DB2 goes hybrid: Integrating native XML and XQuery with relational data and SQL

K. Beyer
R. Cochrane
M. Hvizdos
V. Josifovski
J. Kleewein
G. Lapis
G. Lohman
R. Lyle
M. Nicola
F. Özcan
H. Pirahesh
N. Seemann
A. Singh
T. Truong
R. C. Van der Linden
B. Vickery
C. Zhang
G. Zhang

Comprehensive and efficient support for XML data management is a rapidly increasing requirement for database systems. To address this requirement, DB2 Universal Database™ (UDB) now combines relational data management with native XML support. This makes DB2® a truly hybrid database management system with first-class support for both XML and relational data processing as well as the integration of the two. This paper presents the overall architecture and design aspects of native XML support in DB2 UDB and its integration with the relational data-flow engine. We describe the new XML components in DB2 UDB and show how XML processing leverages much of the infrastructure which is used for relational data.

INTRODUCTION

XML is the de facto standard for exchanging data between different systems, platforms, applications, and organizations. XML first became a W3C** (World Wide Web Consortium) Recommendation in February 1998, as a standard way to delimit text data.[1] It has emerged in the industry as the predominant mechanism for representing and exchanging structured and semistructured information across the Internet, between applications, and within an intranet. Virtually every industry is working to standardize XML representations for their common business objects. As one industry analyst put it, "Hundreds of vertical schemas, in fields as diverse as government, biology, finance, and travel, are publicly available and being actively used. Undoubtedly, there are thousands more in private hands."[2]

Among the key benefits of XML are its vendor and platform independence and its high flexibility. XML is a data model suited for any combination of structured, unstructured, and semistructured data. XML data is easy to extend because new tags can be defined as needed. XML documents can easily be transformed into "different looking" XML and even into other formats such as HTML. Furthermore, XML documents can easily be checked for compliance with a schema. All this has become possible through widely available tools and standards such as XML parsers, XSLT (Extensible Stylesheet Lan-

guage Transformation), and XML Schema. They greatly relieve applications from the burden of dealing with the particularities of proprietary data formats. In an era where message formats, business forms, and services change frequently, XML reduces the cost and time required to maintain application logic.

With the advent of Web Services and service-oriented architectures, it is quite common for intra-company and intercompany interactions to be processed by use of XML messages. In such cases, the message is more than the transaction request; it is also a business artifact, such as a purchase order, an order inquiry, an invoice, and so forth. Such messages need to be retained for many reasons, including auditing, regulatory compliance, and nonrepudiation (i.e., a way to guarantee that the sender of a message (or document) cannot later deny having sent the message and that the recipient cannot deny having received the message). For example, a large securities clearing house interacting with member brokers using Web Services is legally obligated to store the XML messages for nonrepudiation. Many of these uses also require extensive search capabilities, and the XML storage must have very high fidelity to preserve digital signatures as required for nonrepudiation. Thus, although its original intent was data interchange, an increasing amount of XML is designed for persistent storage, and enterprises are even retaining XML messages used for data exchange for later analysis.

Another reason for using XML as a permanent storage format is that XML can be a more suitable data model than a relational schema. For example, in life-science applications the data is highly complex and hierarchical and yet may contain significant amounts of unstructured information, which is challenging to store with a relational schema. Most of today's genomic data is still kept in proprietary flat file formats, but major efforts are under way to move to XML.[3]

Many industries rely heavily on existing relational databases and applications to run their businesses and on SQL (Structured Query Language) to manipulate them. Much of the information within XML documents is generated from these databases, and much of the information from XML documents is stored in them. Integration of this well-structured relational information with self-describing XML data is an important evolutionary advance in the data industry.

There are three key factors that led us to build a hybrid relational and native XML database system:

1. *XML and relational data coexist and complement each other in enterprise solutions*. Some types of data are best modeled and stored in a relational format; other types are best suited for XML. Although XML data can be normalized into relational tables, it may not be appropriate to do so. For example, if XML data comes from a multiplicity of schemas, the aggregate size of the relational schema to model all the data may be unacceptable given the usage; for example, an organization may have 1500 electronic forms and require over 30,000 relational tables to represent this data, despite the fact that most forms are seldom used. If XML data has a highly variable schema with respect to time, the impact of changing the corresponding relational schema frequently may make it impractical to model the data in a relational database. This is particularly pronounced when the corresponding schema change would require normalization, such as making a single-valued attribute into a multi-valued attribute. If XML data contains many sparse attributes that are only accessed in the context of the parent object, the cost of normalization may be prohibitive and denormalization may be impractical because of limits on the maximum width of a row or the maximum number of columns in a table.

2. *A successful XML repository requires much of the same infrastructure that already exists in a relational database management system*. Such a repository must support all the traditional database properties, such as transactional (ACID) properties, availability, scalability, reliability, usability, manageability and installability. The data must be quickly and efficiently updatable with existing, well-understood isolation levels (i.e., the degree to which concurrent transactions in a database management system can affect each other) and semantics. It must have performance characteristics similar to a relational system. For high-performance bulk processing of XML data, it is important to have an underlying model that is based on a set-at-a-time processing (i.e. applying an operation to a set of rows), as also argued by

Jagadish et al. in Reference 4. Relational database engines are highly scalable as a result of many years of research and tuning. The XML data must be indexable for both parametric and full-text search predicates, and it must be stored in a way that avoids unnecessary joins. This is especially true for common operations like full document retrieval. Furthermore, well-known query rewrite and optimization techniques can be applied to XQuery. The database community has years of innovation in this area, and the database industry has a large investment in systems that solidly support these characteristics. In a hybrid system, much of the infrastructure that supports these characteristics can be reused.

3. *XML query languages have considerable conceptual and functional overlap with SQL.* XQuery[5] and SQL/XML[6] are the two industry-standard languages that have emerged to query business artifacts encoded in XML. XQuery provides a rich query language that supports the hierarchical structure of XML. SQL/XML extends the relational model with an XML data type, constructs for querying this new data type in conjunction with relational data, and functions for converting between relational and XML data. Despite the slightly different focus of these two languages, they include many similar concepts, including set-based processing, sequence-based processing (i.e., processing an *ordered* set of items), joins, selections (i.e., returning a subset of a table's rows), projections (i.e., returning a subset of a table's columns), and quantification. Regrettably, they are not directly convertible,[7,8] but a significant portion of a relational data-flow engine is directly applicable to the processing of XML query operations. The principal difference is the introduction of intradocument structure-dependent operations due to the hierarchical nature of XML data. The potential for such extensibility has already been proven through the integration of object capabilities in SQL. We further demonstrate the extensibility of such systems with navigational support for XML.

The other consideration that shaped the overall design and architecture of the system was the need to support schema evolution. A repository must support schema evolution to minimize the impact of changes to applications and existing XML data. Schemas for data represented as XML must be very flexible and can be highly volatile over time. XML is increasingly used to represent actual business artifacts, such as derivatives contracts, mortgages, and legislative and legal documents. Most of these artifacts have very long retention requirements: decades (for derivatives contracts), centuries (for mortgages or insurance forms), or indefinitely (for legal and legislative documentation). Therefore, it is critical that an XML repository respond seamlessly to schema changes. This evolution is either impractical or impossible in more rigidly structured relational systems because XML documents must retain their original structure, schema, and content to preserve their digital signatures. Although an individual document typically has an associated schema when it is inserted, a large collection of documents is unlikely to conform to a single schema.

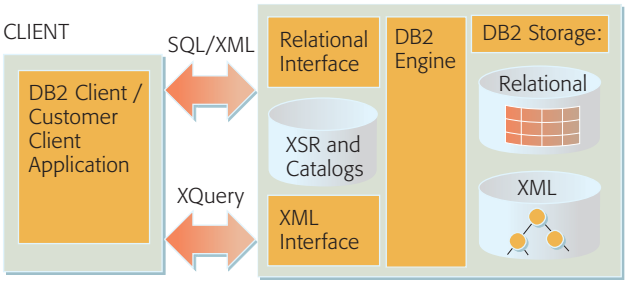

Figure 1
Integrating XML and relational data in DB2 UDB

In this paper, we present the native XML features of the upcoming version of DB2* Universal Database* (UDB) and DB2 for z/OS* and describe the underlying architecture and design concepts. We discuss examples illustrating the XML capabilities as well as the integration of XML with SQL and relational data management. (Throughout this paper, unless otherwise indicated, DB2 refers to DB2 UDB.)

Overview of DB2 with native XML support

A high-level view of DB2 with native XML support is shown in ***Figure 1***. The DB2 storage component manages both conventional relational data storage and the new native XML storage. Both types of storage are accessed by the DB2 engine, which processes SQL/XML and XQuery queries in an integrated manner. DB2 unifies new XML native storage, indexing, and query processing with existing relational storage, indexing, and query process-

ing. Additionally, DB2 provides an XML Schema repository (XSR) to register and maintain XML schemas, and uses those schemas to validate XML documents. Finally, database utilities are enhanced with XML import and export capabilities, as well as with a new graphical XQuery builder.

The addition of native XML support has no impact on existing SQL applications. A client application can continue to use SQL to communicate with the DB2 server through the relational APIs (application programming interfaces) to access and manipulate data in the relational data store. The SQL/XML extensions allow publishing of relational data in XML format and full document retrieval from the native XML storage. Additionally, the new SQL/XML querying functions provide SQL applications with subdocument-level search and extract capabilities, by embedding XQuery statements into SQL statements.

An XML application can interact with the DB2 server through the XML interface by using the XQuery language, which is supported as a stand-alone query language independent of SQL. XQueries can optionally contain SQL statements to combine and correlate XML data with relational data. Application support and API enhancements for XML are discussed in the section "XML API and application support."

Because an update language for XML is not yet close enough to standardization, the DB2 server supports full document updates for now. A stored procedure that provides applications with a flexible interface for subdocument level updates is available for XML updates. Currently, this procedure rewrites the full document, but eliminates the need to send documents for update from the DB2 server to the client and back.

DB2 treats both SQL and XQuery as primary query languages. Both operate on their respective data models and can be used independent of each other. However, database applications can benefit immensely from the integration of the two languages that DB2 supports. Because many applications deal with existing relational data and XML simultaneously, queries need to combine and correlate these two types of data. In particular, DB2 has separate parsers for SQL and XQuery statements, but uses a single integrated query compiler for both languages. No translation from XQuery to SQL is performed. DB2's compiler and optimizer are extended to handle SQL and XQuery in a single modeling framework.[9]

Queries enter the system through either language and are then compiled into an execution plan, as described in the section "XQuery compilation." After parsing, the distinction between SQL/XML and XQuery is discarded in favor of a unified internal representation. The query is modeled as a query graph, using an extended query-graph model.[10] We exploit rich data-flow modeling to perform powerful cross-language optimizations. We extend traditional rewrite optimizations to work with the extended query model and introduce some rewrites specific to the XML query languages. After the rewrite phase, the portions of the query that can be answered by an XML index are detected. This is significantly more challenging than for relational indexes. The query then enters our enhanced cost-based optimizer to plan the new XML index and navigation operators. Once the execution plan is generated, the query can be evaluated by the combined relational-XML runtime engine, which is a relational runtime extended with XML navigation and indexing capabilities.

The XML data type

At the heart of DB2's native XML support is the XML data type. This is a first-class data type in DB2, just like any other SQL type. SQL/XML aligns the XML data type with the XQuery Data Model,[11] which closes the algebra and allows XML values to be passed back and forth between SQL/XML and XQuery. (An algebra is closed under a data model if every operation takes as input and produces as output an instance of the data model.) By building a hybrid system, DB2 enables seamless flow of XML data from SQL applications into an XQuery processor and vice versa.

The XML data type can be used in a "create table" statement to define one or more columns of type XML, as shown in ***Figure 2***. Since XML has no different status than any other type, tables can contain any combination of XML columns and relational columns. Each column of type XML can hold one well-formed XML document for every row of the table. Though every XML document is logically associated with a row of a table, XML and relational columns are stored differently. The rela-

tional columns are stored in traditional row structures, whereas the XML data is stored in hierarchical structures. The two are closely linked for efficient cross-access. Native XML storage is described in the section "Native XML storage."

An XML schema is not required in order to define an XML column or to insert or query XML data. An XML column can hold schema-less documents as well as documents for many different or evolving XML schemas. Schema validation is optional and may be performed on a per-document basis. Thus, the association between schemas and documents is per document and not per column, providing maximum flexibility.

The XML type has no length associated with it. The XML storage and processing architecture imposes no limit on the size of an XML document. Currently, only the client-server interfaces limit XML documents to 2 GB per document.

Values of type XML are processed in an internal representation that is not a string and not directly comparable to strings. Rather, SQL/XML defines a set of functions that consume or produce the XML data type. One such function, XMLSERIALIZE, converts an XML value into a string value, which is the textual representation of the same XML document. Another function, XMLPARSE, converts a string value that represents an XML document into the corresponding XML value.

