An expression-based XML index is the solution. Navigational queries, such as opening a folder, go down a hierarchy one level at a time in breadth-first order. If this type of query is prevalent in a workload, it is beneficial to create an index for the parent-child relationship. ID/IDREF sets up linking within an XML instance which is different from document order. Primary XML index is not geared toward efficient traversal of IDREF links. Instead, an index can be created on the IDREF links for efficient traversal of IDREF links.

XML index maintenance can be performed by reconstructing the index rows corresponding to the modified XML instance. Alternatively, it can be done incrementally, and ORDPATH is especially suited to handle such changes. This is an interesting topic for future investigation, as also is an experimental comparison between our indexing scheme and the comparable ones.

Acknowledgment

The authors would like to thank their colleagues Adrian Baras, Denis Churin, Wei Yu, Sameer Verkhedkar, Goetz Graefe and Soner Terek for their invaluable discussions on indexing of XML data; José Blakeley, Goetz Graefe and the anonymous reviewers for their suggestions on improving the content and the presentation of the paper.

References

[1] R. Bayer and K. Unterauer. Prefix B-trees. ACM Transactions on Database Systems, 2(1):11 26, 1977.

[2] P. Bohannon, J. Freire, P. Roy, J. Simeon. From XML Schema to Relations: A Cost-Based Approach to XML Storage. ICDE 2002.

[3] Extensible Markup Language (XML) 1.0. http://www.w3.org/TR/REC-xml.

[4] J. Cowan, R. Tobin, eds. XML Information Set. http://www.w3.org/TR/2001/WD-xml-infoset-20010316.

[5] M. Fernandez, Y. Kadiyska, A. Morishima, D. Suciu, W-C Tan. SilkRoute : a framework for publishing relational data in XML. ACM TODS, vol. 27, no. 4, December, 2002.

[6] D. Florescu and D. Kossmann. Storing and Querying XML Data Using an RDBMS. IEEE Data Engineering Bulletin, 22(3):27-34, 1999.

[7] R. Goldman, J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. VLDB 1997.

[8] T. Grust. Accelerating XPath Location Steps. SIGMOD 2002.

[9] L. Guo, F. Shao, C. Botev, J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. SIGMOD 2003.

[10] H. Jiang, H. Lu, W. Wang, J. X., Yu. Path Materialization Revisited: An Efficient Storage Model for XML Data. 2nd Australian Institute of Computer Ethics Conference, 2000.

[11] Microsoft® SQL Server™. http://www.microsoft.com/sql.

[12] S. Pal, M. Fussell, I. Dolobowsky. XML support in Microsoft SQL Server 2005. http://msdn.microsoft.com/xml/default.aspx?pull=/library/en-us/dnsql90/html/sql2k5xml.asp.

[13] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller. ORDPATHs: Insert-Friendly XML Node Labels. SIGMOD 2004.

[14] A. Schmidt, M. Kersten, M. Windhouwer, F. Waas. Efficient Relational Storage and Retrieval of XML Documents. In Proc. of WebDB 2000, pp. 47-52.

[15] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, R. Busse. XMark: A Benchmark for XML Data Management. VLDB 2002.

[16] J. Shanmugasundaram, R. Krishnamurthy, I. Tatarinov. A General Technique for Querying XML Documents using a Relational Database System. SIGMOD 2001.

[17] A. Silberschatz, H. F. Korth, S. Sudarshan. Database System Concepts, 4th edition, McGraw-Hill, 2001.

[18] XML Path Language (XPath) 2.0. http://www.w3.org/TR/2003/WD-xpath20-20031112.

[19] XQuery 1.0: An XML Query Language. http://www.w3c.org/TR/xquery.

[20] I. Tatarinov, E. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita. Storing and Querying Ordered XML Using a Relational Database System. SIGMOD 2002.

[21] M. Yoshikawa and T. Amagasa. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. ACM Transactions on Internet Technology, vol. 1, August 2001, pp. 110-141.

APPENDIX — XMARK Benchmark

For completeness of the presentation, we present the XMARK queries adapted for our system. The data is contained in the following tables:

```
Create table PEOPLE (p_id  int IDENTITY PRIMARY
        KEY, p_xmlperson     xml)
Create table ITEMS (i_id  int IDENTITY PRIMARY
        KEY, i_xmlitem       xml)
Create table open_auctions(oa_id  int IDENTITY
        PRIMARY KEY, oa_xmlopen_auction      xml)
Create table closed_auctions(ca_id  int IDENTITY
        PRIMARY KEY, ca_xmlclosed_auction    xml)
Create table categories(c_id  int identity primary key,
        ct_xmlcategory  xml)
```

Some of the queries described in the XMark benchmark and discussed in this paper are presented below along with their implementation in our system.

