Query Rewrites with Views for XML in DB2

P. Godfrey, J. Gryz, A. Hoppe, W. Ma, C. Zuzarte

Abstract—There is much effort to develop comprehensive support for the storage and querying of XML data in database management systems. The major developers have extended their systems to handle XML data natively. These have the advantage over stand-alone XML database systems that relational and XML data can be queried mutually. Indeed, recent SQL standards specify means to query relational and XML data together (called SQL/XML). These systems also now support XQuery, in addition to SQL. It is thus possible to mix the processing of relational and XML data via either query language.

While there has been significant progress in efficient native storage systems for XML, there remain numerous challenges to handle efficiently queries over XML. There are efforts to adapt the strong optimization techniques used for relational (“SQL”) queries for XML (and mixed) queries as well. One such technique, the materialized view, has been well studied, and well adopted, over the last decade as an effective technique for optimizing relational queries.

Our work extends the use of materialized views for SQL/XML, and could be applied to XQuery. Within IBM DB2 9 (Viper), we implement query rewrite rules that enable the use of materialized views in the evaluation of queries over XML. To accomplish this, it was necessary to extend the existing query matching and compensation framework in DB2 with new functionality. We consider what types of query rewrites based on XMLTable are possible, and which are feasible. We present a linear-time algorithm to determine the locality (self-containment) of XPath expressions within a schema-unaware environment, which we have implemented. We demonstrate the efficacy of our techniques via an experimental evaluation over a representative suite of SQL/XML queries and materialized views, executed over our DB2 prototype.

I. INTRODUCTION

XML technology is used in practically all industry due to its versatility and neutrality for exchanging data across diverse devices, applications, and systems from different vendors. This, along with its transparent, self-describing nature and its ability to handle the spectrum of structured, semi-structured and unstructured data, has made XML into a universal standard for data interchange. As a result of XML industry standards becoming prevalent, the amount of data stored as XML has grown rapidly. Meanwhile, a significant amount of data is—and always will be—stored in a relational format. Many applications must process both XML and relational data. Some types of data are best modeled and stored in the relational format, while other types are more suited for XML. Most DBMS vendors have committed to support these two storage standards within a single database management system. (Mostly, they have extended their successful relational platforms for also handling XML.) IBM DB2 9 (Viper) is both a full-fledged relational and XML system [1]. It offers first-class support for both XML and relational data processing, as well as the integration of these two standards. It incorporates a new native XML storage system called XML Store, and supports XML indexes that can be used also within the existing relational storage and the query processing engine. Data stored in DB2 can be queried by SQL and by XQuery. DB2 9’s SQL covers most of the language features specified by the SQL 2003 standards, as well as that standard’s SQL/XML component [2]. This allows for XML documents to be (logically) stored in tables as values in columns of type XML.

These architectural changes to the system do not affect existing SQL applications; the interfaces and parsers for SQL and XQuery are independent from one another. In the back-end, however, DB2’s compiler and optimizer have been extended to handle the new functionality of SQL and XQuery within a single modeling framework. This allows applications that use XML to benefit from the many features that are already available in the database management system.

Of course, this is an ongoing evolution. Many features remain missing. Components to support these require further enhancements and modifications, and perhaps, new, orthogonal functionality to be added to DB2. While there is a rich base for query optimization for relational (“SQL”) queries, the same does not exist for XML, and mixed, queries.

One such technique, the materialized view, has been well studied, and well adopted, over the last decade as an effective technique for optimizing relational queries. A view is simply a named relational query that can then be referenced as if it were
a table itself. When a view is declared as materialized, the system maintains the results of the view’s query as a physical table in the database. The system undertakes the responsibility of ensuring that the materialized view’s table correctly reflects the view’s query. Thus, on the one hand, materialized views incur expense since they must be updated when the tables upon which they rely change, and they take up storage space. On the other hand, a good choice of materialized views in a database can greatly improve the performance of a query workload.

SQL/XML is considered SQL-centric. It provides SQL programmers a way to access and mix XML data with relational data from within SQL. The key extensions the standard provides which allow for this are the first-class data type XML for typing columns and the table function XMLTable. An XML document can be stored in a row in an XML column. Constraints can be added to an XML column as well that enforces that only XML documents that conform to given XML schemas may be stored there. However, there are no such constraints by default, so the collection of documents stored in a given column then must be considered as schema free.

XMLTable provides a way to query and navigate the documents stored in XML columns in the database within the SQL framework. (We briefly introduce its syntax and semantics in §III-B.) The result is a relational “table” [3]. Thus, it fits within the relational algebra as just another relational operator which respects algebraic closure. The contents of an XML column in the result table are copied elements from the XML elements in the database. This is good as the result table is independent of the rest of the database, and it can be further processed without need to reference the database. This is bad, however, as this restricts the possible XML navigation because these elements are outside of the context of the original documents (for example, we may lose information about the ancestors of a node).