The XML type can be used not only as a column type but also as a data type for host variables or parameter bindings in languages such as C, Java**, and COBOL. The section "XML API and application support" provides details on this extension to the DB2 APIs. The XML type is also allowed as a parameter and as variables in SQL stored procedures, user-defined functions (UDFs), and external stored procedures written in C and Java. This is important for flexible application development.

NATIVE XML STORAGE

The amount and nature of XML data can be quite variable, depending on the application. Small XML documents often do not exceed 3000 bytes, but the largest XML documents can be multiple gigabytes in length. Small collections of documents have a few thousand documents, but large collections have billions. Some applications retrieve the entire document; others select only a small portion. Some of these documents, once created, are strictly read only, while others are frequently updated.

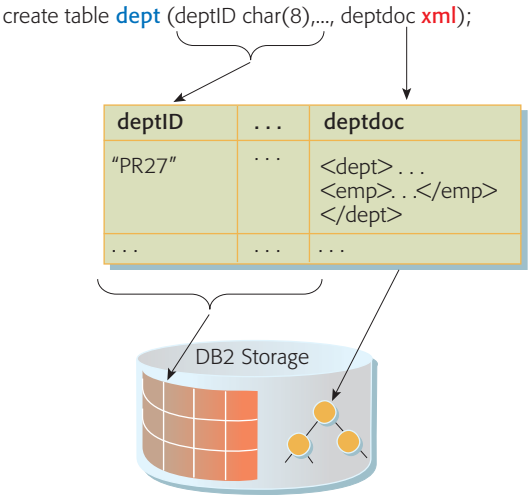

Figure 2
Table with a column of type "XML"

Designing a storage system for such widely varying workloads is challenging. Documents must be able to span disk pages, and even a single text node could be larger than a page. It is not feasible to traverse every node of a gigabyte document to retrieve a small subtree. Therefore, the system must support direct node access and XQuery's node reference semantics in a concurrent system under all of SQL's isolation levels, and it must support rollback and recovery.

To meet these requirements, DB2 introduces a new native XML storage format to store XML documents as instances of the XQuery Data Model in a structured, type-annotated tree. By storing the binary representation of type-annotated XML trees, repeated parsing and validation of the document is avoided. The binary representation maintains the salient features of the document, however, so that any digital signatures on it are preserved. As each node in every document contains its type information, our storage supports schema evolution fairly easily. Because type information is on the document level, rather than on the column level, each document in a column can conform to a different schema, or to different versions of an evolving schema. Furthermore, every node contains pointers to its parent and children to support efficient

navigational queries. Path expressions are evaluated directly for the native format on buffered pages without copying or transforming the data.

To insert XML data into the database, client applications send XML documents in their textual representation to the DB2 server. The server uses a SAX (Simple API for XML) parser to check that incoming documents are well-formed and to perform optional validation. The SAX events are converted into a hierarchical representation of the XML document. For the sample document in Example 1, this hierarchy looks like the document tree in the upper part of *Figure 3*.

Example 1

```
<dept loc = "CA">
   <employee id="901">
      <name>John Doe</name>
      <phone>408 555 1212</phone>
      <office>344</office>
   </employee>
   <employee id="902">
      <name>Peter Pan</name>
      <phone>408 555 9918</phone>
      <office>216</office>
   </employee>
</dept>
```

Replacing tags with StringIDs, as shown in Figure 3, not only reduces memory consumption but also contributes to higher performance for navigational queries. With StringIDs, operations such as node comparisons now operate on integers instead of strings. The mapping table is used for all XML columns in the database, not only to achieve even greater compression, but also, which is more important, to allow the system to easily perform navigation on a mixed set of nodes from several XML columns. Whenever XML nodes or documents are returned as query results, the nodes are serialized back to their text form, including namespaces.

The document tree in the lower part of Figure 3 is similar to the format in which inserted documents are stored on disk pages. Extra information is stored with each node, such as the type annotation if the document was validated. Each element node has a set of child slots for its associated attribute and ordered children. These child slots have hints within them to give an indication of what the child represents. This allows fast navigation across a context node's set of children to find potentially qualifying children without actually visiting each child node. This is important because a child node may be on another page and require additional I/Os. For example, when looking for an employee with a child having a specific name, the 'id' attributes of the employee element and all children without the hint of the target name can be skipped. A unique identifier gives each node both a logical and a physical addressability that can be used in indexing and query evaluation.

If a document tree is too large to fit on one page, it is split into regions, as shown in *Figure 4*. At any level of the document a subtree of nodes can be cut off and become a region. The regions of a document can be stored on separate pages, which do not have to be in physically consecutive order. Our general optimization approach is to keep the number of regions per document as small as possible. Multiple regions can be stored on one page, especially if documents are much smaller than a page and each document is a single region. Nodes with large content can be 'chunked' across multiple pages, and nodes with a large fan-out of children can be 'continued' across multiple pages.

If a document spans multiple pages, its regions are connected by the regions index. A regions index is a system index that is created automatically for every table that contains one or more XML columns. In Figure 4, the document tree is split into three regions shown in gold, orange and green. The gold and the orange regions occupy a full page each. The green region is smaller and fits on a page already containing another small region.

The use of regions and the indirection through the regions index provide important advantages over other approaches, such as the use of physical links between nodes on different pages. From the regions index, DB2 can efficiently prefetch regions for large documents. Since the logical node identifiers are independent of physical node locations, it is much easier to insert, update, and delete nodes or subtrees and to perform document or page reorganization if needed.

The XQuery language uses node reference semantics in its results to allow additional navigation. This is

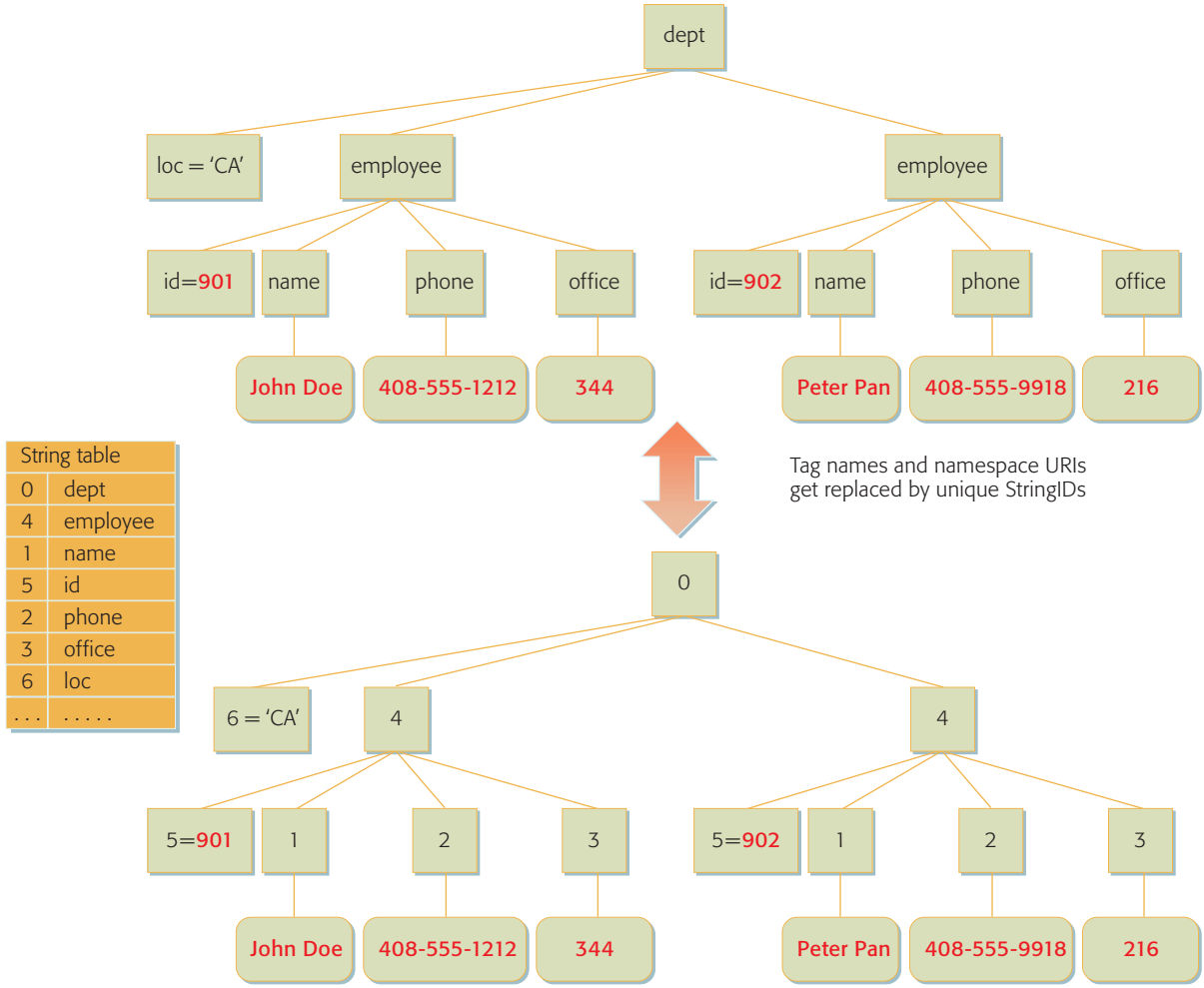

Figure 3
StringIDs in XML storage

achieved by versioning. As a document is updated, its regions are versioned, leaving multiple versions of a node in the regions index. Only the current version is available in the XML value indexes; new readers thus always find the latest version. Readers that have already qualified a node continue to see a consistent version of the document. Old region versions are removed when there are no references to them; this is detected by checking the oldest reader of the table.

The paged storage of XML documents leverages existing components in DB2, such as the buffer pool manager, the table space layer, the lock manager, and the log manager. Reuse of existing relational engine services for transactions, concurrency, scalability, and recoverability simplifies the implementation and is essential for coexistence with relational data. Buffer pool services keep active pages in memory. Record management services handle the placement of nodes within a given page, and the logging of nodes or sets of nodes enables failure recovery.

XML INDEXES

XML applications that manage millions of XML documents are not uncommon. As a result, indexing support for XML data is required to provide high query performance. However, the rich structure of XML introduces new challenges. The obvious interpretation of an index on a relational column is that the values of the column are organized so that the system can quickly locate the rows that satisfy range predicates on the column. The meaning of creating an index on an XML column depends on the

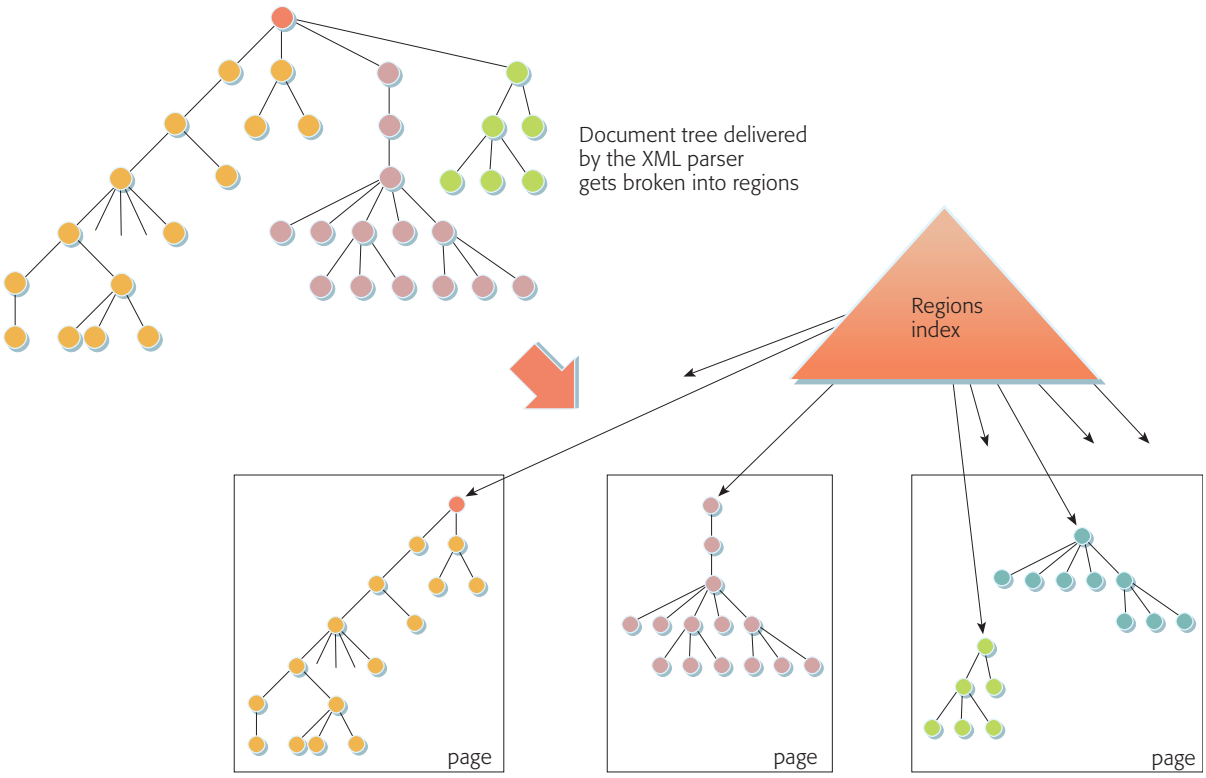

Figure 4
Interlinked document regions

class of XML index that is created. We consider three classes:

1. *Structural indexes*, which map distinct node names, paths, or tag-based path expressions to all matching node instances (see, for example, Reference 12). They may also map node identifiers to nodes instances (e.g., the regions index).
2. *Value indexes*, which allow quick retrieval of nodes based upon the node's data value.
3. *Full-text indexes*, which map tokens (e.g., words) to the nodes that contain them.

