XQuery Implementation in a Relational Database System

Shankar Pal, Istvan Cseri, Oliver Seeliger, Michael Rys, Gideon Schaller, Wei Yu, Dragan Tomic,

Adrian Baras, Brandon Berg, Denis Churin, Eugene Kogan

Microsoft Corporation
One Microsoft Way, Redmond, Washington, USA
{shankarp, istvanc, oliverse, mrys, gideons, weiyu, dragant, adrianb, branber, denistc, ekogan}@microsoft.com

Abstract

Many enterprise applications prefer to store XML data as a rich data type, i.e. a sequence of bytes, in a relational database system to avoid the complexity of decomposing the data into a large number of tables and the cost of reassembling the XML data. The upcoming release of Microsoft's SQL Server supports XQuery as the query language over such XML data using its relational infrastructure.

XQuery is an emerging W3C recommendation for querying XML data. It provides a set of language constructs (FLWOR), the ability to dynamically shape the query result, and a large set of functions and operators. It includes the emerging W3C recommendation XPath 2.0 for path-based navigational access. XQuery's type system is compatible with that of XML Schema and allows static type checking.

This paper describes the experiences and the challenges in implementing XQuery in Microsoft's SQL Server 2005. XQuery language constructs are compiled into an enhanced set of relational operators while preserving the semantics of XQuery. The query tree is optimized using relational optimization techniques, such as cost-based decisions, and rewrite rules based on XML schemas. Novel techniques are used for efficiently managing document order and XML hierarchy.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

1. Introduction

Enterprise applications use XML [3] for modelling semi-structured and markup data in scenarios such as document management and object property management [13]. Powerful applications can be developed to retrieve documents based on document content, to query for partial contents such as sections whose title contains the word "background", to aggregate fragments from different documents, and to find all the phone numbers of a person.

Storing XML data as a sequence of bytes representing a rich data type has several advantages. XML schemas for real-life applications are complex so that decomposing XML data conforming to those schemas into the relational data model results in a large number of tables. This makes the decomposition logic complex, the re-assembly cost high, and the queries very complicated. Furthermore, changes to the XML schema require a significant amount of maintenance of the database schema and the application. XML as a rich data type also permits structural characteristics of the XML data, such as document order and recursive structures, to be preserved more faithfully.

The upcoming release of Microsoft's SQL Server 2005 [10] allows storage of XML data in a new, rich data type called XML [1][8][13]. This data type stores both rooted XML trees and XML fragments in a binary representation ("binary XML"). The query language on XML data type is a subset of XQuery [15][16][22], an emerging W3C recommendation (currently in Last Call) that includes the navigational language XPath 2.0 [20]. It is supported using the relational query processing framework with some enhancements. SQL Server 2005 also supports a data modification language on XML data type for incremental updates, which is not discussed further in this paper [1][13].

This paper discusses the XQuery processing architecture in SQL Server 2005 and how XQuery expressions are compiled into query trees containing relational operators and a small number of new operators

introduced for the purpose of XQuery processing. An XQuery expression is parsed and compiled into an internal structure called the *XML algebra tree* on which rule-based optimizations are applied. This is followed by a transformation of the XML algebra tree into the relational operator tree. This paper describes some of the interesting aspects of the implementation instead of being a comprehensive manual on the subject.

XML as a richly structured data type introduces new challenges for query processing, data modification and indexing. Query processing must retain document order, perform structural navigation, provide sequence operation, and support dynamically constructed XML nodes. These requirements are not supported by a relational query processor and appropriate extensions to it are necessary.

At runtime, the XML data ("XML blob") must be available in a parsed state (the so-called XQuery Data Model [23]) to evaluate an XQuery expression. The data may be parsed multiple times to evaluate several XQuery expressions on the same data, or to evaluate complex XQuery expressions using a streaming parser, such as the XmlReader in the .NET framework [9], to avoid the overhead of keeping the data in memory (e.g. DOM). Runtime parsing is costly and often fails to meet the performance requirements of enterprise applications. For better query performance, SQL Server 2005 provides a mechanism for indexing the XML data [12] based on its Data Model content [2]. An XML index retains structural fidelity of the data, such as document order and hierarchical relationships among the XML nodes, and speeds up different classes of queries on the XML data.

XQuery compilation produces a query tree that uses relational operators, such as SELECT and JOIN, on the primary XML index [12], if one exists, on an XML column. For non-indexed XML columns, the query plan contains operators to parse each XML blob, locate nodes matching simple path expressions, and generate rows resembling XML index entries that represent the subtrees rooted at those nodes. From this point onward, the processing for both the XML indexed and the XML blob cases is largely the same – multiple rowsets are manipulated using relational operators to yield the query result. The queries that return XML results aggregate the rows representing the resulting XML sequence into the binary XML form as the final processing step.

The XQuery compiler performs static type inference by annotating operator-nodes in the query tree with type information. Type incompatibility between the inferred type and the expected type raises static errors. This fits well with the static type guarantees in the SQL language and the relational query processor's ability to optimize query plans using statically known constraints. As a result, many runtime checks are avoided.

The compiled query plan is optimized using well-known relational optimization techniques such as costing functions and histograms of data distributions. Query compilation produces a single query plan for both relational and XML data accesses, and the overall query tree is optimized as a whole. SQL Server 2005 also introduces optimizations for document order (by eliminating sort operations on ordered sets) and document hierarchy, and query tree rewrites using XML schema information.

Relational query optimization, however, impacts XQuery semantics and introduces new challenges. The query optimizer shuffles operators around in the query tree to produce a faster execution plan, which may evaluate different parts of the query plan in any order considered to be correct from the relational viewpoint. Consequently, path expression based navigational accesses are not guaranteed to be executed top-down and may be evaluated bottom-up. This may yield dynamic errors, such as type cast errors, when none would occur with top-down evaluation. For this reason, SQL Server 2005 currently converts dynamic errors to empty sequences. In most contexts this yields correct results, but not always (e.g., in the presence of negation).

A significant number of XQuery functions and operators are supported in the system. Wherever possible, these functions and operators are compiled into the analogous SQL functions and operators for efficient execution. In all other cases, additional code in the server executes the XQuery function or operator while preserving XQuery semantics.