Query Q1: Return the name of the person with ID 'person0'

```
select p_xmlperson.value('(/site/people/person/name
    /text())[1]', 'nvarchar(4000)')
from people
where p_xmlperson.exist('/site/people/person/@id[.=
"person0"]') =1
```

Query Q2: Return the initial increases of all open auctions.

```
select oa_xmlopen_auction.query('<increase> {
        /site/open_auctions/open_auction/bidder[1]
                /increase/text() } </increase>')
from open_auctions
```

Query Q4: List the reserves of those open auctions where a certain person issued a bid before another person

```
select oa_xmlopen_auction.query('<history>
        {/site/open_auctions/open_auction/reserve/text()}
    </history>')
from open_auctions
where oa_xmlopen_auction.exist('
        (/site/open_auctions/open_auction/bidder/
            personref[@person = "person18829"])[1] <<
        (/site/open_auctions/open_auction/bidder/
            personref[@person = "person10487"])[1]')=1
```

Query Q6: How many items are listed on all continents?

```
select count(i_id) from items
```

Query Q7: How many pieces of prose are in our database?

```
SELECT  SUM(c) as pieces_of_prose
FROM (SELECT COUNT(i_id) AS c FROM items
WHERE i_xmlitem.exist('//description') = 1UNION all
SELECT COUNT(oa_id) AS c FROM open_auctions
WHERE oa_xmlopen_auction.exist('//description') = 1
UNION all
SELECT COUNT(ca_id) AS c FROM closed_auctions
WHERE ca_xmlclosed_auction.exist('//description') = 1
UNION all
SELECT COUNT(ct_id) AS c FROM categories
WHERE ct_xmlcategory.exist('//description') = 1
UNION all
SELECT COUNT(i_id) AS c FROM items
WHERE i_xmlitem.exist('//annotation') = 1 UNION all
SELECT COUNT(oa_id) AS c FROM open_auctions
WHERE oa_xmlopen_auction.exist('//annotation') = 1
UNION all
SELECT COUNT(ca_id) AS c FROM closed_auctions
WHERE ca_xmlclosed_auction.exist('//annotation') = 1
UNION all
SELECT COUNT(ct_id) AS c FROM categories
WHERE ct_xmlcategory.exist('//annotation') = 1
UNION all
SELECT COUNT(i_id) AS c FROM items
WHERE i_xmlitem.exist('//email') = 1
UNION all
SELECT COUNT(oa_id) AS c FROM open_auctions
WHERE oa_xmlopen_auction.exist('//email') = 1
UNION all
SELECT COUNT(ca_id) AS c FROM closed_auctions
WHERE ca_xmlclosed_auction.exist('//email') = 1
UNION all
SELECT COUNT(ct_id) AS c FROM categories
WHERE ct_xmlcategory.exist('//email') = 1) as i
```

Query Q9: List the names of persons and the names of the items they bought in Europe.

```
SELECT t.p_xmlperson.query('
    <person name="{(/site/people/person/name)[1]}">
          {sql:column("i_name")}</person>')
FROM (
    SELECT p_xmlperson, p_id, i_id, i_xmlitem.value('
              (/site/regions/namerica/item/name)[1]',
              'nvarchar(400)') i_name
    FROM    people LEFT OUTER JOIN closed_auctions
              ON (p_xmlperson.value('
                    (/site/people/person/@id)[1]',
                    'nvarchar(400)') =
                    ca_xmlclosed_auction.value('
                    (/site/closed_auctions/closed_auction/
                    buyer/@person)[1]', 'nvarchar(400)')
              LEFT OUTER JOIN items
              ON (ca_xmlclosed_auction.value('
                    (/site/closed_auctions/closed_auction/
                    itemref/@item)[1]', 'nvarchar(400)') =
                    i_xmlitem.value('
                    (/site/regions/namerica/item/@id)[1]',
          'nvarchar(400)')) t ORDER BY p_id, i_id
```

Query Q10: List all persons according to their interest; use French markup in the result.

```
select  person.value('(.)[1]', 'varchar(50)') category,
          cp.p_xmlperson.query(
            '<personne><statistiques>
<sexe>{/site/people/person/gender/text()}</sexe>,
<age>{/site/people/person/age/text()}</age>,
 <education>{/site/people/person/education/text()}
</education>, <revenu>
{/site/people/person/income/text()}</revenu>
</statistiques><coordonnees>
<nom>{/site/people/person/name/text()}</nom>,
<rue>{/site/people/person/address/street/text()}</rue>,
<ville>{/site/people/person/address/city/text()}</ville>,
<pays>{/site/people/person/address/country/text()}
</pays>, <reseau>
<courrier>{/site/people/person/emailaddress/text()}
</courrier>
<pagePerso>{/site/people/person/homepage/text()}
</pagePerso></reseau></coordonnees>
<cartePaiement>{/site/people/person/creditcard/text()}
</cartePaiement></personne>')
from people cp cross apply
cp.p_xmlperson.nodes('/site/people/person/profile/interest
/@category') n(person) ORDER BY category
```

Query Q16: Return the IDs of those auctions that have one or more keywords in emphasis.

```
select ca_xmlclosed_auction.query('
        <person id ="{(/site/closed_auctions/
          closed_auction/seller/@person)[1]}"/> ')
from closed_auctions
where ca_xmlclosed_auction.exist('
    /site/closed_auctions/closed_auction/annotation/
    description/parlist/listitem/parlist/listitem/text/
    emph/keyword/text()') =1
```

Query Q20: Group customers by their income and output the cardinality of each group.

```
SELECT CAST( ('<result>' + '<preferred>' +
           cast(sum(case when income>=100000 then 1
           else 0 end) as nvarchar(10))+ '</preferred>' +
         '<standard>' +          cast(sum(case when
income<100000 and
           income>=30000 then 1 else 0 end) as
           nvarchar(10))+ '</standard>' +
        '<challenge>' +
           cast(sum(case when income<30000 then 1 else 0
           end) as nvarchar(10)) + '</challenge>' +
        '<na>' + cast(sum(case when income is null then 1
                 else 0 end) as nvarchar(10))+
        '</na>' + '</result>' ) AS XML)
FROM  (SELECT p_xmlperson.value('
          (/site/people/person/profile/@income)[1]','float')
          as income FROM people) i
```