XQuery is considered, of course, as XML-centric. It will usually be the better choice for applications that focus primarily on the XML data. The XQuery standards provide for means to query relational data in conjunction with the XML data. DB2 supports XQuery, as do the other major database systems. The result of an XQuery query is not tabular. The results from XQuery are references to elements to the context of their original documents. This is good as it supports the full XML navigation semantics as reflected in XPath, and so allows for powerful subsequent processing of the results. This is bad, however, as such subsequent processing can require returning often to the context of the elements in the database, and so can be highly inefficient.

With XMLTable, we can leverage the existing view facilities of the database system for XML data. An SQL query using XMLTable and with XML columns in its output can be declared as a view, just as any other SQL query can be. SQL queries with XMLTable can be materialized views too. However, one does not get for free that these materialized views will be used to optimize SQL/XML queries. The mechanisms to match correctly the views to queries is needed.

Our work extends the use of materialized views for SQL/XML. Our framework could be further extended to help optimize XQuery with the use of the same materialized views. Within DB2 9,1 we implement query rewrite rules for SQL/XML queries that enable the use of materialized views in their evaluation. To accomplish this, it is necessary to extend the existing query matching and compensation framework in DB2 with new functionality. In DB2, materialized views are called materialized query tables (MQTs). We shall refer to these as MQTs when discussing our implementation.

We consider the types of query rewrites based on XMLTable that are possible, and furthermore, which of these are feasible. We discuss a linear-time algorithm that determines the locality (self-containment) of XPath expressions within a schema-unaware environment, which we implemented. We then demonstrate the efficacy of our techniques via an experimental evaluation over a representative suite of SQL/XML queries and materialized views executed on our DB2 prototype.

The paper is organized as follows. In §II, we discuss related work. In §III, we provide brief background on query processing in DB2 9 and the XML/SQL standard. In §IV, we present our solution for using materialized views for queries over XML. In §V, we present an experimental evaluation of the techniques. In §VI, we consider future directions and conclude.

II. RELATED WORK

Query rewriting with materialized views [4] has been investigated in the context of relational database systems for a while, with algorithms having been implemented in all major commercial systems. One approach to rewrites of XQuery queries is to transform the collection of XML documents into relational format, and then use these existing algorithms. There are several strategies for storing and accessing XML data in a relational DBMS. An XML document is either shredded into relational table tuples [5], [6], or simply saved as a Character Long Object (CLOB) [7] or a Binary Long Object (BLOB) [8]. An XQuery query is then translated into an SQL query, which is evaluated by the SQL query engine. As argued in [9], there are many problems with these approaches:

a) some XQuery constructs cannot be expressed in SQL;

b) order preservation of nodes from the XML documents is difficult to maintain;

c) a simple XQuery query may be translated into an SQL query which requires evaluating a large number of joins (or other expensive operations); and

d) expensive re-parsing of the XML documents during query runtime is necessary.

There have been important theoretical results established regarding the complexity of XPath and XQuery rewrites. The complexity of query containment of subclasses of XPath expressions has been of particular interest [10]. For the most part, the news is bad. Even for quite limited subclasses of

1We implement a prototype of these techniques as a code branch-off of the DB2 9 code base.
XPath, the complexity is worse than polynomial. For other subclasses, it is even undecidable. Moreover, query containment is only one part of the process in query rewrite. We might know that a given query is contained in a view, but still not be able to provide a rewrite. Consider the view defined as \( /a, \) “return all descendant nodes of element ‘a’ from the document,” and the query \( /a, \) “return all child nodes ‘a’ from the document.” While the view certainly contains all the nodes that are returned by the query, given that only copies of nodes are stored in the view, we cannot select them from the view. We cannot tell which nodes correspond to (direct) children of the root node of the document (which the query requests), and which are not.

XPath rewrites were investigated in [11]. However, this work is not applicable to rewrites of SQL/XML due to the non-convertibility of XQuery and SQL/XML. The techniques presented in [12] are not useful in our context either, as they require the additional information of the node references and the paths to a nodes to also be stored.

The only practical technique for rewrites of XML/SQL by means of materialized views, to our knowledge, is presented in [13]. Their objective, however, is significantly different from our own. In fact, their goal was to avoid accessing materialized views that contain published relational data as XML. This was due to high cost of accessing and querying XML stored as CLOBs. For us, XML is stored under a full-fledged, first-class data type, with native storage designed and implemented for high performance processing. Native solutions for XML—and relational data as well—have better performance and functionality [14]. For us, choosing to evaluate with a materialized view will often provide a significant performance improvement.

