Querying XML Documents by Dynamic Shredding

Hui Zhang
Department of Computer Science
University of Manitoba
Winnipeg, MB, Canada, R3T 2N2
+1-204-474-8625
hzhang@cs.umanitoba.ca

Frank Wm. Tompa
School of Computer Science
University of Waterloo
Waterloo, ON, Canada, N2L 3G1
+1-519-888-4567, x4675
fwtompa@db.uwaterloo.ca

ABSTRACT

With the wide adoption of XML as a standard data representation and exchange format, querying XML documents becomes increasingly important. However, relational database systems constitute a much more mature technology than what is available for native storage of XML. To bridge the gap, one way to manage XML data is to use a commercial relational database system. In this approach, users typically first "shred" their documents by isolating what they predict to be meaningful fragments, then store the individual fragments according to some relational schema, and later translate each XML query (e.g., expressed in W3C's XQuery) to SQL queries expressed against the shredded documents.

In this paper, we propose an alternative approach that builds on relational database technology, but shreds XML documents dynamically. This avoids many of the problems in maintaining document order and reassembling compound data from its fragments. We then present an algorithm to translate a significant subset of XQuery into an extended relational algebra that includes operators defined for the structured text datatype. This algorithm can be used as the basis of a sound translation from XQuery to SQL and the starting point for query optimization, which is required for XML to be supported by relational database technology.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: System—*textual databases, query processing*; I.7.1 [**Document and Text Processing**]: Document and Text Editing—*document management*

General Terms

Algorithms, Languages, Management

Keywords

XML, XQuery, dynamic shredding, relational algebra, text ADT

1. INTRODUCTION

As XML becomes a standard data representation and exchange format, querying XML data draws increasing attention. For this purpose, several XML based query languages have been proposed, including W3C's XQuery [2] which is becoming predominant.

There are many alternative ways to process XML queries. Existing approaches can be classified as native, semistructured, object-oriented, relational and object-relational. In common with other researchers [27, 18, 26, 12], we choose to build on the mature foundations of relational technology to provide XML support. The advantages are that most of the world's business data are stored in relational or object-relational format, relational systems are capable of storing and processing large volumes of data, they have been developed as the best possible general-purpose query processors, and vendors are continuously improving their products.

Relational approaches proposed to date are predicated on some pre-defined mappings between XML documents and relational tables. These mapping schemas are sometimes tailored to specific documents or document collections by using available DTDs (Document Type Definitions) [27], identifying structure patterns in XML documents [13], or utilizing XML data statistics and query workloads in a cost-based manner [7, 28]. Alternatively, mappings can be defined generically to reflect the components of an arbitrary XML document [18, 23].

However, the fact that XML documents do not require the existence of DTDs causes the DTD-dependent mapping methods to be inapplicable, and the generic mapping often exhibits poor performance in query processing, especially when "shredded" documents must be recomposed in answer to a query. Other mapping methods may have some advantages for specific workloads, but performance suffers if they are applied to a wide, unknown range of XML environments. Most unfortunately, in all of these approaches, once a mapping is defined, it cannot be changed without a major data reorganization and corresponding application reimplementation. Since adopting a fixed mapping limits the flexibility of an XML database system, and it will be ill-suited for some XML applications, we propose to shred XML data dynamically in response to user queries. Our approach does not require specific mapping schemas between XML documents and relational data. It constructs relational views of XML data on the fly and keeps the original XML text untouched while providing access to various components.

Since XML can be used to represent tabular data as well as documents, it is desirable for a query processor to work effectively with both kinds of data. While exploiting a relational database system as much as possible, we adopt the proposal to enhance SQL to include support for a structured text ADT [8]. By taking full advantage of the powerful query processing capabilities of the RDBMS with the ability to convert dynamically between documents and relations, this approach promises to support a sophisticated XML query engine in the best possible way. Our approach is useful for any XML-enabled DBMS, such as the DB2 XML Extender which stores some significant structural units as BLOB or VARCHAR data. Our research should be of potential benefit to RDBMS vendors, as our approach provides an easy way to integrate text and relational technology to process XML queries.

In this paper, we also present an algorithm to translate from an XQuery canonical form [29] that covers a significant subset of XQuery to an extended relational algebra with support for text[8]. In this algebra, functions on a text datatype, including a tree pattern matching sub-language, provide support for XPath queries [1].

