Bringing the Internet to Your Database: Using SQL Server 2000 and XML to Build Loosely-Coupled Systems

Michael Rys
Microsoft Corporation (http://www.microsoft.com/sql)
mrys@microsoft.com

Abstract

Loosely-coupled, distributed system architectures need 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. A well-established approach is to use XML as the lingua franca for the integration layer that hides the heterogeneity among the components and provides the glue that allows the individual components to take part in the loosely integrated system.

This presentation focuses on how to provide the basic technology to enable a relational database to become a component in such loosely-coupled systems and it will provide an overview over the features that are needed to provide access via HTTP and XML.

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

1 Introduction

Loosely-coupled, distributed system architectures need 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. A well-established approach is to use a lingua franca for the integration layer that hides the heterogeneity among the components and provides the infrastructure that allows the individual components to take part in the loosely integrated system.

Over the past two years, XML and the languages/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 (including data-driven website design). One of the major reasons for the emergence of XML in this space is, that XML is a simple, platform independent, Unicode based syntax 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.

Often the individual components of the distributed systems are using a different (data) language than the integration level and need to provide some wrapping and transformation of their internal data structures and models into the integration model. An important aspect of this wrapping step and the integration-level model is, that any internal changes to any of the components in the distributed system does not require any change to the internals of any other component.

One of the more important building blocks of such a component is a database that locally stores and manages the data. Often, this data may be used independently of the integration context or used in multiple integration contexts. Thus, it is crucial that the internal modeling of the data structures needs to be 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 as easy and efficiently as possible into the XML language used in the specific application domain.

This paper focuses on how to provide the basic technology to enable a relational database (in particular Microsoft's SQL Server 2000) to become a component in such loosely-coupled systems. It will provide an overview over the features that are needed to provide *database access via HTTP*, transform relational query results into the XML language that is used by the integration layer, and provide queryable and updateable *XML views*. Next we will also present, how to elegantly provide rowset abstractions over XML to shred XML into relations and how to efficiently bulk load XML data through the XML view mechanism into the relational database. The discussion of generating XML for SQL queries will discuss canonical, heuristic-based and user-defined strategies and point out their use scenarios and potential shortcomings with respect to the decoupling of external and internal data formats.

2 Architecture of the XML Access to SQL Server

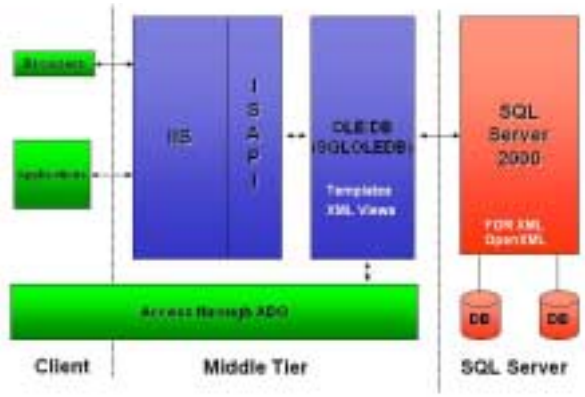

Figure 1: Architectural overview of the XML Access to SQL Server

Figure 1 shows a high-level architectural block diagram of SQL Server 2000's XML support. It is interesting to note that depending on the overall system or service architecture, different access components and/or protocols are preferable. If there is a limited amount of business logic, or most of the business logic can either be completely pushed to the client (for example via ECMAScript) or to the database server (into stored procedures), a simple HTTP access mechanism suffices. For two-tier architectures or where the business logic needs to be performed on the webserver (the middle-tier), a closer coupling of the business logic to the database access is often used due to performance and programmability reasons. Thus, the XML database access story should also be accessible through the standard access components such as OLEDB and ADO.

For these reasons, SQL Server 2000 provides all access to its XML features (except bulkload) via the SQLOLEDB provider and ADO as well as via an IIS ISAPI extension that provides access to the functionality via HTTP. In the following, we will discuss each XML feature in more detail and show where they fit into the architecture and how they can be accessed both via HTTP and ADO.

