Generating SQL/XML Query and Update Statements

Matthias Nicola
IBM Silicon Valley Lab
555 Bailey Avenue
San Jose, CA, USA
mnicola@us.ibm.com

Tim Kiefer[1]
Technische Universität Dresden
Database Technology Group
01062 Dresden, Germany
tim.kiefer@inf.tu-dresden.de

ABSTRACT

The XML support in relational databases and the SQL/XML language are still relatively new as compared to purely relational databases and traditional SQL. Today, most database users have a strong relational and SQL background. SQL/XML enables users to perform queries and updates across XML and relational data, but many struggle with writing SQL/XML statements or XQuery update expressions. One reason is the novelty of SQL/XML and of the XQuery expressions that must be included. Another problem is that the tree structure of the XML data may be unknown or difficult to understand for the user. Evolving XML Schemas as well as hybrid XML/relational schemas make it even harder to write SQL/XML statements. Also, legacy applications use SQL but may require access to XML data without costly code changes.

Motivated by these challenges, we developed a method to generate SQL/XML query and update statements automatically. The input is either a GUI or a regular SQL statement that uses logical data item names irrespective of their actual location in relational or XML columns in the database. The output is a SQL/XML statement that queries or updates relational and XML data as needed to carry out the original user statement. This relieves the user and simplifies schema evolution and integration. We have prototyped and tested the proposed method on top of DB2 9.5.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages–*Query languages*

General Terms

Languages, Design, Algorithms,

Keywords

SQL, SQL/XML, Translation, Generation, Mapping, XQuery

1. INTRODUCTION

The development of the relational data model and well-defined concepts for data normalization, relational algebra, and query optimization have laid the foundation for the broad commercial success of relational databases. A corner stone of that success is the Structured Query Language (SQL). Today, commercial applications in every industry rely heavily on SQL and relational databases. For many companies, these applications include mission-critical systems and constitute a tremendous investment. Thus, today's database administrators and database application developers typically have a skill set that is heavily geared towards relational databases and SQL.

XML has continued to emerge as the de-facto standard for data exchange. The main reasons include that XML is extensible, flexible, self-describing, and suitable for combining structured, unstructured and semi-structured data. Many enterprises also *store* large amounts of business data permanently as XML, to fulfill auditing and compliance requirements or when XML is more suitable than a relational schema. For example, highly variable data is often easier to handle in XML than in relational format.

In response to these needs, the major database vendors have added XML capabilities to their products [11][12][13]. Additionally, the SQL standard has been extended to include an XML data type as well as XML-specific functions and predicates. This is known as SQL/XML [1]. The functions XMLQUERY and XMLTABLE as well as the predicate XMLEXISTS enable users to include XPath, XQuery, and XQuery Update expressions [14] in SQL statements. SQL/XML allows users to query or update XML and relational data in an integrated manner. This is necessary because most companies do not manage XML data separately from their relational data. Instead, a hybrid database design is commonly used where tables contain a mix of XML and relational columns.

Over several years we have worked with many companies to assist them in the design, implementation, and deployment of XML applications on top of DB2. This includes applications in retail [4], government [5], health care [6], finance [10], and others [7]. All observations, challenges, and assumptions described in this paper are based on our experiences with such real applications[2].

We frequently observe that the adoption of SQL/XML faces several challenges. When relational legacy applications require access to new XML data, it is often too expensive or risky to convert them from SQL to SQL/XML. Another frequent challenge is to actually write queries and updates with SQL/XML and XQuery. We see that their use poses a number of problems:

- Users need to learn these new languages, which are often perceived as difficult to master. This stems from the differences between the XML data model and the relational data model.
- SQL/XML involves path expressions that navigate the tree structure of XML documents. To write path expressions, users must know the structure of the XML data in detail. It is not enough to know which data items exist, it is also necessary to know their exact case-sensitive name, namespace, and location within the document structure. But, this structure is often complex, difficult to understand, or even unknown to the user.

[1] Author worked on this project while employed at IBM US.

[2] For privacy reasons we do not correlate specific observations or challenges to individual companies in this paper.

- As more XML documents are accumulated in a database, newer documents may have a different XML Schema than older ones. This requires queries and updates to work across documents for different schemas, which compounds the complexity of writing SQL/XML statements. Also, existing XML queries may need to be changed when the XML Schema evolves.
- In a hybrid database, where some data is stored in relational format and some in XML format, users need to know which data is in which format before they can write correct queries.

We have observed that these issues make it difficult, sometimes even impossible, for users to write correct SQL/XML statements. Therefore we have designed and prototyped a method to generate SQL/XML queries and updates automatically. In the case of updates, XQuery transform expressions [14] are generated and included in SQL update statements. We introduce our approach with focus on SQL/XML queries and revisit updates in Section 6.

The input to our generation algorithm is a simple representation of the user query. The input can come from a GUI, QBE (query by example), or web interface where users see logical data items that can be selected for projection, filtering, grouping, or ordering. The input can also be a regular SQL statement that uses logical data item names as column names in the select, where, group-by, and order-by clauses. This paper focuses on the translation from SQL to SQL/XML, but similar concepts work for GUI or QBE input.

The translation process uses a mapping from *logical data item names* to their *actual locations*. Actual locations of data items are described by a relational column name or an XML column name with an XPath expression, all required namespaces, and other metadata. We also present a semi-automatic method to generate the mapping information and to store it in a mapping table.

The mapping table provides benefits beyond the query translation: (1) It serves as a metadata repository for all XML and relational data in the database. While relational columns are documented in the system tables of the database, XML columns are often a black box for the end user. The mapping table contains information about all XML attributes and elements that carry business data. Even if an XML Schema exists, it may consist of many schema documents and its notation is too complex to serve as metadata for users who write queries and updates. (2) The mapping helps users and applications to deal with schema evolution. If XML data for a new (version of a) XML Schema is added to the database, the mapping table can be updated or regenerated while SQL queries remain unchanged. Instead, the translation algorithm uses the updated mapping to generate new SQL/XML statements that take the new schema information into account. The mapping table is the single point of maintenance for schema evolution and relieves users from manually modifying all existing SQL queries across all applications that use the database. This resilience was a specific requirement from several companies that we worked with.

We have implemented the mapping table generation and the query translation on top of the SQL/XML support in DB2 [12]. The query translation can run either as a Java library in an application or as a Java stored procedure in DB2. The stored procedure takes an SQL statement as parameter, generates and executes a corresponding SQL/XML statement, and returns the desired result set.

