# On the Path to Efficient XML Queries

Andrey Balmin
IBM Almaden
Research Center
abalmin@us.ibm.com

Kevin S. Beyer
IBM Almaden
Research Center
kbeyer@almaden.ibm.com

Fatma Özcan
IBM Almaden
Research Center
fozcan@almaden.ibm.com

Matthias Nicola
IBM Silicon Valley Lab
mnicola@us.ibm.com

## ABSTRACT

XQuery and SQL/XML are powerful new languages for querying XML data. However, they contain a number of stumbling blocks that users need to be aware of to get the expected results and performance. For example, certain language features make it hard if not impossible to exploit XML indexes.

The major database vendors provide XQuery and SQL/XML support in their current or upcoming product releases. In this paper, we identify common pitfalls gleaned from the experiences of early adopters of this functionality. We illustrate these pitfalls through concrete examples, explain the unexpected query behavior, and show alternative formulations of the queries that behave and perform as anticipated. As results we provide guidelines for XQuery and SQL/XML users, feedback on the language standards, and food for thought for emerging languages and APIs.

## 1. INTRODUCTION

XQuery [16] and SQL/XML [8, 9] are supported in major database systems such as DB2 [5, 12], Microsoft SQL Server [14], and Oracle [11]. All of these systems support storing, querying, and indexing XML documents with or without XML schemas, but differ in their implementation and scope of functionality. However, compared to decades of experience with plain SQL, most database application developers are very new to XQuery and SQL/XML. There is a great need for "best practices" and "do's and dont's" to master the complexities of these new languages. From the experiences of early XQuery and SQL/XML adopters we identified a number of common mistakes that novice users of these languages tend to make. Some of these mistakes

lead to unexpected query results and thus are easy to spot, but other issues can impact query performance. Many of these problems are not implementation-specific in any particular system but inherent in the XQuery and SQL/XML languages.

There are particular language features that impede the exploitation of XML indexes for predicate evaluation. The goal of a database system is to detect indexable predicates regardless of how a query is expressed. However, seemingly identical queries are not always equivalent due to sometimes subtle differences in semantics, which novice users are not aware of. This can lead to surprises in terms of performance if indexes are not used for semantic reasons - and unexpectedly so for the user.

Also, relational techniques that are applicable to SQL are not directly applicable to XQuery and SQL/XML. For example, XQuery has properties which SQL does not have, such as node identities, which make query rewrites more complex.

In this paper we focus on fundamental issues that are avoidable only by understanding the semantics of the query languages. We discuss these language issues in the context of two common application characteristics: (1) Applications have to manage large numbers of small to medium sized XML documents. For example, we observe that applications which process millions of documents under 1MB per document are much more common than those which process one or few large documents. Financial applications, web services (SOAP messages), and web feed formats such as RSS [7] and Atom [1] are just a few examples. Even document- and content-oriented XML applications with larger documents typically manage many documents, not just one. Therefore XML indexes are needed most of all to filter documents (context). This is in contrast to indexing schemes which only focus on XPath processing within a single document.

(2) Since flexibility is usually one of the main reasons for using XML in the first place, we focus on supporting XML datasets without schemas, with multiple or evolving schemas, or with schemas that include extensibility points. Such schema flexibility has recently been identified as a killer application for XML databases [15] and matches our observations. A prime example for extensible schemas is RSS, which allows elements of any namespace anywhere in the document. The documents can use the xsi:type mechanism to dynamically define the data type of the nodes. The power

of XQuery and SQL/XML can tackle this dynamic environment. However, with the power of these languages come some complexities and pitfalls that users must be aware of in any XML database.

These issues in XQuery and SQL/XML are discussed in terms of concrete query examples in DB2 Viper, but these are language-specific issues and not particular to DB2. The upcoming DB2 Viper release provides native XML storage, indexing, navigation and query processing through both SQL/XML and XQuery. The key design principles of the XML support in DB2 Viper include truly hierarchical XML storage, path-specific XML indexing, and a high degree of schema flexibility [5, 12]. DB2 allows applications to assign XML schemas to documents on a per-document basis instead of a per-column basis. This maximizes schema flexibility and enables DB2 to support schema evolution, a key requirement for XML applications [15].

Our main goals are three-fold. First, we strive to educate the users of XQuery and SQL/XML and enable them to successfully deploy systems using these languages. To that end, we provide tips for avoiding common mistakes and performance problems. Second, we provide important feedback to the language standards bodies by identifying language features that impede efficient query processing. In Section 4 we summarize the issues that we think should be revisited for the next versions of the standards. Finally, this work should be of interest to the new fledging communities that rely on XML to provide messaging, content integration and other functionality. New data formats and API's such as RSS [7], Atom [1], and JCR [10], are likely to give rise to new query languages in the future. Our experience should be relevant to the design of such new languages.

In Section 2 we describe the XML index architecture in DB2 and the basic rules for index eligibility. Section 3 then discusses 10 areas where semantic language characteristics prevent XML index usage or lead to unexpected query results. These topics include SQL/XML query functions, joins, namespaces, document and text nodes, "between" predicates, and others. Usage tips are given throughout these sections.

## 2. DB2/XML OVERVIEW

DB2 Viper stores XML data in native XML-type columns in relational tables. The physical storage format for the XML type preserves all the information in the XQuery data model. An important feature of DB2 is that it does not require an XML schema to be associated with an XML column. An XML column can store documents validated according to many different and evolving schemas, as well as non-validated documents, all in the same column. Hence, the association between schemas and XML documents is per document, for highest flexibility.

In DB2, documents conforming to new schemas are easily added to the system at any time, and the new schema may be similar to previous schemas. For example, an element type may require a new child element. This ability is crucial to storing the data of evolving systems, e.g. many Web services [17, 6].

DB2 applications can access XML data using either SQL/XML or XQuery. The two languages are composable: SQL can be invoked from XQuery, and XQuery can be invoked from SQL. The key to this dual behavior is SQL's new XML data type [9], which is based on the XQuery data model (XDM) [19]. Supporting arbitrary XDM results from these functions enables the user to transition back and forth between SQL and XQuery.

SQL programmers manipulate XML data using XQuery subqueries from SQL's `xmlQuery`, `xmlExists`, and `xmlTable` functions. The XQuery arguments to `xmlTable`, `xmlQuery` and `xmlExists` can be arbitrarily complex, including FLWORs and joins with other tables. SQL also provides `xmlCast` to convert XML data into SQL data, and the XML publishing functions (e.g., `xmlElement`) to construct new XML data from relational inputs.

DB2 also supports a stand-alone XQuery interface and provides access to XML columns stored in tables through the `db2-fn:xmlcolumn` function which simply imports an entire XML column as a sequence of items. Further details on DB2's XML support can be found in [5, 12, 13, 4, 2].