In short, we make the following contributions:

- Describe a dynamic shredding approach to query XML documents;
- Present an extended relational algebra with support for text data type to process XML queries;
- Develop a translation algorithm from an XQuery canonical form to the proposed algebra and prove its correctness.

The rest of this paper is organized as follows: Section 2 reviews related work, and Section 3 describes our dynamic shredding approach. Section 4 describes our algebra, and Section 5 presents a translation algorithm from an XQuery canonical form to our algebra. Section 6 outlines the ongoing implementation work and concludes the paper.

2. RELATED WORK

2.1 Shredding XML documents to relational tables

Many researchers have investigated using relational technology to process XML queries, with a focus on either defining mappings between XML documents and relational tables [13, 18, 27, 9, 16, 26, 21] or translating XML queries on the XML documents into SQL queries on the relational data [23]. There are also industrial efforts to build query engines that support XML queries, such as Microsoft's and Software AG's XQuery implementation prototypes, among others [5]. The mapping problem has been addressed in both directions: shredding XML documents to fit into suitable relational schemes and wrapping relational data with XML views. In this paper, we are concerned with the former only.

The STORED project [13] applies a data mining algorithm to identify regular structures in a set of XML documents without DTDs. Each "stable" pattern found in the data is mapped to one table, and data items that do not belong to any stable pattern are stored in a semistructured storage. Florescu and Kossmann [18] model XML documents as ordered, oriented graphs, and propose several ways of storing edges and values in relational tables. Their method is generic and can be applied to any document, including documents without DTDs. Shanmugasundaram et al. propose three mappings based on simplified graph structures of DTDs [27]. Compared to Florescu and Kossmann's method, this approach results in large database schemas but provides better query performance. By modeling the target application with an XML schema, data statistics, and a query workload, and by using a standard cost-based relational optimizer, Bohannon et al. find an efficient mapping for a target XML application [7]. Similarly driven by cost, Wang et al. propose an adaptive method based on generic algorithms to find optimal mappings for given XML data and query workloads [28].

2.2 Algebras for XML

Several XML algebras have been proposed. An early one [15], which has become the basis of W3C's XQuery Formal Semantics [3], does not seem to be a good choice for efficient implementation purposes. SAL [6] is another algebra proposed for XML documents modeled as a graph, whose basic unit for manipulation is a node. However, there is no formal translation from an XML query to their algebra.

The YATL algebra [10] is an extension to relational algebra which introduces a *Bind* operator to create tuples of variable bindings and a *Tree* operator to construct the XML result according to a given tree structured construction specification. The functionalities of these two operators are quite similar to that of our extraction and construction operators (which will be described in the next sections). The major differences to our approach are that the *Bind* operator does not preserve context when forming variable bindings and the *Tree* operator is quite complex which makes it difficult to optimize at the algebra rewriting level.

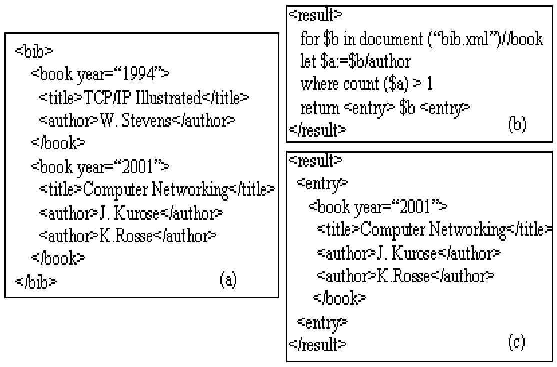
TAX [20] is a tree algebra whose operators manipulate collections of ordered labeled trees instead of relations. Based on this algebra, the authors propose a translation algorithm which is different from ours. Whereas the authors are working on the efficient implementation of their algebra, our algebra is more easily integrated into existing RDBMSs.

Fiebig and Moerkotte [17] have studied XML construction and optimization in Natix [22] from the algebraic point of view. However, the language under consideration is YATL [10] rather than XQuery, and their work is focused on physical rather than logical algebra. In contrast, our work aims at a general algebraic framework for XML query processing and optimization, which includes both the query part and the construction part. However their work is complementary to ours in the sense that their construction plans can be an underlying physical implementation of our logical construction operators.

