Native XML Support in DB2 Universal Database

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

Bert van der Linden
IBM Silicon Valley Lab
555 Bailey Avenue
San Jose, CA
USA
robbert@us.ibm.com

Abstract

The major relational database systems have been providing XML support for several years, predominantly by mapping XML to existing concepts such as LOBs or (object-)relational tables. The limitations of these approaches are well known in research and industry. Thus, a forthcoming version of DB2 Universal Database® is enhanced with comprehensive *native* XML support. "Native" means that XML documents are stored on disk pages in tree structures matching the XML data model. This avoids the mapping between XML and relational structures, and the corresponding limitations. The native XML storage is complemented with XML indexes, full XQuery, SQL/XML, and XML Schema support, as well as utilities such as a parallel high-speed XML bulk loader. This makes DB2 a true hybrid database system which places equal weight on XML and relational data management.

1 Introduction

XML is the de-facto standard for exchanging data between different systems, platforms, applications, and organizations. 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 semi-structured data. XML data is easy to extend because new tags can be defined as needed. Also, 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, and XML Schema. They greatly relieve applications from the burden of dealing with particularities of proprietary data formats. In an era where message formats, business forms and services change frequently, XML reduces the cost and time it takes to maintain application logic correspondingly.

Beyond XML for data exchange, enterprises are keeping large amounts of business critical data permanently in XML format. This has various reasons. Some businesses must retain XML documents in their original format for auditing and regulatory compliance. Typical examples are legal and financial documents as well as eForms, particularly in the government sector.

Another reason for using XML as a permanent storage format is that XML can be a more suitable data model than a relational schema. This is not only true for content-oriented applications, but also for certain data-oriented applications. For example, in life science applications the data is highly complex and hierarchical in nature and yet may contain significant amounts of unstructured information. Most of today's genomic data is still kept in proprietary flat file formats but major efforts are under way to move to XML [14].

Relational databases have been offering support for storage, manipulation, search, and retrieval of XML data. This is usually based on storing XML documents in LOBs or mapping and shredding XML to a relational schema. These solutions have inherent functional and performance constraints. Generally, LOB-based storage allows for fast insert and retrieval of full documents but suffers from poor search and extract performance due to XML parsing at query execution time. This can be moderately improved if indexes are built at insert time. While this incurs XML parsing overhead, it may speed up queries that look for documents which match given search conditions. Yet, extraction of document fragments and sub-document level updates still require expensive XML parsing.

**Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005**

Shredding XML to relational tables is expensive at insert time due to costly XML parsing [10] and multi-table inserts. But once XML is broken into relational scalar values, queries and updates in plain SQL promise higher performance. Still this approach suffers from drawbacks: XML schemas can have many nested and repeating elements such that the corresponding relational schema would consist of dozens or even hundreds of tables. Defining such a mapping from XML to a relational schema is a complicated task. Once data has been inserted, any changes to the relational schema -due to changes in the XML Schema- are almost always infeasible. This severely restricts the flexibility which XML is often used for in the first place. Also, the required multi-way joins to reconstruct XML documents can be expensive when dealing with large amounts of data [12]. Beyond that, complex XQueries can even be untranslatable into SQL [5].

This motivates native XML database technology. DB2 Universal Database® has been extended with comprehensive native XML support. In this paper we present the XML features in the upcoming version of DB2 and describe some of the key implementation concepts. We discuss examples to illustrate the XML capabilities as well as the integration of XML with SQL and relational data management.

After pointing to related work in section 2, we provide an overview of DB2's native XML solution and its high level architecture in section 3. Sections 4 and 5 then present the native XML storage and XML indexing mechanism, respectively. The XQuery and SQL/XML support is explained in section 6. This is followed by a description of XML schema support and the DB2 schema repository in section 7. Section 8 explains why shredding remains an important XML feature and presents DB2's new shredding solution. Application support and API enhancements for XML are covered in section 9. Then section 10 gives an overview of various tools for XML, such as XML import, export and load, and the graphical XQuery builder. Finally, this paper concludes with a summary.

