XML programming with SQL/XML and XQuery

by J. E. Funderburk
S. Malaika
B. Reinwald

Most business data are stored in relational database systems, and SQL (Structured Query Language) is used for data retrieval and manipulation. With XML (Extensible Markup Language) rapidly becoming the *de facto* standard for retrieving and exchanging data, new functionality is expected from traditional databases. Existing SQL applications will evolve to retrieve relational data as XML data using database or SQL extensions for XML. New XML data will be stored, searched, and manipulated in the database as a "first class" citizen along with existing relational data. Furthermore, new applications will emerge that solely operate in terms of XML. These new XML applications operate on the same database using an XML query language, XQuery. In this paper, we describe an integrated database architecture that enables SQL applications with XML extensions as well as XQuery applications to operate on the same data. The architecture allows for a seamless flow from relational data to XML and back.

The recent expansion of the Internet and invention of the XML (Extensible Markup Language) data format has created both the opportunity and the need for businesses to exchange information, and to interoperate in a uniform way that has not been achieved nor been possible before except in isolated segments within the business community. Today's economic practices often create businesses and business processes formed from many incompatible systems. The need to integrate and exchange data within a company is as profound as it is between companies. Much of the data being exchanged are operational: data that enable transactions, determine the course of business processes, and in the aggregate, become business intelligence data that affect decisions of business leaders.

The XML data format provides a way of regularizing the storage of semi-structured data, historical data, and other information requiring content management. XML can be used to store the content itself and data mined from the content. Data mined from the content can be used to form catalogs, similar in concept to card catalogs in libraries, which contain existence and location information and possibly other interesting summary information. Information mined from content can be stored and used in business or scientific intelligence queries.

Information that is low in quantity and importance can be stored using a variety of simple techniques. However, business-critical data and data in large quantities require a data storage system that can properly manage the data. Relational database managers fulfill vital responsibilities in complex information systems by consolidating storage and distribution of data. They provide a uniform, high-function interface and support other features such as security, data consistency, control, and the regularization and automation of backup procedures. These re-

0018-8670/02/$5.00 © 2002 IBM

sponsibilities have been established and relied upon for many years; most systems are already in place and are entrenched. It is vital that existing data stores and business processes be extended to support XML technologies in a variety of ways. Systems integrators need to interact with existing data, implementing new business processes using XML without the need to write and continually rewrite low-level code to interact between XML data and the interface technology supported by databases.

The most basic requirement is that data already stored in relational database systems must be publishable as XML. Subsets of the data (or, in general, the results of queries) must be able to be formed into XML documents and sent to applications or consumers elsewhere. Usually, XML documents must match a particular XML format or schema. Such formats are specified by DTD (Document Type Definition), XML Schema, or by documentation. Existing data must be transformed into forms that comply with the standard of interchange. These forms may or may not be under the system integrator's control and often evolve over time. The ability to adapt easily is required.

XML programming model evolution

SAX (the simple API [application programming interface] for XML)[1] was the first popular interface for XML programming. A SAX application was a set of event handlers, each called when the parser encountered an element or some text in the document. Without any control over its execution, one had to build XML applications as state machines in order to do any nontrivial work. SAX became a *de facto* standard, with the Java** language being the official binding of the API. Non-Java language bindings cropped up, but they were particular to a specific implementation of a SAX parser.

DOM (Document Object Model)[2] followed and provided a navigational interface that standard programming languages could use. A DOM application was in control of its execution flow. This made the application a normal program that could be organized using familiar approaches. Navigation is performed with individual steps using DOM function calls (such as getFirstChild( ) and getNextSibling( )). Processing a document while validating its semantics or syntax (supplementing a DTD) often required a rather verbose and somewhat fragile program. Evolution of the structure of the input document, although a problem under any circumstances, required a detailed adjustment of individual navigation steps and loops written in a mix of programming language statements and DOM interface calls. Without standard language bindings, DOM applications were, by default, trivially dependent on a particular parser. Traditionally, DOM-capable XML parsers have to read the entire XML document into memory.

XSLT (Extensible Stylesheet Language Transformations)[3] 1.0 arrived, was the first recognized XML programming language, and was substantially complete. An XSLT program transforms its input document into an output document using a programming construct called *templates*. The input document is read into memory, and traversed. At each point, template-matching patterns are tested. When a match is found, the template body is executed and a portion of the output document is formed. A sublanguage called *XPath* is used to form template-matching patterns and to navigate within the input document tree, selecting the desired element and attribute data to use in expressions to form output or conditional logic. XPath allows several navigations to be performed in succession and various predicates can be applied at each step. XSLT processes the input document without type information and therefore processes data in the document as text. It is also possible to explicitly form expressions that perform basic operations on double floating-point numbers.

Because XSLT uses optimistic recursion when trying to find matching templates, it is the responsibility of the programmer to make sure that templates will only match where they are intended. Longer patterns are more restrictive than shorter ones, but programs are often written with short patterns. XSLT determines which template matches based on whether the pattern matches, the selectivity of the pattern, the mode of the template, and the set of nodes that were selected for matching. Templates can be imported from various sources. Adding templates to an import file could affect the output of programs. In addition, imports can be nested, and imported templates have a precedence scheme. Programmers must expend effort to maintain control over the power of templates in nontrivial programs. Even so, XSLT processors give XML programmers freedom from the drudgery of SAX and DOM programming. XSLT programs work very well with document-oriented XML documents such as XHTML,[4] but for program-to-program, data-oriented operations, the potential for error and lack of data typing is a concern.

XML Schema[5] was developed to add data types to XML and to provide better document validation than DTDs.[6] XML Schema is also important because DTDs do not support XML Namespaces.[7] Because XML Namespaces are an integral part of the XSLT processing model, XSLT processors typically did not validate their input documents. XML Namespaces now play a part in any new XML effort. XML Schema or a suitable replacement is a necessity.

What is the next advance in XML programming? It is language.

SOAP (Simple Object Access Protocol)[8] is an XML messaging format. It defines a simple standard envelope for messages, which can be used in a wide variety of protocols. When combined with HTTP (HyperText Transfer Protocol), it is lightweight and accessible, eliminating the difficulties that must be faced when using other protocols.

At this time, we have XSLT processors running XSL stylesheets, and handwritten DOM and SAX applications. The overall processing model is one where one step processes a whole document and then dumps the output document out to a file or sends it over the wire to another step that processes the document as input. Integrators and Webmasters use a variety of small, single-function processors and APIs for interacting between XML and databases, messaging systems, and Web servers.

What is the next advance in XML programming? It is *language*. Language has been a factor in almost every advance in software and data management. Language attracts and draws in. The SQL (Structured Query Language) is largely responsible for the replacement of hierarchical and navigational databases by relational ones. Even in the face of greater efficiency and performance, on average, successful languages win over nonlanguage approaches. However, language is only useful within the context of its environment. Because of this, the next advance in XML programming will be XML-related languages combined with XML databases/data stores. Whereas single-document, file-oriented, processors are useful tools, an XML language is much more useful when combined with a database—just as SQL is. Even at a pragmatic level, combining the two affords better function, support, and performance. An XML-capable database manager can become a central highway in the next generation of XML programming. Capability will be measured in terms of language.

Using relational data in a world of hierarchical messages

The use or transmission of data in a system requires representation of the data values and the semantic relationships among the values. The modeling of semantic relationships varies most among systems. In addition to defining a way to model semantic relationships, most systems also require abiding by the cardinality rules for storing data and maintaining a semantic relationship.

In relational systems, data are separated according to cardinalities and according to logical dependencies (normalization). Separate tables are required to store a single, but nontrivial, piece of knowledge. Data in each table form regular records, called *tuples*, that contain the same number of fields. Data in each of the separate tables are correlated by column values that serve as keys. Key values may represent actual external data or be manufactured values solely for the purpose of correlating tuples. The organization of related data into tables is dependent on cardinalities in the following way.

1. Data that are one-to-one with respect to other data can be stored in the same tuple.
2. Data that are one-to-zero with respect to other data can be stored in the same tuple, with null representing not-present. The data can be stored in a separate table as well.
3. Data that are one-to-many with respect to other data must be stored in a separate table.
4. Data that are many-to-many with respect to other data must be stored in another table, and a correspondence table must be created to record the correspondence between the two tables by storing key pairs from the respective tables.

Simple relationships are maintained by storing data in the same tuple. Complex relationships are maintained by using separate relations and keys. All the data of a category (by cardinality or logical dependence) are stored in the same table. This places information together that is unrelated, but which has the same type and structure properties.

