Integration of SQL and XQuery in IBM DB2

F. Özcan
D. Chamberlin
K. Kulkarni
J.-E. Michels

Relational database systems have dominated the database industry for a quarter century. However, the advent of the Web has led to requirements for storage of new kinds of information in which the order of information is important and data structure can vary over time and from one document to another. These evolving requirements have given rise to Extensible Markup Language (XML) as a widely accepted data format and to XQuery as an emerging standard language for querying XML data sources. A set of extensions to the Structured Query Language (SQL) called SQL/XML enables XML data to be stored in relational databases, taking advantage of the mature infrastructure of relational systems and combining the advantages of SQL and XQuery. However, building a bridge between SQL and XQuery is challenging due to the many syntactic and semantic differences between the two languages. This paper describes how IBM DB2® deals with this challenge and provides users with a flexible system for storing and processing both relational and XML data.

INTRODUCTION

Since the introduction of the first relational database systems in the early 1980s, the commercial database field has seen mostly evolutionary changes. Most large-scale commercial database systems introduced since that time have been based on the relational data model and Structured Query Language (SQL). Recently, however, a new data format, Extensible Markup Language (XML), and a new query language, XQuery, have emerged to challenge the predominance of pure relational systems. XML and XQuery represent a significant new approach to database management. In this paper, we examine the motivation for this new approach, and we compare the SQL and XQuery languages to determine how they can be used cooperatively by sharing a common infrastructure, thereby taking advantage of the large existing investment in relational technology.

In understanding the need for a new approach to storing and retrieving data, the concept of *metadata* is crucial. Metadata is defined as "data about data"—that is, it is information that describes the structure of stored data. All database systems provide some means for storing metadata, and all query languages make use of metadata in processing queries. In relational systems, metadata is stored

separately from the data itself, typically in a set of tables called the *system catalog*. This is possible because relational systems are designed for data that has a regular, repeating structure that can be described independently of any data instance; for example, every row of a table contains the same columns and the same data types. Relational database systems are well optimized for traditional business applications in which the metadata is well known and changes slowly, if at all.

SQL is a mature relational database language that takes advantage of the regular structure of data stored in tables. For example, consider the following SQL query:

```
SELECT itemno, price
FROM product
WHERE description = 'sweater' AND price < 100
ORDER BY price
```

In processing this query, an SQL implementation can rely on information about the `product` table that is stored in the system catalog. For example, each row of the table might be known to have exactly one `price` value of type `DECIMAL` (possibly null). The semantics of the query language need not be concerned with rows that have no `price` column, or have more than one price, or have a price of an unexpected type.

As database systems have evolved toward handling more complex kinds of information, the need for a more flexible data format has become evident. For example, Web services and other forms of e-commerce exchange information in the form of messages with complex and flexible formats. Documents available on the Web vary widely in their structure and very often rely on an intrinsic (non-value-based) ordering of their parts.

Even among documents of a single type, a great variation may exist from one document to another. For example, in medical records, patients may vary widely in the numbers of doctors visited, types of insurance, diseases, medications, procedures, and so on. Types of data that are present in one patient record may be absent in another. Each medical record may be *sparse*, meaning it contains only a few of the many possibilities. Because of these variations, each record must be self-describing; that is, it must contain metadata that describes its own structure and content.

The need for documents to be self-describing led to the notion of *markup*—descriptive information associated with the parts of a document. Markup originated in the publishing industry and was first used by editors to specify aspects of appearance, such as font and size. Gradually, markup evolved toward a logical description (for example, "citation" rather than "italics"), which enabled document components to be rendered differently on different devices and to be more readily understood by applications such as information retrieval systems. A standard notation for logical markup called Standardized General Markup Language (SGML) was adopted by the International Organization for Standardization in 1986.[1] Today's XML notation is a direct descendant of SGML.

In self-describing data, such as XML documents, metadata is separated into two types—*markup* and *schema*. Markup contains information about individual instances of stored data—for example, a piece of data might be identified as an address or as a part number. A schema, on the other hand, contains global information about how documents are assembled from their component parts. (Note that this use of the term *schema* is different from the use of the same term in the SQL Standard.[2]) A schema for a purchase order, for example, might specify that a purchase order consists of a date, a customer, a ship-to address, an optional bill-to address, and an array of one or more items that in turn contain lower-level data structures. A schema for a given type of document specifies the degree of flexibility that is allowed in constructing documents of that type, such as alternative content, optional content, and constraints on the number of occurrences of various parts. Within the constraints allowed by a schema, markup is used to identify the structure of an individual document. At one extreme, the structure of a document can be completely specified and constrained by its schema. At the other extreme, a document may have no schema at all and may rely entirely on markup for its metadata.

XML provides separate specifications for inserting descriptive markup into an individual document[3] and for creating a schema that describes the (possibly flexible) structure of a class of documents.[4] An XML schema corresponds roughly to a

relational system catalog, whereas XML markup is a form of metadata that is largely absent and unnecessary in relational systems.

Although both XML and the relational data model are completely general in their ability to represent all types of data, differences exist in the types of data typically stored in these respective formats. Some of these differences are as follows:

- XML data is often sparse—that is, a given data element may contain only some of many possibilities. Sparse data can also be represented in relational format, but relational representations of sparse data tend to be more complex and inefficient because they lack the self-describing property of XML.
- Relational data has no intrinsic order that is independent of its values, but XML is often used to store intrinsically ordered data, such as paragraphs in a book.
- XML documents often contain text, which increases the importance of specialized forms of search. Text search requires linguistic operations, such as stem matching, and often needs to combine precise with imprecise forms of search in a single query. Relevance ranking is an important form of search in XML data.
- Compared to a typical relational system catalog, XML schema information is often more complex and subject to change. An XML query may operate over multiple documents conforming to different schemas or to multiple versions of a schema. Some documents may not have a schema. An XML database must be prepared to cope with *schema evolution*—an environment in which schema information is heterogeneous and rapidly changing.

COMPARING SQL AND XQUERY

The first languages to be widely used in retrieving information from XML documents were XPath[5] and XSLT.[6] XPath was designed as a notation for navigating within an XML document, which is structured as a hierarchy of elements and attributes. XPath can isolate the elements and attributes that satisfy a given search criterion, but it is limited in that it cannot construct a new element. For this reason, it is not a complete query language. XSLT is more powerful than XPath, but was designed primarily for transforming one document into another. The expressive power of XSLT is sufficient for a query language, but its recursive pattern-matching paradigm is difficult to optimize and is better adapted for document transformation than for queries.

Recognizing the limitations of XPath and XSLT, the World Wide Web Consortium (W3C**)[7] organized a workshop in 1998[8] to begin consideration of a new query language for XML data sources. One outcome of the workshop was the formation of a new W3C working group on XML Query, which has produced a draft specification of a new language called XQuery. Participants in the XML Query Working Group have included database experts and information retrieval specialists, engineers and language theorists, users, and software vendors. Some of the reasoning that shaped the basic architecture of XQuery has been documented in Reference 9. The XQuery specification[10] has now completed its Last Call period, and in November 2005, it was adopted as a W3C Candidate Recommendation, an important step in the W3C standardization process. In the meantime, several software vendors have developed products based on the evolving XQuery specification. A current list of XQuery implementations is maintained on the XML Query Working Group's public Web page.[11]

■ In understanding the need for a new approach to storing and retrieving data, the concept of "metadata" is crucial ■

One of the first activities of the XML Query Working Group was to define a formal representation for XML data called the *XQuery 1.0 and XPath 2.0 Data Model*.[12] This data model served as the basis for the development of XQuery. In this paper, we refer to this data model simply as the XQuery Data Model (XDM).

Another early activity of the XML Query Working Group was to evaluate the possibility of extending SQL to serve as a query language for XML data sources. Because of XML's hierarchic structure, intrinsic ordering, and integrated metadata, the working group concluded that XML would be better served by a new query language rather than by extensions to SQL (for a more complete analysis of

this decision, see Reference 9). Nevertheless, in comparing XQuery with SQL, it is apparent that the languages have many similarities:

- Both are declarative query languages (though arguably, XQuery is somewhat less declarative than SQL).
- Both are functional languages defined in terms of a set of expressions that are closed under a specific data model.
- The two languages are roughly equivalent in expressive power (both support joins, quantification, recursion, and user-defined functions, but not second-order functions).
- Both languages have type systems that include simple and complex data types and inheritance.
- Both languages have set-oriented operators, including union and intersection, as well as set-oriented search operators. To compare these set-oriented search facilities, the following examples illustrate how the two languages would express a query against a database of orders and items to find the dates on which a customer named `Jones` ordered an item with the description `hat`:

SQL:

```
SELECT date
FROM order, item
WHERE order.customer = 'Jones'
AND order.orderno = item.orderno
AND item.description = 'hat'
```

XQuery:

```
/order[customer = "Jones"] [item[description =
  "hat"]]/date
```

A closer examination of the two languages, however, reveals that they also have many significant differences:

- XML data, unlike relational data, has an intrinsic order. This fact affects the design of XQuery in many ways, including positional predicates, "before" and "after" predicates, and operators, such as path expressions, that preserve document order.
- Relational databases represent information only by values, whereas XML also uses the concept of nesting (element hierarchies). As a result, some joins that would be explicit in SQL are implicitly represented by path expressions in XQuery. In addition, some kinds of search that would be complex and recursive in SQL can be represented compactly by using XQuery operators such as //, which searches for an object at an unknown depth in a hierarchy.
- Because XML mixes markup with data, it is possible in XQuery to express queries that span both data and metadata, such as, "What kinds of things are red?" Data and metadata are separated in relational databases; therefore, SQL does not provide a facility for expressing this kind of query.
- SQL has a null value, which is needed because every row of a table has the same set of columns. XQuery, on the other hand, has no notion of a null value because XML permits missing data to be represented by elements that are empty or simply absent.
- SQL logical operators (`and`, `or`, and `not`) are three-valued because of the presence of nulls (comparison of a null to any other value returns the "unknown" truth value). XQuery, on the other hand, uses traditional two-valued Boolean logic. Certain XQuery operators (such as `eq`) return an empty sequence when one operand of a comparison is an empty sequence. The XQuery operators `and` and `or` treat the empty sequence as false. XQuery users desiring three-valued logic are free to define their own logical operations as user-defined functions.
- XQuery has two sets of comparison operators, called *value comparisons* (which operate on single values) and *general comparisons* (which operate on sets and look for any matching pair). Thus, in XQuery, `author = "Gray"` is true if any author of a given work is equal to Gray, whereas `author eq "Gray"` requires that Gray be the only author. This distinction is important in XML, where the cardinality of an expression such as `author` may vary from one element to the next. In SQL, on the other hand, a value expression always returns a single value. SQL has the notion of a *quantified predicate* such as `= ANY` or `= ALL`, but these predicates can be used only in restricted circumstances (when the right side of the comparison is a table subquery).
- XQuery has a concept of *identity* that is not present in SQL. In XQuery, *nodes* (which correspond to XML concepts such as elements and attributes) have identity, but *atomic values* (such as `47` and `"Hello"`) do not. The concept of identity affects the XQuery language in several ways. The

language has expressions called *constructors* that create new nodes with new identities. Also, XQuery set operators, such as `union` and `intersect`, and path expressions eliminate duplicates from their results based on node identity rather than on value, as in SQL.