Each of these index classes is useful for some queries, but we believe that value indexes are significantly more useful than structural indexes for our expected query workloads. For example, in a query workload on employee records, a query to find employees with any recorded interests is much less likely than a query to find employees interested in nanotechnology. The relational analogy is the likelihood of queries to find records with a particular value in some column versus queries looking for null values. Nevertheless, our value indexes do support some structural predicates.

Because our XML data is commingling with existing relational data and will be used by future versions of existing applications, we require our value indexes to support all the features of the existing relational system. This includes transactions, concurrency, recovery, scalability, fast insertion, efficient update, reorganization, backup and restore, and import/export.

As with relational systems, applications typically cannot afford to index every item. XML compounds this issue because of the sheer quantity of items that can be indexed. For example, a range predicate can be found on not only any simple node in the document (the "leaf" elements and attributes), but also the processing instructions, the comments, the text nodes (which differ from their containing element), and the interior nodes (such as the concatenation of all text nodes below a node). If we only supported indexing every item in the XML document, the index storage would be many times

larger than the original document. Moreover, the number of I/Os required to transactionally maintain the indexes would be prohibitively expensive. Therefore, DB2 supports path-specific value indexes on XML columns so that elements and attributes frequently used in predicates and cross-document joins can be indexed. XML value indexes are described in the following section.

As a semistructured data model, XML is a bridge between the rigid structural world of relational systems and the free-form world of text documents. Thus, DB2 also supports XML-aware full-text indexing.

XML value indexes

The following series of examples describe how XML data can be indexed. The following example shows the simplified CREATE INDEX Data Definition Language (DDL) for XML value indexes:

```
ddl ::= CREATE INDEX index-name
        ON table ( xml-column )
        USING 'xmlpattern' AS type
xmlpattern ::= namespace-decls?
( ( / | // ) axis? ( name-test | kind-test ) )+
axis ::= @ | child:: | attribute:: | self::
  | descendant:: | descendant-or-self::
name-test ::= Qname | * | nsprefix:* | *:ncname
kind-test ::= node() | text() | comment()
  | processing-instruction( ncname? )
```

The following statement defines an XML value index on all employee names in all documents in the XML column `deptdoc`, based on the sample table and document in Example 1:

```
create index idxl on dept(deptdoc) generate key
using xmlpattern '/dept/employee/name'
  as sql varchar(35)
```

The `xmlpattern` path identifies the XML nodes to be indexed. It is called `xmlpattern` and not XPath because only a subset of the XPath language is allowed in index definitions. For example, wildcards (//,*) and namespaces are allowed, but XPath predicates such as `/a/b[c=5]` are not supported. Because we do not require a single XML schema for all documents in an XML column, DB2 may not know which data type to use in the index for a given `xmlpattern`. Thus, the user must specify the data type explicitly in the `as sql <type>` clause. The following types can be used: VARCHAR(n) (for nodes with values of a known maximum length), VARCHAR HASHED (for nodes with values of arbitrary length; in this case, the index contains hash values of the actual strings; such an index can be used for equality predicates but not for range predicates), DOUBLE (for nodes with any numeric type), and DATE and TIMESTAMP (for nodes with corresponding XML values).

Because the SQL type system is not exactly the same as the XML type system, special mechanisms compensate for key differences. For example, the DB2 index manager has been enhanced to explicitly handle special values from the XML type system, that is, +0, –0, +INF, –INF, and NaN (i.e., not a number). The reuse of the existing relational index manager introduces a few minor restrictions on the supported XML data: for example, we cannot index arbitrarily large XML strings unless the strings are hashed. Also, the way XQuery treats `xdt:untypedAtomic` data is challenging for indexing. The general comparisons (=, >, etc.) dynamically cast an untyped operand based upon the type of the other operand, which implies that untyped data must be indexed in every data type that it can be cast to. For example, is the untyped value '1234' a character string, a number, or a hexadecimal binary string? The answer could be any of these, depending on how any particular query treats the value. To avoid casting untyped data to every possible data type, the index requires a specific target data type.

Ultimately, an index is created on the cast of the node to the indexed type, taking into consideration the type annotation of the node as derived during validation. This implies that some string-valued nodes appear in a numeric index and that all matching nodes appear in a string index. These semantics are critical when we determine index eligibility. We carefully considered the proper action to take for a node that matches the pattern but cannot be cast to the index type and decided not to create any index entry for it because the node might never be cast to that type during an actual query. Unlike indexes on relational data, a single document may contain zero, one, or multiple nodes that match the `xmlpattern` and thus create a corresponding number of index entries for a single row in the table.

The following statement defines a unique index of all employee ID attributes. Uniqueness is enforced

within a document and across all documents in the XML column.

```
create unique index idx2 on dept(deptdoc)
  generate key using xmlpattern '/dept/employee/@id'
  as sql double
```

In some applications it is difficult to predict which elements or attributes will be searched. For such cases, the following index definitions can be used to index all text nodes and all attributes, respectively, if needed. In this example we are prepared for elements with arbitrary-length values and expect attributes to be numeric:

```
create index idx3 on dept(deptdoc) generate key
using xmlpattern '//text()' as sql varchar(hashed)

create index idx4 on dept(deptdoc) generate key
using xmlpattern '//@*' as sql double
```

To match and index nodes in a particular namespace, the `xmlpattern` path can contain namespace declarations and namespace prefixes:

```
create index idx5 on dept(deptdoc) generate key using
xmlpattern 'declare namespace
    m="http://www.me.com/";
    /m:dept/m:employee/m:name'
as sql varchar(45)
```

To reduce the size of index entries, each unique path in the documents of an XML column is represented by an integer PathID (path identifier). This is similar to the concept of StringIDs for tags described in the section "Native XML storage". The so-called path index maps each distinct reverse path (revPath) to a generated PathID. A reverse path (revPath) is a list of node labels from leaf to root, compressed into a vector of StringIDs. The path index is cached for performance and is typically small because only unique paths are registered.

Each entry in a value index includes the PathID that identifies the path of the indexed node, the value of the node cast to the index type, a RowID, and a NodeID. The RowIDs identify the rows containing the matching documents, similar to regular relational indexes. The NodeIDs identify the matching nodes and regions within the documents. Index entries are ordered by PathID and value. Placing the PathID first allows for quick retrieval of specific path queries. For example, if an index on `//name` were created, which might match many paths, then a query on `/book/author/name` would still access consecutive index entries. A disadvantage of this placement is that a query like `//name='Maggie'` needs to examine every location in the index per matching path (i.e., for every path that ends in `/name`).

Typically, indexes are defined with `xmlpattern`s that identify atomic nodes. A node is atomic if it is an attribute, a text node, or an element that has no child elements and exactly one text node child. However, it is also possible to define indexes on nonatomic nodes. In our example, the XML pattern `/dept/employee` would be considered nonatomic because each employee element has three child elements with one text node each. This results in a single index entry for each employee element. The value of such an entry is the concatenation of all text nodes in the subtree under 'employee'. This complies with the XML data model. If the intention is to index all employee names, offices, and phone numbers as separate values, then the `xmlpattern` path `/dept/employee/*/text()` or three separate create index statements should be used. Nonatomic indexes are rarely useful for data-centric XML, but can be useful for mixed content in text-oriented XML.[13]

A given index can be used to evaluate an XPath predicate only if the data type used in the predicate matches the one in the index and if the XPath qualifies a subset of the indexed nodes. In the previous example, index `idx3`, could be used to evaluate the predicate `/dept//name[text()="Joe"]`. However, `idx2` could not be used to evaluate the predicate `//@id-"A167"` for two reasons: (1) `idx2` is a numeric index, but the predicate asks for a string comparison, and (2) the predicate searches for `@id` attributes anywhere in the document, but `idx2` only covers those under `/dept/employee`. Details are given in the section "Index eligibility."

XML full-text indexes

Full-text search is a common operation in document- and content-centric XML applications. The existing text-search capabilities of DB2 have been extended to work with the new XML column type. Full-text indexes with awareness of XML document structures can be defined on any native XML column. The documents in an XML column can be fully or partially indexed. The latter is useful if it

is known in advance that only a certain part of each document will be subject to full-text search, such as a "description" or "comment" element. Correspondingly, text-search expressions can be applied to specific paths in a document.

The following statement defines a text index that fully indexes the documents in the XML column `deptdoc` in the table `dept` in the database `personneldb`:

```
create index myIndex for text on dept (deptdoc)
  format xml connect to personneldb
```

The following query exploits this index but restricts the search to a specific element. The query retrieves all documents where the element `/dept/comment` contains the word `Brazil`:

```
select deptdoc from dept where
contains (deptdoc, 'sections ("/dept/comment")
          "Brazil" ') = 1
```

Text search in specific parts of documents is a critical feature for many applications. Standard text search features are also available, such as scoring and ranking of search results as well as thesaurus-based synonym search. For best performance of insert, update, and delete operations, the text index is maintained asynchronously, that is, not within the context of a DML (Data Manipulation Language) transaction but after the transaction has ended (this is known as a "lazy" update). An "update index" command is available to explicitly force synchronization of the text index.

QUERYING XML

DB2 supports the two industry-standard languages for querying XML data: XQuery and SQL/XML. XQuery is a powerful language defined by the W3C, that queries both structured and semistructured data. It provides *path-expressions*[14] to navigate through XML trees and extract XML fragments, as well as expressions to create, sort, aggregate, combine, and iterate over sequences (i.e., examine or manipulate each item in a sequence), and construct new XML data. XQuery is a reference-based language; therefore, subsequent expressions on the result of a path expression may traverse the document in both forward and reverse directions. SQL/XML, which is standardized by ANSI (American National Standards Institute) and ISO (International Organization for Standardization), defines a new XML data type. SQL/XML defines second-order query functions such as `XMLQUERY`, `XMLTABLE`, and `XMLEXISTS` that take an XQuery statement as input and execute it over the XML values passed from SQL. SQL/XML also includes functions to construct new XML data and to convert XML to relational data types and vice versa.

The XQuery data model is based on the notion of *sequences*, which are ordered collections of zero or more items. SQL/XML aligns the XML data type with the XQuery data model, which closes the algebra and allows XML values to be passed back and forth between SQL/XML and XQuery. By building a hybrid system, DB2 enables seamless flow of XML data from SQL applications into an XQuery processor and vice versa.

Relational tables can define columns that use the new XML data type. This enables existing SQL applications to augment their current relational database designs with additional XML data and provides an evolutionary path for XML support.

The following subsections describe how XML and relational data can be queried with XQuery, SQL, or a combination of the two. Throughout this discussion our examples will refer to the following two tables, and we assume that the XML column `deptdoc` contains documents like the one shown in Example 1.

```
create table
  dept(deptID char(8) primary key, deptdoc xml)
create table
  unit(ID char(8), name char(20), manager char(20))
```

Querying XML data with XQuery

In DB2, XQueries can operate on XML documents in one or more XML columns, as well as documents passed as runtime arguments. Access to XML data stored in relational columns is provided by two DB2 input functions: `db2-fn:xmlcolumn` and `db2-fn:sqlquery`. The `db2-fn:xmlcolumn` function takes as input a string literal that identifies an XML column and returns an XML sequence that consists of all document nodes in the specified column. If a column value is null, then there is nothing in the resulting XML sequence for that row. The following example shows how this function is used:

```
for $e in db2-fn:xmlcolumn("DEPT.DEPTDOC")
  /dept/employee
where $e/office = 344
return $e/name
```

The db2-fn:xmlcolumn function can be used multiple times in a single XQuery to reference different XML columns in the same or separate tables, or to reference the same XML column several times. This is a very common usage scenario. However, it may at times be desirable to restrict the input to an XQuery based on conditions placed on relational columns in the same or related tables. This can be accomplished with the function db2-fn:sqlquery, which accepts any select statement that returns a single XML column. The following query is an example where the set of input documents to XQuery is filtered by using a join and a predicate on another relational table.

```
for $e in db2-fn:sqlquery ('select deptdoc
          from dept, unit
          where dept.deptID=unit.ID and
          unit.manager = "Jim Qu"')/dept/employee
where $e/office = 344
return $e/name
```

This highlights the power of integrating XQuery and SQL. Users can leverage all of their existing relational data and indexes to qualify XML documents for XQuery processing.