2 Related Work

Various native XML databases have been in existence for several years, such as Tamino, XHive, Ipedo, NeoCore, Xyleme, and others [4]. The XML storage approach described in [7] is similar to ours in the sense that large documents are split into subtrees of nodes.

In Oracle 10g XML documents can be stored with indexing support as CLOBs, shredded to object-relational tables, or a combination of both [11]. The XML support in Microsoft SQL Server 2005 stores XML documents as byte sequences in BLOB columns [12]. A primary XML index can be defined to avoid parsing the XML BLOBs at query time [12]. 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 sections 4 and 5. In DB2, XML parsing is never required at query time and indexes can be defined on specific paths. The upcoming XML support in DB2 is based on a prototype described in [2]. The more general modeling and architectural concepts can also be found in [2] and are not covered in this paper. Further related work is discussed in [9] and [2].

3 Overview: "The Big Picture"

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 plain SQL, SQL/XML [6] and XQuery [3] in an integrated manner. Different parsers are used to read SQL and XQuery statements but then a single compiler is used for both languages. There is no translation from XQuery to SQL. DB2's compiler and optimizer are extended to handle SQL and XQuery in a single modeling framework [2]. Corresponding processing support is built into the index manager, the runtime system, memory management, the data dictionary, concurrency control, the storage layer, and database utilities.

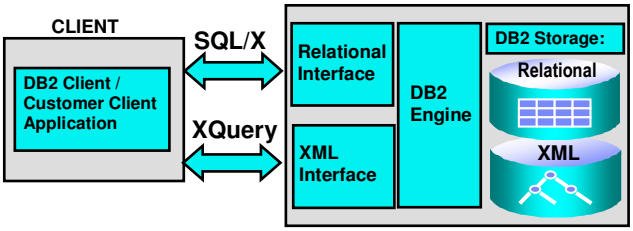

Figure 1: Integrating XML and Relational in DB2

There is no impact on existing SQL applications. A client application can continue to use SQL to communicate with the DB2 Server through the relational APIs to access and manipulate data in the relational data store. The SQL/XML extensions also allow publishing of relational data in XML format. Additionally, SQL allows full document retrieval from the native XML storage. New SQL/XML functions provide SQL applications also with sub-document level search and extract capabilities, i.e. by embedding XPath or XQuery into SQL statements.

An XML application can interact with the DB2 Server through the XML interface using the XQuery language. XQueries typically access the native XML store. XQuery is supported as a standalone query language independent from SQL. Yet, XQueries can optionally contain SQL statements to combine and correlate XML with relational data. Since an XUpdate language is not yet close enough to standardization, the DB2 server supports full document updates for now. An XML update stored procedure is available which provides applications with a flexible interface for sub-document level updates. This also eliminates the need to send documents for update from the DB2 server to the client and back.

3.1 The XML Data Type

At the heart of DB2's native XML support is the XML data type. XML is now a first-class data type in DB2, just like any other SQL type [6]. The XML data type can be used in a "create table" statement to define one or more columns of type XML (Figure 2). Since XML has no different status than any other types, tables can contain any combination of XML columns and relational columns. An XML-only application may define tables that contain XML columns only. A column of type XML can hold one well-formed XML document for every row of the table. The NULL value is used to indicate the absence of an XML document. Though every XML document is logically associated with a row of a table, XML and relational columns are stored differently. Relational and XML data are stored in different formats that match their respective data models. The relational columns are stored in traditional row structures while the XML data is stored in hierarchical structures. The two are closely linked for efficient cross-access.

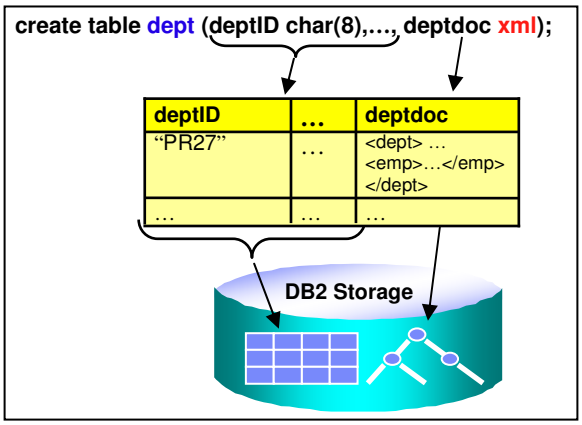

Figure 2: Table with a column of type "XML"

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 on a per-document basis. Thus, the association between schemas and documents is per document and not per column, which provides maximum flexibility.