- In SQL, every data item has a data type, such as `INTEGER` or `VARCHAR(100)`, that governs its behavior in various expressions. XQuery supports, in addition to specific data types resembling those of SQL, the notion of untyped data. Untyped data is typically found in documents that have no schema. XQuery provides special rules for handling untyped data. Generally, for usability and compatibility with XPath 1.0, XQuery attempts to cast untyped data to an expected type based on the context where it is encountered. For example, untyped data encountered in an arithmetic expression is cast to the type `xs:double`.
- SQL supports overloaded functions and performs function selection based on the data types of the operands. XQuery, on the other hand, does not support overloaded functions. Each XQuery function (identified by its function name and number of parameters) has a well-defined signature, and an XQuery function call attempts to coerce its arguments to the expected types. This approach is consistent with the XQuery principle of operating on untyped data and casting it to an expected type based on its use.
- Names are more complex in XQuery than in SQL because they conform to the conventions of QNames in XML namespaces.[13] For example, the XQuery names `a:foo` and `b:foo` might match if the namespace prefixes `a` and `b` are both bound to the same namespace Uniform Resource Identifier (URI).
- The type system of XQuery, which is based on XML Schema,[4] has many more primitive data types than the SQL-type system and it also has a different inheritance model. In SQL, user-defined data types are encapsulated and can be inspected only by their observer methods. XML values, on the other hand, are not encapsulated, and their state can be freely inspected. XQuery supports derivation by extension, which corresponds roughly to the subtype concept in SQL, and also supports other XML Schema concepts, such as derivation by restriction and substitution groups. These concepts, required for compatibility with XML Schema, contribute greatly to the complexity of XQuery.

The remainder of this paper is organized as follows: the next section introduces a set of SQL extensions called SQL/XML, which are designed to bridge the gap between SQL and XQuery, allowing the languages to work closely together. The paper then describes how SQL/XML and XQuery are supported in IBM DB2* Universal Database* Version 9.1 on the Linux**, UNIX**, and Windows** platforms, and it discusses some of the challenges that arise in interfacing SQL and XQuery due to their differences in low-level syntax, semantics, and type systems. It then concludes by summarizing how SQL/XML allows users to combine the advantages of SQL and XQuery, using each language where it is most appropriate.

■ A set of SQL extensions called SQL/XML bridges the gap between SQL and XQuery, allowing the languages to work closely together ■

BRIDGING SQL AND XQUERY: SQL/XML

With the increasing acceptance of XML as a standard data format in many applications, the need is growing for systems that can store and manage persistent XML data while supporting a full set of tools and infrastructure that include backup and recovery, concurrency, and access control. Of course, XML data can be stored in special-purpose database systems designed exclusively for that purpose, but storing XML data in a relational database system makes it possible to reuse the extensive existing infrastructure. Because the database industry has a huge investment in SQL implementations, application development tools, and packaged applications, there is a great deal of interest in extending SQL-based systems to handle the storage and manipulation of XML data along with relational data. The resulting hybrid database systems will allow XML and relational data to coexist and complement each other in an enterprise information architecture.

In addition to capabilities for storing and retrieving XML data, SQL-based systems usually offer another important XML-related feature: an ability to transform (or "publish") relational data in XML format.

The data to be published is extracted from tables by SQL queries and may be either native relational data or data that was derived by shedding (decomposing XML documents into rows and columns). It is easy to see why a publishing facility is important: Vast quantities of business data are stored in SQL database systems, and there is a great demand for presenting this data in XML form to various client applications. Converting relational data to XML format requires extensions to SQL.

Because some of the techniques for handling XML data in relational systems require SQL language extensions, the organizations responsible for the standardization of SQL (the ANSI INCITS H2 committee in the United States and the ISO/IEC JTC1 SC32 WG3 committee internationally) have initiated activities to standardize the required extensions. These efforts led to the publication in 2003 of a new part of SQL, Part 14: XML-Related Specifications (SQL/XML),[14] referred to in this paper as SQL/XML:2003. A revised version, referred to in this paper as SQL/XML:2006, is expected to appear in mid 2006.[15] The approach taken by the SQL/XML specification is consistent with the decision of the W3C XML Query Working Group to develop a new query language for XML, because it embeds the XQuery language in SQL and relies on it for querying XML data stored in relational systems.

The primary goal of SQL/XML is to act as a bridge between SQL and the XML world, which includes the XML standard itself,[3] XQuery,[10] and XML Schema.[4] SQL/XML provides SQL language extensions in the following major categories:

XML data type—SQL/XML introduces a new SQL data type called XML for storing XML documents and provides a set of functions for converting between the new data type and other SQL data types.

Publishing functions—SQL/XML provides a set of functions for publishing relational data in XML format.

Query functions—SQL/XML provides a set of functions for embedding XML queries inside SQL queries.

In addition, SQL/XML provides a set of mapping rules that specify how names, values, and data types can be mapped between the SQL and XML worlds, which follow quite different conventions. SQL/XML:2003 specified many of the language extensions listed previously, and the remaining extensions will be covered by SQL/XML:2006. In the following sections, we briefly describe these facilities, with emphasis on how they are used in IBM DB2. The reader is referred to Reference 15 for a more complete description.

The examples in the remainder of this paper are based on a relational database containing three tables. In the `books` table, each row represents a book, and the content of the book is contained in a column of type XML. The `articles` table contains ordinary relational data about articles in journals. The `authors` table contains information about the authors of articles and books. Each row of the `authors` table contains an association between one author and one book or article, identified by its `pubid` as a foreign key. The SQL `CREATE TABLE` statements below show the details of the column names and data types in the three tables.

```
CREATE TABLE books(
  id CHAR(20) NOT NULL PRIMARY KEY,
  bookdoc XML)

CREATE TABLE articles(
  id CHAR(20) NOT NULL PRIMARY KEY,
  title VARCHAR (250),
  year INTEGER,
  journal VARCHAR (200));

CREATE TABLE authors(
  id CHAR (20) NOT NULL PRIMARY KEY,
  name CHAR (60),
  affiliation CHAR (50),
  pubid CHAR (20));
```

XML data type

SQL/XML provides a new SQL built-in data type called XML. Each instance of the XML data type encapsulates an instance of the XDM—that is, it contains a sequence of zero or more nodes or atomic values. (In SQL/XML:2003, the XML type was based on an extended version of the XML Infoset.[16] The basis for the XML type will change to the more flexible XDM in SQL/XML:2006.) A value of the XML data type is distinct from a textual representation of the same XML data. The XML data type can be used in the same ways as any other SQL data

type, such as to specify the type of a column or a function parameter.

The following example creates an SQL table named books with two columns. The column named id is of type VARCHAR(20), and the column named bookdoc is of type XML:

```
CREATE TABLE books (
  id VARCHAR(20) NOT NULL PRIMARY KEY,
  bookdoc XML)
```

SQL/XML provides a set of modifiers that can be used with the XML data type to constrain its instances in various ways—for example, XML(DOCUMENT) denotes an XML data type that is constrained to contain only well-formed XML documents. DB2 does not support these explicit type modifiers. In DB2, each instance stored in a column of type XML must be a well-formed XML document, but the result of a query or view definition may be any XDM instance.

As the XML data type is distinct from other SQL data types, such as VARCHAR and CLOB, SQL/XML provides a set of functions that can be used for converting between XML and other data types and for performing other useful operations on instances of the XML data type. These functions are summarized in the following subsections.

XMLParse

The XMLParse function converts an SQL character or binary string into an instance of the XML data type. It parses the input string according to the rules of XML Version 1.0[3] and returns a value of type XML. The syntax of the XMLParse function is illustrated by the following example, which inserts a value into an XML column. The value is obtained by parsing the string contained in the variable :h_var, which must be of an SQL character or binary string data type (CHAR, VARCHAR, BLOB, or CLOB). The required keyword DOCUMENT indicates that the input string must contain a well-formed XML document. The optional keywords PRESERVE WHITESPACE indicate that all white-space characters in the input string are preserved in the stored document.

```
INSERT INTO books (id, bookdoc) VALUES ('1256',
  XMLParse (DOCUMENT :h_var PRESERVE WHITESPACE))
```

A potential problem with statements such as the one in the previous example is that DB2 performs code-page conversions on character strings when they are exchanged between the client application and the database server if the client and server are using different code pages. Code-page conversions should be avoided in the case of XML data in order to avoid introducing inconsistencies with the XML encoding declaration. To avoid code-page conversions, a user can use a binary host variable (BLOB) to contain XML data or a special syntax in the SQL DECLARE SECTION to specify that a host-language variable contains XML data. When a variable declared in this way is used as an input variable, no code-page conversion is performed, and the content of the variable is

■ The most important new functionality in SQL/XML:2006 is the ability to embed an XQuery expression inside an SQL statement ■

automatically parsed into a value of type XML. The syntax for declaring a host variable containing XML data is illustrated in the following example:

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL TYPE IS XML AS CLOB (1M) hv_book;
EXEC SQL END DECLARE SECTION;
```

XMLSerialize

The XMLSerialize function is the inverse of XMLParse—it converts a value of type XML into an SQL character or binary string. It serializes the input XML value into a value of type CHAR, VARCHAR, BLOB, or CLOB by using the serialization rules of XQuery.[17] The syntax of the XMLSerialize function is illustrated by the following example, which selects the content of a book and serializes it as a value of type CLOB:

```
SELECT XMLSerialize (bookdoc AS CLOB)
FROM books
WHERE id = '2457';
```

Serializing an XML value into a character string host variable has the same potential problem with code-page conversion described earlier for XMLParse. Again, the solution to this problem is to serialize into a variable of a binary data type or to use the XML AS CLOB notation in the SQL DECLARE SECTION. Whenever a variable that is declared using XML AS

CLOB is used as an output variable, its content is automatically serialized, and no code-page conversions are performed.

XMLCast

The XMLCast function is a variation of the SQL CAST function in which either the source or target data type is XML. In the following example, an SQL integer value is cast into the XML data type, resulting in an XDM instance that contains a single item of type xs:integer:

```
XMLCast (1234 AS XML)
```

It is important to distinguish the semantics of XMLParse from those of XMLCast and XMLSerialize. Consider an SQL character string containing the following content:

```
<part color="red">Gear</part>
```

If this string is passed as input to the XMLParse function, the result will be an instance of the XML data type containing a document node, an element node, an attribute node, and a text node. On the other hand, if the same string is passed to the XMLCast function with the target data type XML, the result will be an XML item containing the original input string as an instance of xs:string.

If the input to XMLCast is an XML value and the target data type is an SQL type, the function first converts the input value to an atomic value by using XQuery atomization rules.[10] The resulting atomic value must have a type that can be converted to the SQL target data type according to the mapping rules in the SQL/XML specification; otherwise, an error is raised. Thus, if the XML value produced by parsing the above example string were passed to XMLCast with a target data type of VARCHAR (100), the resulting value would be Gear. On the other hand, if the same XML value were passed to XMLSerialize, the result would be the original string: <part color="red">Gear</part>. If both the input data type and the target data type are XML, XMLCast returns references to the input nodes without actually copying the nodes.

XMLValidate

The XMLValidate function validates an instance of the XML data type according to the validation rules of XML Schema.[4] If the input value is a valid document as defined by the given schema, the function returns a copy of the input value in which the element and attribute nodes have been augmented with default values and type annotations. If the input is not a valid document as defined by the given schema, an error is raised.

XML schemas are commonly represented as serialized XML documents, frequently in a file with the extension xsd that resides at some URL outside the database. It is considered undesirable for a database server such as DB2 to obtain an XML schema over the Web, where its contents can change at the whim of the owner or may simply become unavailable. Therefore, DB2 requires users to register all XML schemas to be used for validation and provides an interface for this purpose. The list of registered schemas is maintained in a DB2 catalog table.

A registered XML schema can be identified by an SQL identifier or by the URI of its target namespace. The schema to be used in an XMLValidate invocation can be specified explicitly in the function call or determined from information contained in the input document, as illustrated by the following examples:

1. This example identifies a registered schema by an SQL identifier:

```
XMLValidate (DOCUMENT X
  ACCORDING TO XMLSCHEMA ID SCOTT.BOOKS)
```

2. This example identifies a registered schema by its target namespace URI:

```
XMLValidate(DOCUMENT X ACCORDING TO
  XMLSCHEMA URI 'http://example.books.com')
```

3. In this example, because no schema is specified, validation is performed against the registered schema identified by the xsi:schemaLocation attribute of the root node of the input document:

```
XMLValidate(DOCUMENT X)
```

Publishing functions

As noted earlier, SQL/XML and DB2 provide a set of functions for converting ("publishing") relational data into XML format. These functions are also referred to as XML constructor functions, as their purpose is similar to that of computed node constructors in XQuery. In this section, we describe these functions and provide examples of their use. All the examples in this section are based on the

Table 1 ARTICLES

ID	TITLE	YEAR	JOURNAL
ID0001	Web and XML	2001	ACM J1
ID0002	XQuery Support in DB2	2005	ACM J1
ID0003	SQL/XML Progress Report	*null*	IEEE J2
ID0004	XSLT and XQuery	2004	Journal X

`articles` table (***Table 1***). When an example query is followed by the symbol =>, the text following the => symbol is the result of the query when executed against the data shown here (query results have been serialized with added white space for readability).

XMLElement

`XMLElement` mirrors the functionality of the XQuery element constructor in that its output is an XDM element node. `XMLElement` is at the heart of the XML publishing functions as it allows for creating an XML element in any shape or form. Its variable parameter list makes it possible to create simple constructs, such as an empty element, or complex constructs, such as many nested elements with attributes or namespace declarations at each nesting level.

The first argument of `XMLElement` is an SQL identifier that serves as the name of the element to be constructed. This required argument can be followed by a variable number of optional arguments that specify namespace declarations, attribute definitions, element content, and a null handling option. If more than one optional argument is supplied, the arguments must be supplied in the just mentioned order. The arguments of `XMLElement` may be provided in any form, including constants, value expressions, subqueries, and calls to other SQL/XML functions. The following example illustrates the simplest form of `XMLElement`, which creates an empty element node.

```
VALUES ( XMLElement ( NAME "Journal" ))

=>

<Journal/>
```

The following example illustrates a more complex invocation of `XMLElement` whose content includes nested invocations of `XMLElement` to construct nested elements.

```
SELECT XMLElement (
 NAME "article"
 XMLElement ( NAME "title", title )
 XMLElement ( NAME "journal", journal ),
 XMLElement ( NAME "year", year))
FROM articles
WHERE id = 'ID0001'

=>

<article>
 <title>Web and XML</title>
 <journal>ACM J1</journal>
 <year>2001</year>
</article>
```

As long as at least one of the values specified for the content of the constructed element is non-null, the element is constructed, and any null values are ignored (i.e., they are dropped from the result). If all of the values specified for the content of the constructed element are null, the result of `XMLElement` is defined by the final parameter, which specifies a null handling option. The following options are supported by DB2:

- `EMPTY ON NULL` (the default behavior): An empty element is constructed and returned.
- `NULL ON NULL`: No element is constructed, and an SQL null value is returned.

The null handling option applies only for the `XMLElement` call in which it is specified. It does not change the behavior of any contained `XMLElement` call or any other value expression.

Unlike XML and XQuery, SQL is not (always) case sensitive. As a result, close attention must be paid to uppercase and lowercase in function calls, such as `XMLElement`, that bridge the two languages. For example, in `XMLElement`, the argument supplied for the name of an XML element is an SQL identifier, which is always made uppercase by the SQL engine unless it is delimited by double quotation marks (a "delimited identifier"). A user wishing to create an element named `Journal` might be surprised that `XMLElement(NAME Journal)` creates an element named `JOURNAL` instead. The desired result can be

obtained by using a delimited identifier: `XMLElement (NAME "Journal")`.

XMLAttributes

One or more attributes can be specified for an XML element by means of the `XMLAttributes` function. For each attribute, this function associates a value specified by a value expression with an attribute name specified by an SQL identifier. The attribute name is optional if the value expression identifies a column of a table. If an attribute name is not specified, the column name serves as the attribute name. If the value of an attribute is the null value, then the attribute is not included in the result. To comply with the XML Recommendation,[3] no two attribute names can be identical. The following example shows how `XMLAttributes` can be nested inside `XMLElement` to create an element with three attributes (in this case, the element has no content):

```
SELECT XMLElement (NAME "article",
 XMLAttributes (journal AS "journal",
                title AS "title",
                year AS "year" ))
FROM articles
WHERE id = 'ID0001'

=>

<article journal = "ACM J1"
         title = "Web and XML" year = "2001"/>
```

XMLNamespaces

Although a namespace declaration is similar in appearance to an attribute definition, it has different semantics. Therefore, namespace declarations are created by a separate function called `XMLNamespaces`. A call to `XMLNamespaces` can create one or more namespace declarations, each of which binds a namespace URI, supplied by a character string literal, to a namespace prefix, supplied by an SQL identifier, as illustrated by the following example:

```
SELECT XMLElement (NAME "lib:article",
  XMLNamespaces
   ('http://example.com/library' AS "lib"),
  XMLAttributes ('yes' AS "lib:bestpaper"),
  XMLElement (NAME "lib:journal", journal),
  XMLElement (NAME "lib:title", title))
FROM articles
WHERE title = 'Web and XML'

=>

<lib:article xmlns:lib="http://example.com/
   library" lib:bestpaper="yes">
 <lib:journal>ACM J1</lib:journal>
 <lib:title>Web and XML</lib:title>
</lib:article>
```

Another form of namespace declaration specifies the default namespace that applies to all unprefixed element names within the scope of a given element. In a call to `XMLNamespaces`, a default namespace can be specified by the keyword `DEFAULT` followed by a namespace URI, or, if the element has no default namespace, by the keywords `NO DEFAULT`.

A namespace prefix (or default namespace) defined by `XMLNamespaces` is valid inside the `XMLElement` in which it is specified. Nested calls to `XMLElement` inherit the namespace prefixes defined by the outer `XMLElement`, unless it is overridden by a nested call to `XMLNamespaces`. The following example illustrates how default namespaces can be controlled by nested calls to `XMLElement` and `XMLNamespaces`. The default namespace specified for the `article` element is valid inside the `article` and `journal` elements, but not inside the `title` element, which has no default namespace.

```
SELECT XMLElement ( NAME "article",
  XMLNamespaces
   ( DEFAULT 'http://example.com/library'),
  XMLElement ( NAME "journal", journal),
  XMLElement ( NAME "title",
    XMLNamespaces ( NO DEFAULT ),
    title ) )
FROM articles
WHERE title = 'Web and XML'

=>

<article xmlns="http://example.com/library">
  <journal>ACM J1</journal>
  <title xmlns="">Web and XML</title>
</article>
```

To comply with Namespaces in the XML Recommendation,[13] the prefixes `xml` and `xmlns` are always implicitly bound to `http://www.w3.org/XML/1998/namespace` and `http://www.w3.org/2000/xmlns`, respectively, and cannot be bound to any other namespace URI. Similarly, no other namespace

prefix can be explicitly bound to either of these URIs.

XMLForest

XMLForest provides a simple mechanism for creating a sequence of XML element nodes. As the name of the function implies, these element nodes are not rooted in a common top-level element, distinguishing this function from XMLElement, which always generates exactly one top-level element node.

The argument to XMLForest is a list of one or more value/name pairs that are used to construct the element nodes. Each element content is specified by a value expression whose declared type can be any SQL predefined data type (including XML). Each element name is specified by an SQL identifier. If the value expression is a column reference, then the name can be omitted, and the column name serves as the element name. If the value of an element is null, then that element is not included in the result. By default, if all element values are null, then XMLForest returns the null value. XMLForest is illustrated by the following example:

```
SELECT XMLForest (
  journal AS "journal",
  title,
  year)
FROM articles
WHERE title = 'Web and XML'