III. BACKGROUND

A. Query Processing in IBM DB2 9 (Viper)

DB2 9 is a hybrid relational and XML system. It introduces a storage system called XML Store to facilitate for storing and querying XML documents. XML Store is independent from the relational data storage system. These two storage systems co-exist within DB2 databases. XML Store stores XML data in a format more amenable to its structure. The basic storage unit is the node, the types of which are defined by the XQuery Data Model (XDM) [15]. Node types include element, attribute, document, text, namespace, processing instruction, and comment. A node record contains pointers to its parent node, first child node, and next sibling, if applicable. A disk page may hold one or more nodes. A node may be stored across multiple disk pages, if the size of the node exceeds the size of a disk page. For example, a long text node in an XML document might. All nodes from the same XML document are logically interlinked to form a tree fragment of the document tree via parent and child pointers. Disk pages in XML Store hold fragments of XML document trees. DB2 calls such a fragment a region. The regions of an XML document tree are laced together to represent the complete XML document tree via an index structure called a region index in DB2. As

RIDs address tuples in the relational data storage, each node is addressed by a unique identifier (an NID), which offers both logical and physical addressability. The NID identifies a node within the document via a Dewey node identifier [16], and provides quick access to a node in the XML Store. Given an NID reference, the node can be located on disk, and its ancestors and descendants can be reached via the parent and child pointers stored at the node, or via the region index. If the node is a document node, then the whole XML document stored in the XML Store is accessible via its NID. This provides for efficient navigation over the XML document, and so is beneficial for evaluating XQuery statements. XQuery uses node reference semantics in its results. The result of any XQuery expression must be a sequence containing either constant values or references (NIDs) to nodes in the document. In XQuery, it is common to navigate further the document fragments that result from an XQuery expression.

DB2 9 introduced a number of changes to the Query Graph Model (QGM) [17], its internal data structure for representing queries, to support XML processing. We explain how QGM represents XQuery expressions.

XQuery is represented during query compilation in DB2 9 as a parse “tree” (called a PID tree). Each node in a PID tree is a PID QGM entity. A new type of PID called XPath Step (XPS) was introduced [12]. Each XPS node has four positional child PID nodes.

1) The first child PID node represents the step axis; that is, for instance, child, parent, descendant, or self.
2) The second child PID node represents the test of the step, it could be a name test, or a kind test, or a wildcard test (‘*’).
3) The third child PID node represents the predicates of the step.
4) The fourth child is another XPS representing next step, or is NULL if the step is the final step of the expression.

If an XPath expression has a context represented by an XQuery variable—this is when a PID tree could reference an XQuery variable—then the first child PID would be NULL and the second child PID would be a variable reference.

Consider the XPath query \(/a/\text{a}/\text{/b}\). The path consists of three steps. The first is a descendant-or-self step. The second is a wildcard and specifies that an anonymous child must exist. The final step says this anonymous element must have an
attribute b. The tree to represent this will be right-deep with a depth of three representing this list of steps. A sketch of the QGM for this is shown in Figure 2.

![QGM for XPath //a/*/@b](image)

**Fig. 2.** QGM for XPath //a/*/@b.

**B. SQL/XML**

As introduced in §I, SQL/XML introduces a set of new functions into the standard of SQL. Its `XMLTable` function provides a general mechanism for generating a table from XML data. As `XMLTable` is a table function, it can be used in the `FROM` clause of an SQL query. An `XMLTable` function invocation uses XQuery to select the elements and compose the results. Rather than returning the direct results of the XQuery expressions, `XMLTable` uses these results to construct a tabular view of results, a “table”. The `XMLTable` function uses XQuery in two ways:

1) for the **row generator**, and
2) for the **navigators**.

The row generator is analogous to the for quantifier in the XQuery language, and the navigator is analogous to the let quantifier. Conceptually, the row generator passes its output items to the navigators one at a time. Each navigator consumes items and produces result values. A single item from the output of the row generator is used to create values for precisely one row of the result table. Each column of the output table is populated by its corresponding navigator expression. The definition of a navigator includes a casting that specifies the data type of its column. That casting can be to column type XML. Thus, the columns in the output table correspond to the navigators in the `XMLTable` function call, and the number of resulting rows is equal to the number of items generated by the row generator.

Let us consider an example. Consider the query in Figure 3 that uses the `XMLTable` function, and the document in Figure 4 as the value in column `bookdoc` of the single row of the input table `books`.

The output of the query then would be as in Figure 5. The `XMLTable` function produces a table with two columns. The row generator is the XPath expression `$book/book/chapter`. It produces three rows of output as the row navigator matches three `chapter` nodes from the input document from the column `bookdoc` (from the one row of input in the table `books`).

The navigator expressions populate the column values of the output rows. They do not generate more rows, multiply them, or eliminate any. The output will have exactly the number of rows that the row generator produced. Rather, the navigators fill in the columns of these rows. The output table has two columns, `title` and `sections`. The navigator for column `title` is an XPath expression `title`. This is evaluated with respect to each node matched by the row generator. The results are cast to VARCHAR. The navigator for column `sections` is an XPath expression `section`. Its results are cast to type XML. Note that the value of `sections` for the second row, for Chapter Related, is a list of XML elements, the two sections from the chapter. In the case of the third row, for Chapter Solutions, the value is null as that chapter element contains no section elements.