Unlike a Varchar or a CLOB type, 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 communication protocol limits XML bind-in and bind-out to 2GB per document. With very few exceptions, this is acceptable for all XML applications.

Values of type XML are processed in an internal representation that is not a string and not directly comparable to strings. The XMLSERIALIZE function can be used to convert an XML value into a string value which represents the same XML document. Similarly, the XMLPARSE function can be used to convert a string value which 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 in languages such as C, Java, and COBOL. Section 9 provides details on this extension to the DB2 APIs. The XML type is also allowed for parameters and 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.

4 Native XML Storage

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 parser to check incoming documents for wellformedness and to perform optional validation. The SAX events are converted into a hierarchical representation of the XML document. For the sample document in Figure 3, this hierarchy looks similar to the document tree in the upper part of Figure 4.

```
<dept>
  <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>
```

Figure 3: Sample Document

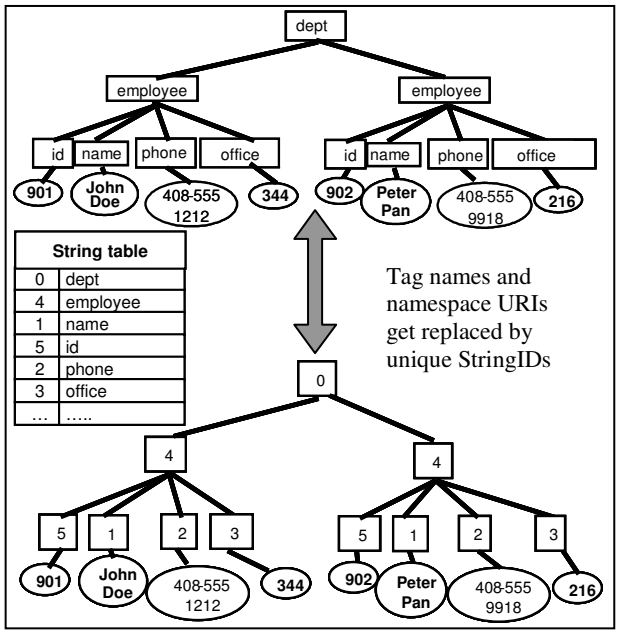

Figure 4: StringIDs in XML Storage

During insert, all tag names and namespace URIs in the document tree are replaced by integer values (StringIDs). The new catalog table SYSXMLSTRINGS holds the mapping from tags to StringIDs for all XML columns in the database. There is only one entry per

unique string. In the example in Figure 3 and 4, the tag "id" occurs twice in the sample document, and possibly many more times in other documents in the current database, but each occurrence of that tag is replaced by the same StringID "5". Upon the first database-wide insert of a tag, a StringID is assigned and registered in the string table. In all subsequent occurrences the tag is replaced with that same StringID. The size of the mapping table is usually very small since it corresponds to the number of unique tags in the database (typically hundreds or thousands). A special purpose cache ensures high performance access to this table.

The document tree in the lower part of Figure 4 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.

Replacing tags with StringIDs not only reduces the space consumption but also allows for higher performance of navigational queries. Operations such as node comparisons now operate on integers instead of strings. Further details on document navigation can be found in [2]. Whenever XML nodes or documents are returned as query results, the nodes are serialized back to their text form. In this process the mapping from StringIDs to actual tags is reversed.

If a document tree is too large to fit on one page it gets split into *regions* (Figure 5). 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. Multiple regions can be stored on one page, especially if documents are much smaller than the page size and each document is just a single region.