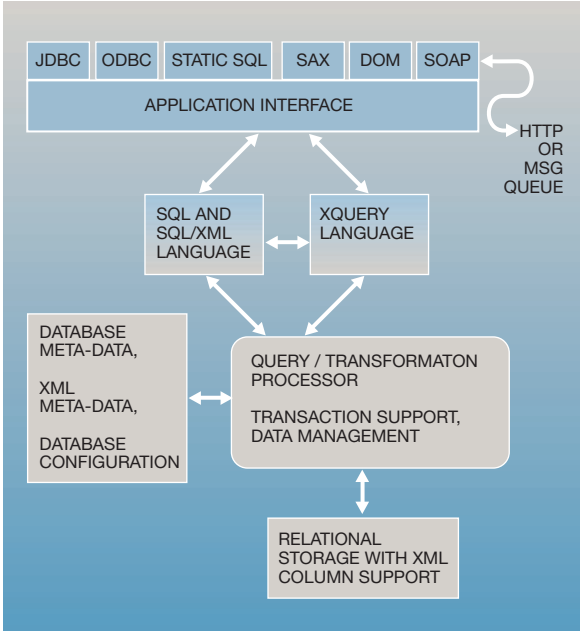
In the relational world, it is as easy to query all the authors a book has as it is to query all the books an author has written. The data objects and relationships between data are in a separate domain of metadata, and such objects and relationships must be understood and materialized by an application. The application does this by utilizing the correct table names and by creating the appropriate "join" expressions. Utilizing a set of tables at a server requires too much involvement with the names and data design for a particular server. A table or set of tables does not make a very appealing message format, because the data require too much correlation and interpretation by the recipient.

In the XML-centric world, the parent/child element and the element/attribute relationships are used to encode semantic relationships. A specific document must choose a specific hierarchy that models the relationships that are meaningful in the specific communication. Data cardinality also influences XML schema design. Here are the most common generalities for XML.

1. Data that are one-to-one with respect to other data can be stored in attributes of an element or descendant elements of the same element—either as element content or in attributes.
2. Data that are one-to-zero with respect to other data can be stored in the same way as the one-to-one case, but the element or attribute housing the data can be absent or the contents of the node can be empty.
3. Data that are one-to-many with respect to other data can be stored in repeating child elements of the element representing the other data.
4. Data that are one-to-many with respect to other data can be stored in child elements where data are duplicated. Data can also be stored in separate hierarchies that are cross-linked by identifiers, or some other application convention.

In the XML world, relationships are encoded within the document itself by using parent/child relationships and element/attribute relationships. Data that are related are adjacent. Data appear when the data are needed, and the structure varies accordingly. This makes an XML document more self-contained and requires less interpretation. The XML format does have its downside. An XML document originates in character form, is organized in a particular hierarchy that favors particular relationships, and may duplicate various facts. Because of this, the XML data format is not desirable as a method of storing data that need to be correlated with many other collections of data, in many relationships, or for storing data that are updated.

Figure 1 XML/relational database manager



Any system that provides many views of data in various relationships must be able to efficiently materialize results at run time. Typical queries form joins and other various predicates and projections that combine and form subsets from the data. Stored data that support these kinds of queries must be readily accessible, in a digested, typed format ready for comparison or computation. Such queries need the assistance of indexes to perform well.

Thus, often the right way to store data is relational but it is not the right way to encode messages. The data manager must enable the creation of any arbitrary number of specific mappings between relational data and XML, and make these mappings or transformations invokeable by applications. Figure 1 depicts the structure of a database manager that is designed to support both XML and relational requests. SQL or XQuery requests are accepted, and specific APIs can be chosen by the application to process the result.

SQL extensions for XML

In this section, we describe how traditional SQL applications can evolve to deal with XML data. Building on the relational data model with SQL and existing data access protocols, we introduce SQL extensions for XML to construct XML data from relational data, as well as store, query, and retrieve XML data. Some of the SQL extensions described here are provided by DB2* (Database 2*) XML Extender, some are proposed for the ANSI/ISO (American National Standards Institute/International Organization for Standardization) SQL standard (SQL/XML),[9,10] and others are discussed in workgroups or exist in prototype implementations. ANSI/ISO approved a project for a new part of SQL:200n, part 14, XML-Related Specifications (SQL/XML).

An informal group of companies, called SQLX (http://www.sqlx.org) including IBM, Microsoft, Oracle, and Sybase began to define XML extensions for SQL in early 2000. The group focuses on SQL capabilities and consciously avoids vendor extensions while encouraging state-of-the-art and projected future developments. SQLX forwards proposals to the INCITS (InterNational Committee for Information Technology Standards) H2 Database Committee for approval.

IBM's vendor extensions were incorporated in the latest version of DB2 XML Extender. XML Extender adds XML functionality to IBM DB2 databases through a set of user-defined types, user-defined functions, and stored procedures. This section is not intended to compare SQL/XML with XML Extender. A comparison would not be useful, because the current SQL/XML proposal only contains the first phase of extensions, and more extensions are in preparation. Both technologies are presented as partly overlapping approaches with similar objectives. The line between SQL/XML and XML Extender is kept fuzzy intentionally, with the projection that these technologies will merge in the course of time.

XML publishing functions. Relational data are the universal backbone of any business. With XML as a universal data exchange format, the capability of constructing XML data from existing relational data, while preserving the power of SQL, tremendously simplifies business-to-business (B2B) application development. In this subsection, we introduce a list of scalar and aggregate functions as SQL extensions:

- XMLElement and XMLAttributes: construct XML elements with attributes
- XMLForest: constructs a sequence of XML elements
- XMLConcat: concatenates XML elements
- XMLAgg: aggregates XML elements
- XMLGen: constructs XML according to an XML element constructor specification

XMLElement constructs a new XML element with attributes and content. The attribute names and values are specified in XMLAttributes through column names or aliases for value expressions. Element content is constructed from a variable list of value expressions. The result of the value expressions is mapped from SQL to XML according to the mapping rules specified in SQL/XML.[9] SQL/XML defines mapping rules to map SQL identifiers and XML identifiers, SQL data types and XML schema types, and SQL data and XML data on a value, table, schema, and catalog level. An example for mapping an SQL table to XML is discussed in the subsection, "Example XQuery view."

In a supply chain sample scenario, a company stores orders and orderItems in relational tables. A business partner posts through a secure Internet connection a query to retrieve all open orders:

```
SELECT XMLELEMENT (
            NAME "order",
            XMLATTRIBUTES (o.oid AS "id"),
            XMLELEMENT (
              NAME "signdate",
              o.contractdate
            ),
            XMLELEMENT (
              NAME "amount",
              (SELECT SUM(orderitem)
               FROM orderItems AS oi
               WHERE i.oid = o.oid)
            )
          )
FROM orders AS o
WHERE status = 'open';
```

The query returns a result set with two rows, with each row containing an XML value:

```
<order id="4711">
   <signdate>2002-03-18</signdate>
   <amount>24000</amount>
</order>

<order id="4712">
   <signdate>2002-03-19</signdate>
```

```
    <amount>44000</amount>
</order>
```

XMLForest simplifies queries as it constructs sequences of XML elements from SQL value expressions in the order of the expressions. XMLForest is a shorthand for a sequence of XMLElement invocations. XMLForest takes a variable list of SQL value expressions as input, and produces for each expression an XML Element with the column name or alias of the expression as the tag name, and the value of the expression as the element content.

```
SELECT XMLELEMENT (
            NAME "order",
            XMLFOREST (
                o.oid AS "id",
                o.name AS "name",
                o.city AS "city"
            )
        )
FROM orders AS o
WHERE status = 'open';
    <order>
        <id>4711</id>
        <name>steel company></name>
        <city>Hamburg</city>
    </order>

    <order>
        <id>4712</id>
        <name>beer company>/name>
        <city>Munich</city>
    </order>
```

XMLConcat takes a variable number of XML value expressions and constructs a single XML value as a sequence of XML values. This function is used to construct an XML element from pieces of independently constructed XML. XMLConcat is a scalar function. The following example produces the same output as the previous example.

```
SELECT XMLELEMENT (
            NAME "order",
            XMLCONCAT (
                XMLELEMENT(
                    NAME "id",
                    o.oid
                ),
                XMLELEMENT(
                    NAME "name",
                    o.name
                ),
                XMLELEMENT(
                    NAME "city",
                    o.city
                ),
            )
        )
FROM orders AS o
WHERE status = 'open';
```

XMLAgg is an aggregate function, which constructs an XML value from a collection of XML value expressions. XMLAgg resolves the 1:n relationships in XML. The following example retrieves information about an order including its orderItems.

```
SELECT XMLELEMENT(
            NAME "order",
            XMLATTRIBUTES(o.oid AS "id"),
            XMLAGG(
                XMLELEMENT(
                    NAME "item",
                    XMLATTRIBUTES(
                        oi.listnbr AS "listnbr"
                    ),
                    XMLFOREST(
                        oi.name AS "name",
                        oi.quantity AS "quantity"
                    )
                )
                ORDER BY oi.listnbr
            )
        )
FROM orders AS o, orderItems AS oi
WHERE o.oid = oi.oid
GROUP BY o.oid;
```

This query returns the result:

```
<order id="4711">
        <item listnbr="1">
                <name>bike</name>
                <quantity>10</quantity>
        </item>
        <item listnbr="2">
                <name>racket</name>
                <quantity>5</quantity>
        </item>
</order>
```

Mapping relational data to XML in DAD description. An alternative method for publishing XML documents from relational tables is through the DB2 XML Extender Document Access Definition (DAD), which itself is an XML document. The DB2 XML Extender[11]

is a component of DB2 that provides various forms of XML support.