In Section 2 we introduce the sample data for this paper and provide a brief recap of key SQL/XML concepts. Section 3 describes the content and generation of the mapping table. The translation of queries and updates is explained in Sections 4, 5 and 6. Section 7 discusses schema diversity and Section 8 compares related work. Section 9 concludes with a summary.

2. SAMPLE DATA AND SQL/XML

The examples in this paper are based on the database schema with two tables in Figure 1. Figure 2 shows one sample document for the `customer` table and two documents for the `order` table. Customer documents are variable in that the `Status` information is either an element or an attribute. In our example, orders arrive from two different order entry systems and therefore have two different XML structures that represent the same information.

```
create table customer (cid int, cdoc xml)
create table order(odoc xml)
```

Figure 1 Database Schema

```
<Customer xmlns="http://mycompany.org/customer"
          xmlns:a="http://mycompany.org/address">
  <Name>John Smith</Name>
  <a:Addr Country = "United States"/>
  <Phone>123-456-7890</Phone>
  <Phone>123-555-6523</Phone>
  <Email>jsmith@hiscompany.com</Email>
  <Email>john147@yahoo.com</Email>
  <Status>1</Status>
</Customer>

<Order Oid="42" xmlns="http://mycompany.org/ord">
  <CustomerRef>27</CustomerRef>
  <Lineitem>
     <Product>Printer</Product>
     <Qty>2</Qty>
     <Price>254.15</Price>
  </Lineitem>
  <Lineitem>
     <Product>Paper</Product>
     <Qty>5</Qty>
     <Price>12.95</Price>
  </Lineitem>
</Order>

<Order Oid="43" xmlns="http://mycompany.org/ord">
  <CustomerRef>83</CustomerRef>
  <Product Qty="10" Price="700">PC X</Product>
  <Product Qty="12" Price="950">Laptop Y</Product>
</Order>
```

Figure 2 Sample Documents

The SQL/XML query in Example 1 produces a phone directory for all customers whose status is 1. It returns the relational column `cid` together with name, country and phone information from the XML column. One row is returned for each `Phone` element. Since `Phone` is a repeating element, the row-generating expression in the XMLTABLE function is `$CDOC/c:Customer/c:Phone`. This row-generating expression provides the context for the column definitions in the COLUMNS clause. There, the elements `Name` and `Phone` and the attribute `Country` are extracted and assigned to SQL data types and relational column names. These column names are referenced in the select and order-by clauses. The XMLEXISTS predicate in the where clause ensures that only customers are selected whose `Status` element or attribute is 1.

A lot of data-specific knowledge is required to write this query. First, a missing or incorrect namespace declaration leads to an empty result set. The element `Addr` is in a separate namespace which requires an extra declaration. Correct handling of namespaces is typically difficult for users who are new to SQL/XML. Also, the location of the `Name` and `Addr` elements relative to the `Phone` element must be known to write correct relative XPath expressions in the column definitions. To avoid truncation or errors, the user must also know that the customer's name can be safely cast to the SQL type varchar(30). It is also critical to know that `Status` can be an element or an attribute; otherwise the query result will be incomplete. Also, it is not obvious to the end user that `Status` is always a number so that a numeric predicate is safe and does not fail with a type error. Since table and column names in SQL are case-insensitive by default, many users even struggle with the case-sensitivity of XML tags. The mapping and translation mechanism that we propose solves these problems.

```
SELECT cid, name, country, phone
FROM customer,
 XMLTABLE(XMLNAMESPACES(
            'http://mycompany.org/cust' AS "c",
            'http://mycompany.org/addr' AS "a"),
   '$CDOC/c:Customer/c:Phone'
   COLUMNS
    name    varchar(30) PATH '../c:Name',
    country varchar(30) PATH '../a:Addr/@Country',
    phone   varchar(12) PATH '.' ) AS T
WHERE
 XMLEXISTS('declare namespace c =
    http://mycompany.org/customer;
    $CDOC/c:Customer[@Status=1 or c:Status=1)]')
ORDER BY name;
```

Example 1 Sample SQL/XML Query

3. MAPPING TABLE

In this section we first explain the information stored in the mapping table and then a method to generate it automatically.

3.1 Mapping Table Structure

The mapping assigns a logical name to each relational column in the database and to each distinct element or attribute name that occurs in an XML column. If the same element or attribute occurs on multiple different paths, then these paths can be mapped to the same or different logical names, as we explain later in this section.

Without loss of generality, we exclude non-leaf elements from the mapping table, such as `Lineitem`, and only list leaf elements, i.e. ones that only contain text nodes with the actual business data. For data centric XML we found that users find it more useful to query the values of leaf-elements, such as `Qty` and `Price`, rather than the values of non-leaf elements, which are defined as the concatenation of all descendant text nodes. The query translation does not depend on this decision. It also works if non-leaf elements are listed and queried, e.g. if mixed content is expected.

The overall mapping consists of two parts which are implemented as two tables. The mapping for our sample documents is shown in Tables 1 and 2. The first table maps logical data item names to:

- A *column* name and a corresponding *table* name. If the column is of type XML, then the mapping also includes:
- One or more *XPath* expressions.
- The *repeat level*, i.e. the deepest level in an XPath expression at which an element can occur more than once. The repeat level is 0 if the path occurs at most once per document. For the path `/ns3:Order/ns3:Lineitem/ns3:Qty` the repeat level is 2 because the repeating element is `Lineitem` and not `Qty`. This information is needed to properly choose the row-generating expression for XMLTABLE functions.
- An SQL *data type* that the XML element or attribute can be cast to. This is required for XMLTABLE functions.

The logical data item *customerID* maps to a relational column, as indicated by the special value *SQL* in the XPath column of the mapping table. The other table maps namespace prefixes to URIs. This separation from the main mapping table allows us to handle XML data where multiple different namespaces occur in a single path, as for the logical data item *country*.