=>

<journal>ACM J1</journal>
<TITLE>Web and XML</TITLE>
<YEAR>2001</YEAR>
```

Like XMLElement, XMLForest accepts a call to XMLNamespaces as its first parameter, thus declaring a set of namespaces that are valid inside all of the element nodes created by XMLForest. Also like XMLElement, XMLForest accepts a null handling option at the end of its parameter list that specifies the handling of constructed elements whose content is null. The options are NULL ON NULL and EMPTY ON NULL, with the same meanings as the equivalent options in XMLElement. However, in XMLForest, the default null handling option is NULL ON NULL, which is the opposite default of XMLElement. This difference in defaults can be a source of confusion when comparing a call to XMLElement with a call to XMLForest. Unlike XMLElement, XMLForest provides no syntax for specifying attributes for the constructed XML elements.

XMLConcat

The XMLConcat function takes a varying number of parameters, each of which is a (possibly null) XML value. It returns an XML value that is a sequence comprising all the non-null input values. If any of the input values are themselves sequences, the result of XMLConcat is "flattened" into a single level (it contains no nested sequences). If all the input values are null, XMLConcat returns a null value. XMLConcat might be thought of as the SQL/XML counterpart of the comma operator in XQuery.

■ Just as there is much interest in transforming SQL data into XML data, the reverse is also true ■

The semantics of XMLConcat are quite different from those of the existing DB2 function named concat, which is a string concatenation function. For example, concat ('abc', '123') returns the single string abc123. XMLConcat, on the other hand, operating on two XML values containing the strings 'abc' and '123', would return an XML value containing the sequence ('abc', '123'). Also, concat returns null if any of its input values is null, whereas XMLConcat returns null only if all its input values are null.

XMLAgg

XMLAgg is different from the other XML publishing functions in that it is an aggregate rather than a scalar function. In that respect it is similar to the sum function for numerical data types, which generates the sum of all values in a column in a given group of rows. Similarly, XMLAgg concatenates all the values of a given column in a given group of rows using the semantics of the XMLConcat function, i.e., null values are ignored and only non-null values contribute to the result. XMLAgg is often useful in grouping XML values at a particular nesting level and reconstructing XML values that have been shredded across multiple tables.

In the following example, a top-level XML element is generated that contains as subelements all of the

articles published in ACM J1. From the `articles` table, all rows qualify whose `journal` column contains the value 'ACM J1'. For each of these rows, an XML element is created whose name is `title` and whose content is the value in the `title` column. All of these XML elements are concatenated, and the resulting sequence forms the content of the top-level XML element named `articles-in-ACMJ1`.

```
SELECT XMLElement (
  NAME "articles-in-ACMJ1",
  XMLAgg (XMLElement ( NAME "title", title )))
FROM articles
WHERE journal = 'ACM J1'

=>

<articles-in-ACMJ1>
  <title>Web and XML</title>
  <title>XQuery Support in DB2</title>
</articles-in-ACMJ1>
```

As rows in a table are inherently unordered, the order of the subelements in the above example is not ensured. To force a certain order `XMLAgg` accepts an optional parameter with which the user can specify the order of the XML values before they are concatenated into a sequence. The following example shows how an ordering can be imposed on the subelements in the previous example:

```
SELECT XMLElement (
  NAME "articles-in-ACMJ1",
  XMLAgg ( XMLElement ( NAME "title", title)
           ORDER BY title DESC ) )
FROM articles
WHERE journal = 'ACM J1'

=>

<articles-in-ACMJ1>
  <title>XQuery Support in DB2</title>
  <title>Web and XML</title>
</articles-in-ACMJ1>
```

Other publishing functions

In addition to the functions listed above, SQL/XML provides publishing functions for comment nodes, processing instruction nodes, text nodes, and document nodes. Each of these functions closely resembles its counterpart node constructor in XQuery. These publishing functions are illustrated by the following examples:

1. This function call returns an XDM comment node:

```
XMLComment ('This is a comment')
```

2. This function call returns an XDM processing instruction node whose target is `telephone` and whose value is `ring`:

```
XMLPI (NAME "telephone", 'ring')
```

3. This function call returns an XDM text node whose string value is `Hello`:

```
XMLText ('Hello')
```

4. This function call returns an XDM document node with a child element node whose name is `color` and whose value is `Red`:

```
XMLDocument (
  XMLElement (NAME, "Color", 'Red'))
```

Query functions

Probably the most important new functionality in SQL/XML:2006 is the ability to embed an XQuery expression inside an SQL statement. This facility provides an easy way to query XML values stored in or generated by an SQL database. SQL/XML:2006 provides three functions for this purpose: `XMLQuery`, which executes an XQuery and returns a scalar value; `XMLExists`, which acts as a predicate and returns `true` or `false`, and `XMLTable`, which executes an XQuery and returns the result in the form of a table (and is therefore used in the SQL `FROM` clause). These functions are described in the following subsections.

XMLQuery

The `XMLQuery` function allows SQL to execute an XQuery expression, optionally passing named parameters to XQuery and receiving the result as a value of type XML. At the language level, the integration of SQL and XQuery is easily achieved, which shows the versatility of both languages. Some of the difficulties and challenges of this integration are discussed in more detail in the section "Syntactic and semantic challenges" later.

The simplest form of `XMLQuery` has a single argument: a character string literal containing the XQuery expression to be evaluated. Successful

execution of this expression returns an XML value (remember that an XML value is an XDM instance, which is a sequence of zero or more items). The XQuery expression is supplied as a character string literal so that it is known at SQL query compilation time and can be optimized together with the enclosing SQL statement.

The following example is a simple `XMLQuery` invocation that executes an embedded XQuery expression without parameters. The result of the query is a value of type XML containing a sequence of integers: 2, 6, 12, 20.

```
XMLQuery ('for $i in (1,2,3,4)
           let $j :=$i + 1 return $i * $j')
```

The `XMLQuery` function is much more useful when it passes a set of named parameters to the embedded XQuery expression. This can be done by means of a `PASSING` clause that specifies the name and value of each parameter.[18] The parameter name is an SQL identifier, which can be referred to inside the XQuery expression as a variable name (with a leading "$" sign). The parameter value may be any SQL expression. The result of evaluating this expression is converted to the XML data type by using the semantics of `XMLCast` (see the description of `XMLCast` in the section "XML data type") bound to the named variable and made accessible inside the XQuery expression.

The following example illustrates how `XMLQuery` can pass a parameter to an XQuery expression. For each row of the `books` table, the SQL query passes the content of the XML column `bookdoc` to XQuery as a parameter named `$book`. The `XMLQuery` function executes the path expression `$book/title` and returns the title of the book as a value of type XML.

```
SELECT XMLQuery ('$book/title'
  PASSING BY REF bookdoc AS 'book')
FROM books
```

In the above example, the `BY REF` keywords indicate that the `bookdoc` XML documents are passed by reference to XQuery. When XML values are passed by reference, no copy is made at the boundary crossing, and node identities and parent linkages are preserved. SQL/XML also defines another parameter-passing option called `BY VALUE`, which copies the nodes, losing node identities and parent linkages. To avoid unnecessary copies and to enable use of the parent axis, DB2 supports parameter passing only by reference. In DB2, the `BY REF` keywords are optional and are omitted in subsequent examples.

As XDM does not have a null value, each named input argument that is null is converted into an empty sequence before the XQuery expression is evaluated. On the other hand, if the result of the XQuery expression is an empty sequence, no conversion is necessary because an empty sequence is a valid XDM value and therefore a valid value of the XML data type.

■ DB2 unifies management of relational and XML data, supporting interfaces for both SQL/XML and XQuery in a unified query model ■

XMLExists

The `XMLExists` function is similar to `XMLQuery` in that it passes named parameters to XQuery and executes an XQuery expression expressed as a string literal. However, rather than returning the result of the XQuery expression, `XMLExists` serves as a predicate whose value is `false` if the result of the XQuery expression is an empty sequence and is `true` otherwise. The `XMLExists` function can be invoked wherever a predicate can be used in an SQL query (for example, in the `WHERE` clause or `HAVING` clause).

Note that if `XMLExists` is used to evaluate an XQuery expression that returns `true` or `false`, the `XMLExists` function itself will always return `true` (as neither `true` nor `false` is an empty sequence). This is a possible pitfall for unwary users. For example, suppose that a user wishes to find the number of books whose title contains the word "Frog." The following incorrect formulation of this query simply returns the total number of rows in the `books` table, because every invocation of `XMLExists` generates a nonempty result and returns `true`:

```
SELECT count (*)
FROM books
WHERE XMLExists ('contains ($book/title, "Frog")'
  PASSING bookdoc as "book")
```

Table 2. Example XMLTable output

year	title
1994	TCP/IP Illustrated
2004	XQuery from the Experts
2003	XML Data Management

The following formulation of the query is correct because, for books whose title does not contain the word "Frog," the embedded XQuery expression returns an empty sequence, the XMLExists predicate is false, and the row is not counted.

```
SELECT count (*)
FROM books
WHERE XMLExists ('$book [contains (title, "Frog")]'
  PASSING bookdoc as "book")
```

XMLTable

Just as there is much interest in transforming SQL data into XML data, the reverse is also true. As discussed earlier in this section, the shredding technique can transform an XML document with regular structure into one or more SQL tables based on schema information. The XMLTable function provides a more general mechanism for generating a table from XML data that can be applied dynamically and requires no schema information. As XMLTable is a table function, it is used in the FROM clause of an SQL query.

Like XMLQuery and XMLExists, XMLTable has arguments that specify an XQuery expression to be executed and a set of named parameters to be passed to the XQuery expression. However, rather than returning the result of the XQuery expression, XMLTable uses this result to construct a table. Each top-level item in the XQuery result generates one row of the table. The XMLTable invocation has a COLUMNS clause that specifies the columns to be generated. Each column is specified by a name, a data type, and a column-generating XQuery expression.

The following example uses XMLTable to generate a table containing the years and titles of books published by "Pub1":

```
SELECT x.year, x.title
FROM books b,
  XMLTable ('$book/book [publisher="Pub1"]'
    PASSING b.bookdoc AS "book"
    COLUMNS
      "year" INTEGER PATH '@year',
      "title" VARCHAR(60) PATH 'title')
    AS x(year, title)
```

In the above example, each invocation of XMLTable returns a table named x with columns year and title, computed as follows:

1. The XQuery expression $book/book [publisher="Pub1"] is evaluated, producing a sequence of items. In this example, the sequence will be of length zero or one, depending on the publisher of the book passed to XMLTable
2. Each item in the sequence produced by the previous step is used to compute a row of the table. During the computation of this row, the given item serves as the XQuery *context item* (the beginning point for path expressions).
3. To generate the value for a particular row and column, the column-generating expression for that column is evaluated with the context item described in the previous step. The result is then cast to the data type specified for that column.

If a column-generating expression is omitted, the column value is generated by an implicit path expression using the column name as a name test. Thus, the column-generating expression PATH 'title' can be omitted from the previous example, as follows:

```
SELECT x.year, x.title
FROM books b,
     XMLTable ('$book/book [publisher="Pub1"]'
        PASSING B.BOOKDOC AS "book"
       COLUMNS
           "year" INTEGER PATH '@year',
           "title" VARCHAR (60))
     AS x(year, title);
```

Table 2 shows what the output of the above example might look like. In this example, an SQL delimited identifier is used for the lowercase column name title to make sure that SQL and XQuery interpret the names in the same way, as XQuery names are case-sensitive and SQL names are case-sensitive only inside delimited identifiers.

NATIVE XML IN DB2

DB2 provides native XML storage, indexing, and query processing through both XQuery[10] and SQL/

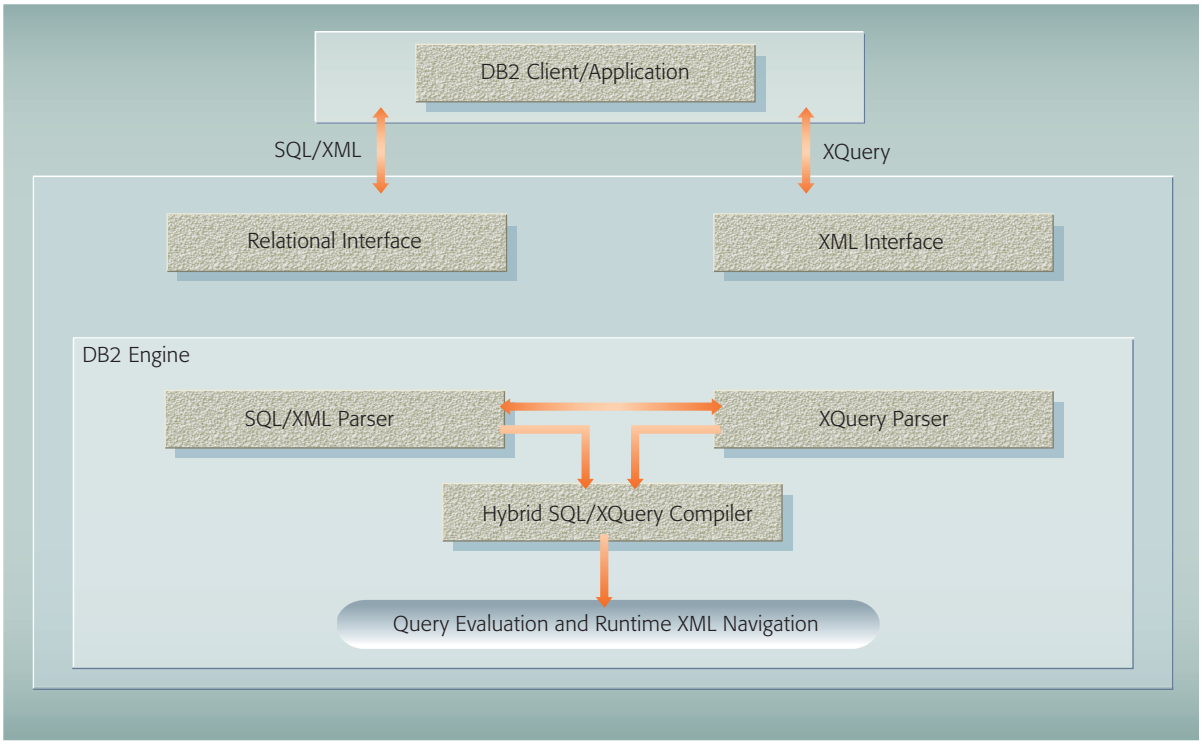

Figure 1
IBM DB2 architecture overview

XML[15] by using the XML data type introduced by SQL/XML. DB2 unifies management of relational and XML data, supporting interfaces for both SQL/XML and XQuery in a unified query model, as shown in *Figure 1*.

DB2 implements all of the SQL language extensions of SQL/XML:2006 (with some minor exceptions), including the XML-data-type `XMLParse`, `XMLSerialize`, `XMLValidate`, and `XMLCast` functions, as well as the publishing and querying functions. Users can query XML data by using XQuery directly or by calling SQL/XML query functions. They can also query both relational and XML data within the same statement by using XQuery or SQL/XML. After parsing, both XQuery and SQL/XML queries are mapped into a unified internal representation and handled by the hybrid query compiler.[19] DB2 exploits this unified model to perform powerful cross-language optimizations. XDM[12] plays a vital role in this process because operating on the same data model allows XML values to be exchanged between SQL and XQuery, providing a seamless integration of the two languages.

DB2 stores XML data in columns of tables. The physical storage format for the XML data type preserves all the information in XDM. An important feature of DB2 is that it does not require an XML schema to be associated with an XML column. An XML column can store documents validated according to many different and evolving schemas, as well as schema-less documents. Hence, the association between schemas and XML documents is on a per-document basis, providing maximum flexibility.

The detailed description of native XML support in DB2 can be found in References 19, 20, and 21. In the following sections, we show how to insert, index, validate, and query XML data in DB2. The examples in these sections are based on the `books`, `articles`, and `authors` tables defined in the previous section.

Storing, validating, and indexing XML data in DB2

DB2 supports the creation of tables having one or more columns of the new XML data type defined by SQL/XML. This enables existing SQL applications to augment their current relational database designs

with additional XML data and provides an evolutionary path for XML support. Currently, each XML instance stored in DB2 must be a well-formed XML document. Conceptually, in an XML column, each row contains an XML document, represented as an XDM instance.

XML columns are created like other SQL columns, and their data is inserted and deleted using SQL `INSERT` and `DELETE` statements. Because XQuery does not yet include update syntax, DB2 only supports full document replacements at present.

To insert an XML document into a table, it must be parsed and converted into native XML storage format. The following example shows how this can be accomplished using the `XMLParse` function:

```
INSERT INTO books (id, bookdoc)
  VALUES ('2457',
   XMLParse (DOCUMENT
     '<?xml version="1.0" encoding="UTF-8"?>
     <book year="1994">
        <title>TCP/IP Illustrated</title>
        <publisher>Addision-Wesley</publisher>
        <price>65.60</price>
        <chapter>
          <title>Introduction</title>. . .
        </chapter>. . .
     </book>'))
```

Note the `bookdoc` column in the `books` table is of type XML but is not associated with any particular XML schema. An XML schema is not required in order to define an XML column or to insert or query XML data.

If the XML document has an associated schema, it can be validated by specifying the schema during insertion. Before an XML schema can be used for validating documents, it must be registered with the database. DB2 provides an XML Schema Repository (XSR)[20,21] to manage XML schemas. Registration of XML schemas with the XSR is done through DB2 commands, stored procedures, or language-specific application programming interfaces (APIs). The following example illustrates registering a schema with a DB2 command:

```
REGISTER XMLSCHEMA
  http://example.books.com FROM books.xsd
AS bookSchema COMPLETE
```

In this example, the target namespace URI of the XML schema is `http://example.books.com`, the file that contains the schema document is `books.xsd`, and the SQL identifier associated with this schema is `bookSchema`.

To validate an XML document during insertion with a given XML schema, we use the SQL/XML functions `XMLParse` and `XMLValidate`. The document is validated after parsing, as shown in the following example:

```
INSERT INTO books
  VALUES ('2457',
    XMLValidate (DOCUMENT
      XMLParse (DOCUMENT
        '<?xml encoding="UTF-8"?>
        <book year="1994">
            <title>TCP/IP Illustrated</title>
            <publisher>Addison-Wesley</publisher>
            <price>65.60</price>
            <chapter>
              <title>Introduction</title>. . .
            </chapter>. . .
        </book>')
      ACCORDING TO XMLSCHEMA ID bookSchema))
```

In this example, the schema to be used in validation was identified explicitly by the SQL identifier `bookSchema` that was passed to `XMLValidate`. If no schema had been explicitly named in the call to `XMLValidate`, the document would have been validated by using the registered schema identified by the `xsi:schemaLocation` attribute of the root element (if no such attribute existed, an error would be raised).

When a document is validated against an XML schema, DB2 not only checks the document for validity but also annotates the nodes in the XML document with type information. XML nodes in a validated document are annotated with their primitive data types, and the runtime system uses this information for dynamic dispatch of functions during query processing. If an XML document is not validated, then all of its element nodes are annotated as `xdt:untyped`, and all of its attribute nodes are annotated as `xdt:untypedAtomic`. Type annotations of nodes play an important role in query processing. For example, a sequence of `price` attributes is sorted as strings if the book documents are not validated, whereas they would be sorted as

decimal values if they were validated and had the type annotation `xs:decimal`.

DB2 supports indexes on particular XML path expressions called *xmlpatterns*, which may contain wildcards, descendant axis navigation, and kind tests. As DB2 does not require a single XML schema for all documents in an XML column, and XML documents in a column may contain both typed and untyped values, DB2 may not know which data type to use in the index for a given xmlpattern. Moreover, untyped values have polymorphic behavior in XQuery predicates: They are cast to the type of the other operand. As a result, the user must explicitly specify a key data type in each `create index` statement, and the values of all elements and attributes represented in the index are cast to this key data type. Because relational and XML indexes share a common implementation, the key data type of an XML index must be an SQL data type, and the result of the index expression must be castable to this type. The following examples illustrate the creation of XML indexes:

```
CREATE INDEX pubIndex ON books (bookdoc)
  GENERATE KEY USING XMLPATTERN '/book/publisher'
  AS SQL VARCHAR (100);

CREATE INDEX yearIndex ON books (bookdoc)
  GENERATE KEY USING XMLPATTERN '//@year'
  AS SQL INTEGER;
```

In the first example, `publisher` element children of the top-level `book` element are indexed as strings, and in the second example, all `year` attributes in the documents are indexed as integers.

Querying XML data in DB2

A DB2 application can access XML data in several ways, using either SQL/XML[15] or XQuery.[10] DB2 allows invoking XQuery within SQL and vice versa. Queries can use `XMLQuery`, `XMLExists`, or `XMLTable` to invoke XQuery and retrieve XML fragments. DB2 also supports a stand-alone XQuery interface.

When XQuery is invoked as a top-level language, DB2 needs a hint to invoke the XQuery parser instead of the SQL parser. This is achieved by prefixing XQueries with the `xquery` keyword. When invoked through the SQL/XML functions `XMLQuery`, `XMLExists`, or `XMLTable`, XQuery gets its input data from the function parameters. When invoked as a top-level language, XQuery needs a source of input data. Because XML data is stored in relational tables, DB2 provides an input function called `db2-fn:xmlcolumn` to provide access to XML data. This function takes the name of an XML column in a relational table or view as an argument and returns the sequence of XML values stored in that column. For example, the following query operates on the `bookdoc` column of the `books` table, returning all chapter titles that contain the word `Czar`.

■ To retrieve XML documents that satisfy a particular condition based on their content, the XMLExists function can be used ■

```
xquery db2-fn:xmlcolumn("BOOKS.BOOKDOC")
  //chapter/title[contains (., "Czar")]
```

DB2 also provides another input function, `db2-fn:sqlquery`, to invoke an SQL query from within XQuery. This function takes in an SQL `SELECT` statement and returns an XML column as output. Note that an SQL query may return an XML result by invoking XQuery, by using SQL/XML publishing functions, or by selecting data from an XML column of a relational table or view. The `db2-fn:sqlquery` function is useful when users want to restrict the XML documents seen by XQuery based on some conditions on relational tables or views, or when they want to provide an inline XML view of their relational data.[22] The following query returns the titles of books written by John Doe. Note that the relational join between the `books` and `authors` tables and the relational condition on the `name` column of the `authors` table restrict the book documents input to XQuery.