2.1 HTTP Access to SQL Server

Several ways exist to access SQL Server via HTTP. The most common one is to write an active server page that accesses the database via ADO. SQL Server 2000 introduces an HTTP access mechanism via an ISAPI extension. The URL formats of the new HTTP access methods start with:

```
http://domainname/vroot
```

The SQL Server ISAPI is registered with IIS to handle messages to a particular *virtual root* (vroot). The ISAPI will receive the requests for that particular vroot, and, after performing authorization, will then pass the appropriate commands via the SQLOLEDB provider to the database.

The virtual root as part of the URL provides an abstraction mechanism that encapsulates the accessed database server and database instances, the access rights and the enabled access methods. Currently, SQL Server 2000 provides the following access methods:

- *Adhoc URL query mechanism:*
  `http://domainname/vroot?sql=select+xmldoc+from+table`
  Allows arbitrary T-SQL statements (including updates and data definition statements). Query results are returned in their native, binary representation. Together with the T-SQL XML extensions, it can return XML. Since this mechanism is dangerous to enable in production systems, we will not further elaborate on this mechanism.
- *Direct query access:*
  `http://domainname/vroot/dbobject/Table[@column1=value]/@column2`
  Provides native access to the binary representation of the data while only allowing a safer set of queries via an XPath-like simple query syntax. See below for an example where it will be used.
- *Template access:*
  `http://domainname/vroot/template/templatefile?param`
  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.
  Elements in the `urn:schemas-microsoft-com:xml-sql` namespace are processed by the template processor and used to return database data as part of the resulting XML document. Elements in other namespaces are returned to the client unmodified. The templates support named parameters to parameterize the queries.
- *XPath XML view access:*
  `http://domainname/vroot/schema/schemafile/XPATH?param`
  This URL provides a way to query an XML view that is specified by the annotated schema referenced by the schema file in an adhoc way using XPath queries. The URL can be parameterized in the same way templates are. For more details on XPaths against annotated schemata, see below.

`dbobject`, `template`, and `schema` designate three types of *virtual names* for the use with the ISAPI extension. They provide an abstraction for the access methods and in the case of the last two types of virtual names also the location of the files that are associated with the access. In addition, all access mechanisms can be parameterized with predefined parameters that among others allow specifying the output encoding, the character set used in the URL, and a server-side XSLT transform.

One of the most important aspects of providing access to a database via HTTP is the security. The URL access

mechanism has to guarantee that only authorized people can access those parts of the database to which they are allowed access. The ISAPI extension provides three authentication modes on a per-vroot basis. This allows the database administrator to set the database internal access rights for the authenticated users inside the database since only those users that are authenticated will be allowed access. The first mode is the standard HTTP/HTTPS-based basic authentication. The second mode allows every user that connects to the server to impersonate the vroot specific Windows or SQL Server user. The user/password combination is associated with the vroot and cannot be changed or retrieved by the user. The final authentication mode takes advantage of Windows' ACL such that the authentication is done by a previous authentication event and the credentials are then passed on to the database. This allows the use of a single vroot with multiple access rights without prompting for the user/password combination.

2.2 Using the XML features through SQLOLEDB and ADO

All the XML features are also accessible through SQLOLEDB. SQL Server 2000's OLEDB provider has been extended with a stream interface that is accessible via ADO's stream interface. *XML* and *XPath* were added as new dialects to SQLOLEDB. XML indicates that the input is a template file, XPath indicates that the input is an XPath query with its associated annotated schema. Using a FOR XML query as described below does not necessitate a new dialect, but as for any XML specific result, the result needs to be returned via the stream interface. Some of the predefined URL parameters such as the output encoding or the XSLT transform are also exposed as SQLOLEDB and ADO properties.