The rest of the paper is organized as follows. Section 2 provides background material on the native XML support in SQL Server 2005. Section 3 introduces the query processing architecture and provides an overview of the XML algebra operators used in the server. Section 4 discusses the transformation of XML algebra trees for XPath and XQuery expressions into relational operator trees. Section 5 deals with the type inference mechanism employed by the XQuery compiler and Section 6 discusses optimizations on the query trees yielding the execution plan for the queries. Related work is discussed in Section 7 while concluding remarks appear in Section 8.

2. XML Support in SQL Server 2005

This section provides a look into some of the XML features of SQL Server 2005 necessary for the discussions in this paper. Detailed information and be found in the product's documentation [10] as well as MSDN whitepapers [13][14].

2.1 XML Data Type

Microsoft's SQL Server 2005 [10] introduces native storage for XML data as a new, rich data type called XML. A table may contain one or more columns of type XML wherein both rooted XML trees and XML fragments can be stored. Variable and parameters of type XML are also allowed. XML parsing occurs either

implicitly or explicitly during assignments of either string or binary SQL values to XML columns, variables and parameters.

XML values are stored in an internal format as large binary objects ("XML blob") in order to support the XML data model characteristics more faithfully such as document order and recursive structures.

The following statement creates a table DOCS with an integer, primary key column PK and an XML column XDOC:

```
CREATE TABLE DOCS (
    PK INT PRIMARY KEY, XDOC XML)
```

2.2 XML Schema Support

SQL Server 2005 provides *XML schema collections* as a mechanism for managing W3C XML schema documents [21] as metadata. XML data type can be associated with an XML schema collection to have XML schema constraints enforced on XML instances. Such XML data types are called "typed XML". Non-XML schema bound XML data type is referred to as "untyped XML".

Both typed and untyped XML are supported within a single framework, the XML data model is preserved, and query processing enforces XQuery semantics. The underlying relational infrastructure is used extensively for this purpose.

2.3 Querying XML Data

XML instances can be retrieved using the SQL SELECT statement. Four built-in methods on the XML data type, namely *query*(), *value*(), *exist*() and *nodes*(), are available for fine-grained querying. A fifth built-in method *modify*() allows fine-grained modification of XML instances but is not discussed further in this paper.

The query methods on XML data type accept the XQuery language [15][16][22], which is an emerging W3C recommendation (currently in Last Call), and includes the navigational language XPath 2.0 [20]. Together with a large set of functions, XQuery provides rich support for manipulating XML data. The supported features of the XQuery language are shown below:

- XQuery clauses "for", "where", "return" and "order by".
- XPath axes child, descendant, parent, attribute, self and descendant-or-self.
- Functions – numeric, string, Boolean, nodes, context, sequences, aggregate, constructor, data accessor, and SQL Server extension functions to access SQL variable and column data within XQuery.
- Numeric operators (+, -, *, div, mod).
- Value comparison operators (eq, ne, lt, gt, le, ge).
- General comparison operators (=, !=, <, >, <=, >=).

The following is an example of a query in which section titles are retrieved from books and wrapped in new <topic> elements:

```
SELECT PK, XDOC.query('
    for $s in /BOOK/SECTION
    return <topic>
             {data($s/TITLE)}
           </topic>')
FROM DOCS
```

The query execution is tuple-oriented – the SELECT list is evaluated on each row of the DOCS table, the query() method is processed on the XDOC column in each row, and the result is a two-column rowset where the column types are integer (for PK) and untyped XML (for the XML result). The query methods are evaluated on single XML instances, so that XQuery evaluation over multiple XML documents is currently not supported by the syntax but is allowed by the architecture. Scalar value-based joins over XML instances are possible.

2.4 Indexing XML Data

Query execution processes each XML instance at runtime; this becomes expensive whenever the XML blob is large in size, the query is evaluated on a large number of rows in a table, or a single SQL query executes multiple XQuery expressions requiring the XML blob to be parsed multiple times. Consequently, a mechanism for indexing XML columns is supported in SQL Server 2005 to speed up queries.

A *primary XML index* [12] on an XML column creates a B^+tree index on the data model content of the XML nodes, and adds a column Path_ID for the reversed, encoded path from each XML node to the root of the XML tree.

The structural properties of the XML instance, such as relative order of nodes and document hierarchy, are captured in the OrdPath column for each node [11]. The primary XML index is clustered on the OrdPath value of each XML instance in the XML column. The other noteworthy columns are the name, type and the value of a node.

XML indexes provide efficient evaluation of queries on XML data, and reassembly of the XML result from the B^+tree. These use the relational infrastructure while preserving document order and document structure. OrdPath encodes the parent-child relationship of XML nodes by extending the parent's OrdPath with a labelling component for the child. This allows efficient determination of parent-child and ancestor-descendant relationships. Furthermore, the subtree of any XML node *N* can be retrieved from the primary XML index using a

range scan over the OrdPath values of *N* and the *descendant limit* of *N*. The latter value can be determined from *N*'s OrdPath alone, which makes OrdPath a very simple yet efficient node labelling scheme.

Secondary XML indexes can be created on an XML column to speed up different classes of commonly occurring queries: PATH index for path-based queries, PROPERTY index for property bag scenarios, and VALUE index for value-based queries are currently provided.

Statistics are created on the key columns of the primary and secondary XML indexes. These are used for cost-based selection of the secondary XML indexes. Choice of the primary XML index is currently a static decision.

The next section describes the architecture for query processing on XML data.

3. XML Query Processing Architecture

As outlined in the previous section, the XML data is persisted in the relational store to leverage the existing relational infrastructure. An XQuery expression is compiled into a query tree that can be optimized and executed by the relational query processor. The hierarchical nature of the XML data is modelled as parent-child relationship using the OrdPath node labelling scheme [11] instead of developing a new, hierarchical store. Query processing for ordered, hierarchical data model requires more work than for the flat relational model. For this reason, the set of relational operators is extended with additional operators for XML processing. This enhancement yields "relational+" operators.

XQuery compilation is performed in multiple stages, starting with the parsing of XQuery expressions and resulting in the generation of the query plan containing the enhanced set of relational operators. The overall architecture is shown in Figure 1. The main steps consist of an *XQuery Compiler*, which includes XQuery parsing, and an *XML Operator Mapper*.