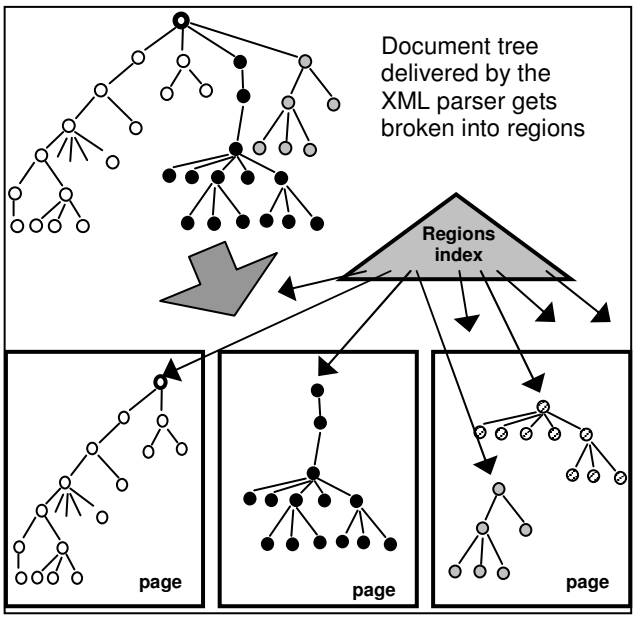

Figure 5: Interlinked document regions

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 5 the document tree is split into three regions colored in white, black and gray. The white and the black region occupy a full page each. The gray region is smaller and fits on a page already containing another small region.

Upon document traversal, a step to a parent, sibling, or child node may not lead to a node on the same page but to a different region. In this case, a regions index lookup finds the page with the corresponding region. In Figure 5, it is possible to navigate from the white region to the gray region of the document without touching the black region. For large documents this means that only those pages need to be fetched from disk which are actually required to evaluate a given query. For partial document access this saves costly I/O.

An alternative to the regions index could have been direct links between regions, similar to the implementation described in [7]. In our system we use the regions index for efficient sub-document level access and intelligent prefetching of regions.

The paged storage of XML documents leverages existing components in DB2, such as the buffer pool manager, the table space layer, and the logging facility.

5 XML Indexes

XML applications that manage millions of XML documents are not uncommon. Thus, indexing support for XML data is required to provide high query performance. 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. DB2 also supports XML-aware full-text indexing.

5.1 XML Value Indexes

Based on the sample table and document in Figure 2 and 3, the following statement defines an XML value index on all employee names in all documents in the XML column "deptdoc":

```
create index idx1 on dept(deptdoc) generate key
using xmlpattern '/dept/employee/name' as sql varchar(35)
```

The xmlpattern is a path which 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. Since 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.
- DATE and TIMESTAMP - for nodes with corresponding XML values.

Since the SQL type system is not exactly the same as the XML type system, special mechanisms are in place to compensate for key differences. One example is that the DB2 index manager has been enhanced to explicitly handle special values from the XML type system, i.e. +0, -0, +INF, -INF, and NaN.

If a node matches the xmlpattern but fails to cast to the specified index type, then no index entry is created for that node without raising an error [2]. A single document may contain zero, one, or multiple nodes that match the xmlpattern. Thus there may be zero, one, or multiple index entries for a single row in the table. This is a significant difference to indexes on relational columns.

As another example, the next statement defines a unique index on 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 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 that exists in the documents of an XML column is mapped to an integer PathID. This is very similar to the concept of StringIDs for tags described in section 4. Again, the mapping information is cached for performance and typically small since only unique paths are registered.

Each index entry 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.