Note that XPath is a proper fragment of XQuery, and any XPath expression by itself is a valid XQuery query. Our work is presently restricted to XPath expressions for the row generators and navigators, so does not permit for the use of full-fledged XQuery constructs for these. This is not so serious a restriction, as most XML/SQL queries, in practice, use only XPath expressions in their row generators and navigators. Indeed, the nature of the row generator and the navigators lend themselves to definition by XPath expressions.

SELECT x.title, x.sections
FROM books,
XMLTable ('$book/book/chapter' PASSING bookdoc AS "book"
COLUMNS "title" VARCHAR (60) PATH 'title',
"sections" XML PATH 'section'
) AS x;

![Example query using XMLTable.](image)

**Fig. 3.** Example query using XMLTable.

```xml
<?xml version="1.0" encoding="utf-8"?>
<book>
  <chapter>
    <title>Introduction</title>
    <section>Goals</section>
  </chapter>
  <chapter>
    <title>Related</title>
    <section>Views</section>
    <section>XML</section>
  </chapter>
  <chapter>
    <title>Solutions</title>
  </chapter>
</book>
```

**Fig. 4.** Sample input document.
IV. SOLUTION

A. Challenges

A matching and compensation framework is needed if we are to match soundly queries to SQL/XML views that can answer them. Matching refers to the process of discovering whether a view is equivalent to, or contains, the query. If the two are equivalent, then the view can be used directly to answer the query. However, if the view contains the query, but not vice versa, we need to compute a compensation; that is, further restrictions—navigation steps and predicates—which, when applied to the view, is equivalent to the query. (In general, the view might contain the query, but no compensation exists. Recall the example in §II.)

One requirement for applying query rewrite with SQL/XML views is that the row generators of the query and the view (MQT) must be the same. The reason for this is that any change to the row generator can change the cardinality of its output; that is, how many rows it produces. Navigators cannot change this cardinality, so there generally exists no compensation if the row generators differ.

On the one hand, this sounds restrictive. On the other hand, row generators tend not to be complex. It is reasonable to be able to select views to materialize to optimize a query workload so that the views share row generators with the queries. Also, many queries will use the same row generator.

The query rewrite in DB2 cannot be based on any XML schema. An XML column only optionally has schema constraints on it. Therefore, our rewrites are schema unaware.

B. Matching

As discussed in §II, the containment and equivalence problem for XQuery is hard. In our work, we restrict our attention to just XPath expressions. Even so, these problems are hard for XPath too. Thus, our goal is to come up with tests that are sufficient to establish containment, and procedures to calculate the compensations, but is not necessary. It is certainly not necessary that we detect all cases. We may miss some opportunities to evaluate the query by rewrite. Our objective is to capture a reasonably large, realistic class of cases for which we can detect containment and compute the compensations.

Two XPath expressions are equivalent if there is an isomorphism between their XPath trees, or XPS representations, as introduced in §III-A. Isomorphism between two labeled, ordered trees can be determined in linear time.

This captures a reasonably large, realistic class of cases for the following reasons.

1) First, our determination of the XPath isomorphism is performed on the internal representation of XPath expressions (XPS) used inside the DB2 engine. One of XPSs main objectives is canonical representation. Therefore, we leverage on this.

2) Second, the determination of XPath isomorphism is done after most of the other query rewrites. Many query rewrites also are targeted to a canonical representation. They remove redundant computations. For example, one rewrite combines self-steps with descendant-steps to obtain descendant-or-self steps.

This means the internal representation of XPath expressions is much closer to a canonical form than the input expressions. Additionally, we relax our requirement of strict isomorphism in a crucial way. Parts of XPS that store predicates are passed to our predicate matching algorithm.

Because of this, it is much more likely that the final representations of the expressions match whenever the expressions are, in fact, semantically equivalent.

Unless navigators are isomorphic, then three conditions for the navigators have to be satisfied to perform a query rewrite:

1) be type convertible,
2) satisfy the prefix property, and
3) satisfy compensation locality.

Type convertibility simply refers to the castings DB2 allows between type XML and other SQL data types. In particular, the type conversions must be deemed safe (that is, the transformation is not lossy).

The XPath expression from the MQT’s navigator must be a containing prefix of the XPath expression from the query’s navigator. That is, each step from the beginning of the expressions must match in that the MQT’s step contains—is less restrictive than—the query’s. The query’s expression can have additional steps after the matching prefix.

A step is less restrictive than another if

- the steps have the same axis navigators,
- the step’s test is less restrictive than the other’s, and
- the step’s predicate implies the other’s.

Those two requirements would be sufficient if a MQT stored references to nodes. However, MQTs in DB2 store copies of nodes. Therefore, for there to exist an XPath expression for the compensation, we must have local, or self, containment.