The XML algebra tree is an intermediate representation on which rule-based (as opposed to cost-based) optimizations are applied. One such optimization is path collapsing described in Section 6. Rewrites using XML schema information are also applied to the XML algebra tree. The output of the XQuery Compiler step is an XML algebra tree that is highly optimized for XML processing.

Using the appropriate XML and relational type information, the XML Operator Mapper converts the XML operators in the XML algebra tree into a relational operator tree that includes the enhanced set of relational operators. This mapping is discussed in more details in Section 4.

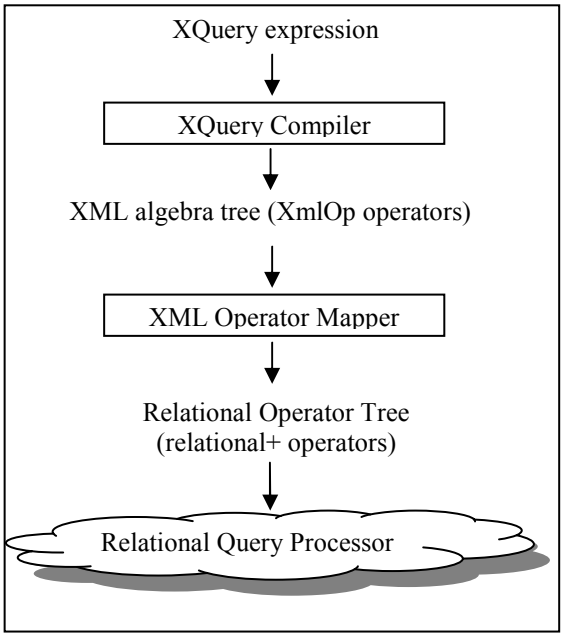

Figure 1. Architecture for XQuery compilation.

XML Operator Mapper recursively traverses the XML algebra tree. For each XML operator in the XML algebra tree, a relational operator sub-tree is generated, which includes enhanced relational operators. The relational operator sub-trees are then inserted into the overall relational operator tree for the XQuery expression.

The mapping of each XML operator to a relational operator subtree depends upon the existence of a primary XML index on the XML column being queried. If it exists, then the query plan is generated to access columns in the primary XML index. If it does not exist, then the query plan is produced to evaluate path expressions without branching on the XML blob and to generate a set of rows representing the subtree of the matching nodes in document order. These rows contain most of the columns of the primary XML index except notably the primary key columns from the base table (used in back join from the primary XML index to the base table) and the Path_ID column that contains the reversed, encoded path from an XML node to the root of the XML tree.

The rest of the query plan is the same if the primary key and Path_ID columns are not needed. Otherwise, it continues to differ.

The relational operator tree for the XQuery expression is grafted into the main query tree for the whole SQL query. Thus, a single query tree is produced, and the query optimizer can optimize the full query plan containing both relational and XML accesses. This also supports interoperability between relational and XML data at the server, making way for richer application development.

The next subsection describes some of the XML operators used in the XML algebra tree.

3.1 XML Operators

The XQuery Compiler parses an XQuery expression and produces an XML algebra tree that includes XML operators. This section describes a handful of the XML operators introduced in SQL Server 2005, some of which are used further in this paper. This list is representative but not exhaustive; detailed descriptions are beyond the scope of this paper.

Each XML operator may accept input such as an ordered XML node list, an unordered XML node set, a Boolean condition, an ordinal condition, a node list condition, and other scalar input.

3.1.1 XmlOp_Select

The XmlOp_Select operator takes a list of items, including ordered XML nodes, as a left child and a condition as right child. It returns the input items in their input order which satisfy the given condition.

3.1.2 XmlOp_Path

The XmlOp_Path operator is used for simple paths without predicates and produces the eligible XML nodes. This operator also uses a path context to collapse paths (see Section 6 for more information).

3.1.3 XmlOp_Apply

The XmlOp_Apply operator takes two item lists as input, and returns one item list. It has an "apply name" property whose value is the variable name bound by the corresponding "for" clause in XQuery. The variable is bound to each of the items in a first item list. The second item list typically contains references to this variable, and is evaluated using the variable binding with the items in the first list.

The XmlOp_Apply operator also takes a "where" and an "order-by" child. It is a complex operator that the XML Operator Mapper translates to a relational operator tree for evaluating the "for", "where" and "order-by" clauses with the appropriate XQuery semantics.

3.1.4 XmlOp_Compare

This is a comparison operator with a field indicating the type of the comparison.

3.1.5 XmlOp_Constant

This operator represents a constant, which can be a literal or the result of constant folding. *Constant folding* is the static optimization that evaluates constant expressions during query compilation to avoid runtime execution costs and to allow more query optimizations.

3.1.6 XmlOp_Construct

The XmlOp_Construct operator creates all the XML node types: elements, attributes, processing instructions, comments, and text nodes. For element construction, the operator takes as input the sub-nodes (attributes and/or children), otherwise the value of the constructed node.

3.1.7 Scalar Operators

The XmlOp_Function operator represents a built in function that returns a scalar or XML nodes. The inputs are the parameters of the function and the output is the result of the function.

The next section describes the mapping of the XML operators for XPath and XQuery expressions to relational operators.

4. XML Operator Mapping

The XML Operator Mapper transforms an XML algebra tree into a relational operator tree. Conventional relational algebraic operators are inadequate to process the hierarchical XML data model in an efficient way. Consequently, the set of relational operators is enhanced with new operators for the purpose of XQuery processing, yielding the relational+ algebra. The relational operator tree is submitted to the query processor for optimization and execution.

We describe the mapping of the XML algebra tree to the relational operator tree in the following subsections. For convenience, we subdivide the discussion into the following categories:

- Mapping of XPath expressions
- Mapping of XQuery expressions
- Mapping of XQuery built-in functions

4.1 XPath Expressions

The XmlOp_Path operator representing a path is mapped to a relational operator in a different way for XML blob than for a primary XML index on an XML column. Each of these scenarios is further subdivided into two cases –

- Simple path expressions without branching in which the full paths from the root of the XML trees are known after path collapsing ("exact paths")
- Paths expressions without branching in which the full paths are not known ("inexact paths").