An XQuery and one or multiple embedded SQL queries are compiled into a single execution plan and comprise a single statement. SQL isolation levels and security privileges apply to the entire statement as a single unit, just as they do to any regular SQL statement.

The result returned by an XQuery statement is treated as a table with a single column of type XML. Each row returned represents an item from the XML sequence that is the result of the XQuery. Thus, existing DB2 mechanisms, which are available for SQL/XML, can also be used to declare and open cursors for XQuery queries (a cursor is a database element that controls record navigation, updateability of data, and the visibility of changes made to the database by other users). The mechanisms can also fetch items from the XML sequence returned by XQuery and close cursors. Note that these items can be anything from XML documents to atomic values such as integers or strings.

Querying XML data with SQL/XML

It is often desirable to use or extend SQL statements to retrieve XML data, because database users are familiar with SQL and because existing relational applications are frequently augmented with XML data. Because XML is a regular SQL data type, full documents can be retrieved from an XML column with a simple select statement, such as:

```
select deptdoc from dept where deptID LIKE "PR%";
```

Additionally, DB2 supports most of the new SQL/XML functions and predicates, including XMLQUERY, XMLEXISTS, XMLTABLE, XMLVALIDATE, XMLPARSE, and XMLCAST. These are described in detail in Reference 6; here, we highlight some of the most useful ways of deploying these functions.

XMLEXISTS is a Boolean predicate, which tests whether an XML document matches given criteria. It returns a value of either 'true' or 'false' for every row. The following sample query returns full department documents as in the previous example but uses XMLEXISTS for additional filtering: only those rows where the department document contains an employee in office 344 are returned. The passing clause establishes the binding between the SQL and the XQuery context.

```
select deptID, deptdoc
from dept d
where deptID LIKE "PR%" and
      XMLEXISTS('$deptdoc/dept/employee[office = 344]'
                passing d.deptdoc as "deptdoc")
```

Apart from document filtering, it is also desirable to extract and return partial XML documents, such as subtrees or atomic attribute and element values. This is achieved with the XMLQUERY function. It evaluates an XQuery expression and returns the actual result as an XML sequence to the SQL application. The query in the following example selects the deptID for all Public Relations departments, and the XMLQUERY function extracts the employee names for all Public Relations employees in office 344. The by ref option avoids copying the XML value when returning from XQuery to SQL.

```
select deptID,
       XMLQUERY('for $e in $deptdoc/dept/employee
                 where $e/office = 344
                 return $e/name'
       passing d.deptdoc as "deptdoc"
       returning sequence)
from dept d
where deptID LIKE "PR%";
```

The XMLTABLE function is useful in converting XML data or the result of an XQuery into tabular format, as shown in the following example. XMLTABLE receives one dept document at a time from the deptdoc table and evaluates the FLWOR (For, Let, Where, Order by, and Return) expression. For each matching employee, a row is returned, containing the employee ID and room number as integers and the name as a character array of length 25.

```
select T.EmpId , T.Name, T.Room
from dept d,
  XMLTABLE('for $e in $deptdoc/dept/employee
            where $e/office = 344
            return $e'
  passing d.deptdoc as "deptdoc"
  columns
  "EmpId" integer path '/@id',
  "Name" char(25) path '/name',
  "Room" integer path '/office') as T
```

XQUERY COMPILATION

Figure 5 gives an overview of the hybrid query compiler. We have implemented a new component, the XQuery parser, and extended all other components in the compiler to process the XQuery data model and the XML query languages. First, an SQL statement or an XQuery expression is compiled into an internal data flow graph. Next, rewrite transformations are applied to normalize, simplify, and optimize the data flow. The optimizer uses this graph to generate a physical plan, which is translated by code generation into executable code. In this section, we describe each component and discuss trade-offs that led to the current design.

Two major decisions were critical in the compiler design. First, DB2 does not implement static typing. XQuery has both static and dynamic semantics, depending on when type checking is enforced. Static typing is too restrictive for schema evolution, as each document insertion or change in schema may result in recompilation and even rewriting of

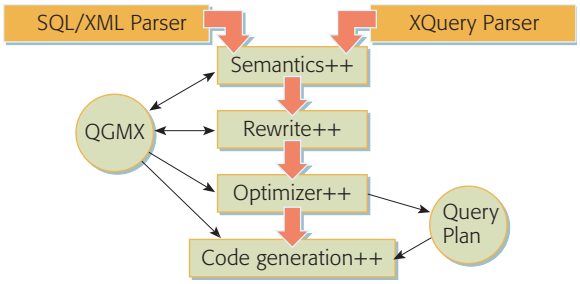

Figure 5
Hybrid SQL/XQuery compiler

applications. Because DB2 does not require that all XML documents in an XML column conform to a single schema, or to a collection of conforming schemas, nonconforming changes may occur between schema versions. For example, optional fields may become mandatory. In addition, because DB2 does not support the schema import feature of XQuery, user-defined data types are not accessible for querying. Although DB2 does not implement static typing, it still exploits type information wherever possible.

Second, DB2 does not normalize[15] the XPath expression into explicit FLWOR blocks, where iteration between steps and within predicates is expressed explicitly. Instead, path expressions that consist solely of navigational steps are expressed as a single expression and are evaluated by our powerful navigation runtime (see the section "Query runtime"). This impacts query modeling, rewrites, and optimization and is discussed in the following sections.

XML query modeling

In DB2, we represent XQuery with an internal query graph model (QGM), which is a semantic network that represents the data flow in a query. Although it is fine-tuned for efficient relational query processing, the data flow graph is more generic than relational algebra. As QGM is designed to be extensible,[16] it is fairly easy to add new entities and capture multiple data models. Note that we are not translating XQuery into relational algebra or into SQL. On the contrary, we are augmenting our internal data flow model with native constructs that are specific to XML and that represent complex navigation of XPath and XQuery.

In its simplest form, a QGM graph consists of operations and arcs that represent the data flow between operations. XQuery provides similar constructs to iterate over XML data and apply predicates to join and sort data. We represent these XQuery operations with the existing QGM entities and introduce new entities to represent path expressions and sequences.

To coalesce the relational and XML data models, we first needed to decide how to represent XQuery sequences within the context of a relational engine. SQL/XML[6] introduces XML as a column in a relational table and is based on the XQuery data model. To accommodate this new SQL data type, we decided to represent an XQuery sequence as a column value in the QGM. Some XQuery expressions consume a sequence as a whole, and others iterate through the items in a sequence. We decided to provide an operation which "unnests" the items in a sequence for set processing and another operation that aggregates a stream of items into a sequence.

The focus of earlier work on XML processing was on efficient representation and execution of path expressions. On one side of the spectrum were *fine-grained approaches*, in which each axis and name test on a single path step was represented as a selection. As a result, a complex path expression required a series of selections and a complex multiway join operation. Although this approach was designed for a system in which the XML data was shredded into relational tables (i.e., XML data was converted to relational format and stored in a set of tables) in order to compose navigation steps with element construction, it turned out to be incapable of handling large queries and representing the full XQuery language. In particular, it implied an order of execution for navigation. On the other side of the spectrum was the *coarse-grained approach*, wherein many binding path expressions were represented in a pattern tree, such as generalized tree patterns.[17] Such systems, however, only dealt with single FLWOR blocks. In DB2, we take a *medium-grained approach* and represent each path expression as a pattern tree in which there is only one bound variable. In this way, we represent each data flow (e.g., each variable) explicitly, so that semantics and rewrite analysis, which are built on explicit data flow representation, can handle the query more efficiently. In addition, this approach allows us to represent not only path expressions as binding patterns but also other FLWOR expressions. It also allows us to support full compositionality of XQuery (i.e., the ability of any XQuery expression to be used within any other XQuery expression). However, as we explain in the section "XQuery rewrite transformations," after query rewrite we try to consolidate all navigation within a query block into a single pattern tree representation.

Query rewrite transformations

The QGM graph output by the XQuery parser needs to capture the full compositionality of the XQuery expressions. As a result, it may not be the most compact or efficient representation of the query. The goal of the rewrite transformations is twofold: First, the data flow is optimized by consolidating some operations, eliminating redundant computation, and applying several logical transformations. Second, the QGM representation is normalized so that the query optimizer gets the same graph as input for semantically equivalent queries and has maximal flexibility.

To support schema evolution, we decided not to apply any schema-based transformations.[18,19] Such transformations may require frequent recompilation and rewriting of queries and applications as schemas evolve. For example, a schema change that converts a single-valued attribute or element into a multivalued attribute invalidates the query plan if such schema-based transformations are applied.

By building XML processing on top of a robust relational engine, we are able to exploit many sophisticated rewrite transformations. These transformations optimize the data flow, and hence, some are also applicable to XQuery. For example, rewrites that merge nested query blocks or eliminate unused variables are directly applicable. A hybrid system also enables query rewrite transformations across language boundaries by seamless compilation (compilation with a single compiler in a single pass) of the XML querying functions of SQL/XML (i.e., `XMLQUERY`, `XMLEXISTS` and `XMLTABLE`)[6] into a single query graph.

In addition, we have developed several transformations specific to XQuery and XML navigation. Specifically, we push navigation down closer to the base data access to avoid navigating intermediate or constructed XML fragments and to exploit XML indexes. We also consolidate all path expressions in

a single FLWOR block into one pattern tree that is annotated with several flags. These flags represent whether a path is a FOR vs a LET path, whether an empty sequence needs to be created when there is no qualifying node (i.e., a node that is returned as a result of an XQuery), whether duplicates should be eliminated, and so forth. This pattern tree computes multiple bindings. Another important rewrite pushes down predicates in the where clause into binding path expressions, enabling XML index matching for value and general comparisons.

Index eligibility

An index is eligible for use during query evaluation if it can be proven that the index contains a superset of the results required for the query. We adapted the XPath containment algorithm described in Reference 20 to identify the indexes that are able to answer a part of a query. At a high level, this includes showing the following:

1. The query implies a predicate of the form: $col/<path-expr> <cmp> <expr>, where cmp is a comparison.
2. The indexed column is used in a FOR binding. LET quantification is not particularly useful because the entire result is required when the predicate is satisfied; that is, either all the results or none of the results are returned.
3. The path expression must produce the same set or a subset of the indexed nodes.
4. The data type of the comparison must match the data type of the index. The data types are not required to be identical; for example, all numeric comparisons match the double index. However, the comparison performed by the index must imply the required comparison. We perform type inferencing on the query graph to determine the type of the comparison based on the types of its arguments. Even with schema evolution, type inferences can be made. For example, literals, casts, arithmetic, and type tests all establish data types of parts of the query.

Example 2

```
for $dept in db2-fn:xmlcolumn('DEPT.DEPTDOC')/
       dept[@loc='CA'],
       $emp in $dept//employee
where $emp/@id > 901
return <empInfo>
                  {$emp/@id, $emp/name}
       </empInfo>
```

For the sample XQuery in the preceding example, we may create the following indexes:

```
create index I1 ...'/dept/@loc'
  as sql varchar(hashed)
create index I2 ...'//@id'
  as sql double
create index I3 ...'/dept/employee/@id'
  as sql double
create index I4 ...'/dept//employee/@id'
  as sql varchar(40)
```

Indexes I1 and I2 are eligible for the query in Example 2: (1) The predicates $doc/dept//employee/@id > 901 and doc/dept/@loc = 'CA' match the required form; (2) the indexed column (i.e. DEPT.DEPTDOC) is used in a FOR binding; (3) the path expression defining the index implies the path expressions in the predicates; and (4) the data types of the comparisons match. Index I3 cannot be used because the data might include /dept/sub-dept/employee/@id, which is not indexed, and I4 cannot be used because a numeric comparison is not compatible with a string index.

Physical plan generation

Physical plan generation is the phase in which the optimizer scans a QGM graph containing relational and XML entities and produces alternative execution plans. The optimizer utilizes data statistics to build a cardinality model, which is then used to estimate costs for the execution plans. Intermediate plans can be pruned, based on costs and plan properties such as the order of input data. The final plan with the cheapest cost is chosen for execution.

