State-of-the-Art XML Support in RDBMS: Microsoft SQL Server's XML Features

Michael Rys, Microsoft Corp. (mrys@microsoft.com)

Abstract

XML is fast becoming the intergalactic data speak alphabet for data and information exchange that hides the heterogeneity among the components of Loosely-coupled, distributed systems and provides the glue that allows the individual components to take part in the loosely integrated system. Since much of this data is currently stored in relational database systems, simplifying the transformation of this data from and to XML in general and from and to the agreed upon exchange schema specifically is an important feature that should improve the productivity of the programmer and the efficiency of this process. This article provides an overview over the features that are needed to provide access via HTTP and XML and presents the approach taken in Microsoft SQL Server.

Keywords*: Loosely-coupled, distributed system architectures, XML, relational database systems*

1 Introduction

Many applications are built today as loosely-coupled, distributed systems where individual components (often called services) are combined together. Since many of these components will be reused for other applications, the architecture needs to be flexible enough to allow individual components to join or leave the heterogeneous conglomerate of services and components and to change their internal design and data models without jeopardizing the whole architecture.

Over the past two years, XML and the vocabularies defined with XML have established themselves as the most prevalent and promising lingua franca of loosely-coupled, distributed systems in the area of business-to-business, business-to-consumer, or generally, any-to-any data interchange and integration. One of the major reasons for the emergence of XML in this space is, that XML is a simple, platform independent, Unicode based syntax that for which simple and efficient parsers are widely available. Another important factor in favor of XML is its ability to not only represent structured data, but to provide a uniform syntax for structured data, semistructured data (data that is sparse or is of heterogeneous types) and marked-up content.

Since the interchanged data may be used independently of the integration context or used in multiple integration contexts, it is crucial that the internal modeling of the data structures is flexible enough to accommodate a changing number of contexts without sacrificing performance. Relational databases have a proven track record in providing the required efficient and flexible management of data in such multiple usage contexts. Local applications already make extensive use of relational systems to manage their data. Thus, in order to provide already existing data to the integration systems and to leverage the flexibility of relational databases, such components are often based on relational systems. However, in order to partake in the integration process as a component, the relational data needs to be translated into XML. In particular, the relational schema needs to be transformed

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

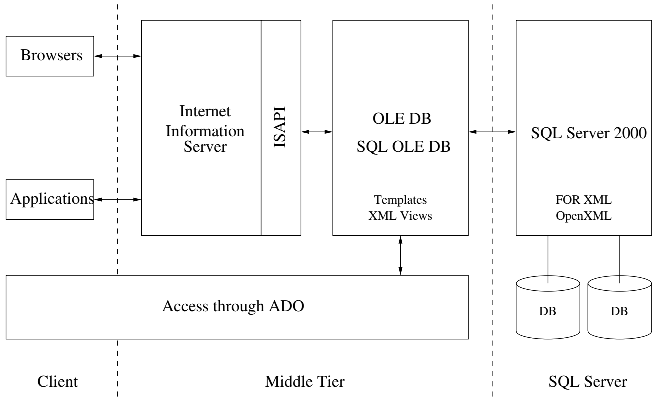

Figure 1: Architectural overview of the XML Access to SQL Server.

as easily and efficiently as possible into the XML language used in the specific application domain and incoming XML data needs to be transformed into relational data in a similar way.

Generally, one can distinguish three major scenarios for producing and storing XML data with (relational) database systems: (i) generating XML from relational data and storing structured XML data in relational format, (ii) providing ways to store and produce semi-structured XML to and from relational data, and (iii) managing arbitrary XML with the database system. Most relational database systems primarily provide support for the first two scenarios because they currently comprise the most common scenarios, support the existing customer base and allow the use of existing relational technology for efficient data management. The third scenario is currently the realm of so called native XML databases, although relational systems may provide some basic technology to support this scenario (such as a CLOB, XML data type, or node table shredders).

This article describes the basic technology to enable a relational database (in particular Microsoft SQL Server) to support these two scenarios. It will describe how to transform relational query results into XML, provide queryable and updateable XML views, and how to elegantly provide rowset abstractions over XML to shred XML into relations. A more detailed description of all features can be found in [3].