As described later in Section 6, segments of simple paths may be concatenated together to produce a longer simple path using the path collapsing technique.

Inexact paths occur in the XML algebra tree when segments of the path cannot be collapsed or a path is split into multiple segments. It occurs most commonly for paths containing wildcard steps, the //-operator, self and parent axes.

The resulting four mappings are discussed below using the path expression /BOOK/SECTION as example. Predicate and ordinal evaluations are discussed later in this section.

4.1.1 Non-indexed XML, Exact Path

The XmlOp_Path operator is mapped to an XML_Reader operator for parsing the XML blob. XML_Reader is a streaming, pull-model XML parser, similar to the XmlReader in the .NET framework [9]. It is chosen for its efficiency in parsing XML data and its relatively low memory requirements, compared to a non-streaming XML parser such as for DOM, for handling large XML instances.

The path /BOOK/SECTION is an argument to the XML_Reader operator and is applied during runtime parsing of the XML blob. The result is a set of rows representing the subtrees of the qualifying <SECTION> nodes and retaining the structural properties of those subtrees using their OrdPath values.

The XmlOp_Path operator can occur at the top-level of the XML algebra tree when the path expression occurs within the query() method, i.e. XDOC.query ('/BOOK/SECTION'). In this case, rows representing the subtree of each <SECTION> node are reassembled into an XML data type result using an XML_Serialize operator. This step is referred to as *XML Serialization* in the rest of the paper. The overall mapping is shown in Figure 2.

4.1.2 Non-indexed XML, Inexact Path

The path, such as /BOOK/SECTION//TITLE, is used by XML_Reader during XML blob parsing to filter the eligible nodes. Thus, the relational operator tree is similar to the one in Figure 2 with the appropriate path as input to the XML_Reader operator.

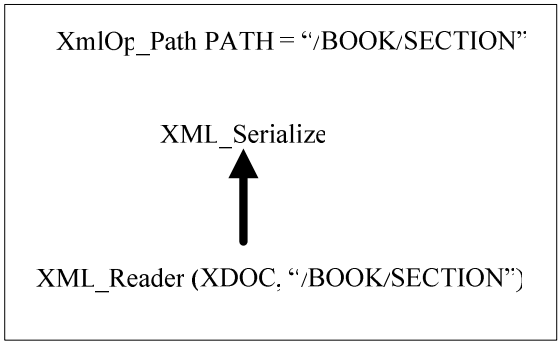

Figure 2. Relational operator tree for the exact path query XDOC.query ('/BOOK/SECTION') in the non-indexed case.

4.1.3 Indexed XML, Exact Path

The XmlOp_Path operator with the exact path is mapped to a relational SELECT operator that filters primary XML index rows (GET(PXI)) by matching the supplied path /BOOK/SECTION with the value in the Path_ID column. The Path_ID column stores the reversed path in an encoded form. The XML Operator Mapper uses a function PATH_ID (/BOOK/SECTION) over the path to generate the search value for the Path_ID column. For simplicity, the result of this function is depicted as #SECTION#BOOK. The resulting relational operator tree is shown in Figure 3.

Top-level XmlOp_Path requires the XML_Serialize operator that receives rows corresponding to the subtree of each <SECTION> node and produces the XML result. The APPLY operator [6] in the relational operator tree is a correlated join between the <SECTION> rows and the right child of the APPLY operator.

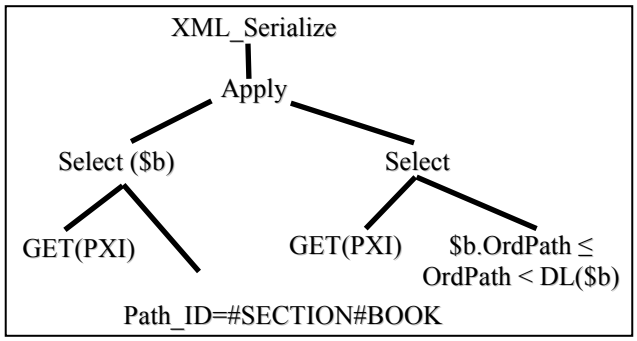

Figure 3. Relational operator tree for the exact path query XDOC.query ('/BOOK/SECTION') for the indexed case

Retrieval of each <SECTION> node's subtree utilizes the OrdPath property for generating the subtree – nodes belonging to the subtree are chosen (SELECT operator in the right child of APPLY) with the OrdPath value in between those of the <SECTION> node and its descendant limit (DL). This is executed efficiently using a range scan over the primary XML index.

4.1.4 Indexed XML, Inexact Paths

Inexact paths are matched on the Path_ID value using the LIKE operator. For example, rows for the <TITLE> nodes in the path expression /BOOK/SECTION//TITLE are found from the primary XML index using the predicate Path_ID LIKE #TITLE%#SECTION#BOOK. Finally, the subtree under each <TITLE> node is serialized in the result. The relational operator tree for this example is shown in Figure 4.

The path expression /BOOK/SECTION/../@id containing the parent axis is split into the evaluation of the paths /BOOK/SECTION and //@id, and then ensuring that the parent of the <SECTION> element is the same as that of the @id attribute. The inexact path //@id is evaluated using the expression Path_ID LIKE #@id%, and benefits from the use of XML indexes. This technique can be used for any path for which the tail end of the path is known.

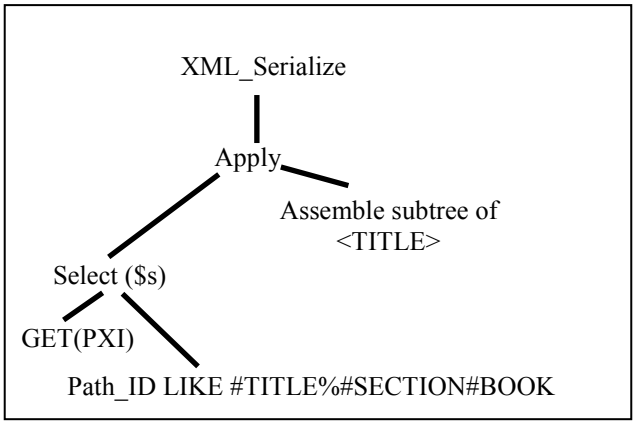

Figure 4. Relational operator tree for the inexact path query XDOC.query ('/BOOK/SECTION//TITLE') for the indexed case.