Typically, indexes will be defined with xmlpatterns 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. All of the index examples above index atomic nodes in the document shown in Figure 3. However, it is also possible to define indexes on non-atomic nodes. In our example, the XML pattern ‘/dept/employee’ would be considered “non-atomic”, 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 is in compliance with the XML data model. If the intention is to index all employee names, offices, and phone numbers as separate values, then the xmlpattern ‘/dept/employee/*/text()’ or three separate create index statements should be used. Non-atomic indexes are rarely useful for data-centric XML, but can be useful for mixed content in text-oriented XML. For example, the following element “title” contains mixed content to indicate a formatting suggestion. In this case, a non-atomic index on “…./title” is useful because then the full title value gets indexed with no regard for the formatting suggestion:

```
<title>The benefits of<bold>XML</bold></title>
```

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. For example, index **idx3** above 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: (a) **idx2** is a numeric index but the predicate asks for a string comparison, (b) the predicate searches for @id attributes anywhere in the document but **idx2** only covers those under /dept/employee. Further details on index eligibility are given in [2] and [1].

5.2 XML Full Text Indexes

Full-text search is a common operation in document- and content-centric XML applications. DB2’s existing text search capabilities 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 indexed or partially indexed, e.g. 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 which fully indexes the documents in the XML column **deptdoc** in our 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 the 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 XML insert, update, and delete operations the text index is maintained asynchronously, i.e. not within the context of a DML transaction. However an "update index" command is available to force synchronization of the text index.

6 XQuery and SQL/XML

DB2 treats both SQL and XQuery as primary query languages. Both operate on their respective data models and can be used independently from each other. However, database applications can benefit immensely from the integration of the two languages that DB2 supports. Since many applications deal with existing relational data and XML simultaneously, queries need to combine and correlate these two types of data. This is described in the following subsections. Throughout this discussion we will refer to two tables in our examples:

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

6.1 Querying XML Data with XQuery

In DB2, XQueries can operate on XML documents in one or more XML columns. Each XML column is interpreted as a sequence of XML document nodes. This is accomplished by using either one of the two DB2 functions "db2-fn:xmlcolumn" and "db2-fn:sqlquery". As shown in the following example, db2-fn:xmlcolumn takes a string literal that identifies an XML column. db2-fn:xmlcolumn returns an XML sequence that consists of all documents in the specified column. Thus, the **for** clause in the example iterates over all documents in the XML column. If a column value is null, then there is nothing in the resulting XML sequence for that row.

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

The function db2-fn:xmlcolumn can be used multiple times in a single XQuery to reference different XML columns in the same or separate tables, or to reference one XML column several times. Each time the db2-fn:xmlcolumn function produces all documents of an XML column as input to the XQuery. This is a very common usage scenario. However, sometimes it can 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 sample query in Figure 6A is equivalent to the query with db2-fn:xmlcolumn above because the embedded SQL statement simply returns all XML documents from the XML column. However, in Figure 6B the input to the XQuery is very efficiently reduced to a single document, because the relational predicate exploits the primary key index on deptID.

Figure 6C shows an example where the set of input documents to XQuery is filtered by using a join and a predicate on another relational table. This highlights the power of integrating XQuery and SQL. Users can leverage all of their existing relational data to qualify XML documents for XQuery processing. The db2-fn:sqlquery function can be used not only to reduce the input to an XQuery but also to extend it. This is illustrated in Figure 6D where we use a UNION query to search all US departments and all UK departments for employee Jane Doe (using tables deptUS an deptUK in a slight variation of the running example).

A
```
for $e in db2-fn:sqlquery('select deptdoc
                           from dept')/dept/employee
where $e/office = 344
return $e/name
```

B
```
for $e in db2-fn:sqlquery('select deptdoc
                           from dept
                           where deptID = "PR27"')/dept/employee
where $e/office = 344
return $e/name
```

C
```
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
```

D
```
for $e in db2-fn:sqlquery('select deptdoc
                           from deptUS UNION
                           select deptdoc
                           from deptUK')/dept/employee
where $e/name = "Jane Doe"
return $e
```

Figure 6: XQueries with embedded SQL

The db2-fn:sqlquery function also enables applications to use XQuery to access and retrieve relational data. This is facilitated by SQL/XML constructor functions that transform relational data into XML format and produce a single column of type XML which can serve as an input to an XQuery. This integration is possible because SQL/XML has adopted the XQuery data model [6].