### 2.1 XML Indexes in DB2

In relational systems, indexing is the most important feature for query performance, and this remains true for XML data. 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 the system can quickly locate the rows that satisfy range and equality predicates on the column. But what does it mean to create an index on an XML column?

As with relational systems, applications typically cannot afford to index every item. XML compounds the issue because of the sheer quantity of items that can be indexed. For example, not only can a range predicate be on any simple node in the document (the "leaf" elements and attributes), but also the processing instructions, comments, text nodes (which differ from their containing element), and interior nodes (as the concatenation of all text nodes below it). If DB2 only supported indexing every item in the XML document, then the index storage would be several-fold larger than the original document. Moreover, the number of I/Os required to transactionally maintain the indexes would be staggering. Therefore, we support the indexing of nodes that are returned from a simple XQuery, as shown in the (simplified) CREATE INDEX DDL:

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

An index entry is created for each node that matches the path expression and is convertible to the index data type. The path expression may contain descendant axes and wildcards, but it cannot contain any predicates. If a node matches the path expression, but it is not a valid in-

stance of the index data type, then the node is simply not added to the index.

This "tolerant" behavior is important for schema evolution. For example, if the data contains U.S. postal codes, then the schema and the queries may treat the data as a number. But when the company begins shipping to Canada, the schema must be changed to use a string for the postal code. Until all the applications are changed to query the postal code as a string, the system may require both a numeric and a string index on the same data. If the old numeric index rejected the non-numeric Canadian postal codes, then we could not accept the new documents until the index was dropped.

Another reason that the indexes are tolerant to type mismatches is the user may decide to create very broad indexes. For example, the administrator may decide to index all of the numeric attributes with an index on `//@*` as a double. Such a broad index is useful for unpredictable query workloads because it would cover a numeric predicate on any attribute in the entire XML column. If an XML index specified a constraint on the data type, then these broad indexes would not be possible because they would inevitably stumble upon a non-numeric attribute and prevent the document insertion.

The index contains an entry for each node that matched the path expression and the data type. Ultimately, we are creating an index on the cast of the node to the indexed type, taking into consideration the node's type annotation derived during validation. This implies that some string-valued nodes appear in a numeric index, and that all nodes appear in a string index.

The result of an XML index scan is set of nodes that matched the query predicate. Under the covers, XML indexes are implemented using B+Trees. The index contains sufficient information to answer a range or an equality predicate on the converted value, additional restrictions on the path, as well as to perform node-level conjunctions and disjunctions of multiple predicates.

In this paper, we are solely concerned with using indexes to locate the subset of context nodes from an entire collection that require further processing. For the present setting, we can think of filtering documents from a collection (i.e., rows from a table), but the discussion applies to node-level filtering as well. We are not considering the use of indexes to locate related nodes from an arbitrary context node; in other words, once we locate a context node, we do not discuss the use of indexes to navigate within that particular document. We limit our focus to context filtering because it is the main way to improve performance on the workloads we observed, which manage large collections with millions of modestly-sized documents. Therefore, we say that an index is *eligible* only if it can eliminate documents from a collection based upon a predicate in the query. The next section describes this process in detail.

## 2.2 Index Eligibility

The question of index eligibility (whether an index can be used to answer a query predicate) is typically trivial in relational query processing. Any index defined on a single relational column can be used to answer any equality or range predicate on this column. This problem, however is more difficult for XML columns. Whereas any index on a

relational column stores all values in this column, an XML index stores only values of nodes that match the XPath pattern in the index definition. An XML index can be used to answer an XML query predicate, only if this index contains all XML nodes that satisfy the query predicate. However, as we show in Section 3.2, this condition is necessary, but not sufficient for the index to be usable.

DEFINITION 1 (INDEX ELIGIBILITY). *We say that an index $I$ is eligible to answer predicate $P$ of query $Q$, if for any collection of XML documents $D$, the following holds: $Q(D) = Q(I(P, D))$. Where $I(P, D)$ is the set of XML documents produced, by probing index $I$ with predicate $P$.*

In other words, applying the query to the documents pre-filtered by the index should produce the same result as applying the query to the full database.

This definition focuses on the indexes that pre-filter documents, in order to simplify the presentation. However, all the problems that we describe in Section 3 would still apply to indexes that return individual XML nodes satisfying the indexable predicate.

It follows from Definition 1 that an index cannot be used to answer a predicate in the query expression if the index expression is more restrictive than the query expression.

Let us illustrate this condition with an example. All the examples in this paper are based on the following schema:

```
create table customer (cid integer, cdoc XML);
create table orders   (ordid integer, orddoc XML);
create table products (id    varchar(13),
                       name  varchar(32));
```

Consider an index `li_price` defined as

```
CREATE INDEX li_price ON orders(orddoc)
USING XMLPATTERN '//lineitem/@price' AS double
```

This index contains values of `price` attributes of all `lineitem` elements that appear anywhere in the `orddoc` column of the `orders` table. Thus, this index can be used only if the query has a predicate on a lineitem price. For instance, Query 1 can use the `li_price` index to filter the documents that contain `price` attributes that match the pattern `//order/lineitem/@price`, and have values greater than 100.

QUERY 1.
```
for $i in db2-fn:xmlcolumn('ORDERS.ORDDOC')
           //order[lineitem/@price>100]
return $i
```

Notice that the index definition is less restrictive than the XPath navigation embedded in the query. The index contains all lineitem prices, and the query asks only for those that occur in the `order` elements. Hence, the XML index contains all the required information to answer this query efficiently.

However, an index cannot be used to answer a query that is less restrictive than the index definition. For example, Query 2 *cannot* use the same index.

QUERY 2.
```
for $i in db2-fn:xmlcolumn('ORDERS.ORDDOC')
           //order[lineitem/@*>100]
return $i
```

This query needs to return orders where any attribute of the lineitem element satisfies the predicate. We cannot check this condition using an index that contains only `price` attributes. For example, if the following document appears in the `orders.orddoc` column, its `order` element should be returned by Query 2.

```
<doc>
  <order @id="1001">
    <date>January 1, 2001</date>
    <lineitem @id="LI100101" @quantity="200">
  <order>
</doc>
```

No node of this document will be indexed in `li_price`, since the document does not include any `price` attributes. Thus, any index access plan that uses `li_price` will miss this `order` element.

Furthermore, to improve query execution time of Query 1, the index will apply the entire `//order/lineitem/@price > 100` predicate. Consider another order document, which does contain a price attribute.

```
<doc>
  <order @id="1002">
    <date>January 1, 2002</date>
    <lineitem @id="LI100201" @quantity="200"
        @price="99.50" />
  </order>
</doc>
```

The `li_price` index stores the reference to the price attribute, along with its value 99.50. This `order` element does not satisfy the conditions of Query 1, and an index scan of `li_price` will filter out this document.