We illustrate this problem on the following example. Consider the query in Figure 6 that returns all ancestors from the id node to the root of the document.

Assume also that the MQT in Figure 7 is available.

It might seem that the rewrite in Figure 8 of the query by means of the MQT is correct. However, the root nodes of the trees returned by the row generators for the original and for the rewritten query are different: it is the root node for the former and the name node for the latter. Thus, the answers of the two queries will differ: the list root, item, description, name for the former and a single name node for the latter.
SELECT xtable.ancestor
FROM t1, XMLTable ('$r/root/item'
PASSING xmldoc AS "r"
COLUMNS
"ancestor" XML PATH
'description/name/id/ancestor::*'
) AS xtable;

Fig. 6. Ancestor query.

CREATE TABLE mqt_name AS (
SELECT xtable.name
FROM t1, XMLTable ('$r/root/item'
PASSING xmldoc AS "r"
COLUMNS
"name" XML PATH
'description/name'
) AS xtable );

Fig. 7. Ancestor MQT.

The correctness of the rewrite can be easily guaranteed if only child and descendant relations are permitted in the compensation XPath expression. We do relax this rather strict requirement by imposing what we call a locality requirement. Briefly, an expression is local, if its computation does not require any information beyond the nodes contained in the sub-trees of nodes stored in an MQT. We leave out the details of the linear algorithm that determines the locality of a query with respect to an MQT.

Predicate matching is often overlooked in research on queries rewrite, and especially in the case of XPath matching. Predicate matching for relational queries is reducible, for the most part, to a logical implication and equivalence. Predicates in the XPath language, however, act differently from relational predicates, so existing database technology cannot be used for matching.

For example, consider the predicate name > 5 and name < 5. If this were a relational condition, it is unsatisfiable. Consider instead that it is a test on an XPath step: root[name > 5 and name < 5]. That can have matches. The meanings are not the same in the two cases. As a step test, this asks that a matching node of element root have at least two children nodes of element name, one of which evaluates less than five, and the other evaluates more than five.

An XPath predicate can be modeled as a binary tree. The internal nodes represent logical operators, and leaves represent base conditions. We design and implement a matching algorithm between two predicate binary trees. It is not necessary for the trees to be ordered the same way, as we do require for the path expressions. Rather, we perform a mapping to see if one subsumes the other. We do restrict, for now, that no base condition is repeated. Thus, if we find a subsumption mapping, it is unique (each base condition in one tree can map to only one condition in another tree). Given a “subsumer” tree of size \( m \) and a “subsumee” tree of size \( n \), this takes \( mn \) steps. However, predicate trees are typically small, so this is acceptable. Figure 9 shows an example mapping.

C. Compensations

With the restrictions we have put on matching, the compensation results from our algorithm. This consists of two parts: the path beyond the matching prefix, plus any restrictions to steps over the MQT’s; and, as tests, any remaining predicate conditions in the subsumee tree that were not matched to the corresponding subsumer predicate tree in the predicate matching. (Pseudo-code of this process is shown in the Appendix.)

V. EXPERIMENTAL EVALUATION

A. Setup

The existing matching and compensation framework in DB2 9.5 (VIPER 2) was extended as discussed in §IV to handle new types of matching using XMLTable. The default configuration for database manager and database instance, except for the number of secondary log files, was used. These queries are not generally sensitive to the database configuration, except for the number of secondary log files; an increased number of secondary log files is necessary to load large XML documents in DB2.

The page size is set at four kilobytes, and the total buffer pool size at 1000 pages, for a four megabyte buffer pool. Thus, the documents used in the experiment do not fit into the buffer pool.

To compare evaluation of queries not using MQTs and using MQTs, a duplicate of each table is made. An MQT is created only for the second table. The query asked against the first table has no applicable MQT, whereas the same query asked against the second table (the duplicate) does; it can be rewritten by using the MQT by the matching and compensation framework.

Each query is posed five times in a single user mode. DB2batch is used to time execution. Each query is submitted to the database engine separately with an exclusive database connection. The elapsed times do not include the output time. That is, the time lapsed measures until all output rows are fetched into main memory.

1344
B. Database and Queries

A data generator was used to produce the databases used in the experiments. The schema is based on a shops document schema as depicted in Figure 10. Connecting lines indicate inclusion relationships between the elements. A dashed line means that the element is optional. The N marker indicates a one-to-many relationship.

Two versions of the shops document are used. The first is ten megabytes, containing 10,000 items sold in five shops. The second is one hundred megabytes, containing 100,000 items sold in ten shops.

Thirteen queries are executed against the shops document for the 10MB and 100MB versions. The first twelve all share a similar pattern. Consider Query 4 of this suite in Figure 11.

```
SELECT xtable.column
FROM shops, XMLTable ('$r/root//item'
PASSING xmldoc AS "r"
COLUMNS
"column" XML PATH 'description/date/instockDate[@year]'
) AS xtable;
```

Fig. 11. Query 4.