2 Architecture of the XML Access to SQL Server

Figure 1 shows a high-level architectural block diagram of SQL Server's XML support. Since different applications have different needs for where to run their business logic, the architecture provides for direct HTTP access when only visualization using XSLT needs to be performed on the mid-tier and the rest of the business logic processing can either be completely pushed to the client or to the database server. For two-tier architectures or where the business logic needs to be performed on the middle-tier, a closer coupling of the business logic to the database access is often used due to performance and programmability reasons. Thus, all access to the XML features is provided via the SQLOLEDB provider for both ADO access as well as HTTP access via the ISAPI extension to the Internet Information Server.

The major mid-tier access methods provided by SQL Server are the template access and the XML view access. Templates are XML documents that provide a parameterized query and update mechanism to the database. Since they hide the actual query (or update) from the user, they provide the level of decoupling that makes building loosely-coupled systems possible. Specially named elements containing queries are processed by the template processor and used to return database data as part of the resulting XML document. Templates can contain either T-SQL statements, updategrams, XPath queries or a combination thereof.

XML views are defined by annotating an XML schema document with the mapping to the relational tables and columns. Hierarchies are mapped from and to the database using a relationship annotation that expresses the outer-join between the parent and the children. This view can then be used to both query it using the XPath 1.0 XML navigation language [2] and to update it using so-called updategrams.

3 Serializing SQL Query Results into XML

People that are familiar with writing SQL select queries want to be able to easily generate XML from their query result. Unfortunately, there are many different ways how such a serialization into XML can be done. SQL Server therefore provides three different modes for the serialization with different levels of complexity and XML authoring capability. All three modes are provided via a new select statement extension called FOR XML. The three modes are: raw, auto, and explicit. The (simplified) syntax of the FOR XML clause is ([] indicates optional, | indicates alternative):

```
FOR XML (raw | auto [, elements] | explicit) [, xmldata]
```

All three modes basically map rows to elements and column values to attributes. The optional directive elements changes the mapping of all column values to subelements in the auto mode (the explicit mode has column-specific control over the mapping, see below). The optional directive xmldata will generate an inline schema using the XML-Data Reduced schema language as part of the result that describes the structure and data types of the XML query result.

All three FOR XML modes share the general design goals that the XML generation should not slow down arbitrary non-FOR XML queries and that the result should be streamed through the serialization process instead of generating the XML in a server-side cache. In order to not impact the database system's relational engine for non-XML queries, the serialization is performed as a post-processing step on the resulting rowset after the query execution is done. As a consequence, information about the instance lineage of the data in case of non-primary key-foreign key joins is not available for the serialization, since one cannot tell if the master data in the join comes from one or multiple rows. The second requirement is to avoid the costly build-ups of large XML fragments on the server. In order to avoid such caching, the serialization rules for hierarchical results in the auto and explicit modes require, that rows containing parent data need to be directly followed by their children and children's children data so that the data can be serialized in a streaming fashion.

3.1 The RAW Mode

The raw mode is the simplest mode. It performs a so-called canonical mapping where any row of the query result is mapped into an element with the name row and any column value that is not null into an attribute value of the attribute with the column name. NULL values are mapped to the absense of the attribute. For example, the query

```
SELECT CustomerID, OrderID
FROM Customers LEFT OUTER JOIN Orders ON Cus-
tomers.CustomerID = Orders.CustomerID
FOR XML raw
```

may return

```
<row CustomerID="ALFKI" OrderID="10643"/><row Cus-
tomerID="ALFKI" OrderID="10692"/>
```

Since the query results do not contain nested rowsets, the raw mode only returns flat XML documents where the hierarchy of the data is lost. It however works with any SQL query and the serialization process is very efficient. On the negative side, such canonical mappings are normally not generating the XML in the semantic format needed for exchanging the data. Thus the additionally required postprocessing step (e.g., using an XSLT stylesheet) adds additional processing overhead which often more than negates the initial performance gains of the serialization.