XML indexes can also be used to support *structural predicates*. Notice that our definition of the value predicates includes the structural part, i.e., XPath navigation. For example, an index on `//lineitem/@price`, can identify `@price` attributes that satisfy the pattern `//order/lineitem/@price`, if the index contains all values, regardless of their data type. A varchar index, by definition, includes all matching values cast into the varchar data type[1]. Thus, a DB2 varchar index can be used to answer structural predicates by scanning a full range of values $(-\infty; +\infty)$, for a given set of paths. However, we believe that the main benefit of indexes will come from supporting the value predicates, which are typically more selective.

More information can be found in [3] which describes the prototype of the XML index eligibility algorithms used in DB2 Viper.

## 3. COMMON PITFALLS

In this section we go over a list of problems that are frequently encountered by early adopters of our system. Some of these issues relate to the some features of the XML query languages that users should be aware of in order to get the expected results for their queries. This class of issues includes use of SQL/XML query functions (Sections 3.2 and 3.3), XQuery LET clauses (Section 3.4), handling of document nodes (Section 3.5), and namespaces (Section 3.7). Another class of issues have to do with maximizing the use of indexes while taking full advantage of the flexibility provided by XML. This includes determining predicate data types (Section 3.1), use of construction (Section 3.6), `/text()` functions (Section 3.8), attributes (Section 3.9), and detecting "between" predicates (Section 3.10). These issues are not particular to DB2 implementation, they will be present in any system that supports the schema-less, incomplete schemas, and schema evolution scenarios.

### 3.1 Matching Index and Query Predicate Data Types

In order to establish index eligibility, the system must prove that the data type of the predicate comparison is compatible with that of the index definition.

An XML index can be defined with one of the four data types: varchar, double, timestamp, and date. By definition the value of each XML node that matches the XML pattern of the index, is type-cast into the data type of the index. If the type-cast operation was successful, the result is inserted into the index B-Tree structure. Thus, an index defined as double will contain only numeric values, or values castable to numbers, while the varchar index contains string representation of all nodes, since any XML node value can be converted into a string using the *string()* function.

For documents that have not been validated against schemas, all element nodes will have the type annotation *untyped*, and all attribute nodes will have the type annotation *untypedAtomic*, and they will be cast to xs:string by the comparison operators. However, in schema evolution scenario, different documents in an XML-typed column might be validated against different (and possibly conflicting) versions of the schema. Since, the type information is on a document-level, as opposed to column-level, the system cannot use any schema information to determine the comparison type during compilation [5, 4]. [2] Instead DB2 relies on information embedded in the query in form of typed constants, casts, and other functions with guaranteed result data types. For example, Query 3 would match a varchar index on `price`, but will not match the `li_price` index. This is because the literal value "100" is in double quotes and therefore is a string, not a number.

QUERY 3.

```
for $i in db2-fn:xmlcolumn('ORDERS.ORDDOC')
            //order[lineitem/@price > "100" ]
return $i
```

Notice that the string predicate in this query will be satisfied by string values such as "20 USD", which will not exist in the index `li_price` defined as double.

Another way to communicate the comparison data type information to the query compiler is to include type-casts in the predicate. This is especially useful for join predicates that normally don't contain any type information, such as constants.

---

[1]See Section 2.1 for more detail of the XML value index data types.

[2]Note that during run-time when the system accesses each document, it will use the type annotations of individual nodes to determine the correct type of the comparison.

QUERY 4.
```
for $i in db2-fn:xmlcolumn("ORDERS.ORDDOC")/order
for $j in db2-fn:xmlcolumn("CUSTOMER.CDOC")/customer
where $i/custid/xs:double(.) = $j/id/xs:double(.)
return $i
```

For example, Query 4 can use either of the following two double indexes, because the *xs:double()* casts on both sides of the equality guarantee that a numeric equality operator will be used, which is valid only for values that can be cast to double. All such values are guaranteed to be present in a double index.

```
CREATE INDEX o_custid ON orders(orddoc)
USING XMLPATTERN '//custid' AS double

CREATE INDEX c_custid ON customer(cdoc)
USING XMLPATTERN '/customer/id' AS double
```

Without the casts, the join predicate can turn out to be, for example, a string equality, for which a double index cannot be used, since it won't contain some of the values. Even a varchar index, which, by definition, contains all the values converted into strings, still cannot be used without the casts. Both sides of the join may turn out to be numeric, but a varchar index cannot enforce some of the rules of numeric comparison, such as $10E3 = 1000$.

TIP 1. *Use type-cast expression in XQuery join predicates. Notice that the* $i/xs:double(.) *notation in Query 4 is more general than* xs:double($i), *since it does not require* $i *to be a singleton.*

## 3.2 SQL/XML Query Functions

To query and manipulate XML data in SQL, SQL/XML provides functions *XMLQuery*, *XMLExists* and *XMLTable* for invoking XQuery from SQL. *XMLQuery* is a scalar function which executes an XQuery expression, and returns an XML result as instance of the XQuery data model (XDM). *XMLExists* is a predicate which executes an XQuery expression and returns *true* if the result is a non-empty sequence, and returns *false* otherwise. Finally, *XMLTable* is a table function which executes multiple XQuery expressions to compute a relational table. Each item in the result of the first XQuery expression in an *XMLTable* function is used as the context item in the rest of the XQuery expressions to populate the columns of the result table.

All of these functions pass named arguments to XQuery and may contain XPath predicates which *might be* index eligible. However, depending on which function is used and how it is used, XML indexes may or may not be eligible. When an *XMLQuery* function is used in the select-list and contains an XQuery expression on data that is passed in from the from-clause, which contains predicates on nodes that are indexed, the XML index is not eligible. Because, the select-list in an SQL statement does not eliminate any rows produced by the from-clause, and even if the result of the XQuery expression is an empty sequence, it needs to be returned to the user. Consider the following query:

QUERY 5.
```
SELECT XMLQuery('$order//lineitem[@price > 100]'
                passing orddoc as "order")
FROM orders
```

The output of this query might be as follows:

| row 1: | &lt;lineitem id = "2" &gt; ... &lt;/lineitem&gt; |
| | &lt;lineitem id = "7" &gt; ... &lt;/lineitem &gt; |
| row 2: | () |
| row 3: | () |
| row 4: | &lt;lineitem id = "9" &gt; ... &lt;/lineitem&gt; |
| row ...: | ... |
| row n: | &lt;lineitem id = "m"&gt; ... &lt;/lineitem &gt; |

This query returns as many rows as there are rows in the orders table. For orders which have a lineitem with a price greater than 100 it returns those lineitems. For those orders which do not have such a lineitem, it has to return an empty sequence. Note that in this query all lineitems of one order which satisfy the condition will be returned as an XML sequence in one row. As the XML index would have only returned lineitems which have a price greater than 100, and incorrectly eliminated orders which do not have such a lineitem, an XML index on the orddoc column of the orders table cannot be used to answer this query. Note that in this example, XML values (documents) are passed one at a time to XQuery. If the input values were supplied as a single sequence via the *db2-fn:xmlcolumn* function, like in the following example, then the li_price XML index would be eligible. The following query needs to return only lineitems which have a price greater than 100 and those orders which do not have such a lineitem would be eliminated in this path expression.