As should be apparent from the discussions above, the indexed and the non-indexed cases differ mainly in the way paths are evaluated on XML blobs or the column Path_ID in the primary XML index. The rest of the processing is done much the same way on columns common to both the primary XML index and the output rows of the XML_Reader. For this reason, in the remainder of this paper, only the indexed case is illustrated for brevity.

4.1.5 Predicate Evaluation

Predicate evaluation is performed by comparing the search value with that in the value column of the primary XML index. The relational operator tree for the path expression /BOOK[@id = "123"] is shown in Figure 5.

The evaluation of the simple paths /BOOK and /BOOK/@id proceed as described above using the Path_ID column of the primary XML index. The specified value "123" is compared with the VALUE column in the same row of the primary XML index as the @id attribute. Since the two paths are evaluated separately, a check for the parent-child relationship is also needed. This is depicted in Figure 5 as the Parent_Check() function. The check uses the OrdPath property that the parent's OrdPath is a prefix of the child's OrdPath except for the rightmost component.

The value of a simple-valued, typed element is stored in the same row as the element, so that predicates on the element are evaluated in the same way as an attribute. Predicates on untyped XML are more complicated to evaluate since values may need to be aggregated from multiple rows, which makes the relational operator tree more complex.

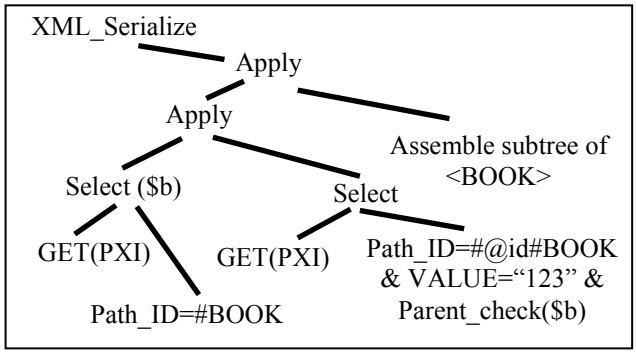

Figure 5. Relational operator tree for the query XDOC.query ('/BOOK[@id="123"]').

The relational operator tree may also contain CONVERT operators if the operands need to be converted to the appropriate types to perform an operation.

4.1.6 Ordinal Predicate

Ordinal predicate evaluation such as /BOOK[*n*] adds a ranking column to the rows for <BOOK> elements and then retrieves the *n*th <BOOK> node. A special optimization exists for the cases *n* = 1 and *n* = last(). The ordinal predicate is mapped to TOP 1 ascending and TOP 1 descending, respectively. TOP *n* is a relational operator that chooses the topmost *n* values from a rowset. When the input set is sorted, such as the rows in the primary XML index, this rewrite avoids ranking all the nodes before the ordinal predicate is evaluated.

4.2 XQuery Expressions

SQL Server 2005 supports the FLWOR clauses "for", "where", "order-by" and "return". XML operator mapping is described in some detail below for these constructs. A formal algorithm for the mapping is not presented in this paper for lack of space. However, fragments of the algorithm are illustrated below using examples.

The XQuery processing framework described in this paper is powerful enough to support "let" but this is not discussed further in the paper.

4.2.1 "for" Iterator

The XML algebra operator for the "for" iterator in XQuery is XmlOp_Apply. It maps to the relational APPLY operator, as shown in the example in Figure 6 for the query

```
for $s in /BOOK//SECTION
where $s/@num >= 3
return $s/TITLE
```

In the example, the Path_ID column is used to match the path /BOOK//SECTION using the LIKE operator. The APPLY operator with the $s binding iterates over the <SECTION> nodes and determines its <TITLE> children. The operators for the "where" condition are discussed later.

Nested "for" expressions and "for" with multiple bindings (e.g. for $i in /Customer, $j in /Order …) turn into nested APPLY operators, where each APPLY binds to a different variable.

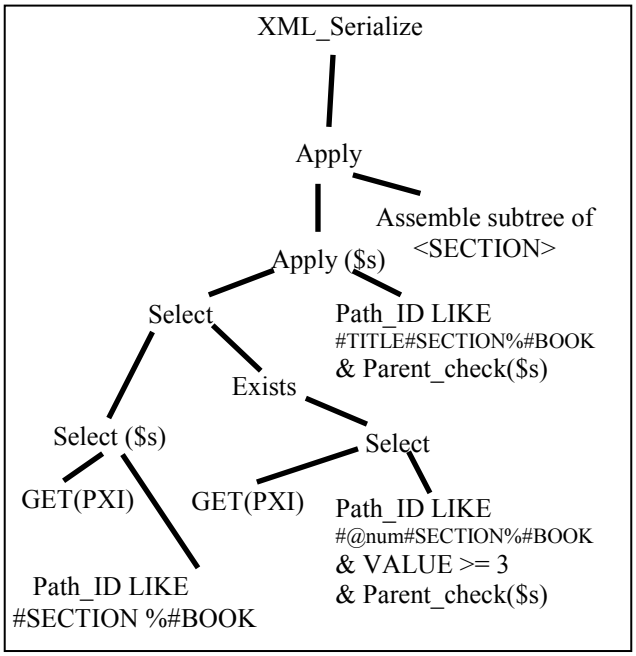

Figure 6. Relational operator tree for the XQuery expression 'for $s in /BOOK//SECTION where $s/@num >= 3 return $s/TITLE' for indexed XML.

4.2.2 "where"

The "where" clause in XQuery is represented by the "where" child of the XML algebra operator XmlOp_Apply. It maps to a relational SELECT operator on the input sequence that filters the rows matching the specified condition. An example is shown in Figure 6. In the example, <SECTION> elements matching the path /BOOK//SECTION are bound to the variable $s. Another path matching using the LIKE operator occurs for the path /BOOK//SECTION/@num obtained by path collapsing on the path $s/@num inside the "where" condition. At the same time, the VALUE comparison is performed using the specified value 3, and to check parent-child relationship between <SECTION> nodes and the @num attributes. The OrdPath values of the the <SECTION> element and its @num attribute satisfies the conditions of the parent-child relationship.