The following example shows an XQuery which constructs a result document that contains unit and department information. The department information is an XML document retrieved from the XML column deptdoc. The unit information comes from a pure relational table. The SQL/XML statement constructs an XML element "Unit" with three child elements whose values are taken from the relational columns of the unit table, i.e. the columns ID, name, and manager.

```
let $d := db2-fn:sqlquery('select deptdoc from dept
                           where deptID = "PR27"')
let $u := db2-fn:sqlquery('select XMLELEMENT(NAME "Unit",
                             XMLFOREST(ID, name, manager))
                          from unit where ID = "PR27"')
return <report>
          <units>{$u}</units>
          <department>{$d}</department>
       </report>
```

An XQuery and one or multiple embedded SQL queries are compiled into a single execution plan and comprise a single statement. SQL isolation levels as well as security privileges apply to the entire statement as a single unit, just like 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 can be used to declare and open cursors, fetch items from the XML sequence returned by the XQuery, and close cursors. Note that these items can be anything from XML documents to atomic values such as integers or strings.

6.2 Querying XML Data with SQL

It is often desirable to use and/or extend SQL statements to retrieve XML data. One reason is that database users are familiar with SQL which makes it a good starting point for managing XML. Also, existing relational applications are frequently augmented with XML data. Therefore, it is a natural approach to extend the exiting SQL applications and even existing SQL statements with XML capabilities.

Since XML is now a regular SQL data type [6], full documents can be retrieved from an XML column with a simple select statement:

```
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 [6], so here we only 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 either true or false for every row. The XMLEXISTS predicate evaluates an XPath or XQuery expression for each value of an XML column. If the result of the XQuery expression is an empty sequence then XMLEXISTS returns false, otherwise it returns true.

The following sample query returns full department documents as in the previous example but with XMLEXISTS for additional filtering. Only those rows are returned where the department document contains an employee in office 344. The **passing by** clause establishes the binding between the SQL and the XQuery context [6].

```
select deptID, deptdoc
from dept d
where deptID LIKE "PR%" and
      xmlexists('$deptdoc/dept/employee[office = 344]'
                passing by ref 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 XPath or XQuery expressions and returns the actual result as an XML sequence to the SQL application. The query in the next example selects the deptID for all PR departments, and the XMLQUERY function extracts the employee names for all PR employees in office 344.

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

In this statement, XMLQUERY returns an empty sequence for each department document where no employee in office 344 is found. To avoid those rows in the query result, a corresponding XMLEXISTS predicate needs to be placed in the where clause.

An XMLQUERY function may also appear in the where clause so that an extracted value can be used in a join condition with a relational column of another table. The extracted XML value needs to be cast to the SQL type of the relational join column. In the example below we join the tables **unit** and **dept** to obtain the name of the unit whose manager happens to be employee number 901. The XMLQUERY function searches the department documents in the **dept** table and extracts the name of the employee with id 901. The xmlserialize function casts the extracted XML value to char(20) so that it can be compared to the manager column from the **unit** table.

```
select  u.name, u.manager, d.deptID
from dept d, unit u
where xmlserialize(content
    xmlquery('$deptdoc/dept/employee[@id=901]/name/text()'
                passing by ref d.deptdoc as "deptdoc"
                returning sequence) as char(20)
                ) = u.manager
```

6.3 Query Execution Plans and Operators

DB2 has separate parsers for SQL and XQuery statements, but uses a single integrated query compiler for both languages. Query execution plans can contain novel XML operators for XML navigation (XSCAN), XML index access (XISCAN), and novel joins over XML indexes (XANDOR). For details see [2] and [8]. DB2 also collects XML-specific statistics for XML data which the query optimizer uses to create efficient query execution plans. Statistics for XML data are more complex than for relational data since not only value distributions but also structural statistics need to be considered. Histograms of element occurrences, attribute occurrences, and their corresponding value occurrences aid in query optimization.

7 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 and external entities. The type annotation produced by the validation is persisted together with the document for use during query execution. DB2 conforms to the XML Query standard, the XML Schema standard, and the XML standard for the above operations.

7.1 XML Schema Registration and Validation

Before XML Schemas and DTDs 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.