QUERY 6.
```
VALUES (XMLQuery('db2-fn:xmlcolumn("ORDERS.ORDDOC")
                //lineitem[@price > 100] '))
```

Query 6 may use an index on the orddoc column of the orders table. However, this query returns a single row, containing all the qualifying lineitems for *all* orders in the orders table and its output would be:

| row 1: | &lt;lineitem id = "2" &gt; ... &lt;/lineitem&gt; |
| | &lt;lineitem id = "7" &gt; ... &lt;/lineitem &gt; |
| | &lt;lineitem id = "9" &gt; ... &lt;/lineitem&gt; |
| | &lt;lineitem id = "15" &gt; ... &lt;/lineitem &gt; |
| | ... |
| | &lt;lineitem id = "m"&gt; ... &lt;/lineitem &gt; |

The most efficient formulation of this query is Query 7, because it can use an index on the orddoc column of the orders table and it does not require aggregating all qualifying lineitems into a single XML sequence.

QUERY 7.
```
db2-fn:xmlcolumn('ORDERS.ORDDOC')//
                    lineitem[@price > 100]
```

Query 7 returns each lineitem as a separate row and its output will look like as follows:

| row 1: | &lt;lineitem id = "2" &gt; ... &lt;/lineitem&gt; |
| row 2: | &lt;lineitem id = "7" &gt; ... &lt;/lineitem &gt; |
| row 3: | &lt;lineitem id = "9" &gt; ... &lt;/lineitem&gt; |
| row 4: | &lt;lineitem id = "15" &gt; ... &lt;/lineitem&gt; |
| row ...: | ... |
| row k: | &lt;lineitem id = "m"&gt; ... &lt;/lineitem &gt; |

Tɪᴘ 2. *If only XML fragments are to be retrieved, then use the stand-alone XQuery interface (like in Query 7) to extract XML values which satisfy a given condition.*

When an *XMLExists* predicate, used in the where-clause of an SQL statement, contains an XQuery expression with a predicate on indexed nodes, then an XML index can be considered to answer such queries. When the result of the XQuery expression embedded in the *XMLExists* predicate is an empty sequence, *XMLExists* returns *false* and filters out the input row. Consider the following query which returns the whole orddoc documents, which contain a lineitem with a price greater than 100, and their ids. The `li_price` index can be used to answer this query.

Qᴜᴇʀʏ 8.

```
SELECT ordid, orddoc
FROM orders
WHERE XMLExists('$order//lineitem[@price > 100]'
                passing orddoc as "order")
```

One needs to be careful when using the *XMLExists* function to evaluate XQuery predicates. When the XQuery expression embedded in an *XMLExists* function is a boolean expression, returning *true* or *false*, *XMLExists* will always return *true*, because the result is always a *non-empty* sequence, i.e. a sequence of one item whose value is either *true* or *false*. For example, the following Query 9 will not eliminate any order documents and will return all rows in the `orders` table.

Qᴜᴇʀʏ 9.

```
SELECT ordid, orddoc
FROM orders
WHERE XMLExists('$order//lineitem/@price > 100'
                passing orddoc as "order")
```

To retrieve only the order documents which contain a lineitem with a price greater than 100, the query needs to be formulated as in Query 8, enclosing the XQuery predicate as a filter. In general, the *XMLExists* function is useful when the whole XML document is retrieved, like in Query 8.

Tɪᴘ 3. *Use XMLExists function if full documents, as well as additional relational columns, are to be retrieved based on a condition on the XML column and make sure that the XQuery expression embedded in XMLExists returns nodes or atomic values, not a boolean value. For this purpose, embed the predicate either in an XPath or a FLWOR expression.*

The *XMLExists* function can also be used to eliminate the empty sequences produced by *XMLQuery* when XML fragments are to be retrieved. The following query would only return lineitems which have a price greater than 100.

Qᴜᴇʀʏ 10.

```
SELECT ordid,
       XMLQuery('$order//lineitem[@price > 100]'
                passing orddoc as "order")
FROM orders
WHERE XMLExists('$order//lineitem[@price > 100]'
                passing orddoc as "order")
```

Note that in Query 10, only the path expression in *XMLExists* is eligible for an XML index. To extract values from XML documents based on a set of conditions, the *XMLTable* function can be more suitable. Because, *XMLTable* would avoid repeating the same XQuery expression, once in the select list for extraction and once in the where-clause for restricting the output. The following query is similar to Query 10.

Qᴜᴇʀʏ 11.

```
SELECT o.ordid, t.lineitem
FROM orders o,
     XMLTable('$order//lineitem[@price > 100]'
              passing o.orddoc as "order"
              COLUMNS
                "lineitem" XML BY REF PATH '.')
          as t(lineitem)
```

Note that while Query 10 returns as many rows as the number of *orders* which satisfy the condition, Query 11[3] returns as many rows as the number of *lineitems* that satisfy the condition. In Query 11, there is an implied join between the `orders` table and the *XMLTable* function, because each order document from the `orders` table is passed into the *XMLTable* function as an argument. A relational table is computed for each row of the `orders` table. An order document may contain more than one lineitem which has price greater than 100 and *XMLTable* will produce a table with as many rows as there are such lineitems in a given order.

There is an important difference between the first XQuery expression in the *XMLTable* function and the ones used to specify the column values in the same *XMLTable* function. The first one, called the "row-producer", determines the number of output rows in the *XMLTable* result, and if it produces an empty sequence, no output table is computed. As the first XQuery expression eliminates rows of the `orders` table, an index on the orddoc column of the `orders` table is eligible to evaluate the first XQuery expression. However, the XQuery expressions which compute the column values always produce a result and do not affect the output cardinality. When one such XQuery expression produces an empty sequence, the result value of the corresponding column is the NULL value. Hence, a predicate in a column XQuery expression is not index eligible. Consider the following query:

Qᴜᴇʀʏ 12.

```
SELECT o.ordid, t.lineitem, t.price
FROM orders o,
     XMLTable('$order//lineitem'
              passing o.orddoc as "order"
       COLUMNS
         "lineitem" XML BY REF PATH '.',
         "price" DECIMAL(6,3) PATH '@price[. > 100]')
     as t(lineitem, price)
```

---

[3]In Query 11, the *BY REF* keywords indicate that the XQuery expression '.' which computes the output column lineitem returns node references. I.e., the result of *XMLTable* contains node references to lineitem elements in the original documents, preserving node identities and parent linkages. The details of *BY REF* and its counterpart *BY VALUE* can be found in [9].