Two distinct notations can be used in DADs to describe how to map from relational tables to XML.

- SQL Composition: A notation that incorporates an SQL SELECT statement, followed by instructions on how the resulting rows should be tagged as XML
- RDB (relational database) Node: A notation that includes a list of the tables whose contents are to be tagged as XML, together with the primary-foreign key relationships between the tables. As in SQL Composition, the list of tables is followed by instructions on how the contents (or more typically a subset of the contents) should be tagged as XML.

There are seven steps to consider when generating a DAD for publishing.

1. Scoping the document content
2. Shaping the document structure
3. Mapping the relational content to the document
4. Controlling the number of documents generated
5. Outputting document header information
6. Validating the generated documents
7. Transforming the generated documents (e.g., to HTML, HyperText Markup Language)

We describe in detail the steps required to produce the following XML documents from relational data:

```
<order id="4711">
  <signdate>2002-03-18</signdate>
  <amount>24000</amount>
</order>
<order id="4712">
  <signdate>2002-03-19</signdate>
  <amount>44000</amount>
</order>
```

Scoping the content of the generated documents. The first part of the SQL Composition DAD contains an SQL SELECT statement that retrieves all the rows whose content is required in the generated document. The SQL statement can include join operations, subselects and SQL functions. For example,

```
<SQL_stmt>
    SELECT
        o.oid AS id,
        o.contractdate AS cdate,
        SUM(oi.orderitem) AS total,
    FROM orders AS o, orderItems AS oi
    WHERE oi.oid = o.oid AND status = 'open'
    ORDER BY id;
</SQL_stmt>
```

The first part of the RDB_node contains a list of tables whose rows form the content of the generated document. The key relationships between the tables are listed in the scoping portion of the DAD. For example,

```
<RDB_node>
    <table name="orders" key="oid"/>
    <table name="orderItems" key="oid"/>
    <condition>
      orders.oid=orderItems.oid
    </condition>
</RDB_node>
```

Another condition can be added to restrict the documents generated to those that have an 'open' status by inserting the following check into the DAD: <condition>status='open'</condition>.

Shaping the structure of the generated documents. For both the SQL Composition DAD and RDB Node DAD, the shape of the output document is governed by the structural tag layout in the second part of the DAD. Multiple hierarchies can be generated in a single document, and the way elements repeat can be controlled. Following is an example for SQL Composition:

```
<root_node>
    <element_node name="order">
        <attribute_node name="id">
            <column name="id"/>
        </attribute_node>
        <element_node name="signdate">
            <text_node>
                <column name="cdate"/>
            </text_node>
        </element_node>
        <element_node name="amount">
            <text_node>
                <column name="total"/>
            </text_node>
        </element_node>
    </element_node>
</root_node>
```

Mapping the relational content to XML. For both the SQL Composition DAD and RDB Node DAD, the mapping is governed by instructions that appear alongside the structural tags in the second part of the DAD.

The tag layout shown in the subsection on shaping, above, represents the mapping information for the SQL Composition example.

Controlling the number of documents generated. For SQL Composition, the number of documents produced can be controlled by the SQL statement in the DAD. The number of documents produced is equal to the number of rows grouped by the first grouping expression. For RDB_node, the number of documents produced can be controlled by the options supplied on the root element in the DAD. Suppose we wanted to produce a single document as follows:

```
<orders>
    <order id="4711">
        <signdate>2002-03-18</signdate>
        <amount>24000</amount>
    </order>

    <order id="4712">
        <signdate>2002-03-19</signdate>
        <amount>44000</amount>
    </order>
</orders>
```

For RDB Node notation, we would use the multi_occurrence option in the DAD to cause a single document to be produced instead of two. Following is an example of how the option is specified:

```
<element_node name="orders">
    <element_node name="order"
          multi_occurrence="yes">
        <attribute_node name="id">
            <RDB_node>
                <table name="orders"/>
                <column name="oid"
                type="varchar(20)"/>
            </RDB_node>
        </attribute_node>
    </element_node>
</element_node>
```

Outputting document header information. Header information such as XML declarations, DTD references, and processing instructions can be generated through the statements in the DAD. Following are some examples:

```
<prolog>?xml version="1.0"?</prolog>
<doctype>
  !DOCTYPE Order SYSTEM "orders.dtd"
</doctype>
```

Validating the generated documents. For both SQL Composition and RDB Node, it is possible to validate the generated documents against an XML schema or a DTD. For XML document validation there are three options.

1. Use the validation option in the DAD as follows:

   ```
   <dtdid>order.dtd</dtdid>
   <validation>yes</validation>
   ```

   DB2 XML extender will perform the validation against a DTD stored in the file system, or stored in a special table called the DTD_REF table.

2. Use the dvalidate UDF (user-defined function) as follows:

   ```
   db2xml.dvalidate( doc, dtd )
   ```

3. For schema validation, the svalidate UDF is available as follows:

   ```
   db2xml.svalidate( doc, xmlschema )
   ```

Transforming the generated documents. It is possible to apply further transformations to the generated documents, for example to convert them to HTML. There are a number of ways to transform.

1. Place an XSL processing instruction in the header information.
2. Use the XML Extender-supplied XSLT UDF, for example XSLTransformToClob(xmldoc, stylesheet, parameters, validate). A CLOB is a character large object.

In this subsection, we have seen DAD fragments only. Following is a complete SQL Composition DAD:

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "dad.dtd">
<DAD>
    <validation>no</validation>
    <Xcollection>
        <SQL_stmt>
            SELECT
              o.oid AS id,
              o.contractdate AS cdate,
              SUM(oi.orderitem) AS total,
            FROM orders AS o, orderItems AS oi
            WHERE oi.oid = o.oid AND status = 'open'
            ORDER BY id;
```

```
        </SQL_stmt>
            <prolog>?xml version="1.0"?</prolog>
            <doctype>
               !DOCTYPE Order SYSTEM "orders.dtd"
            </doctype>
            <root_node>
               <element_node name="order">
                  <attribute_node name="id">
                  <column name="id"/>
                  </attribute_node>
               <element_node name="signdate">
                  <text_node>
                     <column name="cdate"/>
                  </text_node>
               </element_node>
               <element_node name="amount">
                  <text_node>
                     <column name="total"/>
                  </text_node>
               </element_node>
               </element_node>
            </root_node>
    </Xcollection>
</DAD>
```

Publishing XML documents from a DAD. Having created a DAD, using a text or XML editor, or through the WebSphere* Studio application development family of tools,[12] it is necessary to invoke a DB2-supplied stored procedure to produce one or more XML documents. The stored procedures, which are a component of the DB2 XML Extender,[11] make it possible to select whether:

- The generated documents are placed in permanent tables, temporary tables, WebSphere MQ queues, or in memory.
- The generated documents should be validated.
- A maximum limit should be set on the number of documents produced.
- The content of the DAD should be overridden.

Following is an example of a stored procedure invocation to generate XML documents from a DAD:

```
EXEC SQL CALL dxxGenXML(
            :orderdad:orderdad_ind;
            :result_tab:rtab_ind,
            :result_colname:rescol_ind,
            :valid_colname:val_ind,
            :overrideType:ovtype_ind,:override:ov_ind,
            :max_row:maxrow_ind,:num_row:numrow_ind,
            :returnCode:returnCode_ind,
            :returnMsg:returnMsg_ind
);
```

where the input parameters are:

- orderdad: contains the DAD
- result_tab: contains the name of a table where the documents to be published (in result_colname) are placed together with an indication (in valid_colname) for each document whether it is valid or not (if validation was requested in the supplied DAD)
- overrideType: contains an indication of the type of scoping override to be supplied in this request. There are three possible values: (1) SQL override: to override the SQL statement in a SQL composition DAD, (2) location path restrictions (an XPath subset): to override the values included in RDB_node-generated documents, and (3) no override.
- override: contains either one SQL statement or a series of location path restrictions
- max_row: the maximum number of rows to be returned.

The main output parameters are the actual number of rows returned, and the result table populated with documents and validation indicators.

DB2 supplies a number of stored procedures for document generation. Table 1 shows the major stored procedures.

Storing XML in the database. SQL provides extensibility features such as UDTs (user-defined types) and UDFs to extend the system. DB2 XML Extender provides a character-based UDT for XML, and a collection of UDFs to operate on XML. SQL extensions are proposed for SQL/XML with a built-in XML SQL data type.[9] The SQL/XML data type is more generic (flexible) and provides XML-specific functionality, rather than the XML extender data types that are built using the SQL UDT and character data type functionality.

For example, an application can create a table "orders" with a column "purchaseOrder" of data type XML with the request:

```
create table orders ( oid integer,
                customer varchar(20), purchaseOrder XML );
```

An SQL INSERT request is used to store an XML purchase order document in the table. XMLParse is used

Table 1 DB2 stored procedures for document generation

Publishing Stored Proc Name	Features
dxxGenXML( ) dxxRetrieveXML( )	Documents are placed in a permanent or temporary table. The name associated with the DAD (known as a collection name) is supplied as input instead of the DAD itself.
dxxmqGen( ) dxxmqRetrieve( )	Documents are placed in or retrieved from a WebSphere MQ queue.
dxxGenXMLClob( ) dxxRetrieveXMLClob( )	One document is placed in or retrieved from an output parameter clob.
dxxmqGenClob( ) dxxmqRetrieveClob( )	One document is placed in or retrieved from a WebSphere MQ queue.

to "cast" an XML character string to an instance of the XML data type. An explicit XMLParse is introduced as opposed to using cast syntax, as additional parameters for whitespace handling or other options might be provided. Furthermore, XMLParse performs much more than just switching a type indicator. It might be an expensive operation to check for well-formedness of the document.