Finally, the algebra used in the Rainbow system, called XAT [14], is similar to the one we have used in our own research. In the XAT algebra, each operator takes one or more XAT tables as input and produces an XAT table as output. An XAT table is an order-preserving table of tuples. Each tuple is a sequence of cells, and each cell stores an XML node or a sequence of nodes. Each column in an XAT table is either a variable binding from a given XQuery or a variable generated internally. The Rainbow project has used this algebra as the basis of its investigation into XQuery support for views and for streams, among other applications.

3. DYNAMICALLY SHREDDING XML

We begin by defining how we address the problem of storing XML data in relations. Unlike other mapping ap-

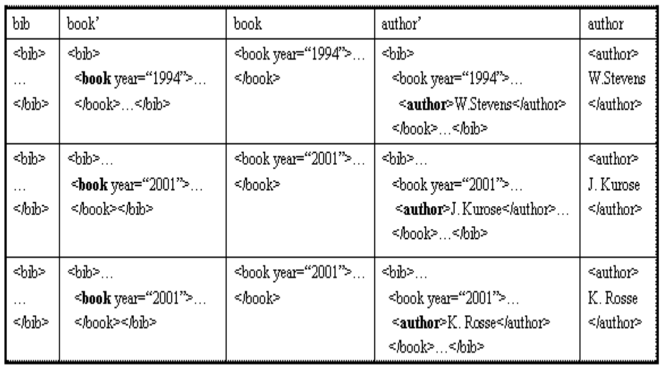


```
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author>W. Stevens</author>
  </book>
  <book year="2001">
    <title>Computer Networking</title>
    <author>J. Kurose</author>
    <author>K.Rosse</author>
  </book>
</bib>                                (a)
```

```
<result>
  for $b in document ("bib.xml")//book
  let $a:=$b/author
  where count ($a) > 1
  return <entry> $b <entry>
</result>                             (b)
```

```
<result>
 <entry>
   <book year="2001">
     <title>Computer Networking</title>
     <author>J. Kurose</author>
     <author>K.Rosse</author>
    </book>
 <entry>
</result>                             (c)
```

Figure 1: (a) an XML bib text (b) an XQuery query posted on bib text and (c) the generated XML result



bib	book'	book	author'	author
<bib> … </bib>	<bib> <**book** year="1994">… </book>…</bib>	<book year="1994">… </book>	<bib> <book year="1994">… <**author**>W.Stevens</author> </book>…</bib>	<author> W.Stevens </author>
<bib> … </bib>	<bib>… <**book** year="2001">… </book></bib>	<book year="2001">… </book>	<bib>… <book year="2001">… <**author**>J. Kurose</author>… </book>…</bib>	<author> J. Kurose </author>
<bib> … </bib>	<bib>… <**book** year="2001">… </book></bib>	<book year="2001">… </book>	<bib>… <book year="2001">… <**author**>K. Rosse</author> </book>…</bib>	<author> K. Rosse </author>

Figure 2: Results of extraction with a tree pattern that flags book and author nodes

proaches, we can store whole documents, or arbitrarily large fragments if preferred by a data administrator, in a column of type *text*. Most importantly, we do not require that documents be chopped into atomic pieces to be squeezed into relational tables and columns. Thus we avoid having to determine at database design time what are the smallest units of a document that might be needed by an application, and we avoid having to reassemble meaningful aggregates from the atomic pieces at run time in response to users' queries.

To enable dynamic shredding of XML data, we define an *extraction* operator, $\chi_{A,S}(R)$, adapted from the function *extract_subtexts*() designed for a text-relational abstract data type (ADT) [8]. This operator takes a table R as input and two parameters, A and S, where A is a column of table R of type *text* and S is a tree pattern to match against each text entry in the given column A. The pattern matching language is a variant of XPath that describes tree patterns instead of path patterns, using hash marks or some similar flags to indicate which nodes are to be returned. Therefore, it differs from an XPath expression by identifying *several* nodes in a tree that correspond to a single match rather than extracting only the last node in some path.