Query 12 returns the lineitems and their prices, as well as the `ordids` of the order documents which contains these lineitems. If the price of a lineitem is not greater than 100, there is still going to be a row for that lineitem in the output, but the value of the price column will be NULL. Hence, the `li_price` index *is not* eligible to filter the XML documents in the `orders` table.

TIP 4. *Use XMLTable if both relational values and XML fragments are to be retrieved based on conditions on XML columns and make sure to express the predicates in the "row-producer" XQuery expression of XMLTable.*

## 3.3 Joining XML Values in SQL/XML

The SQL/XML query functions, *XMLQuery*, *XMLExists* and *XMLTable* can be used to express joins between relational columns and values in XML documents. The join condition can be expressed either in SQL or in XQuery, resulting in slightly different queries as XQuery and SQL comparisons have different semantics. Consider the following query, which returns the name of products, and the lineitems which order them.

QUERY 13.
```
SELECT p.name, XMLQuery('$order//lineitem'
                            passing orddoc as "order")
FROM products p, orders o
WHERE
  XMLExists('$order//lineitem/product[id eq $pid]'
             passing o. orddoc as "order",
             p.id as "pid");
```

The join condition in Query 13 is expressed in XQuery. Note that only an XML index can be considered to answer this query, because the join condition is expressed as an XQuery condition, using XML schema data types. For example, an XML index on *//lineitem/product/id* is eligible. No relational index on the id column of the `product` table is eligible, because relational indexes implement SQL comparisons using SQL data types. The semantics of the comparison operators are different in two languages. For instance, while trailing blank characters are ignored in SQL, they are significant in XQuery. Note that the *$pid* variable in XQuery inherits its subtype from the SQL side (xs:string in this example), providing the compiler enough information to decide the data type of the join condition.

The following query, which expresses the join on the SQL side, looks similar at surface, but has subtle and important differences:

QUERY 14.
```
SELECT p.name, XMLQuery('$order//lineitem'
                            passing orddoc as "order")
FROM products p, orders o
WHERE p.id =
   XMLCast(
       XMLQuery('$order//lineitem/product/id'
                   passing o. orddoc as "order")
       as VARCHAR(13))
```

In particular, this query is different from Query 13 in two important ways: First, this one uses SQL comparisons for the join condition and hence only a relational index on the id column of the `products` table is eligible. Second, the *XMLQuery* function extracts all the ids of products in a given order document, and the *XMLCast* function will insist that there is only one such id in an order document (otherwise *XMLCast* will raise a type error). Hence, if an order contains more than one lineitem with a product, then Query 14 will fail with a type error, while Query 13 will succeed. *XMLCast* in Query 14 will also fail if the value of an id element has length greater than 13, whereas Query 13 will succeed, because there is no length limit on xs:string type. Moreover, if we change the value comparison (*eq* operator) in Query 13 to a general comparison (= operator), then the XQuery comparison would succeed even if a given product has more than one id. These examples demonstrate that special attention needs to be paid to the hierarchical relationships and the relative cardinalities when joining XML values.

TIP 5. *When joining an XML value and a relational value, express the join condition on the SQL side if there is an index on the relational column and express it on the XQuery side if there is an XML index on the XML column.*

When two relational tables are to be joined on their XML values, there are two options: Either extract the values and express the join in SQL, or pass the XML values into an *XMLExists* function and express the join in XQuery. Query 15 and Query 16, which return the lineitems and the names of customers who ordered them, illustrate these two alternatives.

QUERY 15.
```
SELECT c.name, XMLQuery('$order//lineitem'
                            passing o.orddoc as "order")
FROM orders o, customer c,
WHERE XMLCast(XMLQuery('$order/order/custid'
                            passing o.orddoc as "order")
            as DOUBLE) =
     XMLCast(XMLQuery('$cust/customer/id'
                            passing c.cdoc as "cust")
            as DOUBLE)
```

As Query 15 expresses the join on the SQL side using the SQL comparison operator "=", no XML index is eligible. A relational index is also not eligible because the join is between two XML columns. This join can also be expressed by passing both XML values into an *XMLExists* function, as in Query 16. In this case, an XML index on *order/custid* on the orddoc column of the `orders` table can be used to compute the join. If the join is between two XML columns, then the join needs to be expressed in XQuery so that XML indexes on these columns can be employed. Since the data type of both XML elements are unknown to the XQuery compiler, explicit type casts are needed.

QUERY 16.
```
SELECT c.name, XMLQuery('$order//lineitem'
                            passing o.orddoc as "order")
FROM orders o, customer c,
WHERE XMLExists('$order/order[custid/xs:double(.) =
                $cust/customer/id/xs:double(.)]'
                passing o.orddoc as "order",
                        c.cdoc as "cust")
```

TIP 6. *Always express XML joins on the XQuery side.*

While SQL/XML [9] enables embedding of XQuery expressions in SQL, it does not unify their type systems and the comparison operators. XQuery operates on XML schema types and has special rules to deal with both typed and untyped data. SQL, on the other hand, operates on its own type system and requires both sides of the comparison to be strongly typed.

## 3.4 XQuery Let-Clauses

A let-clause binds its variable to the result of the associated expression, even when the result of the expression is an empty sequence [18], unlike a for-clause which does not produce any iteration for an empty sequence. A FLWOR expression represents a join (in general a cartesian product) between the for-bindings, and also an outer-join between the let-bindings and the for-bindings, where the let-bindings are the NULL-preserving side of the outer-join. As all values of the let-bindings need to be returned, we cannot use an XML index to compute the expression in a LET binding, unless we can prove certain properties. Consider the following query:

QUERY 17.
```
for $doc in db2-fn:xmlcolumn('ORDERS.ORDDOC')
for $item in $doc//lineitem[@price > 100]
return <result> {$item} </result>
```

In this query, we can use the `li_price` XML index as the result of the query will not contain the orders which do not have a lineitem with a price greater than 100. However, if we replace the second for-clause in Query 17 with a let-clause, then the index can no longer be used.

QUERY 18.
```
for $doc in db2-fn:xmlcolumn('ORDERS.ORDDOC')
let $item:= $doc//lineitem[@price > 100]
return <result> {$item} </result>
```

Note that Query 17 and Query 18 are semantically different and produce different results. Query 17 returns a result element for each qualifying lineitem while Query 18 returns a result element for each order document. If a given order document contains a lineitem with price greater than 100, it returns the lineitems of that order which have a price greater than 100 in a result element. However, if there is no such lineitem then an empty result element is returned. Since, no order documents are eliminated with the predicate on price, and a result element is returned for each order document, not just the ones that satisfy the predicate, the `li_price` XML index cannot be used to answer Query 18.