```
xquery db2-fn:sqlquery("SELECT b.bookdoc
  FROM books b, authors a
  WHERE a.name = 'John Doe'
  AND b.id = a.pubid")/book/title
```

Conversely, it is possible to invoke XQuery from within an SQL query by using the `XMLQuery`, `XMLExists`, and `XMLTable` functions. Users can employ the `XMLQuery` function to retrieve fragments of XML documents. For example, the following

query constructs and returns `bookinfo` elements containing the book title and chapter titles of books in which any chapter title contains the word XML:

```
SELECT XMLQuery ('for $b in $book/book
    where contains ($b//chapter/title, "XML")
    return <book-info>
            {$b/title}
            <chapter-titles>
              {$b//chapter/title}
            </chapter-titles>
          </book-info>'
  PASSING bookdoc AS "book")
FROM books
```

The XML column `bookdoc` is passed to XQuery by means of the variable `book`. Execution of the XQuery expression extracts the title of book and chapters, applies the predicate, and constructs a new element `book-info`. Note that the `bookdoc` document nodes are passed by reference to XQuery, thus maintaining node identities across the language boundary and avoiding the copying of large documents.

The SQL query in the previous example returns a result for each row of the `books` table. For books that have "XML" in a chapter title, the result is an XML value containing a `book-info` element. For other books, the result is an XML value containing an empty sequence.[23]

To retrieve XML documents that satisfy a particular condition based on their content, the `XMLExists` function can be used. For example, the following query returns values of the `bookdoc` column that contain books written after 1996. In each case, the entire XML document is returned.

```
SELECT bookdoc
FROM books
WHERE XMLExists ('$book/book[@year > 1996]'
  PASSING bookdoc AS "book")
```

If it is desired to retrieve only fragments of XML documents that satisfy a given condition, the `XMLTable` function can be used. For example, the following query retrieves the titles of books written after 1996. In this example, each invocation of `XMLTable` is passed one book in XML form. If the `year` attribute of the book is not greater than 1996, `XMLTable` returns no rows. If the `year` attribute is greater than 1996, `XMLTable` returns a row containing the title of the book. (If a book element contains more than one title, then all titles of the book are returned as a sequence in a single row.) As the SQL query joins the `books` table with the result of `XMLTable`, only the titles of books written after 1996 appear in the query result.

```
SELECT t.booktitle
FROM books b,
  XMLTable('$book/book[@year > 1996]'
    PASSING b.bookdoc AS "book"
    COLUMNS
      "title" XML BY REF PATH 'title')
  AS t(booktitle)
```

Some applications may require grouping and analysis. Although XQuery does not yet have an explicit group-by construct, grouping queries can be expressed by using nested queries and self-joins. The following XQuery computes the number of books published by each publisher. This query contains a `let` clause that computes the group of books for each distinct publisher.

```
xquery
  for $pub in distinct-values(
    db2-fn:xmlcolumn('BOOKS.BOOKDOC')
      /book/publisher)
  let $group :=
    db2-fn:xmlcolumn('BOOKS.BOOKDOC')
      /book[publisher=$pub]
  return
    <result>
      {$pub}
      <count>{count($group)}</count>
    </result>
```

A similar query, which also returns the number of books for each publisher although in a slightly different format, is given in the following example. This query uses `XMLTable` to extract the publishers of each book and computes the groups on the SQL side.

```
SELECT t.publisher, count(*)
FROM books b,
  XMLTable('$book/book'
    PASSING b.bookdoc AS "book"
    COLUMNS
      "publisher" VARCHAR(100)) AS t(publisher)
GROUP BY t.publisher
```

In this example, for each book, XMLTable returns a computed one-column table containing the publisher of the book. By "breaking out" the publisher into a separate column, it becomes possible to use the grouping feature of SQL to count the number of books from each publisher. This query avoids scanning the books table twice and performing a self-join by exploiting the explicit GROUP BY construct of SQL.

Defining XML and relational views

DB2 provides both a relational and an XML interface. Pure relational applications can access XML data by creating relational views of XML data, and XML-centric users can query XML data and XML views of relational data through XQuery. Users can use SQL/XML publishing functions to create XML views of their relational data. The view defined in the following example consists of a table that contains only a single row and a single column of type XML. The XML value contained in this single-cell table consists of a bib element that contains many book elements. Each of the book elements in turn contains zero or more author elements, computed from relational data stored in the authors table.

```
CREATE VIEW bib(doc) AS
  (VALUES XMLELEMENT (NAME "bib",
    (SELECT XMLAgg(
      XMLElement (NAME "book",
        XMLATTRIBUTES (b.id),
        (SELECT XMLAgg(
           XMLElement (NAME "author",
             XMLAttributes (au.affiliation),
             XMLElement (NAME "name", au. name)))
        FROM authors au
        WHERE au.pubid = b.id),
        XMLQuery ('$book/book/*'
          PASSING b.bookdoc as "book")))
        FROM books b)))
```

For each row of the books table, the XMLQuery function extracts all the element children of the top-level book element in the XML column bookdoc. The enclosing XMLElement function creates a new book element that has the same element children as the original book element, but also has an id attribute and author element children. The view definition joins the books and authors tables to compute the authors of each book. An illustrative fragment of the resulting bib element is shown below.

```
<bib>
  <book id= "2457">
  <author><name>W. Richard Stevens</name></author>
  <title>TCP/IP Illustrated</title>
  <publisher>Addison-Wesley</publisher>
  <price>65.60</price>
  <chapter>
  <title>Introduction</title> ...
  </chapter>
  ...
  </book>
  ...
</bib>
```

Conversely, SQL/XML query functions can be used to create relational views of XML data. In particular, XMLTable can be used to define an on demand shredding of XML data into relational tables according to the needs of the application. The following view definition uses the XMLTable function to generate a relational table that contains the title, year, publisher, and price of each book, extracting this data from the stored XML content.

```
CREATE VIEW
  bookContent (title, year, publisher, price) AS
  (SELECT t.title, t.year, t.publisher, t.price
  FROM books b,
    XMLTable ('$book/book'
      PASSING b.bookdoc AS "book"
      COLUMNS
        title     VARCHAR(100) PATH 'title',
        year      INTEGER      PATH '@year',
        publisher VARCHAR(100) PATH 'publisher',
        price     DECIMAL (6,3) PATH 'price')
    AS t)
```

The view definition invokes the XMLTable function once for each row in the books table. Each invocation of XMLTable returns as many rows as there are book elements in the input bookdoc elements (probably one in our example data). *Table 3* shows what the resulting view might look like.

SYNTACTIC AND SEMANTIC CHALLENGES

Integration of the SQL and XQuery languages is made difficult by various mismatches in their syntax and semantics. In this section, we discuss some of these differences and their implications for XML processing in DB2.

Constructors and node identities

SQL is a value-based language and does not have a notion of identity. Values can be freely copied

Table 3 BOOKCONTENT

TITLE	YEAR	PUBLISHER	PRICE
TCP/IP Illustrated	1994	Addison-Wesley	65.60
Hello World	1997	My Pubs	56.40
XML and Web	2000	Sons and Co.	67.50

between SQL operations. XQuery, on the other hand, is a reference-based language, and the notion of node identity plays an important role in XDM. For example, each step of a path expression eliminates duplicates based on node identity, and XPath allows reverse (i.e., parent and ancestor) traversals. In general, nodes cannot be copied between XQuery operations because copying a node does not preserve its identity and its parent linkage. Therefore, the result of an XQuery expression contains node references rather than copies of nodes. One exception is XQuery constructor expressions, which generate new nodes with new identities. For this reason, constructor expressions are nondeterministic. Because SQL/XML:2006[15] is based on XDM, the publishing functions of SQL/XML are also nondeterministic. This poses several challenges for query optimization. In particular, certain rewrite transformations cannot be applied in the presence of nondeterministic functions, and the order of operations must be enforced, limiting the options available to a cost-based optimizer. For example, consider the following query:

```
xquery
  for $i in (1,2,3)
  let $j:=<comment>hello world</comment>
  return $j
```

Here, the `let` clause needs to be executed once for each `$i` value because each `comment` element returned by this query has a different identity. Hence, the optimizer cannot change the order of execution, although there is no dependency between the computations of `$i` and `$j`. The following seemingly equivalent query returns a different result: it returns the same `comment` element three times (that is, it returns three references to a single node with the same identity).