Consider if we had the general MQT in Figure 12 in the database. Let the MQT be named mshops.

```
SELECT xtable.column
FROM shops, XMLTable ('$r/root//item'
PASSING xmldoc AS "r"
COLUMNS
"column" XML PATH 'description/date/instockDate[@year]'
) AS xtable;
```

Fig. 12. An MQT.

Then we could rewrite Query 4 to use this MQT (mshops, as in Figure 13, using the compensation date/instockDate[@year] as the navigator on column.

```
SELECT xtable.column
FROM mshops, XMLTable ('$r'
PASSING mqt_column AS r
COLUMNS
"column" XML PATH 'date/instockDate[@year]'
) AS xtable;
```

Fig. 13. Rewrite of Query 4.

All twelve queries in the suite and the associated MQTs take the same form. The abstracted pattern is as follows. Each query has the form as in Figure 15.

```
SELECT xtable.column
FROM shops, XMLTable ('$r/root//item'
PASSING xmldoc AS "r"
COLUMNS
"column" XML PATH 'query_navigator'
) AS xtable;
```

Fig. 15. Query template.

Each MQT has the form as in Figure 16.

```
CREATE TABLE mshops AS (
SELECT xtable.mqt_column
FROM shops, XMLTable('$r/root//item'
PASSING xmldoc AS "r"
COLUMNS
"mqt_column" XML path 'mqt_navigator'
) AS xtable
);
```

Fig. 16. MQT template.

Each rewritten query has the form as in Figure 17.

```
SELECT xtable.column
FROM mshops, XMLTable ('$r'
PASSING mqt_column AS "r"
COLUMNS
"column" XML PATH 'compensation'
) AS xtable;
```

Fig. 17. Rewrite template.

Figure 14 shows the specifics of the twelve queries with respect to our template, the MQT that is used in the rewrite, and the compensation used in the rewritten query. As discussed in §IV, any applicable MQT must employ the same row generator as the query. For each query (and matching MQT), this is shown in row generator in the chart. Query Navigator shows the navigator on column in the query, and MQT Navigator, the navigator on column in the applicable MQT. Compensation shows the navigator on column in the rewritten query.

Notice that we are not considering twelve different MQTs, one for each query! In this suite, eight MQTs are represented. The MQTs are meant to be general, and so could be used in rewrites for many queries. Eight are represented here simply because we are demonstrating different scenarios of rewrites that our techniques accommodate. Generally, one
<table>
<thead>
<tr>
<th>#</th>
<th>Row Generator</th>
<th>Query Navigator</th>
<th>Compensation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/root/shops/shop/item</td>
<td>prices/price</td>
<td>prices/price</td>
</tr>
<tr>
<td>2</td>
<td>/root/item</td>
<td>prices/price</td>
<td>prices/price</td>
</tr>
<tr>
<td>3</td>
<td>/root/item</td>
<td>description/date/instockDate</td>
<td>description/instockDate</td>
</tr>
<tr>
<td>4</td>
<td>/root/item</td>
<td>description/date/instockDate[@year]</td>
<td>date/instockDate[@year]</td>
</tr>
<tr>
<td>5</td>
<td>/root/item</td>
<td>description/date/instockDate[@year and @month and @day]</td>
<td>date/instockDate[@year and @month and @day]</td>
</tr>
<tr>
<td>6</td>
<td>/root/item</td>
<td>description/info/date/instockDate[@year&gt;2006]</td>
<td>description/instockDate[@year&gt;2006]</td>
</tr>
<tr>
<td>7</td>
<td>/root/item</td>
<td>description[contains(info,&quot;car&quot;)]/name</td>
<td>description[contains(info,&quot;car&quot;)]/name</td>
</tr>
<tr>
<td>8</td>
<td>/root/item</td>
<td>description/date/productionDate[@year&gt;2005]/../instockDate</td>
<td>description/instockDate[@year&gt;2005]/../instockDate</td>
</tr>
<tr>
<td>9</td>
<td>/root/item</td>
<td>description/date/name/productName</td>
<td>description/name/productName</td>
</tr>
<tr>
<td>10</td>
<td>/root/item</td>
<td>description/info and date]/name/productName</td>
<td>description/info and date]/name/productName</td>
</tr>
<tr>
<td>11</td>
<td>/root/*/item</td>
<td>description/date</td>
<td>description/date</td>
</tr>
<tr>
<td>12</td>
<td>/root/*/shop</td>
<td>name/abbreviation</td>
<td>abbreviation</td>
</tr>
</tbody>
</table>

Fig. 14. The Queries and MQT’s.

could optimize for a large query workload with just a small handful of MQTs.