In addition to the explicit let bindings, each step of an XPath expression, and all XQuery expressions have implied let semantics [18]. In other words, each XQuery expression produces a sequence as output. For example, path expressions in the return-clause of a FLWOR expression have implied let-semantics. Query 19 produces a result element for each order element, irrespective of the condition on the price attribute. In particular, there is an outer-join between the order documents and the lineitems which have a price greater than 100. The lineitems sequence is the empty-preserving side such that empty result elements are returned for orders that do not have qualifying lineitems with a price greater than 100. As a result, there is no filtering and thus no XML indexes can be considered for predicates embedded in constructors in return-clauses.

QUERY 19.
```
for $ord in db2-fn:xmlcolumn('ORDERS.ORDDOC')/order
return <result>
          {$ord/lineitem[@price > 100]}
       </result>
```

Similarly, the path expressions in the where-clause have implied let semantics. For example, Query 20 and Query 21 are equivalent.

QUERY 20.
```
for $ord in db2-fn:xmlcolumn('ORDERS.ORDDOC')/order
where $ord/lineitem/@price > 100
return <result> {$ord/lineitem} </result>
```

QUERY 21.
```
for $ord in db2-fn:xmlcolumn('ORDERS.ORDDOC')/order
let $price := $ord/lineitem/@price
where $price > 100
return <result> {$ord/lineitem} </result>
```

Despite the let semantics in Query 20 and Query 21, we can consider using an XML index. Because, in the where-clause an empty sequence evaluates to *false* and hence eliminates binding tuples. It is important to note the difference between Query 18, where the predicate on price is used to qualify lineitems, and Query 21, where the predicate on price is used to eliminate orders. In general, we can consider an XML index for a let binding when there is a where clause predicate which eliminates the empty sequence. Also note that for a for-clause it does not matter whether the predicate is embedded in the path expression, which computes the for-binding, or is in the where-clause: XML indexes can be used in both cases.

Although the empty sequence results need to be preserved in general, there are two other XQuery operations (in addition to the where-clause) which discard the empty sequences. First, an iterator produces no result when its input sequence is an empty sequence. For-clauses of FLWOR expressions, the in-clauses of quantified expression, and bind-out are three places where there is an implied iteration. Second, sequence concatenation also discards empty sequences, as XQuery does not have nested sequences [19]. If an empty sequence is to be discarded later on, then we can consider using an XML index to evaluate the expression that generates that empty sequence in the first place. For example, consider the following query:

QUERY 22.
```
for $ord in db2-fn:xmlcolumn('ORDERS.ORDDOC')/order
return $ord/lineitem[@price > 100]
```

Even though the path expression in the return clause has outer-join semantics, we can still consider using the `li_price` XML index to answer Query 22. Because there is an implied iterator in bind out and empty sequence results will not be returned to the user. Note the difference between Query 19 and Query 22. The element constructor in the return clause of Query 19 creates an empty result element when there is no satisfying lineitem.

TIP 7. *Unless you want an empty element returned for non-qualifying nodes, do not express predicate conditions in XPath expressions within element constructors.*

## 3.5 Document versus Element Nodes

While writing path expressions, one also needs to be careful whether the context node is a document node or an element node. If the context node is a document node, then the leading slash will evaluate to the document root and the first step expression will start navigation from the root. However, if the context node is an element node, then the navigation starts from that particular element node. Consider the following query:

QUERY 23.

```
db2-fn:xmlcolumn('ORDERS.ORDDOC')/order/lineitem
```

In this query, the *db2-fn:xmlcolumn* function returns document nodes. The first step expression, which is *child::order*, will match the top-most order elements. However, in Query 24, the path expression *$ord/my_order* will return an empty sequence. Because the context node, i.e the *$ord* variable, is bound to my_order element nodes, and the first step expression, which is *child::my_order* will not find any my_order element children of the context node.

QUERY 24.

```
for $ord in
  for $o in db2-fn:xmlcolumn('ORDERS.ORDDOC')/order
  return <my_order> {$o/*} </my_order>
return $ord/my_order
```

The leading slash in XQuery, which is used to express absolute path expressions, is a shorthand for *fn:root(.) treat as document-node()*. Hence, if the context node is not a document node, then the leading slash may result in type errors. For example, the absolute path expression *$order[//customer/name]* in Query 25 will result in a type error, because the *$order* variable is bound to a *new_order* element node.

QUERY 25.

```
let $order :=
    <new_order>
      {db2-fn:xmlcolumn('ORDERS.ORDDOC')/
                      order[custid > 1001]}
    </new_order>
return $order[//customer/name]
```

TIP 8. *Be careful when writing path expressions and recall that there is an extra level of navigation when the context node is a document node. Moreover, do not use absolute path expressions, if the context node is in a tree rooted by an element node, such as constructed elements.*

## 3.6 Node Construction

XQuery and SQL/XML provide construction of new nodes to create new structures and reshape old structures. Element construction is the way to group information in XQuery, like tuples in a relational database, but enriched with nesting and repetition. Thus, construction is the basis for XML view definitions, which are a staple in relational databases to hide information for security or to hide the complexity of relating information. The nearest relational equivalent of construction is projection, which creates and reshapes tuples.

However, construction is nondeterministic because it generates distinct node identifiers on each evaluation (e.g., `<a>5</a> is <a>5</a>` is false). Node identity defines the deduplication in path expression and union operations. Therefore, query transformations that eliminate construction operations and push down predicates are more difficult than their relational counterparts.

Construction also affects the interpretation of items placed inside the node. For example, node construction replaces the type of atomic values with untypedAtomic, concatenates sequences of atomic values into a single space-separated untyped string, redefines the node identity of copied nodes and preserves or erases the type annotations on nodes based upon the "construction mode", and raises an error for duplicate attribute names. These rules are designed to ensure that nodes created by XQuery are valid XML data. When transforming the query to eliminate construction, these semantics must be preserved.

For example, consider Query 26. Someone defined the variable `$view` to reshape the order data, expose a limited subset of the data, and relabel some of the information. A user of the view selected a small subset of the `ordered` elements and project just the `price` attribute.

QUERY 26.

```
let $view :=
  for $i in db2-fn:xmlcolumn('ORDERS.ORDDOC')/
            order/lineitem
  return <ordered>{
            $i/@quantity,
            $i/product/@price,
            <pid>{ $i/product/id/data(.) }</pid>
         }</ordered>
for $j in $view
where $j/pid = '17'
return $j/@price
```

From relational systems, users have come to expect that such selections and projections will be pushed down to simplify the query and improve the performance by enabling indexes. Similarly, XQuery users expect that this query would be simplified into Query 27.

QUERY 27.

```
for $i in db2-fn:xmlcolumn('ORDERS.ORDDOC')/
          order/lineitem
where $i/product/id/data(.) = '17'
return $i/product/@price
```

However, many issues arise that can prevent this transformation:

1. If `product/id` has a numeric type, then Query 27 will produce an error, but Query 26 will succeed. The reason is the value of the new `pid` element is untypedAtomic, which is comparable to a string, but numbers are not. The system must add a cast to preserve the semantics: `where $i/product/id/xdt:untypedAtomic(data(.)) = '17'`, but will likely interfere with index eligibility.