```
INSERT INTO orders
values (1000, 'Steel Inc.', XMLParse('<?xml
        version><purchaseOrder> . . . </purchaseOrder>');
```

An additional function, XMLValidate, is being defined to perform XML Schema validation on an XML instance.

Retrieving an XML instance requires serialization of the XML instance into a particular format and encoding. XMLSerialize provides options to deal with data type issues, encoding, and so on.

```
SELECT XMLSerialize(purchaseOrder)
FROM orders
WHERE oid = 1000;
```

XMLSerialize produces a character string (CHAR, VARCHAR, or CLOB) on the database server side. Besides XMLSerialize, host language mapping rules will be defined to retrieve XML values into a client application as serialized XML or even a DOM document.

The DB2 XML Extender[11] provides two ways of storing XML data in DB2:

- XML Column: where the XML data are stored intact with optional hierarchical indexes for speedy search
- XML Collection: where the data are shredded into relational form and the tags are removed. XML Collections are sets of relational columns (whose data contain no XML tags) within one or more relational tables that can be composed into XML documents or processed in a routine way by regular relational tools and applications.

In the previous subsection, a DAD was used to map from relational data to XML data when publishing XML documents from traditional relational database content. In this subsection, we illustrate that the DAD also maps from XML data to relational data when storing XML data in DB2.

In the case of XML Column, the DAD defines the indexes (known as side tables) that DB2 builds and maintains, as documents are inserted and modified by DB2 XML Extender. Location path notation, a restricted form of XPath, is used in the DAD to specify the portions of the document that are to be indexed.

In the case of XML Collection, the DAD defines the mapping between XML content (elements and attribute values) and relational columns across many tables. As documents are shredded or are generated, DB2 consults the relevant DADs to determine how to proceed.

XML column storage. Intact XML documents, incorporating all the tags, can be stored in three user-defined types in relational tables:

1. XMLVarchar: documents stored in DB2 and up to 3K in length
2. XMLCLOB: stored in DB2 and up to 32K in length
3. XMLFILE: documents stored in the local file system

In all three cases, as XML documents are inserted through SQL INSERT requests, indexing tables known as side tables are built in accordance with an XML Column DAD.

To index the "id" attribute in the following document:

```
<order id="4711">
    <SIGNDATE>2002-03-18</SIGNDATE>
    <AMOUNT>24000</AMOUNT>
</order>
```

The following XML column DAD could be used:

```
<?xml version="1.0"?>
<!DOCTYPE DAD SYSTEM "dad.dtd">
<DAD>
        <dtdid>order.dtd</dtdid>
        <validation>yes</validation>
        <Xcolumn>
            <table name="order_side_tab">
                <column name="order_key"
                        type="integer"
                        path="/Order/@id"
                        multi_occurrence="no"/>
            </table>
        </Xcolumn>
</DAD>
```

The path="/Order/@id" option indicates the name of the attribute to be indexed in a side table. The option multi_occurrence="no" indicates that the order, and hence its attribute called id, will appear at most once in each document.

Multiple single occurrence element or attribute values can be placed in one side table. Each time an intact XML order document is inserted into an XML Column, DB2 inserts one row into the table order_side_tab. The row in order_side_tab represents the order. For speedy access, the row in order_side_tab includes the key of the row that contains the intact XML order. DB2 creates a default relational view that incorporates the side tables and the table containing the XML column. The view provides a simple interface to retrieve the intact documents programmatically by content, in an efficient and simple way, without requiring XML parsing.

Consider storing the following document, that contains multiple orders, in an XML column:

```
<orders>
        <order id="4711">
                <signdate>
                   2002-03-18
                </signdate>
                <amount>
                   24000
                </amount>
        </order>
        <order id="4712">
                <signdate>
                   2002-03-19
                </signdate>
                <amount>
                   44000
                </amount>
        </order>
</orders>
```

We would adapt the DAD as follows:

```
<Xcolumn>
    <table name="order_side_multitab">
        <column name="order_key"
                type="integer"
                path="orders/order/@id"
                multi_occurrence="yes"/>
    </table>
</Xcolumn>
```

Each time an orders document is inserted, DB2 inserts multiple rows into the table order_side_multitab, each of which includes the key of the row containing the intact XML orders document. Each row in the side table represents a single order.

When XML documents are inserted into XML columns, they can be validated against XML schemas or DTDs through the svalidate and dvalidate functions mentioned in the previous section. Alternatively, they can be validated on insertion by specifying validation yes in the XML Column DAD, as illustrated previously.

XML Collection storage. The following SQL request shreds XML data into relational form:

```
EXEC SQL CALL
   DB2XML.dxxShredXML( :dad:dad_ind;
                       :xmlDoc:xmlDoc_ind,
                       :returnCode:returnCode_ind,
                       :returnMsg:returnMsg_ind);
```

Stored procedures for shredding are described in Table 2.

Table 2 Stored procedures for shredding

Shredding Stored Proc Name	Features
dxxShredXML( ) dxxInsertXML( )	A document is shredded into many rows in many tables. The name associated with the DAD (known as a collection name) is supplied as input instead of the DAD itself.
dxxmqShred( ) dxxmqShredall( ) dxxmqInsert( ) dxxmqInsertall( )	These are similar to the stored procedures in the row above, except that they shred documents held in WebSphere MQ queues into DB2 tables. dxxmqShredall( ) and dxxmqInsertall shred all the documents in a queue whereas dxxmqShred and dxxmqInsert shred the first document only.
dxxmqShredCLOB( ) dxxmqShredAllCLOB( ) dxxmqInsertCLOB( )	These are similar to the stored procedures in the row above, except that they shred large documents held in WebSphere MQ queues into DB2 tables.

XML storage guidelines. With the DB2 XML Extender, where data content is updated often and speedy performance of updates is very important, we recommend XML Collection as the storage method. XML Collection also makes it possible to apply analytical tools for relational data to the XML content. Where it is desired to view documents precisely as they were on input to the system, and document updates are not frequent, we recommend XML Column, which also provides fast search capability. Often, documents are stored in XML Column for nonrepudiation purposes.

Some applications use a combination of XML Collection and XML Column. For example, an insurance claims system may store and index the claims in the form in which they were input to the system, for easy subsequent access. The claims may also be shredded into XML Collections to drive a claims processing system.

Working with XML data in SQL. Using XML data in SQL requires the ability to search, update, extract, and shred XML data. SQL by itself is not sufficient to perform these operations, as it does not provide language to navigate and traverse XML data. Operating on XML values requires an XML query language such as XPath and/or XQuery. In this subsection we describe SQL extensions for XML to provide means to search, update, extract, and shred XML in the context of SQL. Some functions are currently discussed in SQLX as a potential proposal for SQL/XML, some functions are implemented in DB2 XML Extender, and others are implemented in various prototypes. The functions are discussed here from the perspective of application requirements without addressing their current status.

XMLExists is a Boolean function, which evaluates an XPath expression on an XML value. If XPath returns a nonempty sequence of nodes, then XMLExists is true, otherwise it is false. XMLExtract returns the result of the XPath query as an XML instance. The following sample query returns all customers and dates of orders that include a shoe item:

```
SELECT customer,
   XMLExtract(
      purchaseOrder,'/purchaseOrder/@orderdate'
   )
FROM orders
WHERE
   XMLExists(
      purchaseOrder,
      '/purchaseOrder[list/item/desc/text( )="Shoes"]'
   )=1;
```

XMLUpdate modifies fragments in an XML value. It takes three arguments as input. XMLUpdate operates on an XML value (first argument), locates XML fragments in the XML value using an XPath expression (second argument), replaces the identified XML fragments with an updated XML fragment (third argument), and finally returns the new XML value. The following example updates the customer name in order XML documents to 'IBM', where the salesperson is 'John Doe'.