The EXISTS operator further on is introduced to filter the <SECTION> rows because of the existential semantics of the >= operator.

4.2.3 "order by"

"Order-by" sorts rows based on the order-by expression and adds a ranking column to these rows. The ranking column is then converted into OrdPath values that yield the new order of the rows to fit the rest of the query processing framework.

4.2.4 "return"

XQuery expressions in the "return" clause are evaluated based on the foregoing principles. If the return sequence of nodes is in document order, then the stored OrdPath values suffice in capturing the structure of the returned result. Thus, the XML_Serialize operator at the end can generate the final, XML result.

New element and sequence constructions are considered in the next subsection. The top-level XML_Serialize converts the constructed rows, based on the new structural relationships, into the final, XML result.

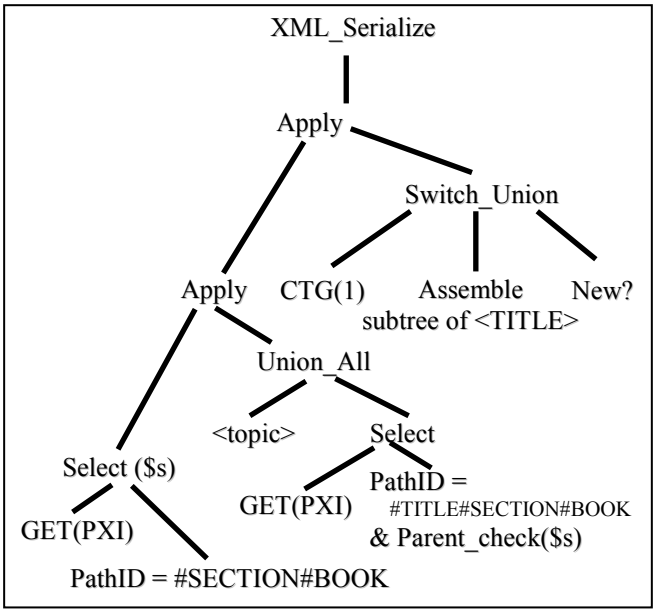

Figure 7. Relational operator tree illustrating element construction.

4.2.5 Construction

New element construction is done by generating new rows for the constructed element with an appropriate OrdPath value. The new element's content requires XQuery evaluation as described in this section, and may require JOIN with the primary XML index to retrieve nodes. Multiple constructed sibling nodes are put together using UNION_ALL operator. For the query

```
for $s in /BOOK/SECTION
return <topic>{$s/TITLE}</topic>
```

the relational operator tree is shown in Figure 7. The <TITLE> nodes are found from the primary XML index as described earlier in this section. An UNION_ALL operator over these rows and the rows generated for the new <topic> elements produces a single set of rows; the hierarchical relationship between the two rowsets for <topic> and <TITLE> is maintained using compile time OrdPath values. Each row also contains a flag to indicate whether the row represents a newly constructed node or an existing node (the "New" flag in Figure 7).

The SWITCH_UNION operator checks the "New" flag on each row. For a newly constructed <topic> row, it outputs a constant row (CTG(1)) with an appropriate OrdPath value. For an existing <TITLE> row, it assembles the subtree of the <TITLE> element from the primary XML index and modifies the OrdPath values in the subtree to maintain the hierarchical relationship with the corresponding <topic> element. Finally, the XML data type result is produced using the XML_Serialize operator.

Sequence construction is handled in a similar way using UNION_ALL, and if needed, a SWITCH_UNION.

4.2.6 Other XQuery Constructs

Other XQuery constructs, such as if/then/else, are expressed in terms of the appropriate relational operators. A discussion of these is too detailed and beyond the scope of this paper.

4.3 XQuery Functions & Operators

Several of the XQuery functions and operators are available in SQL Server 2005. These are selected based on customer requirements rather than completeness.

The XQuery built-in functions and operators are mapped to the underlying relational functions and operators wherever possible. An example is the fn:count() function, which is evaluated using the count() function in SQL. For XQuery types, functions and operators that cannot be mapped directly, additional support has been added to the query processor.

A couple of the aggregate functions, namely, fn:data() and fn:string() are specially optimized since they are frequently used and expensive. Each of these functions aggregates the values from multiple rows. This would normally result in JOIN operations over those rows. Special operators have been introduced to perform these aggregations directly for runtime efficiency.

5. XQuery Type System

The XQuery 1.0 and XPath 2.0 type system is based on W3C XML Schema [21]. The type system is used for type inferences and for generating static type error during query compilation. This fits well with the static type system of the relational data model. In fact, XPath 1.0 [19] has a dynamic type system which prevents error detection during query compilation, and does not fit as well into a relational database system.

Most of the SQL types are compatible with the XQuery type system (e.g. decimal). A handful of types (e.g. xs:duration) are stored in an internal format and suitably interpreted for compatibility with the XQuery type system.

Compilation of XQuery expressions requires annotation of the XML algebra tree with type information, which can be obtained from type definitions or by using type derivation. The type information must be supplied using XML Schema. For example, a node whose value must be of type xs:integer is prevented at compilation time from being supplied with a value of an incompatible type such as xs:string. When such schema information is unavailable, e.g. for "untyped" XML, a limited set of type inferences using the XQuery type system is still possible. Static type checking of XQuery expressions can raise static errors during query compilation and report errors without executing the expressions. This improves the responsiveness of the database system in a big way. Dynamic errors are still possible during query execution time, such as failure of type casts, and return in empty XML or a NULL result instead of an error.

5.1 Type Inference Mechanism

The XQuery Compiler loads type information from XML schemas in the XML schema collection associated with an XML instance into a *symbol table*. It annotates the nodes in the XML algebra tree with type information retrieved from the symbol table. The type of a node is also based on the inferred types of the earlier processed nodes. The resulting operator tree is referred to as the "annotated XML algebra tree" (AXAT).

For example, when adding two inputs annotated with the type xs:decimal, the node for the addition is annotated with the type xs:decimal.

The symbol table contains a normalized form of the type information that makes retrieval of type information from it very efficient.