2. If the type of `product/id` is a long integer, and '17' is replaced with a large long integer value, then Query 26 will convert both values to a double floating-point value,

but Query 27 will compare them as long integers. The result of the two comparisons differ when large values are used due to floating point rounding. Again, the extra cast to untypedAtomic will preserve the semantics, but will likely interfere with index eligibility. Of course, users may also be surprised by the collisions on large long values caused by the conversion, but query rewrites cannot change the semantics of the language.

3. If a `product` has multiple `id` children, say `<id>p1</id>` and `<id>p2</id>`, then the value of `pid` will be the concatenation of them, `<pid>p1 p2</pid>`. If we replace '17' with 'p1 p2', then Query 26 will produce a result, but Query 27 will not find a result. Conversely, if we use 'p2', then Query 27 will find a result, but Query 26 will not. To preserve the semantics, the transformed query must be `where xdt:untyped-Atomic(string-join($i/product/id/data(.),' ')) = 'p1 p2'`. Again, these extra conversions are an impediment to index eligibility.

4. If a `lineitem` has multiple products each with a `@price` then Query 26 will produce a duplicate attribute error, but Query 27 will succeed.

5. If Query 26 is itself a query fragment and its result is subject to arbitrary further processing, then the system must ensure that the node identity semantics are preserved. For example: `$view/@price except db2-fn:xmlcolumn('ORDERS.ORDDOC')/order/lineitem/ product/@price` will return all the `price` nodes. However, if the system incorrectly simplfies `$view/@price` to `db2-fn:xmlcolumn('ORDERS.ORDDOC')/order/lineitem/ product/price`, then the result is empty.

Similar issues arise at the border between SQL and XQuery. The data is transformed from SQL types to XQuery types or vice-versa, and different comparison rules apply. For example, SQL string comparison is insensitive to trailing space characters, but they are significant in XQuery.

There are many conditions that need to be checked before construction can be safely eliminated. Often a system might not be able to prove that the transformation can be applied due to lack of information on one of these conditions. Moreover, the system might know that the transform is not safe, but the user does not understand why an index is not used and their query runs so slowly.

TIP 9. *Write predicates on the data before any implicit cast operations: Write queries directly on the base collection instead of a constructed document whenever possible. If you are reshaping a document, write the query predicates before the construction. Avoid using SQL comparison on XML data and vice-versa.*

## 3.7 XQuery Namespaces

One often overlooked part of an XPath step is the namespace. It can be empty, "*" wildcard, or implicitly or explicitly specified. Index definitions can include namespace specification just like queries. Of course namespaces have to be taken into account when determining if an index contains all documents needed for a query. Consider the following query with two namespace declarations.

QUERY 28.

```
declare default element namespace
        "http://ournamespaces.com/order";
declare namespace
        c="http://ournamespaces.com/customer";

for $ord in db2-fn:xmlcolumn("ORDERS.ORDDOC")
                /order[lineitem/@price > 1000]
for $cust in db2-fn:xmlcolumn("CUSTOMER.CDOC")
                /c:customer[c:nation = 1]
where $ord/custid = $cust/id
return $ord
```

This query looks like it could use index `li_price` defined in Section 2.2, as well as an index on customer nation codes, such as:

```
CREATE INDEX c_nation ON customer(cdoc)
USING XMLPATTERN '//nation' AS double
```

In reality, neither of these indexes is eligible for this query. Both index definitions do not mention namespaces, which means that they only store XML elements with *empty* namespaces. The query asks for "nation" elements with "customer" namespace, which are not in the index. In order to match this query, an index definition should either include the correct namespace, or contain namespace wildcards. For example, any of the following index definitions would do the trick.

```
CREATE INDEX c_nation_ns1 ON customer(cdoc)
USING XMLPATTERN 'declare default element namespace
                "http://ournamespaces.com/order";
                //nation' AS double

CREATE INDEX c_nation_ns2 ON customer(cdoc)
USING XMLPATTERN '//*:nation' AS double

CREATE INDEX li_price_ns ON orders(orddoc)
USING XMLPATTERN '//@price' AS double
```

Notice that default namespaces do not apply to XML attributes, thus index `li_price_ns` will be eligible for this query, since it does not include any namespace restrictions. Contrast this with `li_price` , which does not mention namespaces, however, by default, restricts the index to `price` attributes of `lineitem` elements that only have empty namespace.

TIP 10. *Make sure that the same namespace definitions are included in your data, queries and indexes. If you don't want to synchronize an index definition with data and queries, use namespace wildcards on* every *element step of the index definition.*

## 3.8 Querying and Indexing XML Text Nodes

There is a difference between an XML text node, and its element parent. If a leaf element of the XML tree contains a single text child, the string value of the element and the text node are the same[4]. However, in general an element can contain any number of children text nodes (as well as

---

[4]However, typed values of an element and its child text node, are still different

other nodes). Thus, for the index to be eligible, the `/text()` XPath steps in a query and an index definition have to align. Consider the following index definition and query.

```
CREATE INDEX PRICE_TEXT ON orders.orddoc
USING XMLPATTERN '//price' AS varchar
```

QUERY 29.
```
for $ord in db2-fn:xmlcolumn("ORDERS.ORDDOC")
    /order[lineitem/price/text() = "99.50"]
return $ord
```

The index and query do not match. If every price element has exactly one text child and nothing else, the index could be used. However, we cannot guarantee this for every document that will ever be inserted into the database. For example, the following document satisfies the query condition; however, its "price" element will be indexed as "99.50USD" instead of "99.50". Thus, using the index for this query would return an incorrect result.

```
<doc>
  <order @id="1003">
    <date>January 1, 2003</date>
    <lineitem @id="LI100101" @quantity="200">
        <price>99.50<currency>USD</currency></price>
  <order>
</doc>
```

TIP 11. *For an index to be eligible, make sure that* `/text()` *steps are aligned between a query and the index definition.*

### 3.9 Attributes and Elements

Users of XPath need to be aware that attribute nodes can be returned only by XPath steps with an "attribute" or "self". Other axes, including "child" and "descendant", will not find attributes. Correspondingly, an index defined on `//*` or `//node()` or `/descendant::node()` will not contain any attribute nodes, even though the `node()` test includes attributes. Recall that the expression `//node()` is a shorthand for `/descendant-or-self::node()/child::node()`. This expression will not match any attribute nodes, due to the child axis in the second step.

TIP 12. *In order to index all attributes in the collection, use the pattern* `//@*`, *or its full notation* `/descendant-or-self::node()/attribute::*`.