3.2 The AUTO Mode

Since the instance level lineage is not available to the serialization, the auto mode applies a heuristic on the returned rowset to determine nesting of the data. It basically maps each row to an element while using the table alias as the element name. Nesting is determined by taking schema level lineage information provided by the SQL Server query processor into account. Basically, the left to right appearance of a table alias in the SELECT clause determines the nesting. Columns of aliases that are already placed in the hierarchy are grouped together even if they appear interspersed with columns of other aliases. Computed and constant columns are associated with the deepest hierarchy so far encountered (or with the top level of the first alias). As a consequence, the auto mode cannot provide differently typed sibling elements.

The serialization process streams the XML by looking at each row that arrives from the query processor and opening a new hierarchy level for the level where all the ancestor data is unchanged, previously closing any lower hierarchies of the sibling. The auto mode's hierarchy serialization together with the instance lineage issue mentioned above means that multiple, indistinguishable parents will be merged to one parent and that parents without children and parents with children without properties will be represented as parents with children without properties. As a consequence, children that are not directly following their parent will add a duplicate parent at the place where it reappears in the rowset stream.
For example, the auto mode query

```
SELECT Cust.CustomerID, OrderID as id
FROM Customers Cust LEFT OUTER JOIN Or-
ders ON Cust.CustomerID = Orders.CustomerID
ORDER BY Customers.CustomerID
FOR XML auto
```

may return `<Cust CustomerID="ALFKI"><Orders id="10643"/><Orders id="10692"/></Cust>`.

3.3 The EXPLICIT Mode

The explicit mode allows generating arbitrary XML without any of the auto mode limitations. However, the explicit mode expects that the query be explicitly authored to return the rowset in a specific format. This format, commonly known as a universal table format, provides enough information to generate arbitrary XML. In particular, the explicit mode can generate arbitrary tree structured hierarchies, collapse or hoist hierarchical levels independently of the involved tables, and can generate IDREFS type collection attributes.

The general format and approach is best explained with an example. Explaining every detail of the explicit mode is beyond the scope of this paper. Therefore, the reader is referred to the documentation for the details. The goal is to generate an XML document of the following form:

```
<Customer cid="ALFKI"><name>Alfreds Futterkiste</name><Order oid="O-10643"/>
</Customer>
<Customer cid="BOLID"><name>Bolido Comidas preparadas</name><Order oid="O-
10326"/>
</Customer>
```

Note that while this example still only consists of a simple hierarchy, one Customer column has to be mapped to an attribute and the other one to a subelement, a task that cannot be accomplished by the auto mode. In order to generate XML of this format, the explicit mode expects a universal table of the following format:

Tag	*Parent*	*Customer!1!cid*	*Customer!1!name!element*	*Order!2!oid*
1	0	ALFKI	Alfreds Futterkiste	NULL
2	1	ALFKI	NULL	O-10643
1	0	BOLID	Bolido Comidas preparadas	NULL
2	1	BOLID	NULL	O-10326

Each row corresponds to an element (with the exception of IDREFS where each row is an element of the list). The columns Tag and Parent are used to encode the hierarchy levels for each row (if the parent tag is 0 or

NULL, the tag is the top level). The column names encode the mapping of the hierarchy levels to the element name; in the given example level 1 corresponds to an element of name Customer. The column names also encode the name of the attribute (or subelement) of the values in that column as well as additional information such as whether the value is a subelement or some other information (such as IDREFS, CDATA section, XML annotation etc.).

The serialization process takes each row and determines based on the tag level and the parent tag what level the element is. It uses the information encoded in the column name to only generate the column attributes and subelements for the current level. Thus all other columns can contain NULL. Due to the streaming requirement, children have to immediately follow their parent, thus the key field columns of the parent often contain the key values like in this example because they were used to group the children with their parent element. However the explicit mode only cares about the order and does not care about the parent's key value.