5.2 Static Type Checking

Compilation errors are returned from syntactically incorrect XQuery expressions. The compilation phase checks static type correctness of XQuery expressions, and raises static type errors if an expression could fail at runtime due to type safety violation. Examples of static errors are addition of a string to an integer, and detecting potential mistakes such as querying for a non-existent node name in typed XML data.

Explicit casting to the proper type allows users to work around static errors, although runtime cast errors are

transformed to empty sequences as a deviation from the W3C recommendation; an option to return dynamic errors could have been provided. The empty sequence may propagate as empty XML or NULL in the query result depending upon the invocation context. SQL Server 2005 requires "cast as" with "?" (0 or 1 occurrence) since any cast can result in the empty sequence due to runtime errors.

Location steps, function parameters, and operators (e.g. eq) requiring singletons return an error if the compiler cannot determine whether a singleton is guaranteed at runtime. The problem arises often with untyped data and path expressions that may yield multiple nodes, and an explicit iteration or an ordinal predicate (e.g. (/book)[1]) selecting a single node needs to be used. If a node is specified as a singleton in an XML schema, the XQuery Compiler uses that information and no error occurs. However, the use of descendant-or-self axis, as in /BOOK//TITLE, loses singleton cardinality inference for <TITLE> element even if the XML schema specifies it to be so. An ordinal predicate is required in this case. In some contexts, the ordinal specification can be removed, even for untyped XML [14].

If the type of a node cannot be determined, such as in an xs:any section with processContents = "skip", it becomes xs:anyType, which is not implicitly cast to any other type. An element may be defined as xs:anyType in an XML schema and navigation using parent axis (e.g. XDOC.query('/book/@genre/../price')) also results in xs:anyType for the parent node type. In both cases, the loss of more precise type information often leads to static type errors, and requires explicit cast of atomic values to their specific types.

6. Optimizations on Query Tree

The relational query optimizer treats the query plan with relational semantics. This is appropriate for many cases, such as for evaluating relational accesses before XML data accesses or vice versa, and results in an optimal execution.

This section describes some of the optimizations supported for XQuery processing.

6.1 Exploiting Ordered Sets

The XML Operator Mapper produces query plans for XQuery processing in structure preserving ways. Order preservation among the nodes of an XML tree is achieved using OrdPath as a ranking column – it not only supports relative order but also encodes hierarchical relationship – so that it can serve as a node identifier column during the intermediate processing steps. The query optimizer honors the directives given by the relational operator tree in its optimization decisions.

Rows representing the subtree of an element, either retrieved from the primary XML index or generated by the XML_Reader, are in ascending order of OrdPath, i.e. the rows are in depth-first order of the nodes in the subtree. This information is made available to further relational operators in the relational operator tree to eliminate sort operations. Serialization of an XML subtree using the XML_Serialize operator serves as an example. Similarly, the fn:data() and fn:string() aggregators avoid sorting on OrdPath when the input is already in document order (e.g. retrieved from the primary XML index).

6.2 XML Index and XML Blob Accesses

XML indexes are used as available. The choice of the primary XML index is made statically by the XML Operator Mapper, which work well in most cases [12]. The choice of secondary XML indexes is cost-based and is made by the relational query optimizer.

For indexed XML data, the XML Operator Mapper produces a query plan that uses lookups on the Path_ID column of the primary XML index for path evaluation. The lookup may be an exact match for a fully specified path such as /BOOK/SECTION. For a path expression containing //, such as //SECTION/TITLE, the LIKE operator is used to match the prefix of the Path_ID column with the ending portion of the specified path. This works since the Path_ID column stores reversed paths.

For non-indexed XML data, XML index rows are generated in document order at runtime using the XML_Reader operator, which is a streaming table-valued function (TVF). To transform an XML blob into XML index rows, XML_Reader uses a streaming, pull-model XML parser, similar to the XmlReader in the .NET framework [9]. Searching for paths is done differently for XML blobs than using the Path_ID column in the indexed case. XML_Reader accepts simple XPath expressions without branching and generates rows for the result of the XPath expression evaluation.

The XML_Reader has a special optimization to minimize the number of passes over an XML blob. The rows generated from an XML_Reader can serve as XML node references for a second, correlated XML_Reader. The latter can use these node references to locate the corresponding XML nodes in a single pass over the blob. An example is the path expression /BOOK[@id = "123"]//TITLE, in which the rows generated by an XML_Reader for /BOOK are used as <BOOK> node references in a correlated XML_Reader. Making XML_Reader a stateful operator allows the second XML_Reader to generate <TITLE> elements in a single pass over the XML blob using the <BOOK> element references. An interesting extension is to merge multiple XML_Reader operators into a single one that evaluates multiple path expressions in a single pass over the XML blob, although this optimization is not in the product.

6.3 Using Static Type Information

Node values within an untyped XML instance are stored as Unicode strings. Operations such as numeric addition

(+) require such values to be converted at runtime to compatible types. Within a typed XML instance, on the other hand, node values are stored as the primitive XML Schema types mapped to the corresponding SQL types. For typed XML, the type conversion of values is eliminated wherever possible for faster execution. Elimination of type conversion also enables range scans over XML indexes; this can yield significantly faster execution for range queries. XML schemas are used for other optimizations as well, notably the determination of singleton cardinality.

The query processor uses the static type information to optimize queries. For instance, if it is known from the XML schema that an element <BOOK> occurs at most once in an XML instance, then parsing of an XML instance for the path expression /BOOK can stop as soon as the <BOOK> element has been found. Furthermore, searching for singleton nodes can collapse multiple node retrievals into a single parse of the XML instance.

6.4 Transformations on XML Algebra Tree

A few optimizations on the XML algebra tree are performed by the XQuery Compiler. In the expression

```
for $i in /BOOK/SECTION
    return $i/TITLE
```

the variable $i is annotated with the path /BOOK/SECTION. Within the scope of $i, a path expression relative to $i, such as $i/TITLE, is expanded into the exact path /BOOK/SECTION/TITLE. This technique is called *path collapsing*. In the indexed case, the collapsed path is mapped to an equality comparison on the Path_ID column of the primary XML index for efficient execution. In the case of XML blob, the path is used during XML data parsing to retrieve the <TITLE> nodes.