The result of $\chi_{A,S}(R)$ is computed by considering each row of R in turn, as illustrated in Figure 2 for the pattern corresponding to `//book#/author#` to extract book-author pairs simultaneously. Each application of a tree pattern (having k flagged node labels) against a text tree produces a $2k$-column table with one row for every distinct tuple of bindings and one column for each of the extracted subtexts that matches a flagged node label plus a second column indicating *where* in the input text this subtext come from, referred to as the *mark column*. Thus given a table R with n rows and a column A of type *text*, an application of extraction χ to A produces n $2k$-ary tables, one per row in R. The resulting tables are then "attached" to R as if by a join that correlates each row in R with all rows produced from the A-value in that row. Hence the result of applying this operator is an expanded and unnested table. The new column names by default are the same as the root names of the extracted texts, with suitable renaming as necessary.

Figure 2 symbolically shows the extraction of 'book' and 'author' from a table with one row and one column containing a bib text shown in Figure 1a. Because the original text contains three possible bindings for book-author pairs, the resulting relation has three rows, each of which is joined to the original bib value. The first column contains the original bib text (typically represented by reference rather than by value, for efficiency), the third and the fifth columns illustrate the extracted book text and author text, while the second and the fourth columns are the associated mark columns (with marks represented here in bold) for book and author, respectively. These marks can be used to simulate "node identity" to manage document order and subtext equivalence, and need not be visible to users, i.e., they are "hidden" columns.

A conventional implementation of the marks produced by the extraction operator is to use document references (e.g., *doc id* + *position* in a document). An alternative is to use FlexKeys as designed for supporting XAT [14]. With any such implementation, the extracted text *value* need not be stored explicitly in its text column either, but instead it can be implemented by storing its length or a reference to its endpoint in the stored text: the mark column value together with this value describe the start and end of the text region from which the subtext itself can be obtained. Thus, extraction materializes relational views of application-selected portions of XML data on the fly, and keeps the original XML text untouched.

4. EXTENDED RELATIONAL ALGEBRA

Our algebra is an extended relational algebra based on SQL tables rather than relations [24] and with support for text functions. Thus, similarly to XAT tables, collections of rows are ordered and permit duplicates, but data values may include structured text as well as integer, string, date, etc. Structured text data is treated as an abstract data type, in which an "element node" together with its attributes and descendants in the XQuery 1.0 and XPath 2.0 Data Model [4] is represented conceptually as an ordered tree [8]. The structured text ADT supports two functions to convert from strings (usually VARCHARs or CLOBs) to text and from text to strings, six functions are used for manipulating marks in texts, two functions query the existence of or the number of marks in a text, two functions extract subtexts from a text, and the last five functions manipulate the DTD corresponding to an XML text. Furthermore, selection and join conditions may include text-related conditions, and the projection list may include text functions as well. When needed, these traditional relational operators deal with texts as if the data were converted to canonical strings first. For example, if we wish to test whether two texts t_1 and t_2 have equal value, we evaluate whether $text\_to\_string(t_1)$ is equal to $text\_to\_string(t_2)$.

An important concept in the text ADT is to preserve the context of selected text as well as extracting the subtext. These contexts are useful in our evaluation of XQuery expressions since they indicate where these extracted subtexts come from, which in turn provide a possible mechanism to simulate 'node identity'. Hence, translation issues related to keeping document order and node identity can be solved by utilizing these contexts. It is observed that these contexts can be 'virtual' so that users or applications do not see them, but the system can manipulate them for the purpose of bookkeeping or enforcing correctness. Therefore, we choose to distinguish between *visible* and *hidden* columns. Visible columns are those available to users or applications, whereas hidden columns are used solely by the DBMS. Similar consideration has appeared elsewhere [11, 25], where a typical virtual attribute is a tuple identifier which is used to represent each tuple uniquely.

Since traditional operators are well understood, we omit them here and focus on the non-standard operators only, as shown in Table 1.