```
update sales_tab
set order = XMLUpdate(order,
```

Table 3 DB2 XML Extender SQL function descriptions

XML Extender SQL Function Family	Features
Validation	Enable the validation of XML documents against schemas and DTDs
Transformation	Enable the transformation of XML documents through XSL transformations
Import	Enable importing XML documents from a file system into DB2
Export	Enable exporting XML documents from a file system into DB2
Extract document fragment	Enable the extraction of one or more well formed document fragments from a document by specifying a location path
Extract element & attribute values	Enable the extraction of one or more well element or attribute values from a document by specifying a location path
Update	Enable the modification of element or attribute values in a document by specifying a location path

```
        '/order/customer/name',
        XMLParse('<Name>IBM</Name>'))
where sales_person = 'John Doe'
```

Shredding of XML takes an XML value as input and returns a derived table with columns containing extracted values. The extracted values are specified through XPath expressions. A context XPath expression provides the set of values. For example, an XML value may contain a list of items. The context XPath expression navigates to each item, produces a row, and additional XPath expressions operate on each item to retrieve column values for the rows. A generic function *XMLTable*, takes as input an XML value, an XPath expression for the context, and N XPath expressions to return N column values. The following example query takes an XML value with an item list as input. The context XPath expression '/items' produces a row per item in this list, and the column XPath expressions './item/@id' and './item/@desc' return column values for item id and item description.

```
select *
from table ( XMLTable( :item list, '/items',
                './item/@id', './item/@desc'))
        As T (id integer , desc varchar(10) )
```

The resulting table is:

ID	DESC
23	Shoes
25	Bungee Ropes

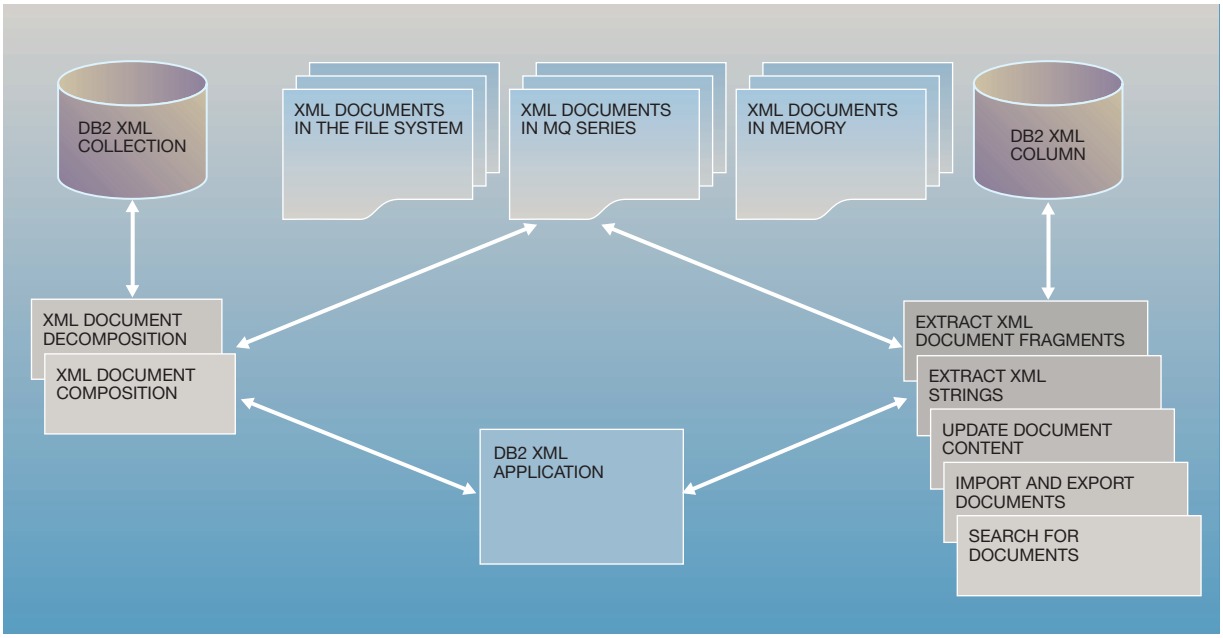
Manipulating XML column content through the DB2 XML extender. DB2 XML Extender provides over one hundred SQL functions that can be used to manipulate XML stored in tables, in memory, in the file system, in WebSphere MQ, or generated through document publishing via the DAD. We highlight just some of the functions and their features. Of course, all of these functions can be used in conjunction with sophisticated SQL requests. Table 3 describes the families of SQL functions available in DB2 XML Extender.

Other XML Extender features. The DB2 XML Extender provides optional validation support against DTDs and XML schemas. The validation can take place prior to storing or shredding the XML document into DB2, or after retrieving, extracting, or generating XML documents from DB2. In addition, it is possible to store and manage DADs and DTDs in DB2 tables. In the future, integration with an XML repository will be provided, so that XML meta-data can be managed in a very general way.

The DB2 XML Extender helps integrate data XML data stored in the file system and in WebSphere MQ with relational data. For example, it is possible to remove a number of items from a WebSphere MQ queue and shred them into relational data, all through a single SQL request.

Figure 2 depicts the components used in implementing the XML data integration functions we have presented for DB2 XML Extender.

Figure 2 XML integration with DB2



XQuery

XQuery is a functional expression language that can be used to query or process XML data or any data that can be represented within the same model as XML. Being purely an expression language, XQuery programs are easier to understand and maintain than XSLT, because they do not include the complexities or management of templates (rule-based system). This is especially true for highly structured data, and for longer programs. XQuery will still be able to effectively process semi-structured data. The query language is small and powerful. It has both an easy, human-readable form and an XML representation. The XQuery language is an activity of the W3C (World Wide Web Consortium). Several publicly available working drafts have been published.[13]

XQuery as the next XML programming language. XQuery can become the next XML programming language. XQuery provides needed concepts, upgraded functionality, and new ideas that will fundamentally change the way XML applications are designed and implemented.

XQuery supports XML data typing using XML Schema as a base, but will also be able to process documents without type information. XQuery includes an upgraded version of XPath whose semantics mesh better with typed data. XQuery provides a powerful FLWR (for, let, where, return) statement that allows joins to be expressed. XQuery also allows the user to construct sequences of items in a given order and to perform arbitrary sorts on any generated sequence. New elements, sequences, and XML data fragments can be formed. The result of expressions can form the output of queries, or as a subexpression, form a temporary data structure that can itself be queried. Variables can be bound to the result of an expression and utilized in multiple places. Variables are not declared with a specific type, but instead take on the type of the expression they are bound to. XQuery includes a type-switch expression that allows programs to test an expression for being a particular type and to form a result based on this test. XQuery has the usual programming constructs, such as if-then-else and arithmetic, Boolean, and comparison operations.

The XQuery language allows one to define functions and invoke them. These functions can be local functions to the query, allowing one to organize one's query in smaller, more understandable fragments. Local functions do not have to be defined to the database system. Some XQuery processors may allow

function libraries to be defined and then accessed in queries. The XQuery language defines an extensive built-in function library. Besides mathematical functions, operations on nodes such as deep-equal, filter, and various functions on sequences will give programmers a wide and detailed level of control over expressions. These features allow XQuery programs to be longer and more understandable than programs in most query languages.

The effective XML programming paradigm involves data transformation, transactions, work flows and messaging. The XQuery 1.0 draft definition covers the core topic of data transformation.

The data manager as the XQuery view processor. The most fundamental requirement for a database manager that supports XML applications is that it must be able to form an XML result. The most natural approach to do this is for the database manager to support XQuery. This makes human or programmatic interaction with data consistent with the XML model and eliminates the need to straddle multiple paradigms such as SQL, XPath, and potentially SAX/DOM/XSLT in the same application (along with their individual programming interfaces and protocols).

XQuery operates on an XML data model, so in order for XQuery to be able to process relational data, stored tables, and columns, an extension mechanism is needed. The extension mechanism can simply provide a simplistic view of a table as XML (actually an instance of the XQuery data model). In XQuery, the way to accomplish this is to invoke a special function that returns an XML document with a format such as the following:

```
<table-name>
    <row>
        <column1-name> data </column1-name>
        <column2-name> data </column2-name>
        <column3-name> data </column3-name>
    </row>
    <row>
        <column1-name> data </column1-name>
        <column2-name> data </column2-name>
        <column3-name> data </column3-name>
    </row>
          . . .
</table-name>
```

The concept of the default view can be applied to any table, view, database or even arbitrary SQL queries.

The view provides the XQuery programmer the raw material to form hierarchic results. There are several proposals for simplistic (sometimes called "default") XML representations of relational tables. These views can span the entire database, whereas others represent only a particular table. The process of standardizing formats is a work in progress, but emerging XQuery processors can easily provide a reasonable extension function providing the required functionality until such standards are in place. SQL/XML is an example of a standardization effort that defines XML views of relational data.

An application could simply request the entire default view and send it to an outside processor such as XSLT to form the desired result. The use of XSLT would be a needed step because the default view is almost never the desired input or output of an application. Using XSLT would eliminate the need for the database to support XQuery, but this approach is wrong for obvious reasons: XML views of entire tables must be exported to the application. XSLT must accept the transformation request, then receive potentially multiple default view documents, and then perform the needed joins. XPath does not naturally express joins, and processors like XSLT are likely to be highly inefficient at processing them.

XQuery should be implemented at the database so that data can be selected (predicated), projected, and joined efficiently within the database process. The database manager also provides highly optimized facilities such as sorts, and resources such as character collation sequences to support the XQuery requests. Since XQuery has the ability to construct new result node hierarchies based on the input data, the desired output format, matching the desired XML schema, can be generated. The database can utilize information about the data and various indexes to efficiently process the query and send a single result document back as a result.

All applications could access default views and form output documents, but this approach would still lead to difficulties. Applications would have to become involved with table and column names and the relationships between the tables in order to form the join conditions needed in forming the required output hierarchies. These expressions often become complex for nontrivial documents. A standard B2B

purchase order would be an example. In order to isolate applications and other requestors, the XML database administrator should be able to create XML views of relational data and define them using XQuery. Applications and other requestors would only need to issue an XQuery request in relation to a provided XML view.

One fundamental way to think about XQuery views is a publishing paradigm. XML database administrators talk with application developers and content providers about the required schemas and hierarchies. The database administrator creates XML views that support the required XML hierarchies including hierarchies required by direct user interaction and those required by applications. The database administrator is free to adjust table and column definitions and storage structures while providing the same view. The schema of the view can also be published, and applications, users, and tools can use that information to formulate queries against the view itself.

Because the database query processor references data in aggregate, queries against views should be composed internally during optimization of the query so the total cost of performing the query can be minimized, just as SQL databases optimize queries with reference to views. This prevents the internal materialization of the view. Applications requiring only small results from large data stores are sent only small amounts of data from an optimized query that is executed at the server. Client, network, and server resources are conserved. Applications requiring large results benefit from the optimized queries and from disk buffering and streaming technologies that the data manager can provide.

By using the XQuery expression language, requestors can provide all the required predicates, joins, unions, and other expressions needed to form the desired result. This is much more flexible than, say, a set of remote procedures each with a fixed set of parameters that produce particular XML results. In such a system, a procedure has to be developed for each output requirement, or the requestor must accept intermediate results and then use another processor (XSLT) against potentially many streams to formulate the desired result.

The idea is to publish a relatively small set of useful views, such as a list of books organized by author and books organized by subject, or purchase orders and requisitions. A user chooses the most applicable view and operates XQuery against it, providing the additional constraints to formulate the desired result. As a result, the XML database administrator is relieved from the details of every application/user, and the application/user does not need to know the keys and join conditions to formulate results from the schema and correlate them hierarchically.

The use of views also isolates the user from the extension mechanism used to access relational data. One does not need to know the various default view formats, how they might evolve into standardized forms, or any special language extensions that apply to the database. The user sees only a pure XML view of hierarchical data.

The XQuery view mechanism also avoids separate and proprietary mapping languages and files. Because the view mechanism uses XQuery itself, there is a seamless integration and a lack of limitations associated with creating mappings, querying mappings, or using multiple mappings in the same query. Different technologies do not need to be learned. Views can be prototyped as queries. With sufficient authority to access tables, any query can be executed without the administrator forming a mapping.

XQuery is the way to process and formulate XML data models.

Views have been used for years in relational systems, but XML views have greater applicability because it makes sense to encode a great deal more information in a hierarchy than in a single row. This is because all the required data and the relationships within the data are self-contained as a single unit, rather than spread across several tables or views.

All XML technologies used by applications, including XML languages such as XPath, XSLT, and XML application interfaces, are best at processing data whose hierarchical structure provides useful and semantic relationships in the context of the specific application. XML documents and messages will be designed with this in mind. Therefore, data stores have the need to materialize hierarchical information in various formats using the same base data. The use of XQuery views provides a clean, simple, and powerful way to solve this problem. Applications are

Table 4 Example of purchase order tables

Order Table

order_id	customer_id	ship_method	date	status
100	777	UPS	1999-10-23	shipped
101	777	USPS	2002-01-25	accepted
102	888	UPS	2002-02-05	shipped

Order_Items Table

order_id	item_number	product_id	quantity	price
100	1	1001	1	108.25
100	2	1002	2	17.42
101	1	1002	5	17.42
102	1	1003	1	104.10

Product Table

product_id	description	stock	price
1001	Sound Blaster Audigy MP3+	1	108.25
1002	Travel Alarm With Radio	1	17.42
1003	5.1 Surround Sound Speaker System	1	104.10

Customer Table

customer_id	name	address
777	William P Barnes	104 West Avery Lane, CA
888	Shirley Jackson	1344 Pennsylvania Ave, OH

made simpler by performing a single XQuery request over these views and receiving satisfactory results in one step.

Highly operational data are normalized and are stored in most systems using the relational model. Selecting, forming, and correlating data hierarchies efficiently is best performed by the data store, because it has the resources available to perform the requests and will incur the least cost in processing the requests.

Example XQuery view. Table 4 depicts an example scenario in which four database tables are used to represent simplistic purchase orders. A simple query over the default view for customers would be:

```
table( "Customer")
```

The query would return a document with the following structure:

```
<Customer>
  <row>
  <customer_id>777</customer_id>
  <name>William P Barnes</name>
  <address>104 West Avery Lane, CA</address>
  </row>
  <row>
  <customer_id>888</customer_id>
  <name>Shirley Jackson</name>
  <address>1344 Pennsylvania Ave, OH</address>
  </row>
</Customer>
```

XQuery can transform the default view into a more logical document:

```
<customerList>{
      for $c in table("Customer")/Customer/row
      return
      <customer id="{ $c/customer_id }">
            <name>{ data( $c/name ) }</name>
            <address>{ data( $c/address) }</address>
      </customer>
}</customerList>
```

The output no longer simply exposes underlying table and column names but instead structures the data with the desired names and hierarchy. Both subtle and dramatic changes are easily accomplished.

```
<customerList>
   <customer id="777">
      <name>William P Barnes</name>
      <address>
         104 West Avery Lane, CA
      </address>
```

Figure 3 XQuery view for purchase order example

```
create view orders as

    <orders>{

        for $o in table("Order")/Order/row return
        <order>{

            <order_id>{ data( $o/order_id ) }</order_id>,
            <date>{ data( $o/date ) }</date>,
            <ship_by>{ data( $o/ship_method ) }</ship_by>,
            <status>{ data( $o/status ) }</status>

            for $c in table("Customer")/Customer/row
            where $c/customer_id=$o/customer_id return
            <customer id ="{ data( $c/customer_id ) }">
                <name>{ data( $c/name ) }</name>
                <address>{ data( $c/address ) }</address>
            </customer>,

            <items>{
                for  $i  in table ("Order_Items")/Order_Items/row,
                     $p in table("Product")/Product/row
                where $i/order_id = $o/order_id and
                      $p/product_id = $i/product_id
                return <item>
                           <item_no>{ data( $i/item_number)}</item_no>
                           <item_desc>{ data( $p/description ) }</item_desc>
                           <item_qty>{ data( $i/quantity ) }</item_qty>
                           <item_price>{ data( $i/price ) }</item_price>
                       </item>
                sortby ( item_no )
            }</items>

        }</order>

    }</orders>
```

```
    </customer>
    <customer id="888">
        <name>Shirley Jackson</name>
        <address>
            1344 Pennsylvania Ave, OH
        </address>
    </customer>
</customerList>
```

Now, an XQuery view to materialize the entire purchase order is seen in Figure 3.

The "orders" XML view constructs an "orders" tag that contains all the orders in the system. The orders tag contains the order id, date and shipping information, a customer information hierarchy, and the items in the purchase order. Here we can see more name changes.

The following query over the orders view returns a single order:

```
view( "orders" )/orders/order[ order_id = 100 ]
```

The document returned by this example is:

```
<order>
    <order_id>100</order_id>
    <date>1999-10-23</date>
    <ship_by>UPS</ship_by>
    <status>shipped</status>
    <customer id='777'>
        <name>William P Barnes</name>
        <address>104 West Avery Lane, CA
        </address>
    </customer>
    <items>
    <item>
        <item_no>1</item_no>
        <item_desc>Sound Blaster Audigy MP3+
        </item_desc>
        <item_qty>1</item_qty>
        <item_price>108.25</item_price>
    </item>
    <item>
        <item_no>2</item_no>
        <item_desc>Travel Alarm With Radio
        </item_desc>
        <item_qty>1</item_qty>
        <item_price>17.42</item_price>
    </item>
    </items>
</order>
```

An application query over the view produces:

```
<summary>{
      view ("order_summary")/ order_summary/order
                                        [@custId = 777]
}</summary>

<summary>
      <order custId = "777">
          <order_id>100</order_id>
          <date>1999-10-23</date>
          <status>shipped</status>
      </order>
      <order custId = "777">
          <order_id>101</order_id>
          <date>2002-01-25</date>
          <status>accepted</status>
      </order>
</summary>
```

When views are accessed, they can be composed with views they reference, making the previous view equivalent to:

```
<order_summary>{
    for $o in table("Order")/Order/row return
    <order custId = "{$o/customer_id}">{
        <order_id>
          { data ( $o/order_id) }
        </order_id>
        <date>
          { data( $o/date ) }
        </date>
        <status>
          {data( $o/ship_method ) }
        </status>
    }</order>
}</order_summary>
```

Composed views can be cached, allowing views to be created against other views with minimal impact. Views do not need to be formed against the default view for efficiency.

Integration with novel storage structures and federation. Highly operational data are data that are used as search and join conditions, in calculations, are updated, or control the behavior of programs. This kind of data is probably best stored using mature relational mechanisms. However, not all data are highly operational and not all data fit well in the relational model. Data can be historical in nature, such as billing records. Some data may be read but almost never written. Some documents are usually retrieved as a complete unit. In this arena, searching, retrieval, and preservation of the original document structure are the significant operations. Cataloging and attaching external information to intact documents is often a requirement. Versioning may be more important than updating. Other kinds of data may involve documents whose structure evolves over time, such as survey forms. Data may also be sparse, such as a catalog of diverse products each with different sets of attributes that can be queried.

Because the relational model is often out of sync with the practicalities of nontraditional data, databases may be enhanced with novel data storage structures. Some of these can be: the ability to store documents in full, the ability to form collections of documents, specialized indexes on XML documents, or XML-aware text search indexes capable of handling semi-structured sections of an XML document. The database may also support storage structures that maintain user-defined data cataloging functions.

One problem with storing operational data in XML documents is that XML must form trees. Data lower

in the hierarchy can often be duplicated, and this presents a problem for efficient storage, buffering of data used to increase performance, and maintenance of consistency when updating. To solve this problem, databases may allow optimized storage of XML fragments. Such fragments can be normalized and indexed. The fragments would be able to be accessed and joined efficiently with other fragments to form complete documents. Here, the ability to create views is important.

As the market demand grows, these new features will emerge from database providers. Specialized servers and data storage techniques are applied to address the critical aspects, the heavy, day-to-day processes of the data. However, it is important to realize that data are never utilized for a single purpose. Eventually we need all data to be integrated. Data must be cataloged and summarized for intelligence reasons. In large corporate data environments, data in one system are often the source of messages or updates that need to be applied to other systems.

When differing storage and indexing schemes exist, there is a need to tie them together. The common ground is XML and its flexible data model. XQuery is the way to process and formulate XML data models. Using the database as the processor will make it possible to efficiently integrate a variety of storage structures and processing models. It is also possible for the data manager to tie in existing federated support in order to further increase the ability to integrate and manipulate various kinds and sources of data.

B2B or scientific intelligence applications may require aggregations of data to be readily available. The database can form these results and they can be stored, or these computations can be specified in views. The data manager can cache view results according to specified policy. In addition, functions to help generate these kinds of queries can be added to the XQuery language by use of extension function libraries.

Special features for XQuery. A number of special features can be added to the database XQuery processor that will make it different than other query languages.

A default view can be generated across the entire database. If this is supported, seamless queries against meta-data and data will be possible. For example, one can ask for all the tables that have a column named salary and have a value larger than 10000. XML query languages naturally query across meta-data (tags) and data (node values). Exposing any XML view affords this ability. As more data are placed in the view, the queries can become more powerful and abstract. For example, a view could also expose type, ownership, and data cataloging information as well as data values.

The XQuery processor can be enhanced with higher order operators. An operator called ExecXQuery could be added to allow a query to form queries under program control, execute them, and process the results.[14] Other, more specialized, higher-order operators could be developed to help access various data sources or to implement customized pointer-to-data operations.

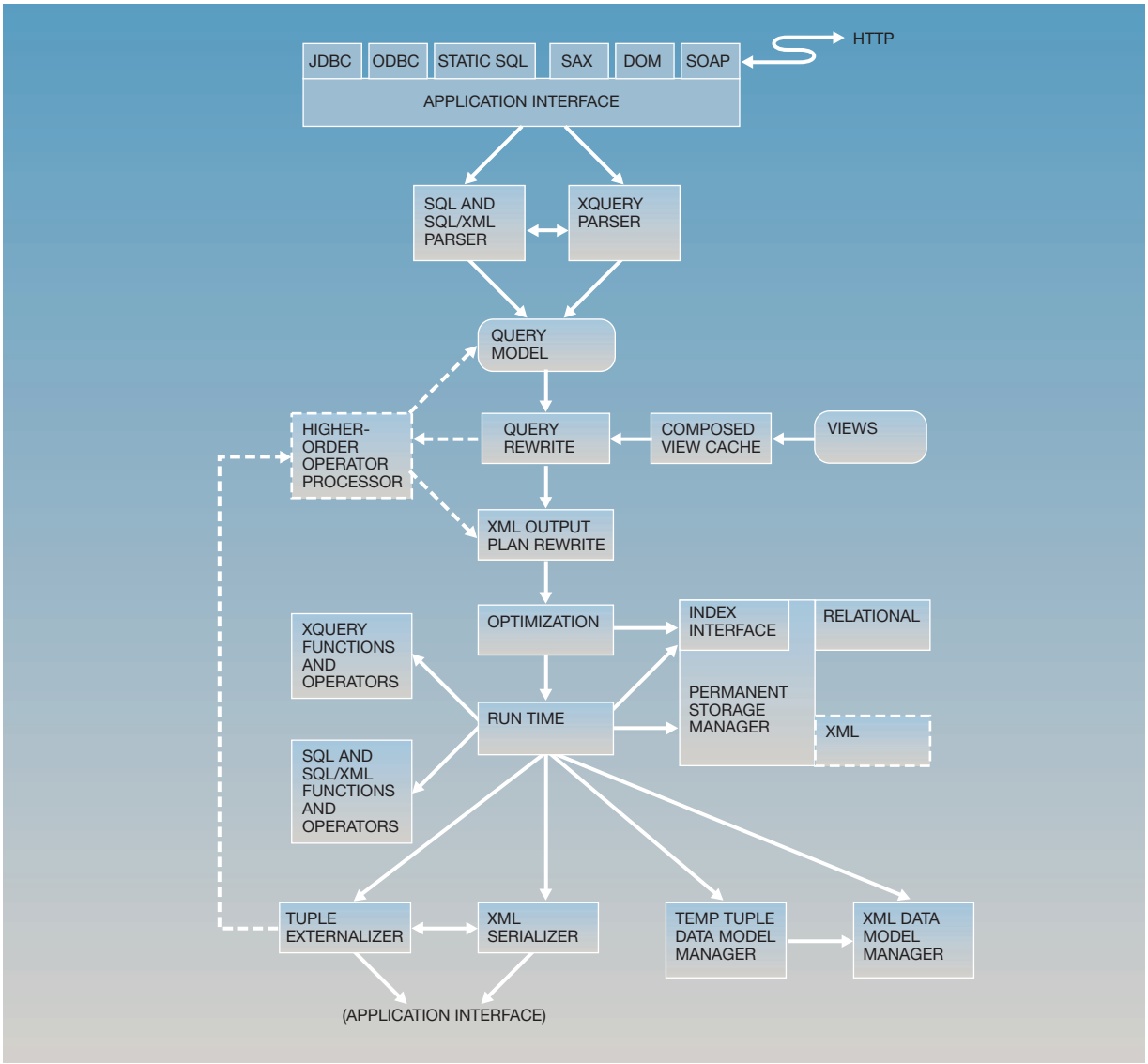
A user chooses the most applicable view and operates XQuery against it, providing the additional constraints to formulate the desired result.

A database manager should be able to publish the XML schema of its views. An integrated way of publishing human readable documentation of the fields of the view should be provided.

Data manager/development environment tools should be able to read existing XML schemas, suggest relational table formats that can retain the information, allow the designer to interactively implement storage structures, and form appropriate XML views upon the storage structures that implement desired schema types.[14] The same technology can be extended to allow users to easily form new XML views from existing storage structures and to interact with XML shredders or automatic transaction generators.

When performance-critical applications are written using SAX, DOM, or XSLT, specialized versions of these processors could be provided that support streaming forms of XML input and/or compressed or binary formats of XML. As the XML programming model evolves, the data manager can provide functionality that matches the needs of business applications.

Figure 4 Integrated database architecture



The database should provide a mechanism to directly issue XQueries using the XML programming model, rather than traditional database application interfaces or protocols. One such way is for databases to respond directly to XQueries via SOAP requests.

These additional features can further increase the power of XML data stores and will allow users, programmers, and applications to interact with data servers in new ways and with greater ease.

Integrated database architecture

The integrated data manager will span relational and XML technologies. Figure 4 depicts an integrated relational and XQuery data manager (optional components are shown with dashed lines):

The integrated data manager can respond to requests presented by new XML interfaces such as XQuery to SOAP, XQuery to SAX or DOM, and to traditional SQL

interfaces. An integrated XQuery SOAP interface allows the database manager to respond to requests over the network without having to install and configure a separate Web server. This approach also simplifies designs including Web servers because it eliminates the need to use an SQL-based interface or protocol in the design.

Data requests may be made using SQL/XML or XQuery. The data manager's application interface determines the type of query and sends it to the appropriate parser. The data manager may allow XQuery and SQL queries to be intermixed either directly or through views. This requires a query model and type system that accurately models both SQL and XQuery semantics.

Query rewrites are applied against the user's query. Views are included and the combined query is optimized by the rewrites. For XQuery, views are parsed, optimized, and cached before merging them with the requesting query. This is important because XQuery views normally contain many XPath queries against the input views, either default views or other user-defined views that can be collapsed into more direct accesses. Caching composed views allows later queries against the view to utilize an already simplified representation of the view. Alternatively, views can be simplified when they are stored by the database. It is likely that XML queries will be treated as dynamic queries (in contrast to traditional static SQL queries), so compiling and optimization time is a part of the total cost of each invocation unless the optimized views/queries are reused. Techniques similar to SQLJ's (SQL interface for Java) customization feature could be developed for XQuery, but even so, not all applications will be customized.