It is important to distinguish between the two orthogonal notions of (a) *multiple paths* per logical data item and (b) *multiple occurrences* of a given path:

(a) A logical data item can be mapped either to the same path in all documents, or to multiple different paths. For example, a customer's status information can be an element in one instance document and an attribute in another. Therefore, the logical data item *status* has multiple associated paths. But, we assume that

Table 2 Namespace Mapping Table

Prefix	Namespace
ns1	http://mycompany.org/customer
ns2	http://mycompany.org/address
ns3	http://mycompany.org/ord

Table 1 Main Mapping Table

Logical Data Item	Table	Column	XPath	Repeat Level	SQL Data Type
name	customer	CDOC	/ns1:Customer/ns1:Name	0	varchar(30)
country	customer	CDOC	/ns1:Customer/ns2:Addr/@Country	0	varchar(30)
phone	customer	CDOC	/ns1:Customer/ns1:Phone	2	varchar(12)
email	customer	CDOC	/ns1:Customer/ns1:Email	2	varchar(30)
status	customer	CDOC	/ns1:Customer/@Status	0	integer
status	customer	CDOC	/ns1:Customer/ns1:Status	0	integer
customerID	customer	cid	*SQL*	-	integer
orderID	order	ODOC	/ns3:Order/@Oid	0	integer
customerRef	order	ODOC	/ns3:Order/ns3:CustomerRef	0	integer
product	order	ODOC	/ns3:Order/ns3:Lineitem/ns3:Product	2	varchar(50)
product	order	ODOC	/ns3:Order/ns3:Product	2	varchar(50)
qty	order	ODOC	/ns3:Order/ns3:Lineitem/ns3:Qty	2	double
qty	order	ODOC	/ns3:Order/ns3:Product/@Qty	2	double
price	order	ODOC	/ns3:Order/ns3:Lineitem/ns3:Price	2	double
price	order	ODOC	/ns3:Order/ns3:Product/@Price	2	double

different paths for the same logical data item do not occur in the same instance document. We verified in several real XML applications that this assumption is reasonable. If two nodes with the same name appear on different paths in the same document, then they almost always have logically distinct meanings, despite their identical names. The notable exception is recursive XML, which we encounter relatively rarely in real XML applications.

(b) Separately, any XML element or attribute on any path may occur multiple times in the same document (e.g. /Customer/phone).

3.2 Generating the Mapping Information

Collecting the required mapping information manually is not practical for any but trivial XML data. We found that for real-world XML data the main mapping table can contain thousands of entries. For example, industry-specific XML Schemas such as FpML, FIXML, OAGIS, HL7, STAR, or UNIFI define thousands of optional XML elements and attributes. Hence, the population of the mapping table must be automated as much as possible.

Our mapping tables are populated semi-automatically, either based on a *master document* or a set of representative *instance documents*. A master document is generated from an XML Schema and includes all mandatory and optional elements and attributes on all possible paths. It also includes elements for *all* the alternatives defined by `xs:choice` constructs. Thus, the master document is not necessarily valid for the schema it was generated from. Existing XML tools (e.g. Stylus Studio) can generate such a master document from XML Schemas. If an XML Schema does not exist, a set of actual instance documents is used as input.

We developed SQL/XML statements to populate both mapping tables from a master document or instance documents. This avoids the implementation of custom code and uses existing XML database capabilities. To populate the namespace mapping (Table 2), an SQL/XML statement with the expression `//(*, @*)` visits each element and attribute in the XML data. The XQuery function `namespace-uri()` obtains each node's namespace.

Populating the main mapping table is done by a recursive SQL/XML statement. It traverses the XML document(s), collects information for each node, and recursively passes it to the next level of the tree. This produces all existing paths from the root to the leaves. Due to space limits, Figure 3 only shows a reduced skeleton of the actual query but still conveys key concepts, e.g. the common table expression for the recursive tree traversal.

The first SELECT block in Figure 3 retrieves a document's root node together with its name and path. The second leg of the UNION ALL takes the already existing rows in the view *pathstable* as input. A document's root element has been added to the *pathstable* in the first step, now all its child nodes are added in the same manner. Their paths are composed of the parents' paths and the node's local name. All newly added nodes are then processed recursively, until no new child nodes are found.

The query in Figure 3 does not produce all necessary information for the mapping table. Additional metadata collection must be added to this skeleton. E.g., a node's level in the tree is a column with a counter that starts at 1 for the root and is incremented for each level of the recursion. Also, a logical data item name is proposed for each node by appending its name to its parent's name.

To determine a suitable SQL type that an element or attribute value can be cast to, the XQuery expression `castable` tests whether a node's value can be cast to `xs:integer`, `xs:date`, `xs:double`, and so on. An SQL data type is then assigned accordingly. This is reasonable although corresponding data types in XML and SQL do not always have identical value spaces. We see that XML data in most real-world applications contain values that are well within the value spaces of SQL data types. Also, the SQL/XML function XMLTABLE is based on the exact same assumption.

```
WITH pathstable (name, xmlnode, xpath) AS (
    SELECT  x.name AS name,
            x.xmlnode AS xmlnode,
            '/' || x.name AS xpath
    FROM mytable,
        XMLTABLE('$XMLDOC/*'
        COLUMNS
        name varchar(100)  PATH './local-name()',
        xmlnode XML        PATH '.') AS x
    UNION ALL
    SELECT  y.name AS name,
            y.xmlnode AS xmlnode,
            xpath|| '/' || y.name AS xpath
    FROM pathstable,
        XMLTABLE('$XMLNODE/(*,@*)'
        PASSING pathstable.xmlnode AS "XMLNODE"
        COLUMNS
        name varchar(100)  PATH 'local-name()',
        xmlnode XML        PATH '.') AS y
) SELECT name, xpath FROM pathstable;
```

Figure 3 Generating Mapping Information

To check whether multiple nodes with the same path occur in a document, an extra group-by clause aggregates and counts nodes that have the same name and parent within a document. Note that the elements `Price`, `Qty` and `Product` occur multiple times per document. They do not repeat within their parent, but the parent `Lineitem` repeats. This information is pushed down to all leaf nodes to assign the correct *repeat level* to each node, i.e. either its parent's repeat level or its own repeat level, whichever is higher.

When all these features are added to the query skeleton in Figure 3, the final statement is 1.5 pages long and available upon request.

This automated process produces the final mapping table, unless an administrator with domain knowledge decides to make adjustments. For example, the administrator can decide that the nodes `/ns3:Order/ns3:Lineitem/ns3:Qty` have the same meaning as `/ns3:Order/ns3:Product/@Qty` and give them the same logical data item name, as in Table 1. She can also define logical names other than the ones generated by the automated process. Information about the relational columns can be added to the mapping table by querying the database catalog. This leads to physical replication of catalog information in our prototype, but can be avoided through the use of views. When XML Schemas evolve or documents for new schemas are added, the mapping tables can be updated incrementally or regenerated.