Registration of XML schemas is done via DB2 commands, stored procedures, or language-specific APIs. The following is an example of registering a simple schema. Its schema URI 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, which is to be used for validation, can either 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 next example shows two insert statements which validate the input document against our previously registered "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 clarify that XML Schema validation in DB2 is a per-document concept and not a per-column concept. 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 homogeneousness.

The next example shows an insert where no schema is referenced explicitly. In this case DB2 tries to deduce the schema from the input document and will try to find it in the repository.

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

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

7.2 XML Schema Evolution and Flexibility

The DB2 schema repository is based on two main design principles. The first principle is that the repository should not and will not require users to modify a schema before it is being registered, or 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. XML applications often deal with large numbers of documents so that 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. For example, changing or introducing new services, products, or business processes can all result in new requirements for information management. All this might result in XML schema changes.

Schema evolution and how best to accomplish it has been a much-debated topic. So far, there is no standard for evolving schemas in sight. However, business pressure force schemas to evolve and XML users find ways to do it. 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 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 also does not prescribe a specific way of doing schema evolution.

DB2 has built-in support for one very simple yet very important type of schema evolution. If the new schema is backwards-compatible with the old schema, then the old schema can be replaced with the new schema in the schema repository. For this operation DB2 verifies that all possible elements and attributes that can exist in the old schema have the same named types in the new schema. This type of schema evolution limits the type of changes one can make to additions of optional elements and attributes, but is simple and useful.

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. We can also write queries against that table to process only documents that conform to the old schema, or only documents that conform 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:

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

This statement returns the schema identifier of the schema which was used for validation of the XML document for department PR27.

8 Annotated Schema Decomposition

Even though the DB2 native XML store can insert and query any XML document, there are cases where it still makes sense to shred XML documents into relational rows and columns. In certain usage scenarios XML is only used to transport data to the database but 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 can also be required because many existing tools for data mining and business intelligence only work on the relational format of the data. Also, the performance of queries over relational data can be superior to queries over XML 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 [13] and Oracle [11]. These annotations are added to the schema by the user and describe which XML elements and attributes map to which 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 piece of annotated schema, the value of the salary element under the payroll element will be inserted into the salary column in table T. The DB2 decomposition annotations are in their own namespace and are using 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 to user to control the decomposition process in great detail:

- The data can be normalized, its white space manipulated, the data manipulated in an expression, or truncated before insertion.
- The data can be inserted conditionally: e.g. only if 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
- Multiple elements or attributes can be inserted into the same table-column pair.

Since XML is a first class type in DB2, decomposing an XML document can include inserting part or the entire 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 multiple XML columns.

9 XML API and Application Support

DB2 introduced a new SQL column type in the database, the XML data type. Applications can bind various language specific data types for input and output of XML columns or parameters. These existing language specific data types only allow the user to work with XML 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 language specific XML types enable the database to be more efficient and enable the database to supply a richer API for the applications. By making XML explicit in the application, the database will avoid unnecessary and/or unwanted code page conversions. XML documents have an internal encoding declaration which makes all but the XML parser's transcoding unnecessary. Avoiding unnecessary code page conversions is often an important performance benefit. Additionally, transcoding an XML document without carefully adjusting the XML encoding declaration might make the XML document invalid.

All the major database interfaces are supporting the XML type natively, i.e. treating XML data as XML, not as a character type. Below, we will touch on JDBC, ODBC, .NET, and embedded SQL.

9.1 JDBC

JDBC is enhanced to make XML data compatible with Strings, Byte arrays, and streams, i.e. XML columns and XML parameters can be bound to Strings, Byte arrays, and streams. IBM is working on standardizing a JDBC XML type. In the mean time a proprietary XML type com.ibm.db2.DB2Xml is available in such a way that application will be able to migrate seamlessly to the future standard JDBC type.