If the database supports higher-order operators, portions of the query below the higher-order operator are executed and the results of the subqueries form new subqueries that are parsed and grafted into the user query.[14]

If XML hierarchies will be output, query rewrites will be performed against the simplified query to implement specific plans that are needed to help the run-time component output correlated, hierarchical, information in an efficient way. For example, the rewrite would apply the Sorted Outer Union approach.[15] With new run-time operators, plans that are even more efficient could be generated.

The query is then optimized to create the final run-time plan. If the database manager supports multiple storage and indexing technologies, the optimizer acts against an abstract interface allowing many such technologies to be plugged into the system. Traditional features of optimizers such as plan caching and plan storage for static SQL continue to be applicable.

The run-time component executes the optimized plan. The run-time component can execute any SQL or XQuery semantic contained in the query model. This allows query rewriting and optimization to be cost-based and discretionary. In order to accomplish these goals, the run-time component must implement the set of both SQL and XQuery functions and operators, because each language specifies different semantics. The run-time component must be able to support XQuery-only operations such as navigations using XPath. The reference-based XQuery data model must be added as a part of the run-time component. The run-time component uses an integrated data model that allows tuples to contain instances of the XQuery data model. Temporary storage of intermediate results includes temporary instances of the XQuery data model held by a reference, and temporary tuples that may contain reference instances of the XQuery data model. If the data manager supports permanent storage in XML format, temporary references may refer to permanent instances or temporary instances.

Finally, XML results are serialized, and converted to character format, if required. XML results will appear either by themselves or be bound out as columns of a tuple. The details are handled by the application interface.

The XTABLES[16] project prototypes the XQuery rewrite portions of the integrated data manager architecture. XQueries are accepted by a SOAP-enabled Web server (Lunar Eclipse) and are passed to an SQL stored procedure for execution at the data server.

The XTABLES stored procedure implements the XQuery parser and query rewriting system. Many simple XQueries against relational data can be fully composed into SQL-compatible queries. Given a number of limitations such as the fact that data in an SQL database are not ordered unless this is explicitly requested, data in SQL databases can be successfully queried and published as XQuery views.

Full support of XQuery requires preserving document order, supporting duplicate node removal, and exact semantics of operations and functions. Implementation of the integrated database architecture will overcome these limitations.

Conclusion and future directions

The future of XML programming will be driven by functionality provided by standards-based languages implemented by data management systems. Data are stored in the relational model and for many reasons will continue to be stored in this way, but XML is the correct way to present messages and to communicate with other systems. Data managers are the start and/or end point of many if not most data flows, and as such should perform the role of query processing and data transformation. SQL/XML is a new part of the SQL standard that allows users to form and query XML data within the relational model. XQuery is an emerging XML-based language that can effectively query and transform both relational and hierarchical data. Nonrelational forms of data are becoming increasingly important, and databases may be enhanced with XML-specific storage structures. XQuery and SQL/XML are critical in supporting and integrating new hierarchical storage structures within the database. Federated databases and mediators will also benefit from the XQuery data model and languages that can integrate nontraditional data sources with existing relational data. Databases with SQL/XML and XQuery functionality will eliminate the need to produce *ad hoc*, customized XML applications. Implementing standards-based XML languages at the data server will eliminate specialized skills needed for application development, long development cycles, and inefficient requests against data servers generated by naive client-based solutions. XML-enabled databases will provide easier, faster, and more efficient systems integration efforts.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc.

Cited references

1. D. Megginson, et al., *SAX: The Simple API for XML*. See http://www.saxproject.org., Version 1.0, May 1988.
2. World Wide Web Consortium, *Document Object Model (DOM) Level 1 Specification, Version 1.0, W3C Recommendation 1 October, 1998*. See http://www.w3c.org/TR/1998/REC-DOM-Level-1-19981001 and see http://www.w3c.org/DOM/DOMTR.
3. World Wide Web Consortium, *XSL Transformations (XSLT), Version 1.0, W3C Recommendation 16 November, 1999*. See http://www.w3c.org/TR/1999/REC-xslt-19991116.
4. World Wide Web Consortium, *XHTML™ 1.0: The Extensible HyperText Markup Language: A Reformulation of HTML 4 in XML 1.0, W3C Recommendation 26 January, 2000*. See http://www.w3.org/TR/xhtml1/ and see http://www.w3c.org/MarkUp/.
5. World Wide Web Consortium, *XML Schema Part 1: Structures, W3C Recommendation 2 May 2001* and *XML Schema Part 2: Datatypes, W3C Recommendation 02 May, 2001*. See http://www.w3c.org/TR/xmlschema-1/ and http://www.w3c.org/TR/xmlschema-2/.
6. World Wide Web Consortium, *Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation 6 October, 2000*. See http://www.w3c.org/TR/2000/REC-xml-20001006.
7. World Wide Web Consortium, *Namespaces in XML, 14-January-1999*. See http://www.w3c.org/TR/1999/REC-xml-names-19990114/.
8. World Wide Web Consortium, *Simple Object Access Protocol (SOAP) 1.1, W3C Note 08 May, 2000*. See http://www.w3.org/TR/SOAP/. See http://www.w3c.org/2000/xp/Group/ for work in progress.
9. "(ISO-ANSI Working Draft) XML-Related Specifications (SQL/XML)," J. Melton, Editor, for FCD Ballot, WG3:ICN-011, H2-2002-063 (March 2002).
10. A. Eisenberg and J. Melton, "SQL/XML and the SQLX Informal Group of Companies," *ACM SIGMOD Record* **30**, No. 3 (September 2001).
11. *DB2 Universal Database Extenders: XML Extender Administration and Programming*, SC27-1234, IBM Corporation (2002).
12. *WebSphere Studio Application Developer Programming Guide*, SG24-6585-00, IBM Corporation.
13. World Wide Web Consortium, XQuery 1.0: An EML Query Language, W3C Working Draft, 16 August 2002. See http://www.w3.org/TR/xquery.
14. J. E. Funderburk, G. Kiernan, J. Shanmugasundaram, E. Shekita, and C. Wei, "XTABLES: Bridging Relational Technology and XML," *IBM Systems Journal* **41**, No. 4, 616–641 (2002, this issue).
15. J. Shanmugasundaram, et al., "Efficiently Publishing Relational Data as XML Documents," *Proceedings of the VLDB Conference*, Cairo, Egypt, September 2000.
16. IBM, Xperanto Technology Demo, based on Xperanto/XTABLES prototype technology. See http://www.ibm.com/software/data/developer/demos/xperanto/.

Accepted for publication August 8, 2002.

John E. Funderburk *IBM Software Group, Silicon Valley Laboratory, 555 Bailey Avenue, San Jose, California 95141 (electronic mail: jfund@us.ibm.com).* Mr. Funderburk is a software developer at IBM's Silicon Valley Lab. He previously worked on DB2's XML Extender and is currently working on XTABLES.

Susan Malaika *IBM Software Group, Silicon Valley Laboratory, 555 Bailey Avenue, San Jose, California 95141 (electronic mail: malaika@us.ibm.com).* Ms. Malaika is a senior software engineer with IBM's Silicon Valley DB2 development group. She works in the area of XML, DB2, and the Web.

Berthold Reinwald *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: reinwald@almaden.ibm.com).* Dr. Reinwald joined the IBM Al-

maden Research Center in 1993, after finishing his Ph.D. degree in computer science from the University of Erlangen-Nuernberg. His Ph.D. thesis on workflow management received the "best Ph.D. thesis" award from the university and was published as a book. At IBM Research, Dr. Reinwald contributed to SMRC (shared memory-resident cache) in DB2 Common Server, query explain tools, workflow management with Lotus Notes®, FlowMark®, and MQSeries, researched and delivered in DB2 Universal Database® support for OLE/COM, OLEDB, XML, and most recently Web services. Dr. Reinwald is active in the design, architecture, and implementation of SQL extensions for XML.