4. QUERY GENERATION

Our current prototype supports a subset of the SQL language as input, where queries can include the following:

- "select", "from", "where", "group by", and "order by" clauses
- conjunction and disjunction of predicates
- aggregation functions in the select clause
- joins between XML columns, between relational columns, and between relational and XML columns
- SQL parameter markers

```
SELECT customerID, name,    | SELECT  cid as customerID, name, country, status
       country, status      | FROM customer,
FROM customer               |      XMLTABLE(XMLNAMESPACES('http://mycompany.org/customer' AS "ns1",
                            |                         'http://mycompany.org/address' AS "ns2"),
                            |               '$CDOC/ns1:Customer'
                            |      COLUMNS  name    varchar(30) PATH 'ns1:Name',
                            |               country varchar(30) PATH 'ns2:Addr/@Country',
                            |               status  integer     PATH '(@Status, ns1:Status)'
                            |             ) AS customerXML
WHERE name = 'John Smith'   | WHERE XMLEXISTS('declare namespace ns1="http://mycompany.org/customer";
                            |                  $CDOC/ns1:Customer/ns1:Name[. = "John Smith"]')
```

Example 2: Original SQL query (left) and generated SQL/XML query (right)

Our future work aims to extend the prototype to also accept "having" clauses, nested sub-selects, SQL "case" expressions, grouping sets, OLAP functions such as rank(), and arbitrary nesting of "and", "or", "not", "any" and "in".

The SQL/XML generation process takes the following steps:

1. Parse and analyze the original query (see section 4.1)
2. Generate XMLTABLE functions (section 4.2)
   a. Perform look-ups in the mapping table
   b. Build the row-generating expression (section 4.2.1), based on the deepest repeat level (section 4.2.2)
   c. Build XMLTABLE column definitions (section 4.2.3)
3. Generate XMLEXISTS predicates (section 4.3)
4. Compose the new query from parts of the original query and the generated SQL/XML functions and predicates (simple)

4.1 Query Parsing

When the original query is parsed, select, from, where, group-by, and order-by clauses are separated from one another. All logical data items referenced in the select, group-by, and order-by clauses are divided into two groups, i.e., those that map to relational columns and those that map to XML columns. The ones that reference relational columns are replaced by the actual physical column names listed in the mapping table. In the select clause, physical column names are assigned their logical name as an alias. All logical data items that refer to XML elements or attributes are grouped by the corresponding XML columns. The logical items that map to the *same* XML column are produced by *one* XMLTABLE function, where each logical item is represented by one column definition. In this way, our algorithm *assigns a set of logical data items to each XMLTABLE function*. This includes logical data items referenced in the order-by or group-by clauses of the SQL query, even if they are not listed in the select clause.

The query in Example 2 returns the ID, name, country and status for the customer `John Smith`. The logical data item *customerID* maps to a relational column and is replaced by the actual column name *cid*. The other logical data items reference XML elements or attributes. Since they are all located in the XML column *cdoc*, our algorithm assigns them to one XMLTABLE function.

For the generated SQL/XML query, the original from clause is augmented by the generated XMLTABLE functions. In the where clause, all predicates on logical data items that reference relational columns are moved to the generated query unchanged. Predicates that involve at least one logical data item that maps to an XML column are translated to XMLEXISTS predicates. In Example 2, the SQL predicate `name = 'John Smith'` uses the logical data item *name* which maps to an XML element. The algorithm turns this into a corresponding XMLEXISTS predicate. (See section 5 for considerations related to the existential semantics of the XQuery general comparison operators.) If a logical data item that maps to an XML element or attribute is used in multiple predicates, then those are combined in a single XMLEXISTS predicate.

4.2 XMLTABLE Functions

The query parsing has determined the number of required XMLTABLE functions (one per XML column that contains referenced XML data) and has assigned a set of logical data items to each. For every logical data item assigned to an XMLTABLE function, step 2a of the generation process retrieves their paths and metadata from the mapping and namespace tables. All namespaces that occur in any of the retrieved path expressions for one XMLTABLE function are combined in an XMLNAMESPACES function. Then the row-generating expression and column definitions are built.

4.2.1 Row-Generating Expression

The row-generating expression defines the context for all column definitions. The XMLTABLE function returns one row for each node found on the row-generating path, i.e., one or multiple rows per input document. We differentiate two cases, referred to as the *simple* and the *nested* case. The *simple case* takes place if each XML element or attribute returned by the XMLTABLE function occurs at most once per document. In this case, each XML document contributes at most 1 row to the result set. In this simple case, the row generating expression is determined as follows. For each logical data item that was previously assigned to the XMLTABLE function, the corresponding path is obtained from the mapping table (step 2a). Then the longest common prefix of these paths is chosen as the row-generating expression and is the context for all relative paths in the COLUMNS clause.

Example 2 clarifies how the row-generating expression is determined in the simple case. The SQL query uses the logical data items *customerID*, *name*, *country* and *status*. Since *customerID* refers to a relational column, it is not assigned to the XMLTABLE function. The three other data items occur at most once per document (simple case). The paths for these logical items are:

- `$CDOC/ns1:Customer/ns1:Name`
- `$CDOC/ns1:Customer/ns2:Addr/@Country`
- `$CDOC/ns1:Customer/ns1:Status`
- `$CDOC/ns1:Customer/@Status`

The longest common prefix is `$CDOC/ns1:Customer` and used as the row-generating expression in the XMLTABLE function.

The *nested case* occurs if at least one logical data item assigned to the XMLTABLE function can appear multiple times per document. In this case, the row-generating expression must generate multiple rows per input document. Repeating items are identified in the mapping table by a repeat level greater than 0. The paths of all repeating items that are assigned to the

XMLTABLE function are candidates for the row-generating expression. Among them we select the one with the deepest repeat level (see Section 4.2.2). If multiple items repeat on the same level, the one that occurs first in the mapping table is picked.

Consider Example 1 in Section 2. The logical data items assigned to the XMLTABLE function are *name*, *country* and *phone,* where *phone* is the only one that occurs multiple times per document. Its path `/c:Customer/c:Phone` repeats on level 2 whereas *name* and *country* are listed with repeat level 0. The path of the logical data item *phone* is therefore the row-generating expression.

The *nested case* also applies to Example 3. The logical data items assigned to the XMLTABLE function are *orderID*, *product* and *qty*. While *orderID* occurs at most once per document, *product* and *qty* have repeat level 2. Since *product* is listed before *qty* in the mapping table, it is selected for the row-generating expression. The logical item *product* is mapped to two paths and the row-generating expression combines both paths using the XQuery comma operator. Thus, multiple relative paths are also used in the column expressions for *orderID* and *qty* to produce the correct column values. This relies on our assumption that different paths for the same *logical* data item do not occur in the same document.