A physical plan is a model of query evaluation at runtime. Each physical operator models a runtime operation. Physical operators can be chosen at different granularities. They can be modeled at a primitive level, so that a complex runtime operation is composed by using a tree of these primitive operators. Alternatively, a complex runtime operation can be modeled by using one physical operator. The choice of physical operator affects cardinality and cost modeling. In DB2, new navigation and index runtime operations are introduced to support native XML processing; correspondingly, new physical operators are needed to model them. Our decision was to use one operator to model each.

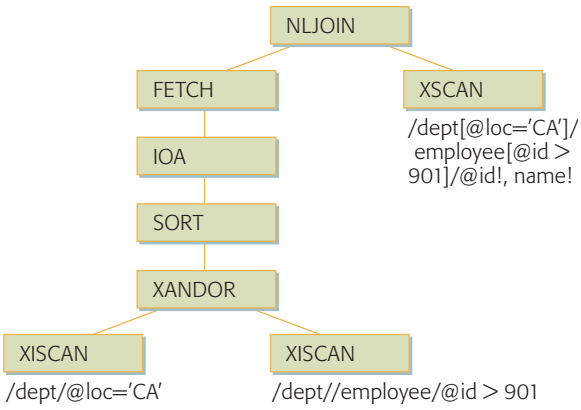

Figure 6
A physical plan for index ANDing

One of the reasons for this coarse-grained approach is that the complex new runtime operations cannot be broken down to trees of primitive operations, and it is not necessary to model the runtime operations in fine detail because there is no alternative in reordering the primitives at runtime. Modeling runtime operators in a coarse-grained manner also enables runtime operations to be flexible and adaptive, based on information available during execution.

We model the XML navigation runtime operation with the physical operator XML Scan (XSCAN, analogous to the relational table scan), and we model the index runtime operation with the physical operator XML Index Scan (XISCAN, analogous to relational index scan). A third new physical operator that we introduced is XANDOR, which models XML index ANDing and ORing. *Figure 6* shows an example that illustrates the execution plan generated for the query in Example 2.

Much of the relational optimizer infrastructure is reused, including rule-based plan generation, join enumeration, join-order and join-method selection, computation and propagation of properties, and the cardinality and costing framework used to cost and prune plans. In particular, by using the extensible rule-based plan-generation mechanism[21] and extensible operator and plan data structures, plans with the new XML operators (XSCAN, XISCAN, and XANDOR) are created by simply incorporating new rules. Because DB2 allows seamless compilation of XQuery and SQL into a single query graph, the optimizer is able to generate plans with mixed relational and XML operators and interchange them to produce alternative plans with different execution orders.

QUERY RUNTIME

To evaluate queries over XML data, DB2's relational query runtime was extended to support the XQuery and SQL/XML operators. There are two major components added for processing queries over XML: (1) XML index runtime and (2) XML navigation. In addition, several adaptations of existing relational runtime operators were required to support the new XML data type.

One issue that influences all aspects of the XML runtime is the dynamic nature of the XML data type. In the relational setting, all the types are known at compilation time. The types of the columns are specified at DDL time and are unambiguous. XML data might have no schema associated with it, a schema that has ambiguous type definitions (e.g., XML Schema[22] union construct), or in the extreme case, each XML element can be annotated with a basic type using the `xsi:type` attribute. To accommodate such uncertainty, the runtime support for XQuery relies on dynamic type dispatch.

XML index runtime

The XISCAN operator finds XML nodes that satisfy a predicate using an XML index. The general form of the predicate is `start-val` ≤ `path-expr` ≤ `stop-val`, which represents a range scan on the values of nodes with a path that matches the nonbranching path expression `path-expr`. Internally, this results in two or three implicit nested-loop join operations.

First, the path index is used to find the set of paths that match the path expression by scanning the range of `revPath` values for the "known" tail of the path expression. Subsequently, the `revPath` value is matched against the full pattern. For the path expression `//dept//name`, all `revPath` values between `name` and `namf` are scanned and then checked for a `dept` element.

For each matching path, the value index is then probed with `value.pathId = path.pathId`, and the bounds specified by setting `start-val` and `stop-val`. However, in our data model, the bounds themselves can be sequences, because cells in an SQL/XML table and `LET`s in XQuery represent sequences. For equality predicates, this results in

one scan of the value index per matching `pathId` and per item in the `start-val` sequence. For range predicates, only the minimum (or maximum) value of the start (or stop) value is required.

The XANDOR operator combines the nodes that are output from multiple XISCAN operations and implements branching path expressions through AND and OR operations, using only the XML indexes. Because we use node identifiers, we access only the nodes with predicates ("leaf" steps) and avoid accessing the large number of branching nodes (which do not have a predicate). The details of the XANDOR operation are beyond the scope of this paper, but the operation is similar in spirit to holistic twig joins.[23]

XML navigation

The XML navigation (XNAV) runtime module evaluates paths and predicate constraints for the native XML store by traversing the XML storage, following the parent-child relationship between the nodes. It returns node references (logical node identifiers) and atomic values to be further manipulated by other runtime operators. XNAV is represented in the optimizer plan by XSCAN operators, as shown in the example of Figure 6. Similar to the relational SCAN operator, XSCAN can also apply query predicates to reduce the size of the data returned by the operator. However, an XML document can correspond to one or several prejoined relational records, or even a whole database. This makes the XSCAN operator more complex in terms of query features as well as robustness.

Design principles

XNAV provides efficient processing of the pattern trees generated by the compiler. In XQuery language terms, it roughly corresponds to a FLWOR block, binding several variables. Unlike other approaches in which every XPath step is modeled as a separate operator,[24,25] a single XNAV operator can evaluate multiple steps from multiple XPath expressions in a query. This reduces the number of operators in the query plan and eliminates the overlap in the evaluation of the individual steps. Internally, XNAV breaks input query steps into path groups, each of which is evaluated by using a one-pass algorithm similar to the one described in Reference 26. As with the relational SCAN operator, XNAV interfaces with the rest of the runtime using a "tuple based" interface (i.e., reading or manipulating data on the tuple level, one record at a time) and returns tuples of bindings. Each binding has the XML data type and can be a singleton or a sequence of items, each item being a reference or an atomic value. During the traversal of a document, XNAV skips the nodes that will not affect the result of the query. Finally, XNAV uses limited memory and paged data structures for the intermediate results.

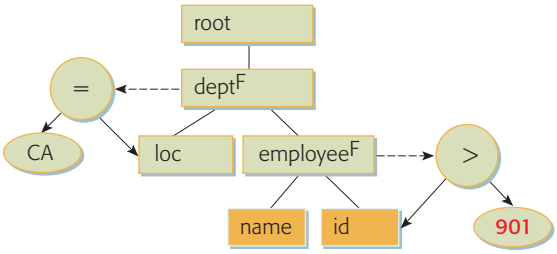

Figure 7
Example of an XNAV tree

XNAV trees

Path evaluation in XNAV is driven by the use of an XNAV tree, a runtime query tree representation. XNAV trees are produced during the threaded code generation phase of the QGM representation of the query. *Figure 7* shows the XNAV tree for the XSCAN operator in the query plan of Figure 6.

XNAV trees have structural and predicate parts. The structural part contains the XPath steps of the connected paths and has a query root node. Each node can have attached predicates. Predicate operator nodes can point back to the structural part of the XNAV tree when a structural node is an argument of an operator. In the example, XNAV reevaluates the predicates evaluated by the index to ensure the correctness of the result. In Figure 7, predicate operators are shown as oval nodes. The output of this example of an XNAV tree is tuples of the *id* and *name* bindings of the qualifying employees. These two nodes are the extraction points indicated in the figure by gold rectangles.

Tuple construction and buffer management

During navigation, nodes that could be part of the result or are needed for predicate evaluation are collected into node buffers. These buffers can contain a reference to a node or its atomic value, depending on the use of the node and the size of its atomic value. XNAV applies several techniques to reduce the number of buffer entries required.

When the result of XNAV is a tuple with multiple bindings, buffers for each of the extraction points (i.e., element and attribute nodes in an XML fragment that are extracted by XPath expressions) are put together into tuples with an algorithm that performs a variant of a merge-join operation over the node identifiers of the ancestors.

Multipass processing

The single-pass algorithm has several advantages: it preserves document order, uses predictive traversal, and often minimizes the number of visited nodes. However, there are several cases where the one-pass algorithm as described previously is not suitable. Although several branches usually can be evaluated in one pass, in some cases, the query might have branches that force navigation in different directions. For example, considering the query `//a[.//b > ../c]` from an a element node, we need to navigate both through the descendants of `a` and upward toward its parent. In such cases, the XSCAN operator builds a set of correlated XML navigations, each evaluating a group of XPath steps with a one-pass algorithm. Packaging more than one XNAV into one operator avoids the expense of the operator invocation and allows for the sharing of latched (or locked) storage pages across all the groups within a single operator.

XML SCHEMA SUPPORT

DB2 supports optional XML schema validation of documents during insert, update, and query operations. In addition, there is limited support for DTDs (Document Type Definitions) and external entities. The type annotations produced by the validation are stored together with the document for use during query execution. DB2 conforms to the XQuery standard,[5] the XML Schema standard,[22] and the XML standard[1] for these operations.

XML schema registration and validation

Before XML schemas can be used for validating documents, they need to be registered with the database. If validation is used, then the database relies on the XML schemas, stores type-annotated documents on disk, and compiles execution plans with references to the XML schemas. Additionally, stable and high performance access to schemas is required for efficient validation in XML insert, update, or query operations. These stability and performance requirements can only be met by storing the schemas in the database itself. Hence, DB2 provides an XML schema repository (XSR).

Internally, the schema repository consists of several new database catalog tables. These tables store the original XML schema documents that comprise an XML schema as well as a "binary representation" of the schema for fast reference during validation of a document. Users can retrieve information from the XSR as from any other catalog table. They can query the XSR tables for schema documents, target namespace, schema location, schema identifier, and other attributes. For example, they can write SQL queries that retrieve the identifier of the schema that was used to validate a stored document and process it accordingly.

Registration of XML schemas is done with DB2 commands, stored procedures, or language-specific APIs. The following is an example of registering a simple schema. Its schema URI (Uniform Resource Identifier) is `http://my.dept.com`, the file that contains the schema document is `dept.xsd`, the schema identifier in the database is `deptschema`, and it belongs to the relational database schema `departments`. Note that the namespace URI is deduced from the schema document itself.

```
register xmlschema http://my.dept.com
     from dept.xsd
     as departments.deptschema complete
```

Documents can be validated in SQL statements with the `XMLVALIDATE` function. The schema to be used for validation either can be specified explicitly, or it can be deduced from the `schemaLocation` hints in the instance documents. A schema can be explicitly referenced by its schema URI or by its schema identifier. The following example shows two insert statements, which validate the input document against the previously registered schema `deptschema`. Both statements specify the schema explicitly, by schema URI and by schema ID, respectively.

```
insert into dept(deptdoc) values
  xmlvalidate(? according to
              xmlschema uri 'http://my.dept.com')
insert into dept(deptdoc) values
  xmlvalidate(? according to
              xmlschema id departments.deptschema)
```

These statements illustrate that XML Schema validation in DB2 is performed on a per-document rather than a per-column basis. Each inserted document can potentially be validated against a different XML schema, demonstrating the flexibility of the DB2 XML store. This flexibility is necessary for document-centric applications, where organization and classification of documents is more important than homogeneity.

The following example shows an insert statement where no schema is referenced explicitly in the XMLVALIDATE function. In this case, DB2 tries to deduce the schema from the input document and find it in the repository.

```
insert into dept(deptdoc) values xmlvalidate(?)
```

Documents that include or refer to DTDs or external entities can also be inserted, but the DTD is used only to resolve entity references and to add default attributes and elements.

XML schema evolution and flexibility

The DB2 schema repository is based on two main design principles. First, the repository should not require users to modify a schema before it is registered or to modify XML documents before they are inserted and validated. In addition, once documents have been inserted and validated, they should never be invalidated and should never require updates to remain valid. Because XML applications often deal with large numbers of documents, bulk updates to make them compliant with a non-compatible schema change are almost always infeasible.

The second design principle for the DB2 XML schema repository is to enable schema evolution. Schema evolution is a sequence of changes in an XML schema over the course of its lifetime. Such changes usually occur due to new or evolving business needs, such as changing or introducing new services, products, or business processes.

How best to accomplish schema evolution has been a much debated topic, and there is currently no standard for evolving schemas. Fortunately, most applications do not need a solution to the general schema evolution problem; instead, they sufficiently constrain the problem so that relatively simple solutions are possible. Therefore, flexibility of the schema repository is of paramount importance. In practical terms, this means that DB2's schema repository does not require the namespace or the schema URI of each registered schema to be unique

■ A single XNAV operator can evaluate multiple XPath expressions in an XQuery ■

because the user might not have control over that. The user does have control over the database-specific schema identifier, which must be unique. The schema repository does not prescribe a specific way to perform schema evolution.