Operator	Definition
$\chi_{A,S}(R)$	Extract components matching pattern S from column A in table R
$\gamma_A(R)$	Partition table R on grouping columns A
$\tau_A(R)$	Sort table R based on sorting columns A
$\mu_A(R)$	Aggregate construction on column A of table R
$\nu_{A^{C_1},A^{C_2},tag}(R)$	Element construction on columns A^{C_1}, A^{C_2} of table R

Table 1: List of operators

4.1 Extraction

Let $S(COLS, PRE\_COLS, OPS, \Phi, F\_COLS)$ be a tree matching pattern, where $COLS$=<$col_1, col_2, \cdots, col_n$> is the vector of items in XPath [1] terminology (renamed if needed for uniqueness); PRE_COLS=<$p_1, p_2, \cdots, p_n$> is the vector of the closest ancestor of each item in $COLS$; OPS=<$op_1, op_2, \cdots, op_n$> is the vector of child or descendent operations on items, i.e., a '/' or a '//' operation; Φ=<$\varphi_1, \varphi_2, \cdots, \varphi_n$> is the vector of text matching constraints on each item (if there is no condition applied on a particular item, an always-true condition is assumed); F_COLS is the vector of items with associated flag #. These vectors are all ordered by the pre-order depth-first traversal on the parse tree for pattern S parsed according to the pattern language. For example, the pattern '`//A#[.//B="s"]/C#`' corresponds to the tree pattern S where $COLS$=<A, B, C>, PRE_COLS=<ϕ, A, A>, OPS=<//, //, />, Φ=<true, equals("s"), true>, and F_COLS=<A, C>.

Let $ex(v_i, c_2, op)$ compute a table with three columns. Each row of the returned table contains the value v_i in the first column and a found item match of c_2 extracted from v_i (with op '/' or '//') in the third column, along with its corresponding hidden mark $\hat{c_2}$ in the second column. In the case that there is no match found at all, $ex(v_i, c_2, op)$ returns a single row with value (v_i, *null*, *null*). Define $ex(c_1, c_2, op) = \bigcup_{v_i \in \, all\ values\ of\ c_1} ex(v_i, c_2, op)$.

Define $pre(col, S)$ to be the item that is the closest ancestor item of the node in the pattern S that will be extracted to column col.

Now our extraction operator(χ) with a pattern S on column A of table R can be formally defined as:

$$\begin{aligned} &\chi_{A,S(COLS,PRE\_COLS,OPS,\Phi,F\_COLS)}(R) \\ &= R \bowtie_A (\pi_{F\_COLS \cup F\_COLS' \cup \{A\}}(\varphi_1(ex(A, col_1, op_1)) \\ &\qquad \bowtie_{pre(col_2,S)} (\varphi_2(ex(pre(col_2,S), col_2, op_2)) \\ &\qquad \cdots\cdots \\ &\qquad \bowtie_{pre(col_n,S)} (\varphi_n(ex(pre(col_n,S), col_n, op_n))) \\ &\quad) \end{aligned}$$

where F_COLS' is the vector of hidden columns corresponding to F_COLS.

4.2 Sorting

A sorting operator τ is used to sort a table according to some sorting criteria. $\tau_A(R)$ takes a table R as input and a list of sorting columns A as a parameter. The result of this operation is the table R, but with the rows sorted in the order indicated by A. The parameter may indicate the sorting is to be done in ascending or descending order, or based on document order rather than values of each column in A. That is, the actual sorting may be performed on respective mark columns associated with each column in A, which are defined to reflect document order.

4.3 Groupby

In order to facilitate eventual optimization involving grouping operations and aggregate constructions, we separate the grouping operation from the computation of aggregate functions. We adopt the *partition* operator γ from Paulley [24] as our groupby operator. For a grouping column *col* of simple type, γ partitions a table such that each row in a partition has the same value for *col*; if the grouping column is of type text, it partitions the table based on the value of the (hidden) node identifier of *col*. Thus the groupby operator can be used to partition a table based on simple values or instead on node identity when a text column is specified.

4.4 Construction operators

To support the conversion of (parts of) relational tables into documents, we include two construction operators:

- *Aggregate constructor* (μ_A) is used mainly for representing the contents of a set-valued column for several tuples as a single tree. Assuming that a groupby operation is performed first, instead of computing an aggregated scalar value for each group, aggregate construction forms a tree from the values over column A in each group, appropriately handling *null* values.