4.2.2 Deepest Repeat Level

In this sub-section we shed more light on using the deepest repeat level to select the row-generating expression. We describe cases where this method works and cases where it can lead to errors, and we argue that it is nevertheless a reasonable approach.

We select the logical data item with the greatest repeat level for the row-generating expression because we assume that (a) it is the one that generates the most rows per input document, and (b) that the remaining items assigned to the XMLTABLE function resolve to singletons when relative paths are built for the COLUMNS clause. Assumption (b) is trivially true for all elements and attributes that occur at most once per document. This is shown in Example 1 where the path `/c:Customer/c:Phone` is the row-generating expression. Using this path as context, the relative paths `../c:Name` and `../a:Addr/@Country` never produce more than one item since *name* and *country* have repeat level 0. Otherwise, the cast to the SQL type varchar(30) would fail.

Whenever *multiple* logical data items with repeat levels greater than 0 are assigned to the same XMLTABLE function, it must be ensured that their column expressions always resolve to singletons. Selecting the logical data item with the greatest repeat level to build the row-generating expression is sufficient in most cases. Example 3 illustrates this observation. The logical data item *product* is selected for the row-generating expression. The path in the column definition for the item *qty*, which has the same repeat level as *product*, resolves to a singleton value although it also occurs multiple times per document. This is because there is at most one `Qty` element for each `Product` element.

Building the row-generating expression from the item that repeats at the deepest level does not guarantee singletons for all column expressions in the XMLTABLE function. One example is a query that tries to return the logical data items *phone* and *email* from the customer data. Both items have the same repeat level. No matter which one is chosen for the row-generation expression, the other one does not resolve to a singleton in its column definition. The reason is that the document structure does not define a relationship between individual phone numbers and email addresses. Both elements repeat *independently* from each other and may occur unequally often. Hence, retrieving phones and email addresses in two columns is not well-defined. A full Cartesian product between all phones and all e-mail addresses typically does not make sense. We therefore use the mapping information (paths and repeat levels) to detect and reject such queries with an explanatory message to the user. Thus, using the deepest repeat level to select the row-generating expression is still reasonable and safe.

4.2.3 Column Definitions

The next step in building an XMLTABLE function generates a column definition for each assigned logical data item. Each column definition consists of a column name, a SQL type, and an XQuery expression. The column name is always that of the corresponding logical data item. Together with the SQL type, this name is obtained from the mapping table.

The XQuery expression in each column definition depends on two logical data items: (1) the logical item for which values are produced in this column, and (2) the logical item that contributed the row-generating expression, which provides the context for the column expressions. For *both* logical data items, two *properties* must be taken into account: (a) whether the logical item is mapped to one or multiple paths in the mapping table, and (b) whether the item occurs once or multiple times per document (i.e. repeat level 0 or >0). Table 3 shows all possible combinations for property (a). Both items, (1) and (2), may be mapped to one or multiple paths. This leads to four categories, identified by roman numerals I to IV in Table 3.Table 4 shows the same matrix for property (b). Both items can occur either once per document (repeat level 0) or multiple times (repeat level >0). The four categories are numbered V to VIII (where VII never occurs because our algorithm always selects the logical item with the greatest repeat level for the row-generating expression). Because the two properties (a) and (b) are orthogonal, each category in Table 3 can occur together with each category in Table 4. This leads to 16 combinations. Some of them can be treated similarly; some do not occur at all.

Table 3: Property (a) - No. of paths mapped to a logical item

		(2) Row-generating expression	
		One path	Multiple paths
(1) Column definition	One path	I 1 path	II list of paths
	Multiple paths	III list of paths	IV list of paths

```
SELECT orderID,
       product, qty
FROM order
```

```
SELECT orderID, product, qty
FROM order,
    XMLTABLE(XMLNAMESPACES('http://mycompany.org/ord' AS "ns3"),
'($ODOC/ns3:Order/ns3:Lineitem/ns3:Product, $ODOC/ns3:Order/ns3:Product)'
    COLUMNS  orderID  integer     PATH '(../../@Oid, ../@Oid)',
             product  varchar(50) PATH '.',
             qty      double      PATH '(@Qty,../ns3:Qty)' ) AS orderXML;
```

Example 3: Combining multiple paths in the row-generation expression and column expressions as needed

Table 4: Property (b) - Logical data items' repeat levels

		(2) Row-generating expression	
		0	> 0
(1) Column definition	0	V simple case	VI nested case
	> 0	~~VII (cannot occur)~~	VIII nested case

The characteristics of an XMLTABLE function (simple vs. nested) and its column definitions can be deduced after identifying the applicable categories in Tables 3 and 4.

When the algorithm detects that a column definition is in category V, a simple case XMLTABLE function must be constructed. This implies that the row-generating expression is the common prefix of all logical items assigned to the XMLTABLE function. Hence, the relative path expression for the column definition is produced by removing this common prefix from the *absolute* path for the logical data item in the column. To illustrate, consider the column definition for the logical data item *name* in Example 2:

```
name   varchar(30)  PATH  'ns1:Name'
```

The XPath expression `ns1:Name` is built by removing the row-generating expression `$CODC/ns1:Customer` from the element's absolute path. (This is the combination (I,V) in Tables 3 and 4.)

If multiple paths are associated with a column's logical item, all relative paths are combined. The column definition for the logical item s*tatus* in Example 2 falls into the combination (III, V). Category III implies that all relative paths for this logical data item are combined with the XQuery comma operator. At the same time, category V means that the relative paths are produced by removing the row-generating expression from each absolute path for this logical data item. Thus, the absolute paths `/ns1:Customer/@Status` and `/ns1:Customer/ns1:Status` are combined into the single column expression `(ns1:Status, @Status)`. The column definitions for other simple case XMLTABLE functions (II, V) and (IV, V) are built identically.

For nested case XMLTABLE functions, column expressions are built differently. First, the logical data item that is used for the row-generating expression has the column expression `'.'`. For all other items, the common prefix of the row-generating expression and the item's absolute path is removed from this absolute path. This produces a relative path. Then we count the number of child steps in the row-generating expression *after* the common prefix. The same number of parent steps is then added to the beginning of the relative path for the column definition. Consider Example 1: the row-generating expression is `$CDOC/c:Customer/c:Phone` and the absolute path for the logical data item *name* is `$CDOC/c:Customer/c:Name`. The common prefix `$CDOC/c:Customer/` is removed from this absolute path, leaving just `c:Name`. The row-generating expression has one child step *after* the common prefix, so we add one parent step to the column definition. Thus, the final column definition and the relative path from `$CDOC/c:Customer/c:Phone` to `c:Name` is:

```
name   varchar(30)  PATH  '../ns1:Name'
```

This column definition belongs to combination (I,VI). Combination (I,VIII) is similar in that only one relative path is used in the column expression. To avoid a cast error with the SQL data type, this relative path must produce a singleton. This is true if the original absolute path up to and including its repeat level is a prefix of the row-generating expression. In this case any repeating elements were removed as part of the common prefix and the remaining relative path has no repeating elements. Otherwise, the algorithm detects that the column item repeats *independently* from the row-generating item and rejects the query with a message that explains why the query cannot produce a meaningful result set.