In principle, the explicit mode does not care about the query that generates the universal table format. One could create a temporary table of this format, insert the data and then perform an explicit mode query over the temporary table that guarantees the right grouping of children and parents to generate the XML. This however would most likely not perform well. Thus, the currently best way to generate this format by means of a single query is to issue a selection for each level, union all them together, and use an order by statement to group children under their parents (similar to the sorted union approach presented in [4]). This basically generates a left outer join where each join partner is placed into its own vertical and horizontal partition of the rowset. Thus the query for generating the universal table and therefore the XML in our example would look like:

```
SELECT 1 as Tag, NULL as Parent,
  CustomerID AS "Customer!1!cid", CompanyName AS "Customer!1!name!element",
  NULL AS "Order!2!oid"
FROM Customers
WHERE CustomerID = 'ALFKI' OR CustomerID='BOLID'
UNION ALL
SELECT 2, 1,
       Customers.CustomerID, NULL,
       'O-'+CAST(Orders.OrderID AS varchar(32))
FROM Customers INNER JOIN Orders ON Customers.CustomerID=Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID='BOLID'
ORDER BY "Customer!1!cid"
FOR XML explicit
```

The query processor pushes the order-by operation down to each individual selection and performs a merge union.

4 Providing XML Views over Relational Data

The previous section presented the SQL-centric approach to generating XML. SQL Server also provides a mechanism that allows defining virtual XML views over the relational database which then can be queried and updated with XML-based tools.

4.1 Annotated Schemata

The core mechanism of providing XML views over the relational data is the concept of an annotated schema. Annotated schemata consist of an XML-based schema description of the exposed XML view (using either the XML-Data Reduced or the W3C XML Schema language) and annotations that describe the mapping of the XML schema constructs onto the relational schema constructs. In order to simplify the definition of the annotations, each schema provides a default mapping if no annotation is present. The default mapping maps an attribute or a non-complex subelement (i.e., content type is text only) to a relational column with the same name. All other elements map into rows of a table or view with the same name. Hierarchies are expressed with annotations. A downloadable visual tool called the SQL XML View Mapper provides a simple and intuitive way to specify the mapping, and thus the annotations, graphically. The view can now be queried using an XML query language and updated via an XML-based update language.

The following example shows a simple XML view (XDR-based) that defines a Customer-Order hierarchy over the Customers and Orders table of the relational database. Everything that is in the default namespace belongs to the XDR schema definition; the annotations are associated with the namespace `urn:schemas-microsoft-com:xml-sql`.

```
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:sql="urn:schemas-microsoft-com:xml-sql">
 <ElementType name="Customer" sql:relation="Customers">
   <AttributeType name="ID"/> <attribute type="ID" sql:field="CustomerID"/>
   <element type="Order">
     <sql:relationship key-relation="Customers" key="CustomerID"
                       foreign-relation="Orders" foreign-key="CustomerID"/>
   </element>
 </ElementType>
 <ElementType name="Order" sql:relation="Orders">
   <AttributeType name="OrderID"/> <attribute type="OrderID"/>
 </ElementType>
</Schema>
```

The sql:relationship annotation provides the hierarchy information as a conceptual left-outer join that describes the how the child data relates to the parent data. Additional annotations exists to define XML specific information such as defining ID-prefixes, CDATA sections, and XML overflow.

The annotated schema does not retrieve any data per se, but only defines a virtual view by projecting an XML view on the relational tables. It actually defines two potential views, customers containing orders and just orders, since the schema does not define an explicit root node. The query and the updategram provide the additional information to actually determine which of the two views will be used.

4.2 Querying using XPath

XPath is a tree navigation language defined by a W3C recommendation [2]. XPath is not a full-fledged query language (it does not provide constructive elements such as projection or sub-tree pruning), but serves as a basis for navigating the XML tree structure. Each XPath basically consists of a sequence of location steps that navigate the tree with optional predicates to constrain the navigation paths.

SQL Server uses a subset of XPath to select data from the virtual XML views provided by annotated schemata. The first location step of the XPath determines the view used of the potential views defined by the annotated schema and returns the serialization of the selected nodes and their complete subtree. The currently supported constructs include easily mapable constructs such as the non-order, non-recursive navigation axes, all datatypes, most operators, and variables. Notably not supported in the current release are the id() function, the order and recursive axes such as the descendent axis.