The output of Query 1 is a one column table that stores price nodes. A price node is returned for each item node that can be reached from the roots of documents stored in the shops table. The item node has to have root, shops, and shop ancestors. The MQT (mshops) stores price nodes for each item, and has the same requirements as in the original query. The compensation in the rewritten query is the same as the navigator in the query, as the MQT’s navigator path subsumes the query’s.

Query 2’s case is similar to the first. The navigators and compensation are the same. The difference is the row generator.

For Query 3, the novelty is the complication of the compensation, as it contains two additional navigational steps. In fact, our rewrite mechanism can accommodate subsequent navigation in the compensation of arbitrary complexity.

Query 4 demonstrates the use of a predicate and an attribute axis, Query 5, a compound predicate, and Query 6, multiple predicates along the path. Query 6 also employs a comparison condition.

Query 7 demonstrates our support of built-in functions. Query 8 shows that we can handle certain, more complex navigation steps, as the parent step in the query’s navigator. (As long as the navigation stays within the scope of the nodes recorded in the MQT, we can accommodate for it in the compensation.)

Query 9 demonstrates that we can match predicates that may appear in the MQT’s navigator to the query’s. Furthermore, the compensation drops the predicate test, as it has been ascertained that it is already satisfied by the MQT. Query 10 demonstrates a more complex example of this. In this case, the query’s predicate is subsumed by the MQT’s, but they are not equivalent. Note that the compensation’s predicate test is more complex than it logically needs to be. (However, it is correct.) Presently, simplification of the predicate is not performed when the matching between predicates in the query and the MQT is not exact. That is future work.

Queries 11 and 12 demonstrate the support of wildcards (‘*’). The wildcard can be used in the context of a node name and a namespace. Query 12 further demonstrates matching against wildcards that our technique accommodates in com-
puting the compensation.

A thirteenth query is used in the experiments, but it does not follow the pattern of the previous twelve. Query 13 is shown in Figure 18.

```
SELECT xtable.count
FROM shops, XMLTable ('$r/root/item'
PASSING xmldoc AS "r"
COLUMNS
  "count" XML PATH
  'count(quantity/original)'
) AS xtable;
```

Fig. 18. Query 13.

The MQT considered for Query 13 is shown in Figure 19.

```
CREATE TABLE mshops AS (
  SELECT xtable.mqt_column
  FROM shops,XMLTable ('$r/root/item'
PASSING xmldoc AS "r"
COLUMNS
  "mqt_column" XML path 'quantity')
  AS xtable);
```

Fig. 19. MQT for Query 13.

The rewritten query for Query 13 using the MQT is shown in Figure 20.

```
SELECT xtable.column
FROM mshops, XMLTable ('$r'
PASSING mqt_column AS "r"
COLUMNS
  "column" XML PATH
  'count(quantity/original)'
) AS xtable;
```

Fig. 20. Rewrite for Query 13.

This scenario demonstrates use of the built-in, aggregate function count. The result of the expression is passed to the built-in function, and the function's result is returned as the final answer. Presently, we support the count function, and have considered how to handle the adding and removing of such aggregate wrapping in compensation. However, support of each built-in function used as wrapper (on the top of an XPath expression) requires more investigation. All functions supported by DB2 can be matched and compensated by our framework when they occur inside the XPath expression.

C. Results

There is, as of the time of our experiments, an important limitation inherited from the XQuery standard in the use of XML/SQL result tables for MQTs in DB2. Every value entry of an XML column in a table must be a well-formed XML document. This means two things: The document must have the proper preamble (<?xml ...), and it cannot be a list but must be a single document. So the output in Figure 5 from the example in §III is not a legal value for an XML column in a table. That output is legal as the response to the SQL/XML query.

Of course, this breaks, in some sense, algebraic closure with respect to SQL/XML queries. Arbitrary results of an SQL/XML cannot be stored as a table. This is recognized as an issue.3

In the meantime, we need to “wrap” the results in the XML column in order to make our MQTs. We call this step document creation. To perform a fair comparison of our rewritten queries (using the MQTs) to the original queries, we consider two versions of the original queries: one that performs the document creation (for each XML column value), and one that does not. (Our rewritten queries also do the document-creation step.) For the meantime, comparing the original queries with document creation against the rewritten queries is the fairer comparison.

The thirteen queries described in the previous section were tested on the AIX 5.2 UNIX operating system running on a IBM 7038-6M2 with eight 64-bit, 1452MHz, PowerPC POWER4 processors and 16GB of main memory. Each original query is evaluated in the two versions: with and without document creation in the navigator. The query suite is evaluated against 10MB and 100MB versions of the database. The timing results are presented in Figures 21 and 22, respectively.

In all cases, the rewrite involving the relevant MQT showed an improvement. With respect to the 10MB database, the

3To be fair, this arises from confusion that still persists in XML query standards in that they do not respect algebraic closure. The result of a query over XML documents need not be a document itself.
improvement of the rewritten query to the original ranged from 30% to 55%. With respect to the 100MB database, this ranged from 27% to 54%.