The XQuery expression in a column definition consists either of a single relative path (category I in Table 3), or of multiple relative paths which are combined with the XQuery comma operator (categories II, III, IV). In the latter cases, a column definition has multiple relative paths to navigate from each path in the row-generating expression to each XML element or attribute that is associated with the logical data item in the column definition.

This is illustrated in Example 3, where the row-generating expression contains two repeating paths, which are:

- `/ns3:Order/ns1:Lineitem/ns3:Product` (p1)
- `/ns3:Order/ns3:Product` (p2)

The logical item *qty* itself also maps to two paths that repeat:

- `/ns3:Order/ns3:Lineitem/ns3:Qty` (q1)
- `/ns3:Order/ns3:Product/@Qty` (q2)

The relative paths that are required in the column expression for *qty* are `../ns3:Qty` to navigate from p1 to q1, as well as `@Qty` to navigate from p2 to q2. The paths p1 and q1 belong to the first type of order documents; p2 and q2 to the second type of orders. Hence, relative paths from p1 to q2 and p2 to q1 are not needed. The algorithm excludes those paths when it detects that p2 and q1 as well as p1 and p2 repeat *independently*, i.e. the repeat level of q1 is 2 but the first two steps of q1 are not a prefix of p2.

4.3 XMLEXISTS Predicates

XMLEXISTS predicates are built from the original predicates for logical data items that reference XML columns. Predicates on the same logical data items are grouped together in the parsing phase. These predicates are combined in one XMLEXISTS predicate. If both sides of a predicate reference XML columns, such as a join, the left-hand side decides which XMLEXISTS the predicate belongs to. Thus, all predicates that are combined in one XMLEXISTS have the same left-hand side.

The first step in generating an XMLEXISTS expression is to parse the paths for all logical data items in the predicates. Namespace prefixes are looked up in the namespace table and namespace declarations are added to the XMLEXISTS predicate as needed.

The context for the XPath predicate(s) within an XMLEXISTS is the path for the logical data item that is the common left-hand side for all predicates in the XMLEXISTS. If multiple paths are associated with the left-hand side, their common prefix is used as context. These cases are shown in Example 2, where the full path `$CDOC/ns1:Customer/ns1:Name` is used as context, and in Example 1 where only the common prefix of the associated paths, `$CDOC/ns1:Customer`, is used as the context.

The next step is to process all right-hand sides. Right-hand sides in the original query can include logical data items, constants, and parameter markers. This determines how a right-hand side is added to the predicate. Constant values can be used as right-hand sides with no or minimal further processing. The format of date and time may need conversion, depending on the formats and

```
SELECT name, orderID
   SUM(qty * price)
   AS amount
FROM customer, order







WHERE
  customerID =
  customerRef
GROUP BY name, orderID
```

```
SELECT name, orderID, SUM(qty * price)AS amount
FROM customer, order,
    XMLTABLE(XMLNAMESPACES('http://mycompany.org/ord' AS "ns3"),
    '($ODOC/ns3:Order/ns3:Lineitem/ns3:Qty, $ODOC/ns3:Order/ns3:Product/@Qty)'
    COLUMNS   orderID integer     PATH '../../@Oid',
              price   double      PATH '(../@Price, ../ns3:Price)',
              qty     double      PATH '.')  AS orderXML,
    XMLTABLE(XMLNAMESPACES('http://mycompany.org/customer' AS "ns1"),
              '$CDOC/ns1:Customer/ns1:Name'
         COLUMNS name    varchar(30) PATH '.') AS customerXML
WHERE XMLEXISTS('declare namespace ns3="http://mycompany.org/ord";
                 $ODOC/ns3:Order/ns3:CustomerRef[. = $CID]')
GROUP BY name, orderID
```

Example 4: XML-to-relational join with aggregation and grouping

database system used. SQL parameter markers are converted to generated variable names and a `passing` clause is added:

```
XMLEXISTS('$ODOC/ns3:Order/ns3:CustomerRef[. =
      $id]'  passing cast (? as integer) as "id")
```

If a right-hand side is a logical data item that refers to an XML column, the associated path is added. If the item maps to multiple paths, all are added as disjunctions. Logical data items referencing relational columns are added by using the column name as an XQuery variable, such as `$CID` in Example 4. In DB2 this variable is implicitly bound to the column value. Alternatively, a `passing` clause can be generated. Example 4 also illustrates how joins between XML and relational columns are generated.

Joins between XML columns are generated similarly. Consider the `products` table which stores detailed information for each product (Figure 4). Example 5 joins the XML columns of the `order` and `product` tables. For each product whose weight is greater than 5, the query retrieves the orderID and product names. The join condition is placed in the row-generating expression because it applies to the logical data item *product* whose repeat level is >0. Placing the join condition in the XMLEXISTS predicate would cause the same problem as in Example 6 (see section 5). If the left-hand side of a predicate is associated with multiple paths, all of them are added to the XMLEXISTS and each path is combined with all right-hand sides and added as a disjunction.

```
create table products(pdoc XML)

<Product xmlns="http://mycompany.org/product">
  <Name>Printer</Name>
  <Weight>15</Weight>
    ...
</Product>
```

Figure 4 Product Table and Sample Document

5. Differences in Language Semantics

The translation of SQL to SQL/XML with embedded XQuery expressions must be aware of semantic differences between the languages and their underlying data models. Some of the challenges are described in [5]. We don't claim that we formally overcome all semantic differences. Instead we have taken a pragmatic approach to address semantic differences so that the implemented solution fits practical use cases that meet our assumptions. For example, the difference in data types is handled by the SQL/XML standard and its casting rules from XQuery types to SQL types in XMLTABLE functions. No additional measures are applied. Another aspect is that XML and XQuery are order preserving while SQL and the relational data model are unordered. We target users and applications that come from the data-centric relational world. They use the SQL order-by clause explicitly if ordering is desired.