### 3.10 Between Predicates

XQuery does not have a special function or operator which is similar to the relational "between" predicate. Furthermore, the existential nature of the XQuery general comparison predicates makes it difficult to express "between". For example, `lineitem[price > 100 and price < 200]`, does not necessarily constitute a between predicate. A lineitem may have multiple `price` children. If one price is above 100, say 250, and another is below 200, say 50, then the `lineitem` satisfies the predicate, even though none of the prices are in the range. Such a predicate cannot be answered with a single range scan over the index– it requires an intersection (ANDing) of the results of two index scans, which may be significantly more costly.

A pair of range predicates on the same item can be interpreted as a "between", and evaluated by a single index range scan, if the system can deduce that the value of the item is a singleton. This can typically be achieved by using value comparisons, the self axis, or attributes, as described below.

If the user is certain that a lineitem can have (and always will have) at most one price, he can write the query using the XQuery value comparisons, which require the comparison operands to be singletons. For example, `lineitem[price gt 100 and price lt 200]` can safely be interpreted as a "between" and evaluated by a single range scan of the `li_price` index. If a lineitem with more than one price child element is encountered, the query will fail at runtime.

As an alternative to the value comparison operators, the self axis can be used to express a "between" predicate. For example, the self axis in the predicate
`lineitem/price/data()[. > 100 and . < 200]`
ensures that both predicates apply to the same price value. This constitutes a "between" predicate since the self axis always evaluates to a singleton. This predicate allows a lineitem to have multiple prices but returns only those of them that have a value between 100 and 200. Notice the `data()` function in the expressions, which is needed to atomize the values of list types. If the system can guarantee that `price` does not have a list type[5], then the following predicate also constitutes a "between":
`lineitem/price[. > 100 and . < 200]`.

If the lineitem price is modeled as an attribute then it can occur at most once per lineitem element. If the system can guarantee that the attribute does not have a list type, the predicate in the following query can be answered by a single index scan.

QUERY 30.

```
for $i in db2-fn:xmlcolumn('ORDERS.ORDDOC')
    //order[lineitem[@price>100 and @price<200]]
return $i
```

TIP 13. *Use value comparisons whenever possible if you are sure that the arguments will be singletons. Otherwise, use predicates of form* `[. > X and . < Y]` *to express the between predicates.*

## 4. CONCLUSION

Indexes are of utmost importance for high performance query processing. This is true for relational queries as well as for XML queries in XQuery or SQL/XML. However, deciding whether a given index is eligible to evaluate a specific query predicate is much harder for XML indexes than for relational indexes. Likewise query rewrite and optimization is more complex for XML queries than for relational queries. In this paper we presented common semantic issues with XQuery and SQL/XML statements which can lead to unexpected query results and prevent index exploitation, in dynamic schema situations as described in the introduction. While XQuery is a powerful new language for formulating

---

[5]For example, our index implementation prohibits the list types from occurring in the indexed documents.

XML queries and construction, a lot more attention is required to ensure compatibility between queries and indexes, so that indexes are used as expected. We showed how common pitfalls in XQuery and SQL/XML can be avoided with careful design of XML index definitions and query predicates. These issues are collected from real-word experiences with XML database systems, but most of the problems are implementation independent and inherent in the XQuery and SQL/XML semantics. Understanding these pitfalls and how to avoid them is crucial for implementers, researchers and users of XML databases alike.

Some of the issues that we raised in this paper, should be revisited in the next version of the XQuery and SQL/XML standards. For example, adding an explicit "between" function would solve the issue of Section 3.10. Most other issues will be harder to address. Small but fundamental changes in the definition of the typed value and element construction can alleviate the problems described in Section 3.6 and Section 3.8. Solving the issues of Section 3.3 would require the unification of the type systems of XQuery and SQL.

## 5. REFERENCES

[1] *The Atom Publishing Protocol*. See `http://www.ietf.org/internet-drafts/ draft-ietf-atompub-protocol-08.txt`.

[2] A. Balmin, T. Eliaz, J. Hornibrook, L. Lim, G. Lohman, D. Simmen, M. Wang, and C. Zhang. Cost-based optimization in DB2 XML. *IBM Systems Journal*, 45(2):271–298, 2006.

[3] A. Balmin, F. Özcan, K. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *Proc. of VLDB*, Toronto, Canada, 2004.

[4] 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. V. der Linden, B. Vickery, C. Zhang, and G. Zhang. DB2 goes hybrid: Integrating native XML and XQuery with relational data and SQL. *IBM Systems Journal*, 45(2):271–298, 2006.

[5] K. Beyer, R. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. Lohman, B. Lyle, F. Özcan, H. Pirahesh, N. Seemann, T. Truong, B. van der Linden, B. Vickery, and C. Zhang. System RX: One part relational, one part XML. In *Proc. of ACM SIGMOD*, Baltimore, Maryland, 2005.

[6] K. Beyer, F. Özcan, S. Saiprasad, and B. van der Linden. DB2/XML: Designing for evolution. In *Proc. of ACM SIGMOD*, Baltimore, Maryland, 2005.

[7] R. Cadenhead, A. Curry, and S. Zellers. RSS history. See `http://blogs.law.harvard.edu/tech/rssVersionHistory`.

[8] A. Eisenberg and J. Melton. Advancements in SQL/XML. *Sigmod Record*, 33(3):79–86, 2004.

[9] International Organization for Standardization (ISO). Information Technology-Database Language SQL-Part 14: XML-Related Specifications (SQL/XML), ISO/IEC 9075-14:2006.

[10] *Java Content Repository (JCR) Standard*. See `http://www.jcp.org/en/jsr/detail?id=170`.

[11] R. Murthy, Z. H. Liu, M. Krishnaprasad, S. Chandrasekar, A.-T. Tran, E. Sedlar, D. Florescu, S. Kotsovolos, N. Agarwal, V. Arora, and V. Krishnamurthy. Towards an enterprise XML architecture. In *SIGMOD Conference*, pages 953–957, 2005.

[12] M. Nicola and B. V. der Linden. Native XML Support in DB2 Universal Database. In *VLDB*, pages 1164–1174, 2005.

[13] F. Özcan, D. Chamberlin, K. Kulkarni, , and J.-E. Michels. Integration of SQL and XQuery in IBM DB2. *IBM Systems Journal*, 45(2):245–270, 2006.

[14] M. Rys. XML and relational database management systems: inside microsoft SQL Server 2005. In *SIGMOD Conference*, pages 958–962, 2005.

[15] E. Sedlar. Managing structure in bits & pieces: the killer use case for XML. In *Proc. of SIGMOD*, pages 818–821, New York, NY, USA, 2005. ACM Press.

[16] W3C XML Query Working Group. See `http://www.w3.org/XML/Query`.

[17] Web Services. `http://www.w3.org/2002/ws`.

[18] *XQuery 1.0: An XML Query Language*, November 2005. W3C Candidate Recommendation, See `http://www.w3.org/TR/xquery`.

[19] *XQuery 1.0 and XPath 2.0 Data Model*, November 2005. W3C Candidate Recommendation, See `http://www.w3.org/TR/xpath-datamodel`.