The XPath query together with the annotated schema is translated into a FOR XML explicit query that only returns the XML data that is required by the query. The implementation goes to great length to provide the true XPath semantics such as preserving the node list order imposed by the parent orders when navigating down the tree and the XPath data type coercion rules. The only two places where the implementation differs from the W3C XPath semantics is with respect to the coercion rules of strings with the $<$ and $>$ comparison operations and with respect to node to string conversions in predicates. In the first case, the implementation does not try to coerce to a number but does a string-based comparison on the default collation, which provides support for datetime comparisons. In the second case, XPath mandates a ”first-match” evaluation semantics that cannot be mapped to the relational system. Instead, the implementation performs the more intuitive ”any-match” evaluation.

For example, the XPath `/Customer[@ID='ALFKI']` against the annotated schema above may result in `<Customer ID="ALFKI"><Order OrderID="10643"/><Order OrderID="10692"/></Customer>`.

4.3 Updating using Updategrams

Updategrams provide an intuitive way to perform an instance-based transformation from a before state to an after state. Updategrams operate over either a default XML view implied by its instance data (if no annotated schema is referenced) or over the view defined by the annotated schema and the top-level element of the updategram. The following example, gives a simple example:

```
<root xmlns:updg="urn:schemas-microsoft-com:xml-updategram">
  <updg:sync mapping-schema="nwind.xml" nullvalue="ISNULL">
    <updg:before>
      <Customer CustomerID="LAZYK" CompanyName="ISNULL" Address="12 Orches-
tra Terrace" >
        <Order oid="10482"/>
      </Customer>
    </updg:before>
    <updg:after>
      <Customer CustomerID="LAZYK" CompanyName="Lazy K Store" Ad-
dress="12 Opera Court" >
        <Order oid="10354"/>
      </Customer>
    </updg:after>
  </updg:sync>
</root>
```

Updategrams use their own namespace `urn:schemas-microsoft-com:xml-updategram`. Each updg:sync block defines the boundaries of an update batch that uses optimistic concurrency control to perform the updates transactionally. The before image in updg:before is used both for determining the data to be updated as well as to perform the conflict test. The after image in updg:after gives what has to be changed. If the before state is empty or missing, the after state defines an insert, if the after state is empty or missing, the before state defines what has to be deleted. Otherwise the necessary relational insertions, updates and deletions are inferred from the difference between the before and after image. Several optional features allow the user to deal with identity and aligning elements between the before and after state. The nullvalue attribute indicates that the fields with the specified value needs to be compared or set to NULL respectively.

In the example above, the customer with the given data (including a company name set to NULL) gets a new company name and address. In addition, the relation to the order 10482 is removed and replaced by a new relation to order 10354.

5 Providing Relational Views over XML

In many cases, data will be sent to the database server in the form of an XML message which needs to be integrated with the relational data after optionally performing some business logic over the data inside a stored procedure on the server. This requires programmatic access to the XML data from within a stored procedure. Unfortunately, neither the DOM nor SAX provides a well-suited surface API for dealing with XML data in a relational context. Instead the new API needs to provide a relational view over the XML data, i.e., it needs to allow the SQL programmer to shred an XML message into different relational views.

SQL Server provides a rowset mechanism over XML by means of the OpenXML rowset provider. OpenXML provides two kinds of rowset views over the XML data: the edge table view and the shredded rowset view. The edge table view provides the parent-child hierarchy and all the other relevant information of each node in the XML document in form of a self-referential rowset. The shredded rowset view utilizes an XPath expression (the row pattern) to identify the nodes in the XML document tree that will map to rows and uses a relative XPath expression (the column pattern) for identifying the nodes that provide the values for each column. The OpenXML rowset provider can appear anywhere in a SQL expression where a rowset can appear as a data source. In particular, it can appear in the FROM clause of any selection.