One semantic difference to overcome is that of comparison operators. General comparisons in XQuery (=, <, >, etc.) operate on sequences and have existential semantics, i.e. $i = $j is true if there is at least one item in $i that is equal to at least one item in $j. In contrast, comparisons in SQL operate on atomic values. This difference can lead to unexpected results when SQL equality predicates "=" are translated into the XQuery general comparisons "=". This is illustrated in Example 6 where both queries are syntactically correct but the SQL/XML query returns a result (Table 5) that does not match the semantics of the original SQL query.

The XMLEXISTS predicate in Example 6 selects every document that has a `Phone` element with the value `123-456-7890`. For each such document, the XMLTABLE function produces rows for *all* `Phone` elements that occur, and not just the `Phone` element that fulfilled the XMLEXISTS predicate. An SQL user views this result as wrong. Using SQL as the original query language means that the result set should follow SQL semantics. To eliminate the unexpected result rows, our algorithm does *not* generate the SQL/XML query in Example 6. Instead, it places the predicate into the row generating expression of the XMLTABLE function: `'$CDOC/ns1:Customer/ns1:Phone[. = "123-456-7890"]'` This is done whenever there is a predicate on a logical data item with repeat level >0. With this expression, the new result set for Example 6 contains only the first result row in Table 5.

```
SELECT orderID, product
FROM order, products








WHERE product = pref AND
      weight > 5
```

```
SELECT orderID, product
FROM order, products,
    XMLTABLE(XMLNAMESPACES('http://mycompany.org/product' as "ns4",
                        'http://mycompany.org/ord' as "ns3"),
        '($ODOC/ns3:Order/ns3:Lineitem/ns3:Product,
          $ODOC/ns3:Order/ns3:Product)[. = ($PDOC/ns4:Product/ns4:Name)]'
            COLUMNS
             orderID      integer      PATH '(../../@Oid, ../@Oid)',
             product      varchar(50)  PATH '.') AS orderXML
WHERE XMLEXISTS('declare namespace ns4="http://mycompany.org/product"
      ; $PDOC/ns4:Product/ns4:Weight[. > 5]')
```

Example 5: XML-to-XML Join – Join condition in the row-generating expression of the XMLTABLE function

```
SELECT name, phone
FROM customer




WHERE
phone = '123-456-7890'
```

```
SELECT name, phone
FROM customer,
    XMLTABLE(XMLNAMESPACES('http://mycompany.org/customer' AS "ns1"),
             '$CDOC/ns1:Customer/ns1:Phone'
        COLUMNS  name  varchar(30) PATH '../ns1:Name',
                 phone varchar(12) PATH '.')  AS customerXML
WHERE XMLEXISTS('declare namespace ns1="http://mycompany.org/customer";
                 $CDOC/ns1:Customer/ns1:Phone[. = "123-456-7890"]')
```

Example 6: Different Semantics – The result set of the SQL/XML query does not match the semantics of the SQL query

Table 5: Result Set from Example 6

Name	Phone
John Smith	123-456-7890
John Smith	123-555-6523

6. SQL/XML UPDATE STATEMENTS

Our algorithm generates SQL/XML update statements with the same concepts as for queries. XQuery transform expressions [14] are generated for updates of XML elements or attributes. The necessary XPath expressions are assembled from one or multiple paths in the mapping table. Predicates on XML elements or attributes are transformed depending on the repeat level. They are converted to XMLEXISTS predicates if the element or attribute occurs at most once per document (repeat level 0). A predicate on an element or attribute with repeat level > 0 means that only one of multiple occurrences of an element should be updated. In this case the predicate is *also* applied in the transform expression. A simple SQL update statement and the corresponding generated SQL/XML update statement are shown in Example 7.

```
UPDATE customer
SET name = 'Peter Smith', status = 3
WHERE customerID = 27;

UPDATE customer
SET cdoc = XMLQUERY('
    declare namespace ns1="http://.../customer";
    copy $new := $CDOC
    modify (
      for $i in $new/ns1:Customer/ns1:Name return
        do replace value of $i with "Peter Smith",
      for $i in $new/(ns1:Customer/@Status,
                  ns1:Customer/ns1:Status) return
        do replace value of $i with "3")
    return $new')
WHERE cid = 27;
```

Example 7: SQL and SQL/XML Update Statements

In Example 8 the logical data item *price* is updated, which occurs multiple times per document. The update is restricted by a predicate on the logical data item *product*, which also occurs multiple times. This predicate is converted to an XMLEXISTS predicate in the where clause of the SQL/XML update statement. It allows the database engine to use an XML index to efficiently locate the required order documents. The predicate is *also* applied in the XQuery transform expression, to update the price of the correct product *within* a given order.

7. SCHEMA DIVERSITY USE CASES

Our mapping and query translation method does not solve schema diversity and schema integration problems of arbitrary nature. However, we found that our method can effectively address several common cases of schema diversity and schema evolution that we have encountered in real XML applications. Industry-specific XML Schemas such as FpML, FIXML, HL7, ARTS, ACCORD, OAGIS, and STAR evolve with each new version that is released. Elements and attributes may be added, removed, or restructured. Some companies also add custom enhancements to these schemas to better fit their business needs. A common requirement that we encountered is to query across documents that belong to different versions of the same schema in one XML column. Our method supports this requirement. If SQL statements use logical data item names, the mapping table can be updated to reflect changed or additional schemas without having to change existing SQL statements. In particular, mapping a single logical item to multiple possible paths helps to address schema diversity.

One company that we worked with has subsidiaries that reside in different countries and use XML tag names in their local languages. Additionally, the document hierarchy has slight variations from country to country although the represented data has the same meaning. Queries and updates across documents from multiple countries were not easily possible. Converting local XML data to a common global XML Schema was not desirable because (a) converting large data volumes on an ongoing basis is costly; (b) maintaining the data conversion process is labor intensive when new countries (schemas) are added; and (c) any data updates based on a global schema would need to be reflected in the original local formats where legacy applications depend on it. The company had 22 critical queries and updates that had to run across XML messages from multiple subsidiaries. Our approach solved this scenario by correctly converting all 22 query and update statements - instead of converting data to a common global schema. The one-time labor to map local element names to global logical data items was considered a worthwhile investment to facilitate the generation of SQL/XML queries and updates.

```
UPDATE order
SET price = 199







WHERE
    product = 'Printer'
```

```
UPDATE order
SET odoc = XMLQUERY('declare namespace ns3="http://mycompany.org/ord";
    copy $new := $ODOC
    modify for $i in
        ($new/ns3:Order/ns3:Lineitem/ns3:Price[../ns3:Product = "Printer"],
         $new/ns3:Order/ns3:Product/@Price[.. = "Printer"]) return
      do replace value of $i with "199"
    return $new')
WHERE XMLEXISTS('declare namespace ns3=http://mycompany.org/ord;
                 $ODOC/ns3:Order[ns3:Lineitem/ns3:Product[. = "Printer"] or
                                 ns3:Product[. = "Printer"]]')
```