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 clause called `FOR XML`. The three modes are: `raw`, `auto`, and `explicit`. The syntax of the `FOR XML` clause is (`[]` indicates optional, `|` indicates alternative):

```
FOR XML (raw | auto [, elements] |
explicit) [, binary base64] [, 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 `binary base64` is required in the raw and explicit modes if a binary column of type `binary`, `varbinary` or `image` is returned. It indicates for all three modes that the binary data should be returned inline in the XML document in base-64 encoding. The auto mode will generate a direct query if no mode is specified. 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 datatypes of the XML query result.

Before we delve into the three modes, we need to understand some of the general architectural design decisions that are common for all the current FOR XML modes.

The first requirement for the current implementation was to not impact the database system's relational engine. Therefore the serialization process has to happen as a post-processing step on the resulting rowset after the query execution is done and thus is not part of the general query processing. As a consequence to the current implementation, FOR XML query results cannot be assigned to columns but need to be returned directly to the OLEDB provider. It also means that information about the lineage of the data in case of non-primary key-foreign key joins is lost, since we cannot tell if the master data in the join comes from one or multiple rows.

The second requirement wants to avoid caching 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 therefore require, that rows containing parent data need to be directly followed by their children and children's children data.

Furthermore, due to the open-world assumption of XML (in contrast to the closed-world assumption of relational systems), relational NULL values are serialized by the absence of the instance value. Finally, when a row is mapped to an element, a query returning multiple top rows will generate an XML fragment. To make it into a well-formed XML document, a root element needs to be added via the template mechanism or via the root property of SQLOLEDB.

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. For example, the query

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

may return

```
<row CustomerID="ALFKI" OrderID="10643" />
<row CustomerID="ALFKI" OrderID="10692" />
<row CustomerID="ANATR" OrderID="10308" />
```

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.

3.2 The AUTO Mode

The `auto` mode applies a heuristics 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). These heuristics together with the loss of the instance level lineage make it impossible to provide differently typed sibling elements: the generated hierarchy will be a simple hierarchy that will introduce a new level for every new table alias.

Due to the streaming requirement, the serialization process then looks at each row that arrives from the query processor, opens a new hierarchy level for the level where all the ancestor data is unchanged, previously closing any lower hierarchies of the sibling.

The hierarchy serialization of the auto mode together with the lineage issue of the first requirement 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. The serialization process also means that if children are not directly following their parent, a duplicate parent will be generated where it reappears in the rowset stream.

For example, the auto mode query

```
SELECT Customers.CustomerID, OrderID
FROM Customers LEFT OUTER JOIN Orders
ON Customers.CustomerID =
Orders.CustomerID
ORDER BY Customers.CustomerID
FOR XML auto
```

may return

```
<Customers CustomerID="ALFKI">
  <Orders OrderID="10643" />
  <Orders OrderID="10692" />
</Customers>
<Customers CustomerID="ANATR">
  <Orders OrderID="10308" />
</Customers>
```

Note the order by clause that is used to group all children with their parent.

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 [3].

The goal is to generate an XML document of the following form:

```
<Customer cid="ALFKI">
  <name>Alfreds Futterkiste</name>
  <Order oid="O-10643" />
  <Order oid="O-10692" />
</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 format given in Figure 2.

Tag	*Parent*	*Customer!1!cid*	*Customer!1!name!element*	*Order!2!oid*
1	0	ALFKI	Alfreds Futterkiste	NULL
2	1	ALFKI	NULL	O-10643
2	1	ALFKI	NULL	O-10692
1	0	BOLID	Bolido Comidas preparadas	NULL
2	1	BOLID	NULL	O-10326

Figure 2: Universal Table format for explicit mode

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 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 (as 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. 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. 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
```

Users who prefer a simpler way to formulate these queries, can use the XML view and XPath mechanism explained below to generate such explicit mode queries under the covers.

4 SQLXML Templates

Templates are XML documents that provide a parameterized query and update mechanism to the database. Templates can contain either T-SQL statements, Updategrams, XPath queries or a combination thereof. Examples for Updategrams and XPath queries are given in later sections. The following shows the basic structure of a parameterized template that retrieves a Customer-Order hierarchy for a specific region and applies an XSLT postprocessing step that transforms the template result into either another XML document or some other format (such as HTML):