Path collapsing can be used in more complex path expressions as well, such as /BOOK/SECTION [TITLE = "Introduction"]. In practice, this simple optimization yields very good results.

The XQuery Compiler rewrites the ordinal predicates 1 and last(), as in /BOOK[1] and /BOOK/SECTION[last()], to TOP 1 ascending and TOP 1 descending, respectively. Primary XML index rows as input to the TOP operator avoid sorting the rows and yields better performance.

7. Related Work

A significant body of research exists on the execution of XQuery on an XML view of relational data, such as SilkRoute [4], XPeranto [17][18], etc. Several commercial products support XML views as well, such as Microsoft's SQL Server 2000 [10]. These approaches compile the XQuery or XPath expression into one or more SQL statements using the XML view definition. The results of execution of these SQL statements are combined into the XML result. Unlike our approach, these techniques still operate on relational data and cannot efficiently handle the full richness of general XML documents or the non-relational aspects of XQuery/XPath. Our approach is to devise query plans for rich XML data type as well as for XML indexing.

This paper differs from our earlier paper on XML indexing [12] in that it describes the internal operators used in the query trees produced for executing XQuery expressions. The XML indexing paper discusses how XML data can be indexed to speed up different query classes and how query plans use those indexes. Thus, these two papers complement one another.

Grust et al. [7] discusses an XQuery implementation on a relational database system. They use the pre-order and post-order node labeling scheme instead of OrdPath, and their approach comes closest to our treatment of indexed XML data. Their approach, however, is built outside the database engine. Our work fits XQuery compilation and execution into the relational query processing framework and pursues optimizations possible within the relational query optimizer.

Florescu et al. [5] describe an XQuery implementation on streaming XML data. By comparison, we consider a persistent, shared state of the XML data on which XQuery processing is studied using a relational query processing framework.

8. Conclusions and Future Work

This paper gives an overview of some of the major features of the XQuery language implemented in SQL Server 2005 using the relational query processor. (A data modification language is also available and fits into the relational query processing framework equally well.) The underlying query processing and data storage frameworks have been built up to provide users with a good set of features that perform and scale well.

SQL Server 2005 is the first release from Microsoft Corporation which implements XQuery. This has been a major undertaking with the primary focus of building up the infrastructure that can support the implementation of the full XQuery specification. Features not currently supported, such as "let" and typeswitch, can be implemented using the same framework. Needless to say, future work includes a long list if items.

From the language perspective, although many built-in functions are available, features such as the remaining XQuery language constructs, remaining XPath axes, user-defined function library, user-defined recursive functions, and many built-in functions and operators can be done in the future. Converting dynamic errors to empty sequences yields correct results as in predicates without negations. However, in the presence of negation and update expressions, wrong results can occur. Therefore, a future

version needs to provide better filtering of spurious errors in the execution framework.

From the query processing viewpoint, more cost-based optimizations, such as the cost-based selection of the primary XML index, can be done. This can be achieved by making the XML Operator Mapper an integral part of the relational query processor. Computed columns based on XML data type methods are useful for property promotion and are supported by SQL Server 2005. In the future, the query optimizer can provide support for matching such columns and in general matching materialized views on XML columns.

Experimental results can be found in the paper on XML indexing [12] for the XMARK benchmark. The query plans were produced for those experiments by the query processing framework described in this paper. As future work, it will be interesting to study the benefits of individual optimizations and to work on further optimizations.

ACKNOWLEDGMENTS

The authors would like to thank their colleagues Denis Altudov, Mike Rorke, Jinghao Liu, Cesar Galindo-Legaria, Milind Joshi, Florian Waas, Torsten Grabs and Joe Xavier for their discussions on XQuery implementation; and specially thank their former colleagues Peter Kukol and Chris Kowalczyk for their invaluable contributions to the project.

REFERENCES

[1] B. Beauchemin, N. Berglund, D. Sullivan. A First Look at Microsoft SQL Server 2005 for Developers. Addison-Wesley, 2004.

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

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

[4] 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.

[5] D. Florescu et al. The BEA/XQRL Streaming XQuery Processor. VLDB Conference, 2003.

[6] C. Galindo-Legaria, M. Joshi. Orthogonal Optimization of Subqueries and Aggregation. SIGMOD 2001.

[7] T. Grust, S. Sakr, J. Teubner. XQuery on SQL Hosts. VLDB Conference, 2004.

[8] J. Melton. ISO/IEC 9075-14:2003, Information technology — Database languages — SQL — Part 14: XML-Related Specifications (SQL/XML), 2004.

[9] Microsoft .NET framework. http://msdn.microsoft.com/netframework.

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

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

[12] S. Pal, I. Cseri, O. Seeliger, G. Schaller, L. Giakoumakis, V. Zolotov. Indexing XML Data Stored in a Relational Database. In *Proceedings of VLDB Conference*, Toronto, 2004.

[13] S. Pal, M. Fussell, I. Dolobowsky. XML support in Microsoft SQL Server 2005. MSDN Online, http://msdn.microsoft.com/xml/default.aspx?pull=/library/en-us/dnsql90/html/sql2k5xml.asp, 2004.

[14] S. Pal, V. Parikh, V. Zolotov, L. Giakoumakis, M. Rys. XML Best Practices for Microsoft SQL Server 2005. MSDN Online, http://msdn.microsoft.com/xml/default.aspx?pull=/library/en-us/dnsql90/html/sql25xmlbp.asp, 2004.

[15] M. Rys. XQuery and Relational Database Systems. In XQuery from the Experts, Howard Katz (ed.), Addison-Wesley, 2003.

[16] M. Rys. XQuery in Relational Database Systems. XML 2004 Conference, Washington DC, Nov 2004. http://www.idealliance.org/proceedings/xml04/abstracts/paper254.html.

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

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

[19] XML Path Language (XPath) 1.0. http://www.w3.org/TR/xpath, 1999.

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

[21] XML Schema Part 1: Structures and Part 2: Datatypes. W3C Recommendation 2 May 2001. http://www.w3.org/TR/2001/REC-xmlschema-1-20010502, http://www.w3.org/TR/2001/REC-xmlschema-2-20010502.

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

[23] XQuery 1.0 and XPath 2.0 Data Model. http://www.w3.org/TR/2005/WD-xpath-datamodel-20050211.