For the general schema evolution problem, one option is to allow the old and new schemas to exist side by side under different names. One can freely mix documents that conform to the old schema with documents that conform to the new schema in the same column of a table. Queries can also be written for that table to process only documents that conform to the old schema, to the new schema, or to both. To enable the application to perform more complicated version-aware operations, DB2 supplies a function to identify the schema that was used to validate a particular document. The following query returns the schema identifier of the schema that was used for validation of the XML document for department PR27.

```
select deptid, xmlxsrobjectid(deptdoc) from dept
where deptid = "PR27"
```

ANNOTATED SCHEMA DECOMPOSITION

Even though the DB2 native XML store can insert and query any XML document, there are cases where it is appropriate to shred XML documents into relational rows and columns. In certain usage scenarios, XML is used only to transport data to the database, and the XML structure is irrelevant once the data is integrated with existing relational data. For example, if an application extracts all relevant data from a Web Services message and decomposes that data into existing tables, then the original XML message might not be needed anymore. Shredding may also be required because many existing tools for data mining and business intelligence work only on the relational format of the data. Also, the performance of queries on relational data may be

superior to queries on XML data if the schema is sufficiently simple.

DB2 offers an improved decomposition product that maps XML data into relational tables. The decomposition process is driven by annotations inside the XML Schema, similar to schema-annotated mappings in MS-SQL Server[27] and Oracle.[28] These annotations are added to the schema by the user and describe the mapping of XML elements and attributes to tables and columns.

DB2 automates the decomposition process by using the annotated schema as input. The following is an example of an annotation. When a document is inserted and decomposed according to this part of an annotated schema, the value of the salary element under the payroll element is inserted into the salary column in table T. The DB2 decomposition annotations are in their own namespace and use the namespace prefix `db2-xdb`.

```
<xsd:element name="payroll">
  <xsd: complexType>
    <xsd:sequence>
      <xsd:element name="salary" type="xsd:string"
           db2-xdb:rowSet="T"
           db2-xdb:column="salary"/>
      <xsd:element name="bonus" type="xsd:integer"
           db2-xdb:rowSet="T"
           db2-xdb:column="bonus" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

The annotations enable the user to control the decomposition process in great detail: the data can be normalized and its white space manipulated, or the data can be manipulated in an expression or truncated before insertion; the data can be inserted conditionally (e.g., only values matching certain criteria should be decomposed into the table-column pairs); foreign key relationships can be described; the same element or attribute can be inserted into multiple table-column pairs; and multiple elements or attributes can be inserted into the same table-column pair.

Because XML is a first-class data type in DB2, decomposing an XML document can include inserting part or all of a document as an XML value into an XML column. Effectively, this allows an application to break an XML document into several pieces and to store only the required pieces in one or more XML columns.

XML API AND APPLICATION SUPPORT

Although the XML data type is stored and manipulated as a hierarchical data type, it is currently only externalized as a serialized XML string to applications. Each of the major database interfaces is optimized to make use of the XML data type natively and manages XML data with a focus on preservation of encoding information. Each interface supports inserting and retrieving serialized XML string data by using existing character and binary application data types.

The new types help avoid unnecessary code page conversions, which can occur when existing character types are used. By default, all XML data accessed through the application interfaces are returned with an XML declaration, including an encoding attribute. Most interfaces provide configuration options to override the default if required.

Applications can bind various language-specific data types for input and output of XML columns or parameters. These existing language-specific data types allow the user to work with XML data only as character or binary types. In order to use XML efficiently and seamlessly, new language-specific XML types are added to the existing client interfaces. These new XML types enable the database to be more efficient and to supply a richer API.

By making XML explicit in the application, the database avoids unnecessary and unwanted code page conversions. XML documents have an internal encoding declaration that makes all transcoding but that of the XML parser unnecessary. Transcoding an XML document without carefully modifying the XML encoding declaration might make the XML document invalid.

All of the major database interfaces support the XML type natively; that is, they treat XML data as XML, not as a character type. In the following subsections, we discuss XML type support in JDBC**, ODBC, .NET, and embedded SQL.

Java Database Connectivity

Java Database Connectivity (JDBC) has been enhanced to make XML data compatible with

strings, byte arrays, and streams, so that XML columns and parameters can be bound to any of these types. IBM is working on standardizing a JDBC XML type. In the meantime, a proprietary XML type `com.ibm.db2.DB2Xml` is available to enable an application to migrate seamlessly to the future standard JDBC type.

The DB2 XML interface has a number of methods that make working with XML data easy. In the following example, a column is retrieved as a DB2 XML object. Then the `getDB2String` method returns the serialized representation of the XML value (without an XML declaration) as a string object. The `getDB2XMLBinaryStream("UTF-16")` method then returns a binary stream with the XML value encoded in UTF-16 (Unicode Transformation Format with 16-bit encoding), including a matching XML declaration.

```
com.ibm.db2.jcc.DB2Xml.xml1=
  (com.ibm.db2.jcc.DB2Xml) rs.getObject
    ("xml_stuff");
String s=xml1.getDB2String();
InputStream is=xml2.getDB2XMLBinaryStream("UTF-16");
```

Call Level Interface

The DB2 Call Level Interface (CLI), a superset of ODBC, has been enhanced to support XML by providing a new SQL type, SQL_XML. Since there is no native XML type in C, the new SQL type can only be used in CLI/ODBC API calls to mark XML values as XML typed. In all other ways, access to serialized XML string data is identical to using a character array. The advantage is that the DB2 client and server know that this is XML data and can avoid unnecessary or unwanted code page conversions. Here is an example of inserting XML data into an XML-typed column:

```
char.xmlBuf[10240];
SQLExecDirect( hStmt, "Insert into t1 values (?)",
                                    SQL_NTS);
SQLBindParameter( hStmt, 1, SQL_PARAM_INPUT,
     SQL_C_CHAR, SQL_XML, xmlBuf, &xmlBufLen);
```

.NET Support

The goal of the DB2 .NET support is to integrate DB2 as thoroughly as possible with the .NET APIs. In this example, an XML document is extracted from DB2, and the application can use the standard .NET interface, `XmlReader`, to manipulate the result.

```
DB2Command cmd = DB2Connection.CreateCommand();
cmd.CommandText = "select c1 from T";
cmd.CommandType = CommandType.Text;
DB2DataReader dr = cmd.Execute();
dr.Read();
// retrieve the column as an XML reader
XmlReader xml = dr.GetXmlReader(0);
```

Embedded SQL

The SQL standard[6] defines new host variable declarations for XML types, and DB2 is using this in its implementation, as shown in the following example:

```
EXEC SQL BEGIN DECLARE;
SQL TYPE IS XML AS CLOB( 10K ) xmlBuf;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT xmlCol INTO :xmlBuf from myTable
       where id = '001';
```

NATIVE XML SUPPORT IN DB2 FOR Z/OS

As XML technology is increasingly embraced by enterprises for their on-demand e-business applications, management of XML data in mainframe databases becomes a necessity. DB2 for z/OS on the powerful IBM mainframe systems is the leading member of the DB2 family and plays a central role in data processing for many of the largest enterprises around the world. In this section, we briefly describe the main features of native XML support in DB2 for z/OS.

The initial functionality of native XML support in DB2 for z/OS is largely a compatible subset of that of DB2 UDB for the Linux**, Unix**, and Windows** platforms. It supports native storage of the hierarchical XQuery data model, indexing, efficient search of XML documents, and composition of new documents using SQL/XML functions. It supports most of the SQL/XML functions with embedded XPath queries and a limited set of types from the XQuery data model. XML schema validation is supported through a DB2-supplied user-defined function, although the type annotation is dropped when storing XML documents (i.e., the persistent XML documents are untyped).

All the utilities of DB2 for z/OS support the XML type. The XML type is also supported for non-Unicode tables and partitioned table spaces and in the data-sharing environment. More XQuery constructs, full XML schema support, and stand-alone XQuery are to be delivered in future releases.

As DB2 for z/OS is targeted at enterprise customers, special attention is given to its performance and scalability. The following is a list of some major design features enhancing the high performance of the native XML support:

1. A custom-made XML parser provided by the z/OS operating system is used for parsing. The parser provides a more efficient buffered token stream interface, instead of a SAX-like interface, to reduce event-handling overhead.
2. Well-tuned segmented table spaces (i.e., the universal table spaces) are used as the storage infrastructure for persistent XML data. Instead of using one record for each node, each record stores a packed subtree or subtrees of nodes in an internal table, significantly reducing the per-node cost. Nodes within each page (as described for DB2 UDB for Linux, Unix, and Windows) are stored within a record for DB2 for z/OS, and a node ID index plays the role of a regions index.

   An internal table space is created for each XML column in a base table. The internal table space contains a table with three columns (`DocID`, `minNodeID`, `XMLData`), where the `XMLData` column is a SQL `VARBINARY` type that contains packed XML data. Each `XMLData` column contains a single subtree or a sequence of subtrees that share a common parent node. The parent node is the context node for the record, containing the absolute node ID in the record header among other context information. Each node contains a relative node ID.
3. An optimal QuickXScan algorithm is used for streaming XPath evaluation. QuickXScan eliminates the combinatorial explosion of matching states commonly seen in streaming XPath algorithms and achieves linear scalability. The QuickXScan algorithm is also customized for even faster XML index key generation.
4. New access methods based on XML value indexes are used, similar to what has been described previously for DB2 UDB for Linux, Unix, and Windows.
5. A next-generation XML schema-validating parser based on parser-generator techniques is used for XML schema validation and annotated schema-based decomposition.

For more details, see Reference 29.

XML UTILITIES AND TOOLS

Standard DB2 utilities have been upgraded to work with the new XML type. For example, XML type column data is supported by DB2's "backup and restore" and by DB2's high-availability data replication for failover and fault tolerance.

The IMPORT and EXPORT functions offer a flexible way to insert or extract data to or from database tables. A single IMPORT command can populate any combination of relational and XML columns in a table. The IMPORT utility can read and import XML documents from any number of separate XML files in the file system. Alternatively, DB2 can import XML documents that are concatenated in a single large input file. Similarly, the EXPORT utility can write XML documents to separate files or concatenate them into a single file.

The IMPORT and EXPORT functions give the user fine-grained control of the XML parsing and validation options. These options are similar to the SQL/XML functions `XMLPARSE` and `XMLVALIDATE`. Validation of documents during import is optional. If validation is used, all imported documents can be validated against a single schema, or schemas can be specified on a per-document basis. Also, it is possible to validate some but not all documents during import. When XML data is exported, a flat file is written in addition to the XML data. This flat file may contain relational data that may have been part of the export. It also contains references to the exported XML documents. Optionally, a schema identifier is included for each exported document that was validated at the time of data insertion. Thus, the relationship between documents and schemas can be exported along with the actual data and can be used for validation when the data is imported into a database again.

XQuery is a functional query language that enables users to query XML data sources, including XML columns. Novice users may find the language fairly complex and difficult, even for simple queries. To alleviate this problem, DB2 provides a GUI-based XQuery builder. The XQuery builder exposes the XQuery language functionality as sets of grid, enabling the user to build fairly complex queries. The tool interprets users' GUI actions and generates the corresponding queries, greatly assisting in the construction and manipulation of XQuery syntax. The DB2 Developer Workbench provides GUI

support for defining XML indexes, XML schema annotations for decomposition, and basic tasks like defining XML columns in tables or viewing XML documents.

RELATED WORK

In recent years, many different approaches have been proposed for XML data management, in both academia and industry. They can be classified into two groups: native XML management systems and systems that reuse an underlying relational DBMS.

In relational-based approaches, the core storage and processing model for XML is the relational model, and a mapping between the XML data model and the relational data model is required. In contrast, a native XML database uses an XML data model, for example, the XML Infoset or the XQuery data model, as the core data model for the processing and storage of XML data. XML is not treated as text, and it is not mapped to a different data model. The data is represented as XML even with respect to its physical storage on disk.

Relational-based approaches either shred the XML documents into relational tables using some sort of encoding[8,30–32] or use a LOB (large object) column to store the XML document as text.[33–35] The main advantage of the relational-based approach is that it requires no modification to existing engines, while exploiting their maturity, extensive tuning, proven scalability, and sophisticated optimizers.

Shredding XML to relational tables is expensive at insertion time due to costly XML parsing[29] and multitable inserts, which require access to many database records. Once XML is broken into relational scalar values, queries and updates in plain SQL can be run with high performance. One important disadvantage of shredding-based methods is their inefficiency for retrieval of the whole or a subpart of the XML document, as well as their inflexibility regarding support of schema evolution. The required multiway joins required for reconstructing XML documents can be expensive when dealing with large amounts of data.[36]

Shredding-based approaches need to translate an XQuery into SQL for evaluation. As Reference 7 argues, due to the semantic mismatch between XQuery and SQL, not all XQuery expressions are translatable into SQL, or they may translate into inefficient SQL statements. A more comprehensive review of methods for XML-to-SQL query translation and their limitations is beyond the scope of this paper and can be found in Reference 37.

Generally, LOB-based storage allows fast insertion and retrieval of full document and full schema flexibility, since any XML document, irrespective of its schema, can be stored. But this approach suffers from poor search and extraction performance due to XML parsing at query execution time. Search performance can be improved if indexes are built at insertion time. Although this incurs XML parsing overhead, it may speed up queries for documents that match given search conditions. Compared to LOB-based approaches, DB2 enjoys a similar fast retrieval of full documents, as it stores the entire document together, but does not suffer from inefficient search, because it uses a directly traversable, parsed structured-tree format.

The second alternative to XML data management is to build a native XML database. Examples of native systems include TIMBER,[38] Niagara,[39] and Natix.[40] Systems such as Niagara and TIMBER break the XML document into nodes and store the node information in a "B+ tree" (a tree structure commonly used for relational database indexes) with all document nodes stored in order at the leaf level. This allows for efficient document or subtree reconstruction by a simple scan of the leaf pages of the tree. All the native XML systems deal only with XML data and do not support SQL or relational storage.

Relational databases have been offering support for storage, manipulation, search, and retrieval of XML data.[41–43] In Oracle 10g XML documents can be stored with indexing support as CLOBs (character large objects), shredded to object-relational tables, or a combination of these functions.[28] Microsoft SQL Server 2005 stores XML documents in a parsed, tokenized format as byte sequences in BLOB (binary large object) columns.[44] A primary XML index can be defined to avoid parsing the XML BLOBs at query time.[28,44] Additionally, secondary XML indexes can be defined to further increase query performance. This is somewhat different from DB2's XML storage and indexing approach described in the sections "Native XML storage" and "XML indexes." In DB2, XML parsing is never required at query time, and indexes can be defined on specific paths.

Finally, there are many XQuery implementations in both academia and industry. A comprehensive list of public XQuery implementations and links can be found on the home page of the W3C XQuery working group (http://www.w3.org/XML/Query).

SUMMARY AND CONCLUSIONS

DB2 Universal Database has been enhanced with comprehensive native XML support to overcome the limitations inherent in mapping XML to relational tables or CLOBs. XML documents are stored as type-annotated trees on disk pages, indexed with path-specific indexes, and queried with XQuery, SQL/XML, or a combination of both. Schema validation is optional and performed on a per-document basis, which allows for flexibility and schema evolution. Enhancements to the major database APIs provide client applications with the required functionality to exploit new XML capabilities in the DB2 server. The native XML solution in DB2 includes XML support in utilities such as XML import and export and a visual XQuery design tool.

We have described the architecture and overall design of native XML in DB2, a hybrid relational and XML data-management system. To the best of our knowledge, this is the first truly hybrid system to support both relational and XML data. We believe such a system is essential to the evolution of enterprise data-management solutions, as XML and relational data will coexist and complement each other.

A hybrid system enables easier incorporation of more traditional data management tools, such as triggers and materialized views, into XML data management. The DB2 XML system allows us to leverage more than 20 years of data-management research to advance XML technology to the same sophistication expected from mature relational systems.

ACKNOWLEDGMENTS

We would like to thank and recognize the large number of engineers at the IBM Toronto Lab, IBM Silicon Valley Lab, IBM Almaden Research Center, IBM Portland Lab, and IBM Thomas J. Watson Research Center for their contributions to integrating native XML support into DB2. We would also like to thank Sriram Padmanabhan for his valuable feedback and suggestions in the preparation of this paper.

CITED REFERENCES

1. M. Birbeck, M. Kay, S. Livingstone, S. F. Mohr, J. Pinnock, B. Loesgen, S. Livingston, D. Martin, N. Ozu, M. Seabourne, and D. Baliles, *Professional XML*, Wrox Press, John Wiley and Sons, Hoboken, NJ (2000).
2. R. P. Bourret, personal communication.
3. Bioinformatics, O'Reilly XML.com, http://www.xml.com/pub/rg/Bioinformatics.
4. H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson, "TAX: A Tree Algebra for XML," *Proceedings of the International Workshop on Data Bases and Programming Languages (DBPL 2001)*, in *LNCS* **2937**, Springer-Verlag, Berlin (2002), pp. 149–164.
5. *XQuery 1.0: An XML Query Language*, S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, and J. Simeon, Editors, W3C Candidate Recommendation (November 2005), http://www.w3.org/TR/xquery.
6. *Information Technology—Database Language SQL—Part 14: XML-Related Specifications (SQL/XML)*, International Organization for Standardization (ISO), ANSI/ISO/IEC 9075-14:2005 draft under development (March 11, 2006).
7. I. Manolescu, D. Florescu, and D. Kossmann, "Answering XML Queries on Heterogeneous Data Sources," *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB 2001)*, Morgan Kaufmann, San Francisco, CA (2001), pp. 241–250.
8. D. DeHaan, D. Toman, M. P. Consens, and T. Özsu, "A Comprehensive XQuery to SQL Translation Using Dynamic Interval Encoding," *Proceedings of the 22nd International ACM SIGMOD Conference on Management of Data (SIGMOD 2003)*, ACM Press, New York (2003), pp. 623–634.
9. K. S. Beyer, R. 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 International ACM SIGMOD Conference on Management of Data (SIGMOD 2005)*, ACM Press, New York (2005), pp. 347–358.
10. H. Pirahesh, J. M. Hellerstein, and W. Hasan, "Extensible/Rule Based Query Rewrite Optimization in Starburst," *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data (SIGMOD 1992)*, ACM Press, New York (1992), pp. 39–48.
11. *XQuery 1.0 and XPath 2.0 Data Model (XDM)*, M. F. Fernandez, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh, Editors, W3C Candidate Recommendation (November 2005), http://www.w3.org/TR/xpath-datamodel/.
12. R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korthet, "Covering Indexes for Branching Path Queries," *Proceedings of the 21st International ACM SIGMOD Conference on Management of Data (SIGMOD 2002)*, ACM Press, New York (2002), pp. 133–144.
13. M. Nicola and B. Van der Linden, "Native XML Support in DB2 Universal Database," *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB*

2005), Morgan Kaufmann, San Francisco, CA (2001), pp. 1164–1174, http://www.vldb2005.org/program/paper/thu/p1164-nicola.pdf.

14. *XML Path Language (XPath) 2.0*, A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, and J. Simeon, Editors, W3C Candidate Recommendation (November 2005), http://www.w3.org/TR/xpath20.
15. *XQuery 1.0 and XPath 2.0 Formal Semantics*, D. Draper, P. Fankhauser, M. Fernandex, A. Malhotra, K. Rose, M. Rys, J. Simeon, and P. Wadler, Editors, W3C Candidate Recommendation (November 2005), http://www.w3.org/TR/xquery-semantics/.
16. L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. J. Shekita, "Starburst Mid-Flight: As the Dust Clears," *IEEE Transactions On Knowledge and Data Engineering* **2**, No. 1, 143–160 (1990).
17. Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, and S. Paparizos, "From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery," *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003),* Morgan Kaufmann, San Francisco, CA (2003), pp. 237–248.
18. M. F. Fernandez and D. Suciu, "Optimizing Regular Path Expressions Using Graph Schemas," *Proceedings of the Fourteenth International Conference on Data Engineering (ICDE 1998)*, IEEE Computer Society Press, J. Wiley, Hoboken, NJ (1998), pp. 14–23.
19. G. Grahne and A. Thomo, "Algebraic Rewritings for Optimizing Regular Path Queries," *Theoretical Computer Science* **296**, No. 3, 453–471 (2003), http://portal.acm.org/citation.cfm?id=782741&dl=ACM&coll=GUIDE.
20. A. Balmin, F. Özcan, K. Beyer, R. J. Cochrane, and H. Pirahesh, "A Framework for Using Materialized XPath Views in XML Query Processing," *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004),* Morgan Kaufmann, San Francisco, CA (2004), pp. 60–71.
21. G. M. Lohman, "Grammar-like Functional Rules for Representing Query Optimization Alternatives," *Proceedings of the ACM SIGMOD Conference on Management of Data (SIGMOD 1998),* ACM Press, New York (1998), pp. 18–27.
22. XML Schema, W3C Architecture Domain, http://www.w3.org/XML/Schema.
23. N. Bruno, N. Koudas, and D. Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching," *Proceedings of the ACM SIGMOD Conference on Management of Data (SIGMOD 2002),* ACM Press, New York (2002), pp. 310–321.
24. D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, and A. Sundararajan, "The BEA/XQRL Streaming XQuery Processor," *VLDB Journal* **13**, No. 3, 294–315 (2004).
25. J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk, "Querying XML Views of Relational Data," *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB 2001)*, Morgan Kaufmann, San Francisco, CA (2004), pp. 261–270.
26. V. Josifovski, M. Fontoura, and A. Barta, "Querying XML Streams," *VLDB Journal* **14**, No. 2, 197–210 (2005).