Example 8: Updating a repeating element (`price`), with a predicate on another repeating element (`product`)

8. RELATED WORK

There has been a lot of work on translating XQuery to SQL, but very few papers have been published that deal with the translation of SQL to XQuery. We are not aware of any work that focuses specifically on the generation of *SQL/XML* instead of *XQuery*. We are also not aware of previous work that generates SQL *update* statements with embedded *XQuery transform expressions*. Although the generation of basic path expressions can be similar for XQuery and SQL/XML, the construction of XMLTABLE functions and XMLEXISTS predicates is different from the generation of FLWOR expressions. For example, column expressions in XMLTABLE must produce singletons, which is a restriction that does not exist for general XQuery. SQL/XML queries can also includes relational predicates or join XML and relational columns, which is not directly possible in XQuery.

An SQL to XQuery translation algorithm to access XML data in a federated environment is presented in [8]. Based on a global relational schema, queries are written in SQL and translated to XQuery before execution on an XML data source. This postulates that a mapping from a global relational schema to a local XML schema exists. However, the paper does not provide any details on this mapping, the information it contains, or how to generate it. Our work describes the metadata necessary to translate SQL to SQL/XML, and presents a method to automatically build the mapping tables. Opposite to [8], our work also shows how to generate SQL/XML updates, XML joins, and queries on mixed XML/relational databases, including joins between XML and relational data. It is not obvious whether the same can be achieved with the XQuery generation in [8]. Our approach also addresses some challenges related to XML Schema variability or evolution. Documents for different or evolving schemas can be integrated in one table and queried transparently. This is not addressed in [8].

In [2], SQL is converted to XQuery for the special case where relational data is exposed to the web using XML as an intermediate format. This approach generates XML documents from flat, relational data. These documents are evenly structured. The SQL to XQuery translation in [2] enables applications to access the relational data as if there was no intermediate XML format. The proposed translation is based on the fixed and known XML document format. Unlike our approach, it is not suited to generate queries for *arbitrarily* structured XML documents.

The algorithm shown in [9] has a similar restriction. It uses a translation from SQL to XQuery to provide applications access to data sources in an integrated Data Service Platform. XQuery functions are wrapped around data services and produce *flat* XML structures that resemble relational tables. This flat XML can be queried with SQL through an SQL to XQuery conversion. Similar to [2], the query generation depends on the known flat structure of the XML data and it is not applicable to arbitrarily nested XML.

The ROX experiments in [3] provide SQL access to XML data by manually defining relational *views* over the XML data. Each repeating element and its non-repeating children are mapped to a separate view, and users write SQL join queries against these views. This approach has the same problem as the traditional shredding of XML data, i.e. it often takes dozens and sometimes hundreds of relational views (or tables) to represent the structure of real-world XML data, such as FpML. This leads to multi-way join queries which are hard to write and to execute efficiently.

9. SUMMARY AND OUTLOOK

Writing correct SQL/XML queries and updates requires application developers to (a) be familiar with SQL/XML, XQuery, and XQuery Updates, (b) have detailed knowledge of the tree structure of the XML documents in the database, and (c) handle schema diversity/evolution when needed. To address these challenges, we have prototyped a method that translates SQL to SQL/XML statements. The translation uses a mapping table that assigns logical data item names to the location of XML elements and attributes in XML documents. Most of the mapping information is generated automatically, a critical prerequisite for the applicability of our method. The query translation enables relational users to perform queries and updates on XML data while mitigating the aforementioned challenges. The mapping layer can provide some resilience against schema changes and eases the integration of diverse XML structures. An administrator with domain knowledge may have to modify selected entries in the mapping table, e.g. if multiple elements with distinct names represent the same logical data item. An area for future work is to minimize this manual labor, reusing work in the area of schema integration.

We have implemented the translation algorithm in Java as a prototype on DB2 and have successfully tested it in real XML application scenarios. It does not yet support all possible SQL queries as input, which is subject for future work along with the integration of our method with object-relational mapping frameworks.

REFERENCES

[1] Eisenberg, A. and Melton, J. "Advancements in SQL/XML", *ACM SIGMOD Record, 33* (2), 79-86, 2004

[2] Escobar, F.J.C. et al. "XML Information Retrieval Using SQL2Xquery", *Tecnologico de Monterrey – Campus cd. De Mexico. Departmento do Computacin*, 2002.

[3] Halverson, A. et al. "ROX: Relational over XML". *Internat. Conference on Very Large Data Bases*, 264-275, 2004.

[4] IBM, DB2 pureXML at Douglas Holding AG, http://www.ibm.com/software/success/cssdb.nsf/CS/LWIS-6XCUYX, Jan 2007.

[5] IBM, DB2 pureXML at New York State Tax, ftp://ftp.software.ibm.com/common/ssi/pm/ab/n/imc14008usen/IMC14008USEN.PDF, Jan 2008.

[6] IBM, DB2 pureXML at UCLA Health System, http://www.ibm.com/software/success/cssdb.nsf/CS/LWIS-7PKLWW, Feb 2009.

[7] IBM, DB2 pureXML Case Studies, http://www.ibm.com/developerworks/wikis/display/db2xml/DB2+pureXML+Case+Studies

[8] Jahnkuhn, H. et al. "Query Transformation of SQL into XQuery Within Federated Environments", *QLQP 2006*, EDBT Workshop, *LNCS 4254*.

[9] Jigyasu, S. et al. "SQL to XQuery Translation in the AquaLogic Data Service Platform", *ICDE 2006*.

[10] Loeser, H., Nicola, M. and Fitzgerald, J. "Index Challenges in Native XML Database Systems ", *BTW 2009*.

[11] Murthy, R. et al. "Towards an enterprise XML architecture", *SIGMOD Conference*, p. 953-957, 2005.

[12] Nicola, M. and Kumar-Chatterjee, P. "DB2 pureXML Cookbook", *IBM Press*, ISBN 0138150478, 2009.

[13] Rys, M. "XML and Relational Database Management Systems: Inside Microsoft SQL Server", *SIGMOD 2005*.

[14] The XQuery Update Facility, http://www.w3.org/TR/xquery-update-10/