This DB2Xml interface has a number of methods that makes working with XML data easy. In the example below, a 'column' is retrieved as a DB2Xml object. Then the getDB2String method returns the serialized representation of the XML value (without XML declaration) as a String object. The getDB2XMLBinaryStream("UTF-16") then returns a binary stream with the XML value encoded in UTF-16, 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");
```

9.2 ODBC

ODBC is enhanced to support XML via a new XML type: SQL_C_XML. However, since there is no native XML type in C, that type can only be used in the ODBC API calls to mark XML values as XML typed. The advantage is that the DB2 client and server know that this is XML data and avoid unnecessary or unwanted code page conversions. Here is an example of inserting XML data into an XML typed column:

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

9.3 ADO.NET

The goal of the DB2 .NET support is to integrate as deeply 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 deptdoc from dept";
cmd.CommandType = CommandType.Text;
DB2DataReader dr = cmd.Execute();
dr.Read( );
// retrieve the column as an XML reader
XmlReader  xml1 = dr.GetXmlReader( 0 );
```

9.4 Embedded SQL

The SQL standard defined new host variable declarations for XML types. DB2 is using this in its implementation.

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

10 XML Utilities and Tools

The standard DB2 utilities are upgraded to work with the new XML type. For example, XML column type data is supported by DB2's backup & restore as well as high availability data replication for failover and fault tolerance.

IMPORT/EXPORT is the 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 which are concatenated in a single large input file. Likewise, the EXPORT utility can write XML documents to separate files or concatenate them into a single file.

IMPORT and EXPORT give the user fine-grained control of the XML parsing and validation options. The 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 sche-

mas 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 which 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 insert time. Thus, the relationship between documents and schemas can be exported along with the actual data and can be used for validation upon re-import into a database.

LOAD is the fast way of inserting data. LOAD is modified to process XML data very efficiently by parallelizing the XML parsing and by bypassing the regular insert flow, directly and formatting writing pages. Parsing multiple input documents concurrently has been shown to significantly boost XML bulk load times [10]. Again, XML Schema validation is optional during load.

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 unintuitive, even for simple queries. To resolve this issue, DB2 provides a GUI-based XQuery Builder. The XQuery Builder exposes the XQuery language functionality as sets of grids. Using a simple drag & drop and drill down paradigm the user can build fairly complex queries. The tool interprets users' GUI actions and generates the corresponding queries, greatly assisting the user in the construction and manipulation of XQuery syntax.

11 Summary

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 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 is rounded off by XML support in utilities such as XML import/export and a visual XQuery design tool.

Acknowledgement

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 T.J. Watson Research Center for their contributions to integrating native XML support into DB2.

References

[1] Balmin, A. et al.: *A Framework for Using Materialized XPath Views in XML Query Processing*, VLDB 2004, pages 60-71.

[2] Beyer, K. et al.: *System RX: One Part Relational, One Part XML*, SIGMOD Conference, 2005.

[3] Boag et al.: XQuery 1.0: An XML Query Language, February 2005, http://www.w3.org/TR/xquery

[4] Bourret, R.: *XML Database Products*. http://www.rpbourret.com/xml/XMLDatabaseProds.htm

[5] DeHaan et al.: *A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding*. Sigmod 2003.

[6] Eisenberg, Melton: *Advancements in SQL/XML*, ACM SIGMOD Record 33(3), pages 79-86, 2004

[7] Fiebig, T. et.al.: *Anatomy of a Native XML Base Management System*, VLDB Journal 11(4), December 2002

[8] Josifovski, V. et al.: *Querying XML Streams*, VLDB Journal, Vol. 14, No 2, April 2005.

[9] Katz, H., (Editor): *XQuery from the Experts*, Addison-Wesley, 2004.

[10] Nicola, M. et al.: *XML Parsing, A Threat to Database Performance*, CIKM 2003.

[11] Oracle XML DB 10g www.oracle.com/technology/tech/xml/xmldb

[12] Pat et al.: *Indexing XML Data Stored in a Relational Database*, VLDB 2004.

[13] SQLXML in MS SQL Server 2000 http://msdn.microsoft.com/sqlxml

[14] XML Efforts in Life Sciences and Bioinformatics, http://www.xml.com/pub/rg/Bioinformatics