27. SQLXML in MS SQL Server 2000, http://msdn.microsoft.com/sqlxml.
28. Oracle XML DB 10g, http://www.oracle.com/technology/tech/xml/xmldb.
29. G. Zhang, "Building a Scalable Native XML Database Engine on Infrastructure for a Relational Database," *Proceedings of the Second International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P 2005)*, ACM Press, New York (2005), http://www.geocities.com/zhanggene/pub/ScalableNativeXMLDB.pdf.
30. M. Nicola and J. John, "XML Parsing: A Threat to Database Performance," *Proceedings of the 12th International Conference on Information and Knowledge Management (CIKM 2003),* ACM Press, New York (2003), pp. 175–178.
31. M. F. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W. C. Tan, "SilkRoute: A Framework for Publishing Relational Data in XML," *ACM Transactions on Database Systems* **27**, No. 4, 438–493 (2002), http://www.cs.washington.edu/homes/suciu/file07_paper.pdf.
32. D. Florescu and D. Kossmann, "Storing and Querying XML Data Using an RDBMS," *IEEE Data Engineering Bulletin* **22**, No. 3, 27–34 (1999).
33. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton, "Relational Databases for Querying XML Documents: Limitations and Opportunities," *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB 1999),* Morgan Kaufmann, San Francisco, CA (1999), pp. 302–314.
34. L. Ennser, C. Delporte, M. Oba, and K. Sunil, *Integrating XML with DB2 Extender and DB2 Text Extender*, IBM Redbooks (2000), http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg246130.pdf.
35. Microsoft TechNet: SQL Server 2000, http://www.microsoft.com/technet/prodtechnol/sql/2000/default.mspx.
36. S. Pal, I. Cseri, O. Seeliger, G. Schaller, L. Giakoumakis, and V. Zolotov, "Indexing XML Data Stored in a Relational Database," *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB 2004)*, Morgan Kaufmann, San Francisco, CA (2004), pp. 1146–1157, http://www.vldb.org/conf/2004/IND5P2.PDF.
37. R. Krishnamurthy, R. Kaushik, and J. F. Naughton, "XML-to-SQL Query Translation Literature: The State of the Art and Open Problems," *Proceedings of the 1st International XML Database Symposium (XSym)*, in *LNCS* **2824**, Springer-Verlag, Berlin (2003), pp. 1–18, http://www.cs.wisc.edu/~sekar/research/xmltosqlsurvey.pdf.
38. H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu, "TIMBER: A Native XML Database," *VLDB Journal* **11**, No. 1, 274–291 (2002), http://www.eecs.umich.edu/db/timber/files/timber.pdf.
39. A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. Viglas, Y. Wang, J. F. Naughton, and D. J. DeWitt, "Mixed Mode XML Query Processing," *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003),* Morgan Kaufmann, San Francisco, CA (2003), pp. 225–236.
40. T. Fiebig, S. Helmer, C.-C. Kanne, J. Mildenberger, G. Moerkotte, R. Schiele, and T. Westmann, "Anatomy of a Native XML Base Management System," *VLDB Journal* **11**, No. 4, 292–314 (December 2002).
41. K. S. Beyer, F. Özcan, S. Saiprasad, and B. Van der Linden, "DB2 XML: Designing for Evolution," *Proceedings of the 2005 International ACM SIGMOD Conference on Management of Data (SIGMOD 2005),* ACM Press, New York (2005), pp. 948–952.

42. R. Murty, Z. H. Liu, M. Krishnaprasad, S. Chandrasekar, A.-T. Tran, E. Sedlar, D. Florescu, S. Kotsovolos, N. Agarwal, V. Arora, and V. Krishnamurthy, "Toward an Enterprise XML Architecture," *Proceedings of the 2005 International ACM SIGMOD Conference on Management of Data (SIGMOD 2005),* ACM Press, New York (2005), pp. 953–957.

43. Michael Rys, "XML and Relational Database Management Systems: Inside Microsoft SQL Server 2005," *Proceedings of the 24th International ACM SIGMOD Conference on Management of Data (SIGMOD 2005)*, ACM Press, New York (2005), pp. 958–962.

44. S. Pal, I. Cseri, O. Seeliger, M. Rys, G. Schaller, W. Yu, D. Tomic, A. Baras, B. Berg, D. Churin, and E. Kogan, "XQuery Implementation in a Relational Database System," *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB 2005)*, Morgan Kaufmann, San Francisco, CA (2005), pp. 1175–1186, http://www.vldb2005.org/program/paper/thu/p1175-pal.pdf.

Accepted for publication January 17, 2006.
Published online April 27, 2006.

Kevin Beyer
IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120-6099 (kbeyer@us.ibm.com). Dr. Beyer is a research staff member at the IBM Almaden Research Center in San Jose, California. He is a member of the DB2 XML project team and a leader of the design and implementation of the XML indexing infrastructure. Prior to his work on XML, Dr. Beyer investigated materialized views and OLAP technologies.

Roberta Cochrane
IBM Software Group, 294 Route 100, Somers, New York 10589 (bobbiec@almaden.ibm.com). Dr. Cochrane is a Senior Technical Staff Member in IBM's Software Group Strategy division. She is a leader in the delivery of advanced query technology to IBM's database products, providing many new advanced features over the last 15 years, including materialized views, triggers, and constraints. She has conducted extensive research in active database systems and played a major role in the definition of the SQL3 standard for triggers and constraints. Dr. Cochrane is a member of the IBM Academy of Technology, a Master Inventor, and was one of IBM's 2002 YWCA TWIN awardees, honoring women in industry. She received a B.S. degree in computer science and mathematics from James Madison University, Virginia and a Ph.D. degree in computer science from the University of Maryland at College Park. The work described in this paper was done under her leadership as a manager of XML Database Technologies in the Exploratory Database department at IBM Almaden Research Center.

Michael Hvizdos
IBM Software Group, 8200 Warden Ave., Toronto Lab, Markham, ON L6G 1C7, Canada (hvizdos@ca.ibm.com). Mr. Hvizdos is an advisory software developer at the IBM Toronto lab, responsible for the design and implementation of the DB2 CLI/ODBC and embedded SQL application interfaces to XML data and XQuery. He has over seven years of experience supporting, maintaining, and enhancing DB2 application interfaces and has contributed to numerous editions of the IBM DB2 UDB Family Fundamentals and Application Development certification examinations. He received a Bachelor of Science degree in electrical engineering in 1998 from the University of Calgary. Mr. Hvizdos continues to enhance the functionality, reliability, and usability of the DB2 application interfaces with particular focus on XML efficiency.

Vanja Josifovski
Yahoo! Research, 701 First Avenue, Sunnyvale, California 94089 (vanjaj@yahoo-inc.com). Dr. Josifovski is a senior research scientist at the Yahoo Research Laboratory where he is a member of the group exploring search and advertisement technologies for the Internet. Previously, Dr. Josifovski was a research staff member at the IBM Almaden Research Center, working on several projects in database runtime and optimization, federated databases, and enterprise search projects. He earned an M.Sc. degree from the University of Florida at Gainesville and a Ph.D. degree from the Linkoping University in Sweden. His technical interests include scalable infrastructure for search, incremental indexing of semistructured and unstructured data, and advanced search technologies for improved result relevance.

Jim Kleewein
Microsoft Corporation, 1 Microsoft Way, Redmond, Washington 98052 (jimk@microsoft.com). Mr. Kleewein has spent roughly the last 20 years working on the design and implementation of data management systems, working on projects ranging from row-level locking on DB2/MVS™, sysplex data sharing, and data federation to geospatial data management. He recently joined Microsoft's unified communications group.

George Lapis
IBM Silicon Valley Laboratory, 553 Bailey Avenue, San Jose, California 95141 (lapis@almaden.ibm.com). Mr. Lapis is a technical manager at IBM's Silicon Valley Lab, managing the DB2 XML compiler development team. He has worked in database software for more than 25 years. He was a member of the R* and Starburst research projects at IBM's Almaden Research Center in San Jose, California. He also was a member of the compiler development team for several releases of DB2 Universal Database. His expertise is mostly in compiler technology and implementation. For the last several years he has led the compiler development team at IBM's Silicon Valley Laboratory in San Jose, California, working on SQL, XML, and XQuery for DB2 Universal Database.

Guy Lohman
IBM Almaden Research Center, 650 Harry Road, B1-434, San Jose, California 95120 (lohman@almaden.ibm.com). Dr. Lohman is manager of Advanced Optimization in the Advanced Database Solutions department at IBM Research Division's Almaden Research Center in San Jose, California. He has over 23 years of experience in relational query optimization. He is the architect of the optimizer for the DB2 Universal Database for Linux, UNIX, and Windows, and was responsible for its development in Versions 2 and 5, as well as for the invention and prototyping of Visual Explain. During that period, Dr. Lohman also managed the overall effort to incorporate into the DB2 UDB product the Starburst compiler technology that was prototyped at the Almaden Research Center. More recently, he was a co-inventor and designer of the DB2 Index Advisor (now part of the Design Advisor), and cofounder of the DB2 Autonomic Computing project, part of IBM's company-wide Autonomic Computing initiative. In 2002, Dr. Lohman was elected to the IBM Academy of Technology. His current research interests involve query optimization, self-managing database systems, and problem determination.

Robert Lyle
IBM Software Group, 555 Bailey Ave., Silicon Valley Laboratory, San Jose, California 95141 (blyle@us.ibm.com). Mr. Lyle is a member of the DB2 XML project team and a leader of the design and implementation of the XML indexing

infrastructure. He has worked in database software for over 17 years. Prior to his work on XML, Mr. Lyle worked on introducing OLAP functions into the DB2 Universal Database for Linux, UNIX, and Windows. Before that, he spent 11 years working on DB2 for z/OS, where he was the technical lead and owner of the indexing component and led the development of the type-2 index manager.

Matthias Nicola
IBM Software Group, 555 Bailey Ave., Santa Teresa Lab, San Jose, California 95141 (mnicola@us.ibm.com). Dr. Nicola is the technical lead for XML database performance at IBM's Silicon Valley Lab. His work focuses on all aspects of XML performance in DB2, including XQuery, SQL/XML, and all native XML features in DB2. Dr. Nicola works closely with the DB2 XML development teams as well as with customers and business partners who are using XML, assisting them in the design, implementation, and optimization of XML solutions. Prior to joining IBM, Dr. Nicola worked on data warehousing performance for Informix Software. He also worked for four years in research and industry projects on distributed and replicated databases; he received his doctorate in computer science in 1999 from the Technical University of Aachen, Germany.

Fatma Özcan
IBM Almaden Research Center, 650 Harry Road, B1-434, San Jose, California 95120 (fozcan@almaden.ibm.com). Dr. Özcan has been a research staff member at IBM's Almaden Research Center since 2001. She is a member of the DB2 XML compiler team and a technical leader in the area of XML query languages, XQuery semantics, and rewrite optimization. She received a Ph.D. degree in computer science from the University of Maryland at College Park in 2001. Her research interests include XML query languages and query optimization, integration of heterogeneous information systems, and software agents. She is a member of ACM SIGMOD and coauthor of the book *Heterogeneous Agent Systems*.

Hamid Pirahesh
IBM Almaden Research Center, 650 Harry Road, B1-434, San Jose, California 95120 (pirahesh@almaden.ibm.com). Dr. Pirahesh is an IBM fellow and a senior manager responsible for the Exploratory Database department at the Almaden Research Center in San Jose, California. He also has direct responsibilities for various aspects of IBM information management products. He received a Ph.D. degree from the University of California at Los Angeles in the area of database systems. Dr. Pirahesh is an IBM Master Inventor and a member of the IBM Academy. He has served as an associate editor of *ACM Computing Surveys* and has served on the program committee of major computer conferences. He was a principal member of the original team that designed the query-processing architecture of the IBM DB2 Universal Database relational DBMS and delivered the product to the marketplace. He has made major contributions to query-language industry standards. His research areas include OLAP and aggregate data management, query optimization, data warehousing, service-oriented architecture, management of semistructured (XML) and unstructured data, and information integration in Web-based federated and distributed systems. He also serves as a consultant to various IBM divisions, including IBM Software Group and IBM Global Services.

Normen Seemann
IBM Software Group, 555 Bailey Ave., Santa Teresa Lab, San Jose, California 95141 (normsee@us.ibm.com). Mr. Seemann is an advisory software engineer in IBM's Silicon Valley Laboratory. He is a member of the DB2 XML compiler team, focusing on XQuery semantics and modeling, XPath, and XQuery rewrite transformations. He received an M.Sc. degree in computer science from the University of Rostock, Germany and joined the IBM Software Group in 2000.

Ashutosh Singh
IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120 (ashutosh@almaden.ibm.com). Mr. Singh is an advisory engineer in the Advance Database Solutions department. He received a Masters degree from the University of Wisconsin at Madison and joined IBM Research in 1999. Mr. Singh's eight years in the IT industry include research, design, and development in the area of relational database systems. He has contributed to the design and development of XML as well as relational technology in the DB2 Universal Database for Linux, UNIX and Windows. His areas of interest include sampling technology in databases, database performance, high availability, query optimization, data and application integration, text search, and data mining. Prior to joining IBM, he was responsible for the architecture and implementation of the stored procedure execution module for the PARADISE database server at the University of Wisconsin at Madison and later at the NCR Corporation.

Tuong Truong
IBM Software Group, 555 Bailey Ave., Santa Teresa Lab, San Jose, California 95141 (tctruong@us.ibm.com). Mr. Truong is a senior software engineer and manager, managing the XML Query Runtime department. His team is responsible for extending the DB2 runtime relational engine to support the evaluation of the XQuery language. Mr. Truong received a B.S. degree in computer science in 1990 and an M.B.A. degree in 1998, both from San Jose State University. He joined the IBM Software Group in 1991. Mr. Truong's 15 years experience in the database industry includes the architecture, design, and implementation of systems in technology areas such as relational databases, business intelligence, data replication, massively parallel systems, and XML databases.

Robbert C. Van der Linden
IBM Software Group, 555 Bailey Ave., Santa Teresa Lab, San Jose, California 95141 (robbert@us.ibm.com). Mr. Van der Linden joined IBM in 2001 to work on the XML project in DB2 as one of the early architects. He came to IBM from a startup company, Propel, where he led the design and implementation of the distributed and fault-tolerant middleware which hosted a scalable e-commerce application. Before that Mr. Van der Linden worked for many years at Tandem Computers on NonStop SQL, a database that runs many critical applications in the financial industry.

Brian Vickery
(bvickery@acm.org). Mr. Vickery has spent roughly the last 10 years working on the design and implementation of data management systems, at various companies, including BEA, IBM, Propel, Tandem, and elsewhere. His contribution to the work described in this paper was done while he was working at the IBM Silicon Valley Laboratory as a member of the DB2 XML team. Mr. Vickery designed and implemented the XML storage layer for DB2 XML.

Chun Zhang
Cosmix Corporation, 444 Castro Street, Suite 700, Mountain View, California 94041 (chunz@kosmix.com). Dr. Zhang is currently a member of the technical staff at Cosmix Corporation in Mountain View, California. Her contribution to the work described in this paper was done while she was a research staff member at the IBM Almaden Research Center and member of the DB2 XML team focusing on the design and development of the query optimizer. She has extensive experience in structured, semistructured, and unstructured data management. Dr. Zhang's current work involves text mining and search engine technologies. She is also interested in storage, query processing, and optimization. Dr. Zhang

received her Ph.D. degree from University of Wisconsin at Madison.

Guogen Zhang
IBM Software Group, 555 Bailey Ave., Santa Teresa Lab, San Jose, California 95141 (gzhang@us.ibm.com). Dr. Zhang is a Senior Technical Staff Member in the development organization of DB2 Universal Database for z/OS. He attained a B.S. degree in computational mathematics from Hangzhou University in China in 1984, an M.S. degree in computer science from Jinan University in China in 1987, a second M.S. degree in computer science from the University of Kansas in 1994, and a Ph.D. degree in computer science from UCLA in 1998. He subsequently joined IBM at the Silicon Valley Laboratory and currently is an architect responsible for XML support in DB2 Universal Database for z/OS. He has received numerous Outstanding Technical Achievement awards and invention achievement awards for his contributions to DB2. Dr. Zhang is a member of the Association for Computing Machinery and the Institute of Electrical and Electronics Engineers. ■