```
xquery
  let $j :=<comment>hello world</comment>
  for $i in (1, 2, 3)
  return $j
```

Constructors also interfere with common subexpression detection. For example, consider the `bib` view definition in the previous section and the following query:

```
xquery
  db2-fn:xmlcolumn('BIB.DOC')
    /book[author/name='John Doe']
  union
  db2-fn:xmlcolumn('BIB.DOC')
    /book[publisher='Pub1']
```

When each reference to the `bib` view in the above query is replaced by its definition, the result is two distinct `BIB` documents, as each invocation of the view produces new nodes with different node identities. In the result of the query, `book` elements that represent books written by John Doe after 1996 will appear twice, once from the first `BIB` document and once from the second. Many SQL query compilers would detect two invocations of the same view as a common subexpression and would evaluate it only once. However, XQuery compilers cannot apply this optimization when the view definition contains node constructors. This limitation is similar to the limitations caused by SQL functions that are declared with the keywords `NOT DETERMINISTIC`.

To avoid the spurious duplicates returned by the previous example, a user might choose to rewrite it as follows:

```
xquery
  let $bib := db2-fn:xmlcolumn('BIB.DOC')
  return ($bib/book[.//author/name='John Doe']
    union
         $bib/book[publisher='pub1'])
```

This query will produce only one instance of each book because the view definition is invoked only once. In effect, the `let` clause explicitly defines the common subexpression.

Because SQL/XML has adopted the XQuery Data Model, the problems caused by nondeterministic functions exist in SQL/XML as well as in XQuery. Before the introduction of the XML data type, there was no semantic difference between an inline view

definition (i.e., `with` clause) and a regular view definition in SQL. However, if the view definition contains an SQL/XML constructor function, then an SQL inline view and a regular SQL view are no longer semantically equivalent. Fortunately, there are some opportunities to remedy this obstacle to optimization. The only operations that are sensitive to node identities are certain XQuery operations that include node comparisons, `union`, `intersect`, `except`, and path expressions. After an XDM instance has been serialized, its node identities are no longer distinguishable. Therefore, if a constructed XML value is not passed to XQuery, then the SQL compiler can detect that node identities are not important and can apply query optimizations, such as common subexpression detection, join ordering, and predicate pushdown. However, this optimization requires global analysis of the query, as opposed to the more local analysis that is commonly used in SQL optimization.

Type mismatches

Another important aspect of the integration of SQL and XQuery is the unification of their type systems. This is made difficult by the fact that the XQuery type system is based on XML Schema,[4] which is different from the type system of SQL. In particular, the primitive data types of the two languages have different domains and properties. For example, XML Schema has durations that mix months and days, whereas SQL does not have such data types. SQL has two types of dates, with and without time zones, and does not allow them to be compared. XML Schema, on the other hand, has a single date type with an optional time zone, and XQuery allows dates without time zones to be compared with dates with time zones. In XQuery, an implicit time zone is used in such comparisons. SQL has many string data types, such as `CHAR`, `VARCHAR`, and `CLOB`, and requires the maximum length of each string-type column to be specified. XML Schema, on the other hand, has a data type called `xs:string` of indefinite length and permits subtypes of limited length to be derived from `xs:string`. Similarly, SQL decimal data types have well-defined precision and scale, whereas XML decimal data types do not necessarily have such constraints. Therefore, SQL/XML defines a mapping from XQuery primitive data types to SQL data types to allow values to be passed back and forth.

These type system mismatches pose several challenges for query processing and indexing and for boundary crossing between SQL and XQuery. Whenever values are passed from XQuery to SQL, they need to be cast or cleansed according to the SQL type system, and vice versa.

Syntax issues

SQL and XQuery have several differences in syntactic and scoping conventions. For example, XQuery is based on Unicode, whereas an SQL database can be in a language-specific code page. When string values are passed from SQL to XQuery, they need to be converted into Unicode.

In addition, the conventions for XML identifiers and SQL identifiers are different. For example, a hyphen is a valid character in an XML identifier, but not in an SQL identifier. Moreover, names in XQuery are QNames, which consist of a namespace and a local name, whereas SQL uses three-part names that consist of a relational schema name,[24] table name, and column name. While namespaces serve to disambiguate names in XQuery, schemas serve the same function in SQL. As a result, different rules and mechanisms are needed to resolve names in XQuery and SQL. The differences in lexical conventions and name resolution prevent the use of a common parser for SQL and XQuery; hence, DB2 uses two different parsers.

■ Although XQuery does not yet have an explicit group-by construct, grouping queries can be expressed by using nested queries and self-joins ■

The low-level syntactic conventions of SQL and XQuery are different in several ways. SQL uses single quotation marks for string literals and double quotation marks for delimited identifiers, whereas XQuery uses either single or double quotation marks for string literals and attribute delimiters. Both languages have conventions for using a pair of quotation-mark characters to represent a single quotation-mark character within a string. Users need to pay special attention to using single or double quotation marks when using `db-fn:sqlquery`, `XMLQuery`, `XMLExists`, and `XMLTable` functions. Consider the following example:

```
xquery
  let $bookTitles :=
    db2-fn:xmlcolumn ("BOOKS.BOOKDOC")
      /book/title
      [contains(., 'Web')]
  let $articleTitles :=
    db2-fn:sqlquery(
      "SELECT XMLELEMENT(name ""art-title"",
        a.title)
      FROM articles a
      WHERE a.title LIKE 'Web%' ")
    return $bookTitles union $articleTitles
```

In this query, we use double quotation marks to delimit the SQL statement in XQuery so that we can use single quotation marks to represent the SQL string literal 'Web%'. Also, note that the SQL identifier art-title is delimited at each end by a pair of double quotation marks.

Similar issues involving nested string delimiters arise when using XMLQuery, as illustrated by the following example:

```
SELECT XMLQuery ('for $chapter in $book//chapter
    where contains ($chapter/title,"Czar")
    return $chapter/title'
  PASSING b.bookdoc AS "book")
FROM books b
```

SQL converts ordinary identifiers into uppercase. For case-sensitive names, SQL requires the use of delimited identifiers, enclosed in double quotation marks. This is especially important when SQL and XQuery are used together in a query. In general, the best practice when bound variables are passed from SQL to XQuery is to use SQL delimited identifiers to name the variables so that they are interpreted in the same way by both languages. If the book variable in the following query were defined using a regular identifier rather than a delimited identifier, the SQL compiler would uppercase it into BOOK, and the $book variable on the XQuery side would be undefined.