We included in the comparison the execution of the original query without document creation, just to show the overhead incurred by the document creation step. Even so, the rewritten queries using the MQTs still perform significantly better, even though they do involve document creation. In the future, this requirement of document creation should be removed from the database engine. Once that is done, the elapsed times of rewritten queries will decrease further.

We also note that in all cases the compile cost of doing the matching and generating the compensation was negligible compared with the query evaluation cost.

VI. CONCLUSIONS

A. In Closing

To the best of our knowledge, this is the first work that considers use of materialized views to improve SQL/XML performance for XML data. While the matching and compensation that we have implemented so far is quite simple from a theoretical perspective (but not simple in the least in implementation, especially in a commercial system), it seems applicable for a wide range of SQL/XML queries. And our experimental results demonstrate that the use of materialized views for SQL/XML queries to be a very viable optimization technique.

B. Future Work

Of course, this work is just a start. There is much more that can be done, and should be.

1) Extend the matching and compensation framework for a wider class of XML/SQL queries.

2) Move the framework into the cost-based optimizer.

3) Make the framework applicable to XQuery.

There is much work that can be done to extend the compensation framework. Presently, the framework only covers the case of one (XML) column in the query and in the MQT. (While this is a quite restricted class, it is a quite prominent one.) Work is needed to extend the compensation framework to the cases of one column in the query but with many in the MQT, many columns in the query but with just one in the MQT, and many columns in the query and in the MQT. None of these is straightforward to address, but we should be able to handle these cases to some degree.

Our present matching algorithm relies on a direct isomorphism. As argued, this is still quite effective in practice, while providing a linear algorithm. Even so, we ought to be able to broaden the matching some to cover more opportunities, while still not sacrificing efficiency.

Much more can be done on predicate conversion, reduction, and elimination. This is likely to come about through incremental iterations and improvements in the framework as it is incorporated into the DB2 code base proper. There is a great deal of experience with this on the relational side, but little experience so far on the XPath and XQuery side. One has to be painstakingly careful in order to abide by XPath semantics, as the issues can be quite subtle.

Our present implementation is a rewrite approach and is implemented in the rewrite engine of DB2. Thus, the decision to use a materialized SQL/XML view to rewrite a query is made before cost-based optimization. The rewrite with the view may not always perform better than the original query, however. It would be far better to make this decision in the cost-based optimizer, so whether to use the rewrite is decided based on cost estimations. General use of materialized views for query optimization is cost-based in DB2 today. We will need to adapt these new techniques into the cost-based optimizer as well.

It is an important goal to use these techniques to optimize XQuery in the database system too. In principle, XQuery expressions could also benefit from materialized SQL/XML views. There is research involved here, though, and solutions are not directly clear. In particular, XQuery and SQL handle empty answers—NULLS and empty lists—in different ways.

This work has been just one demonstration in how known, effective optimization techniques for relational queries can be adapted for queries over XML data. We have focused on materialized views. Many other optimization techniques from relational query processing can also likely be adapted.

REFERENCES


APPENDIX

Figure 23 provides a pseudo-code sketch of the matching and compensation process as discussed in §IV.
// local variables
bool equivalence = true;
vector<condition> ser_family;
vector<condition> see_family;

// initialization
find_matching_between_basic_conditions(subsumer, subsumee);
combine_unmatch_conditions(subsumer);
combine_unmatch_conditions(subsumee);
if (has_unmatch_or_condition(subsumee))
  return no implication between predicates;
if (has_unmatch_and_condition(subsumee))
equivalence = false;
if (has_unmatch_and_condition(subsumee))
  return no implication between predicates;
if (has_unmatch_or_condition(subsumee))
equivalence = false;
remove_unmatch_conditions(subsumer);
remove_unmatch_conditions(subsumee);

// main loop
while (more_than_one_condition_in_predicate(subsumer))
begin
  ser_con = subsumer.get_leaf();
  see_con = subsumer.get_match();
  find_family (ser_con,ser_family);
  find_sibling (see_con,see_family);
  if (ser_con.in_or_relation()) then
    begin
      if(exist a condition that is in see_family and
         the condition.get_match() isn’t in ser_family)) then
        return no implication between predicates
      if(exist a condition that is in ser_family and
         the condition.get_match() isn’t in see_family) then
        equivalence = false
    end
  else //if the relation is and
  begin
    if(exist a condition that is in ser_family and
       the condition.get_match() isn’t in see_family) then
      return no implication between predicates
    if(exist a condition that is in see_family and
       the condition.get_match() isn’t in ser_family)
      equivalence = false;
  end
  replace_family_in_predicate_by_condition
  (ser_family,subsumer, ser_con);
  replace_family_in_predicate_by_condition
  (see_family,subsumee, see_con)
end //end loop
if (equivalence == true)
  return equivalent predicates
else
  return subsumer implies subsumee

Fig. 23. XPath Matching Algorithm.