One of the advantages of this rowset-oriented API for XML data is that it leverages the existing relational model for use with XML and provides a mechanism for updating database with data in XML format. Using XML in conjunction with OpenXML enables multi-row updates with a single stored procedure call and multi-table updates by exploiting the XML hierarchy. In addition, it allows the formulation of queries that join existing tables with the provided XML data.

In order to have access to some of the implicit meta information in the tree such as hierarchy and sibling information, a column pattern can also be a so-called meta property of the node selected by the row pattern. Examples of such meta properties are @mp:id that provides the node id (the namespace prefix mp binds to a namespace that is recognized by OpenXML as providing the meta properties), @mp:parentid that provides the node id of the parent node, @mp:prev that provides the node id of the previous sibling, and the special metaproperty @mp:xmltext that allows to deal with unknown open content (the so-called overflow). For more information on these meta properties, please refer to the documentation.

The following presents the syntax of the OpenXML rowset provider ([] denote optional parts, | denotes alternatives):

```
OpenXML(hdoc, RowPattern [, Flag]) [WITH SchemaDeclaration | TableName]
```

The hdoc parameter is a handle to the XML document that has been previously parsed with the built-in stored procedure `sp_xml_preparedocument`. RowPattern is any valid XPath expression that identifies the rows or, in case of the edge table view, the roots of the trees to be returned. The optional Flag parameter allows to designate default attribute- or element-centric column patterns in the shredded rowset view. If the WITH clause is omitted, an edge table view is generated, otherwise the explicitly specified schema declaration or the implicitly through TableName given rowset schema is used to define the exposed structure of the shredded rowset view. A schema declaration has the form `(ColumnName ColumnType [ColPattern], ...)`. The ColumnName provides the name of the column, ColumnType the relational data type exposed by the rowset view, and ColumnPattern the optional column pattern (if no value is given, the default mapping indicated with the flag parameter is applied). Note that XML data types are automatically coerced to the indicated SQL data types. For example, the following T-SQL fragment parses a hierarchical Customer-Order XML document and uses the rowset views to load the customer and order data into their corresponding relational tables.

```
create procedure Load_CustOrd (@xmldoc ntext) as
declare @h int
-- Parse document
exec sp_xml_preparedocument @h output, @xmldoc
-- Load the Customer data
insert into Customers select * from OpenXML(@h, '/load-
doc/Customer') with Customers
-- Load the Order data. Get the customer id from the parent element
insert into Orders(OrderID, CustomerID, OrderDate)
  select * from OpenXML(@h, '/loaddoc/Customer/Order', 2)
    with (oid int, customerid nvarchar(10) '../@CustomerID', OrderDate datetime)
-- Remove the parsed document from the temp space
exec sp_xml_removedocument @h
go
```

6 Conclusion and Future Work

This paper presents the basic technologies required to enable a relational database to become an XML-enabled component. Based on SQL Server's XML support, it gives an overview over the different building blocks such as queryable and updateable XML views, rowset views over XML and XML serialization of relational results. Rowset views over XML and the XML serialization of relational results can be characterized as providing XML support for users feeling comfortable in the context of the relational world, whereas the XML views provides XML-based access to the database for people more familiar with XML. Some of the discussed features are at the time of the publication not part of the shipped version of SQL Server 2000, but are released as a web release that can be downloaded from `http://msdn.microsoft.com/sqlserver`. In particular, the SQL XML View Mapper, Annotated W3C XML schemata, updategrams, bulkload and the nullvalue attribute are part of the web release.

Future work will include support for the upcoming W3C XQuery language [1] against the annotated schema view.

References

[1] XML query language (XQuery) version 1.0; W3C working drafts. http://www.w3.org/XML/Query.html.

[2] J. Clark and S. DeRose. XML path language (XPath) version 1.0; W3C recommendation 16 November 1999. http://www.w3.org/TR/xpath.

[3] M. Rys. Bringing the Internet to your database: Using SQL Server 2000 and XML to build loosely-coupled systems. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 465–472. IEEE Computer Society, 2001.

[4] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Linday, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *Proceedings of the 26th International Conference on Very Large Databases, 2000, Cairo, Egypt*, 2000.