```
SELECT XMLQuery ('$book/book/title'
  PASSING b.bookdoc AS "book")
FROM books b
```

Because of the case sensitivity of XQuery, users must be careful in typing name tests in their path expressions. For example, if an XML document contains an element named customerName, that element would not match the name test customername or CustomerName.

SQL null values compared with XQuery empty sequences

SQL uses null values to represent missing or unknown data. However, the concept of a null value does not exist in XQuery. In an XML document, missing or unknown data can be represented simply by the absence of an element. When a path expression searches for a value that is not present, XQuery returns an empty sequence, which is a valid XDM value.

Because XQuery does not have a null value, conventions are needed for how to handle null values whenever a parameter is passed from SQL to XQuery. The functions XMLQuery, XMLTable, and XMLExists all convert null values of named arguments to empty sequences before invoking XQuery. The XMLCast function returns null when its input is null (which is consistent with other SQL scalar functions), and XMLAgg simply ignores null values when computing its output sequence (which is consistent with other SQL aggregate functions). XMLElement and XMLForest, on the other hand, have syntactic options (EMPTY ON NULL, etc.) that permit the user to choose among several conventions for handling null inputs, as described in the section on bridging SQL and XQuery.

Conventions for handling null values are also needed by the XQuery input functions db2-fn:xmlcolumn and db2-fn:sqlquery. Like any other relational column, a column of type XML may contain null values. The db2-fn:xmlcolumn and db2-fn:sqlquery functions simply ignore null values when computing the input sequence. No special conventions are needed by SQL/XML functions for returning an empty sequence, because an empty sequence is a valid XDM instance and is therefore a valid value of the XML data type.

Similar functions in XQuery and SQL may have different behavior with respect to missing values. For example, cast expressions in SQL always return a null value when the input is null. In XQuery cast expressions, on the other hand, whether an empty sequence is allowed or not is controlled by the occurrence indicator specified with the target data type. The XQuery expression $x cast as

xs:integer? will produce an empty sequence if its input is empty, because the occurrence indicator ? signifies that an empty sequence is allowed. The XQuery expression $x cast as xs:integer raises a type error, however, if the input is empty. XQuery can also cast values from one data type to another by means of constructor functions, which always accept empty arguments. Thus, in XQuery, the function xs:integer($x) is equivalent to $x cast as xs:integer?, whereas in DB2 SQL, the function integer(b.col) is equivalent to cast (b.col as integer)

IMPLEMENTATION CHALLENGES

One of the challenges in XQuery processing is caused by multiple interpretations of predicate expressions (enclosed in square brackets). The interpretation of a predicate expression depends on the data type of its result. If the predicate expression returns a single numeric value, then it is interpreted as a positional predicate, whereas if it returns a value of any other data type, it is interpreted as a Boolean filter, and its effective Boolean value is computed. This poses several problems for query transformations as certain transformations cannot be applied when there is a positional variable. If the compiler cannot prove that a predicate expression will return a non-numeric value, it must treat the predicate as potentially a positional predicate.

Consider the path expression .//name, which is shorthand for ./descendant-or-self::node()/child::name. This three-step path expression can be rewritten into a more efficient single-step version descendant::name only if there is no positional predicate on the last step. Note that .//name[2] is different from descendant::name[2]: The former expression asks for the second name child of each descendant of the context node, whereas the latter one asks for the second name descendant of the context node.

Another problem relating to positional predicates arises in the case of a predicate expression that can return either a numeric value or an empty sequence. For example, an arithmetic expression returns an empty sequence if one of its operands is an empty sequence. In a predicate context, an empty sequence is interpreted as a Boolean filter and evaluates to false. Therefore, without static typing,[25] it is not possible to safely interpret the expression //book[year+1] as a positional predicate. For books that have a year element child, this is a positional predicate, but for books that do not have a year element child, it is a Boolean predicate.

Users can avoid the problems introduced by multiple interpretations of predicates by explicitly specifying either a positional or a Boolean predicate. A positional predicate can be specified as [position()=exp], and a Boolean predicate can be specified by using the fn:boolean()function, as in [fn:boolean(exp)].

■ SQL is a value-based language and does not have any notion of identity ■

XML Schema and XQuery have generic data types, which pose special challenges for query processing. When an XML document is not validated, the element nodes in the document are annotated as xdt:untyped and the attribute nodes are annotated as xdt:untypedAtomic. In XQuery general comparisons, if one of the operands is untyped, then it is dynamically cast to the data type of the other operand. This semantic of XQuery is challenging for indexing and requires that untyped data be indexed in every data type that it can be cast to. For example, the untyped value '2679' can behave as either an integer or a string, depending on the context of its use. To avoid casting untyped data to every possible data type, DB2 XML indexes require that the user specify a particular target data type for each index at index creation time.

Different XQuery functions and operators have different conventions for handling untyped values. In general, each function or operator attempts to cast untyped values into a data type that it knows how to handle. For example, the max and min functions cast untyped values to xs:double, whereas order by clauses and the distinct-values function treat untyped values as instances of xs:string.

Even validated documents may contain instances of generic data types, such as xs:anyType. Moreover, XML also permits an element to have a type-defining attribute like xsi:type="decimal". For example, an XML schema may declare a given element to have a

generic data type like `xs:anyType`, and each instance of the element could have a different specific data type, like `xs:string`, `xs:integer`, etc. In this case, the XQuery functions and operators must be dispatched dynamically because each input instance may be different.

To support evolving schemas, DB2 does not implement the static typing feature of XQuery.[19,20,21] Static typing would be too strict when documents have changing schemas, and it would reject many valid and useful queries at compilation time. As a consequence, DB2 relies on dynamic dispatch of XQuery functions and operators, even when schemas are strongly typed, i.e., they do not have generic data types. Each operation and function first checks the type annotations of its inputs and performs the appropriate operation.

The lack of concrete type information can affect the search for an eligible index when processing a given predicate. For example, when joining two XML values, if their data types are not known by static analysis, the compiler cannot pick an index. For this reason, it is suggested to insert explicit cast functions into join predicates to aid in index selection.

XQuery allows an expression to be evaluated without accessing all relevant data if accessing further data could not change the outcome except by raising an error. For example, an XQuery engine can evaluate logical expressions in any order and may not evaluate one of the operands; while evaluating an `and` expression, it may stop as soon as one of the operands evaluates to `false`. Similarly, an XQuery engine can stop evaluating a `some` expression (which searches for some item in a sequence that satisfies a condition) as soon as it finds the first item that satisfies the condition, even if some other item might have raised an error. DB2 exploits this type of optimization and is therefore nondeterministic in the presence of certain types of errors, as permitted by the XQuery specification.

CONCLUSION

Relational database systems are very well adapted to traditional business applications in which data has a known regular structure. The advantages of the relational data model are as important as ever, and no one expects XML data and languages to replace relational data and languages in traditional database applications.

Native storage of XML data, however, offers some important advantages for new types of applications. It allows storage of very diverse forms of information while preserving the ability to search or aggregate that information. It provides a natural storage model for data that has an intrinsic order, a hierarchic structure, or a large number of sparsely populated attributes. It is well adapted to a world of "schema evolution" where it is necessary to store and process documents conforming to many different schemas, including some documents lacking schemas, and where the set of schemas is rapidly changing. These kinds of applications are increasing in importance due to the influence of the Web and e-commerce.

In many ways, the state of XML database languages and systems resembles that of relational languages and systems in the early 1980s. Many research papers have been published, and a standard query language is under development. Commercial systems are beginning to appear, but optimization technology is still immature. Some important use cases are still not covered by the existing languages (for example, XQuery still lacks an update capability). Some skeptics are doubtful about the value of the new approach and its ability to be implemented efficiently.

SQL/XML allows users to combine the advantages of SQL and XQuery by using each language where it is most appropriate. It allows XML data to be stored in relational systems, taking advantage of the mature infrastructure provided by these systems and, at the same time, preserving and exploiting the special characteristics of the XML data. By implementing SQL/XML with XQuery support, DB2 gives users the flexibility to choose among several techniques for storing XML data, including shredding and native XML storage.

Native storage of XML data represents a major investment for the database industry. Many XML database products are beginning to appear from major DB vendors and others. If XML databases are successful, they will pass through the same evolutionary stages encountered earlier by relational databases. XQuery will be extended with update operations and possibly with additional features,

such as better error recovery, explicit grouping, and other analytic features. Customer experience will determine which use modes are most important to the language. XQuery will continue to evolve to meet user requirements, and developing optimization techniques for access to XML data will be an active research topic for years to come.

*Trademark, service mark, or registered trademark of International Business Machines Corporation.

**Trademark, service mark, or registered trademark of Massachusetts Institute of Technology, Linus Torvalds, The Open Group, Microsoft Corporation, or Sun Microsystems, Inc. in the United States, other countries, or both.

CITED REFERENCES AND NOTES

1. *ISO 8879:1986,* International Organization for Standardization (ISO), Information Processing—Text and Office Systems—Standard Generalized Markup Language (SGML).
2. *ANSI/ISO/IEC 9075-2:2003,* International Organization for Standardization (ISO), Information Technology—Database Languages—SQL—Part 2: Foundation (SQL/Foundation).
3. *Extensible Markup Language (XML) 1.0 (Third Edition)*, T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, Editors, W3C Recommendation (February 4, 2004), http://www.w3.org/TR/2004/REC-xml-20040204.
4. *XML Schema*, The XML Schema Working Group, http://www.w3.org/XML/Schema.
5. *XML Path Language (XPath) Version 1.0*, J. Clark and S. DeRose, Editors, W3C Recommendation (November 16, 1999), http://www.w3.org/TR/xpath.
6. *XSL Transformations (XSLT) 1.0*, J. Clark, Editor, W3C Recommendation (November 16, 1999), http://www.w3.org/TR/xslt.
7. World Wide Web Consortium, http://www.w3c.org.
8. *QL'98: The W3C Query Languages Workshop,* Boston, MA (1998), http://www.w3.org/TandS/QL/QL98.
9. D. Chamberlin, D. Draper, M. Fernández, M. Kay, J. Robie, M. Rys, J. Siméon, J. Tivy, and P. Wadler, *XQuery from the Experts: A Guide to the W3C XML Query Language*, H. Katz, Editor, Addison-Wesley, Boston, MA (2003).
10. *XQuery 1.0: An XML Query Language*, S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon, Editors, W3C Recommendation (November 3, 2005), http://www.w3.org/TR/xquery.
11. W3C XML Query (XQuery), http://www.w3.org/XML/Query/.
12. *XQuery 1.0 and XPath 2.0 Data Model*, M. F. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh, Editors, W3C Working Draft (November 3, 2005), http://www.w3.org/TR/xpath-datamodel/.
13. *Namespaces in XML*, T. Bray, D. Hollander, and A. Layman, Editors, W3C Recommendation (January 14, 1999), W3C Recommendation, http://www.w3.org/TR/REC-xml-names/.
14. *ANSI/ISO/IEC 9075-14:2003,* International Organization for Standardization (ISO), Information Technology—Database Language—SQL—Part 14: XML-Related Specifications (SQL/XML).
15. *ANSI/ISO/IEC 9075-14:2006,* International Organization for Standardization (ISO). Information Technology—Database Language SQL—Part 14: XML-Related Specifications (SQL/XML); document expected to be published in mid 2006.
16. *XML Information Set (Second Edition)*, J. Cowan and R. Tobin, Editors, W3C Recommendation (February 4, 2004), http://www.w3.org/TR/xml-infoset/.
17. *XSLT 2.0 and XQuery 1.0 Serialization*, S. Boag, M. Kay, J. Tong, N. Walsh, and H. Zongaro, Editors, WC3 Candidate Recommendation (November 3, 2005), http://www.w3.org/TR/xslt-xquery-serialization/.
18. SQL/XML also provides an optional mechanism for passing an unnamed "context item" to an embedded XQuery expression; this mechanism is not currently supported by DB2.
19. K. S. Beyer, R. J. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. M. Lohman, B. Lyle, F. Özcan, H. Pirahesh, N. Seemann, T. C. Truong, B. van der Linden, B. Vickery, and C. Zhang, "System RX: One Part Relational, One Part XML," *Proceedings of the 24th ACM SIGMOD International Conference on Management of Data*, Baltimore, MD (2005), pp. 347–358.
20. K. S. Beyer, F. Özcan, S. Saiprasad, and B. van der Linden, "DB2/XML: Designing for Evolution," *Proceedings of the 24th ACM SIGMOD International Conference on Management of Data*, Baltimore, MD (2005), pp. 948–952.
21. M. Nicola and B. van der Linden, "Native XML Support in DB2 Universal Database," *31st International Conference on Very Large Data Bases*, Trondheim, Norway (2005), pp. 1164–1175.
22. The `db2-fn:sqlquery` function does not provide any mechanism to pass parameters from XQuery to SQL.
23. SQL/XML defines an XMLQuery option called `RETURNING CONTENT`, which implicitly adds a document node to the XML value before returning it. DB2 does not support this option because it causes unnecessary node construction. If a document node is desired, it can be generated easily by the embedded XQuery expression.
24. Note the SQL use of the term *schema* is different from that in XML Schema. In SQL, a schema denotes a collection of tables, data types, and functions, which is roughly equivalent to an XML namespace.
25. Even with static typing, it may not always be possible to distinguish between positional and Boolean predicates due to generic schema types, such as `xs:anyType` and optional elements.

Accepted for publication October 14, 2005.
Published online April 27, 2006.

Fatma Özcan
IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120 (fozcan@us.ibm.com). Dr. Özcan has been a research staff member since 2001. She received a Ph.D. degree in computer science from the University of Maryland. Her research interests include XML query languages and query optimization,

integration of heterogeneous information systems, and software agents. Dr. Özcan is a member of ACM SIGMOD, and coauthor of the book, *Heterogeneous Agent Systems*.

Don Chamberlin
IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120 (chamberl@almaden.ibm.com). Dr. Chamberlin is an IBM Fellow and represents IBM on the W3C XML Query Working Group. He is best known as a coinventor of SQL and author of two books on DB2. He holds a Ph.D. in electrical engineering from Stanford University and an honorary degree from the University of Zurich in recognition of his work on database query languages. He is an ACM Fellow, a member of the National Academy of Engineering, and a recipient of the SIGMOD Innovations Award.

Krishna Kulkarni
IBM Silicon Valley Laboratory, 555 Bailey Road, San Jose, California 95141 (krishnak@us.ibm.com). Dr. Kulkarni is a member of the IBM Information Management Standards and Open Source Group. He serves as the primary IBM representative to the ANSI/INCITS H2 committee and as head of the United States delegation to the ISO/IEC JTC1 SC32/WG3 committee, responsible for the standardization of SQL/XML. He holds a Ph.D. degree in computer science from the University of Edinburgh. He has contributed extensively to the evolution of the SQL standard for over a decade. Dr. Kulkarni has published a number of papers on database topics and is a coauthor of *Object-Oriented Databases: A Semantic Data Model Approach*, published by Prentice-Hall.

Jan-Eike Michels
IBM Silicon Valley Laboratory, 555 Bailey Road, San Jose, CA 95141 (janeike@us.ibm.com). Mr. Michels represents IBM on the ANSI/INCITS/H2 and ISO/JTC1/SC32/WG3 committees responsible for standardizing SQL and SQL/XML. Additionally, he is the IBM representative on the JSR 225 Expert Group, responsible for standardizing the XQuery API for Java™ (XQJ), and the editor of the XQJ specification. Mr. Michels holds an M.S. degree in computer science from the Technical University of Ilmenau, Germany. ■