```
<root xmlns:s =
    "urn:schemas-microsoft-com:xml-sql"
      s:xsl="CustOrd.xsl" >
  <s:header nullvalue="NULL" >
    <s:param name="state">%</s:param>
  </s:header>
  <s:query>
   SELECT Customers.CustomerID, OrderID
   FROM Customers LEFT OUTER JOIN
   Orders ON Customers.CustomerID =
   Orders.CustomerID
   AND Region LIKE @state
   ORDER BY Customers.CustomerID
   FOR XML auto
  </s:query>
</root>
```

The optional `s:xsl` attribute on the template root element points to the file that the server-side XSLT transformation will apply to the template results. The transformation will take place inside the SQLOLEDB provider. The `s:header` contains the parameter declarations. Each parameter declaration provides the name of the parameter and a default value. The optional attribute `nullvalue` declares that the specified string (`"NULL"` in the example) will be interpreted as the NULL value if passed as the parameter value. The parameters will be referenced by name inside the queries using their native variable reference mechanism. `s:query` can contain arbitrary T-SQL statements (best placed into a CDATA section to avoid problems with < or other special characters). In the example above it is a FOR XML query. The result of the template before the application of the XSLT transform with the default parameter values may look like:

```
<root xmlns:s=
    "urn:schemas-microsoft-com:xml-sql" >
  <Customers CustomerID="LAZYK">
    <Orders OrderID="10482" />
    <Orders OrderID="10545" />
  </Customers>
  <Customers CustomerID="TRAIH">
    <Orders OrderID="10574" />
    <Orders OrderID="10577" />
    <Orders OrderID="10822" />
  </Customers>
</root>
```

5 Providing XML Views over Relational Data

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

5.1 Annotated Schemata

The core mechanism of providing XML views over the relational data is the concept of an *annotated schema*. Annotated schemata consist of a schema description of the exposed XML view and annotations that describe the mapping of the XML schema constructs onto the relational schema constructs. Currently SQL Server 2000 uses the XML-based XML-Data Reduced (XDR) schema language, that provides a basic description of XML documents similar to DTDs and in addition provides information about the datatypes. Since the schema documents are XML documents and their content model is open, the annotations can be placed inline.

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. Since SQL Server 2000 does not support nested relations, hierarchy is not mapable without an annotation.

The following example shows a simple XML view 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 familiar XML SQL namespace.

```
<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 `Customer` element is mapped to the `Customers` table using the `sql:relation` annotation, its attribute `ID` is mapped to the table's `CustomerID` column with the `sql:field` annotation. Finally, the similarly mapped `Order` element is parented under the `Customer` element. 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 and CDATA sections, and annotations to define limit values to deal with value-driven horizontal partitions.

In order to define the annotations in a graphical way, a utility called the SQL XML View Mapper is available for download from the Microsoft website [1].

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 XDR does not define an explicit root node. Thus, we need additional information to actually determine which of the two views will be used. This information will be provided implicitly by the query and the Updategram as explained below.

5.2 Querying using XPath

XPath is a tree navigation language and is defined in a W3C recommendation [4]. 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.

SQLServer 2000 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 exact view used of the potential many views defined by the annotated schema by determining the first hierarchy. Instead of returning only a collection of nodes, the selected nodes and their complete subtree is serialized in the resulting XML fragment.

In principle, XPath constructs that are easily mapped to the relational constructs of SQL Server are supported. The currently supported constructs include the non-order, non-recursive navigation axes, all datatypes, all relational and Boolean operators, all but one arithmetic operation, and variables (see the documentation [3] for the currently not supported constructs).

Since the views are virtual, 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 and the XPath datatype 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 the following XML

```
<Customer ID="ALFKI">
  <Order OrderID="10643" />
  <Order OrderID="10692" />
</Customer>
```

The XPath query can be passed via a URL, a template, or via the XPath dialect of the SQLOLEDB provider. The following shows each access method for the query example given above (assuming that the schema file is called `CustOrd.xdr`). First the URL:

```
http://domainserver/dbvroot/schema/CustOr
d.xdr/Customer[@ID='ALFKI']
```

The following template wraps the result with a `root` element and parameterizes the ID. The `mapping-schema` attribute on the `s:xpath-query` element indicates the location of the annotated schema relative to the template file.

```
<root>
  <s:header xmlns:s=
    "urn:schemas-microsoft-com:xml-sql" >
    <s:param name="cid">ALFKI</s:param>
  </s:header>
  <s:xpath-query
   mapping-schema="CustOrd.xdr"
   xmlns:s=
    "urn:schemas-microsoft-com:xml-sql" >
    /Customer[@ID=$cid]
  </s:xpath-query>
</root>
```

Finally, the following Visual Basic fragment shows how to use ADO to post an XPath query:

```
conn.Open strConn
Set cmd.ActiveConnection = conn
cmd.Dialect =
 "{ec2a4293-e898-11d2-b1b7-00c04f680c56}"
cmd.CommandText="/Customer[@ID='ALFKI']"
cmd.Properties("Output Stream").Value =
  Response
cmd.Properties("Base Path")="c:\schemas"
cmd.Properties("Mapping schema") =
  "CustOrd.xdr"
cmd.Execute , , adExecuteStream
```

5.3 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 Orchestra Terr." >
        <Order oid="10482"/>
      </Customer>
    </updg:before>
    <updg:after>
      <Customer CustomerID="LAZYK"
        CompanyName="Lazy K Store"
        Address="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` provides the state to which the data 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 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.

5.4 Bulkloading

Neither Updategrams nor the OpenXML mechanism described below are well-suited to load large amounts of XML data into the database since they both require the whole XML document to be loaded into memory before they perform the insertion of the data. Bulkload is provided as a COM object that allows loading large amounts of XML data via an annotated schema into either an existing database or after creating the relational schema implied by the annotated schema. Transacted and non-transacted load mechanisms are available and Bulkload can be integrated into a data transformation system (DTS) workflow.

For efficiency, the XML data is streamed in only once via the SAX parser to perform the load. Therefore, the loaded XML data needs to satisfy certain conditions in order to provide the correct loading of the data. For example, all the data that generates a new row needs to appear in the same element context. In order to avoid buffering a potentially large amount of XML data, the information that contains the key of the parent has to appear before any of its children.

5.5 Annotated Schemata and BizTalk

The XDR schema language is also used by Microsoft's BizTalk initative [2]. Thus, if one needs to publish data from the database in a BizTalk compatible format or load such data, it is very compelling to use an XPath query or an Updategram against an annotated copy of the BizTalk schema.

6 Providing Relational Views over XML

In many cases, data will be sent to the database server in 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. Since neither the DOM nor SAX provides a well-suited surface API for dealing with XML data in a relational context, 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 2000 provides such 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.

In order to have access about 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. For more information on these meta properties, please refer to the documentation [3].

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 `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 following form:

```
(ColumnName1 ColumnType1 [ColPattern1],
ColumnName2 ColumnType2 [ColPattern2],…)
```

The `ColumnName` provides the name of the column, `ColumnType` is the relational datatype 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 LoadCO (@xmldoc ntext)
as
declare @h int
-- Parse document
exec sp_xml_preparedocument @h output,
@xmldoc
-- Load the Customer data (attr-centric)
insert into Customers
  select * from
  OpenXML(@h, '/loaddoc/Customer')
  with Customers
-- Load the Order data (elem-centric).
-- use explicit schema declaration to
-- get fk value.
insert into Orders(OrderID, CustomerID,
OrderDate)
  select * from
  OpenXML(@h,'/loaddoc/Customer/Order',2)
  with (oid int, customerid nvarchar(10)
'../@CustomerID', OrderDate datetime)
-- Cleanup temp space
exec sp_xml_removedocument @h
go
```

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. Utilizing 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.

7 Conclusion and Future Work

This paper presents the basic technologies required to enable a relational database to become an XML-enabled component in an XML-based loosely-coupled system. Based on SQL Server 2000's XML support, it gives an overview over the different building blocks such as HTTP access, 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. For additional information on each of these areas please refer to SQLServer's Books OnLine documentation [3] and the SQL XML Web release site at [1].

8 References

[1] http://msdn.microsoft.com/xml
[2] http://www.biztalk.org
[3] SQL Server 2000 Books Online, Microsoft Corp.
[4] http://www.w3.org/TR/xpath