  We adopt the *catenate* operator originally defined in [19] to manipulate vectors. A *vector* is an ordered collection of trees organized in a larger tree structure, with the root labeled *vector* and the roots of the trees in the collection being the children. The catenate operator takes two vectors and returns a single vector including all subtrees of the arguments. Hence, the root of the generated tree of μ_A is labeled with "vector". This root is a "virtual" root in the sense that when one vector V_1 is concatenated with a second tree to form a new tree T, the root of V_1 is discarded and all its children become the children of T directly. In the presence of *null*, catenating a vector T with *null* returns T itself, and catenating two *null* trees returns *null*. Therefore, if the group being aggregated is the empty set, then the aggregate result is *null*. Again by default, the column name in the resulting table corresponding to the aggregated value of applying μ is the same as before, i.e., A.

To formally define this construction operator, let *T(tag, attributes, subelements)* denote a structured text. When we aggregate several text values $e_1, e_2, \cdots, e_n$ from one column of a relational table, we form a new text value having $e_1, e_2, \cdots, e_n$ as subelements, having no attributes, and we represent the tag by the special symbol *vector*. Thus a text value generated by an aggregate construction has the form $T(vector, null, <e_1, e_2, \cdots, e_n>)$. (We restrict structured texts such that whenever the tag is *vector*, the corresponding attributes are by definition *null*.)

Now we define our *catenate* ($\circ$) operator as follows: If $T_1 = T(t_1, a_1, <e_1, e_2, \cdots, e_m>)$ and $T_2 = T(t_2, a_2, <f_1, f_2, \cdots, f_n>)$ are two text values and

$$E = \begin{cases} <e_1, e_2, \cdots, e_m, f_1, f_2, \cdots, f_n> \\ \qquad \text{if } t_1 = vector \text{ and } t_2 = vector \\ <e_1, e_2, \cdots, e_m, T_2> \\ \qquad \text{if } t_1 = vector \text{ and } t_2 \neq vector \\ <T_1, f_1, f_2, \cdots, f_n> \\ \qquad \text{if } t_1 \neq vector \text{ and } t_2 = vector \\ <T_1, T_2> \\ \qquad \text{if } t_1 \neq vector \text{ and } t_2 \neq vector \end{cases}$$

then $T_1 \circ T_2 = T(vector, null, E)$.

Consequently, the aggregate constructor μ_A can be formally defined as follows:

Let $v_1, v_2, \cdots, v_n$ be the list of values in a group on column A. Applying the aggregate constructor on this group generates the text value $v_1 \circ v_2 \cdots \circ v_n$, and μ_A is the result of applying an aggregate constructor to each group in the table. Note that the order of the subelements of the result is defined to match the order of the rows in the table.

- *Element constructor* (ν) is another construction operator. It takes three parameters, A^{C_1}, A^{C_2}, and *tag*, where (1) A^{C_1} is a list of columns that are to become the XML attributes of the resulting element being constructed, here denoted *elet*. The order that columns occur in the list A^{C_1} does not matter. (2) A^{C_2} is a list of columns that are to become the subelements of *elet*. The order that columns occur in the A^{C_2} list is also the order that these subelements occur in the resulting element. (3) *tag* is the tag name of *elet*. By default, the column corresponding to *elet* in the resulting table is named *tag*, but if *tag* conflicts with an existing column name, some renaming is necessary.

  Element constructor is applied to each tuple and the result is computed as: concatenate the value T_i of each column i appearing in A^{C_2} to construct a tree T with all T_i as children. Set the tag of the result to *tag* and the (XML) attributes of the result to the named set of values in A^{C_1}, where names correspond to the (table) attribute names in A^{C_1}. Since the order of each column appearing in A^{C_2} is important, we apply the *catenate* operator in a way such that each child is in the same order as it is in A^{C_2}.

  This can be formally defined as:

  For a single row in a table, let c_{1j} be the value of column C_{1j} for each C_{1j} in A^{C_1}; and let $c_{21}, c_{22}, \cdots, c_{2m}$ be the values of columns in A^{C_2}. Applying the element constructor to this row produces the text value $T(tag, \{C_{1j} = c_{1j} \mid C_{1j} \in A^{C_1}\}, c_{21} \circ c_{22} \circ \cdots \circ c_{2m})$. Finally, $\nu(tag, A^{C_1}, A^{C_2})$ is the result of applying such element constructor to each row of the table.

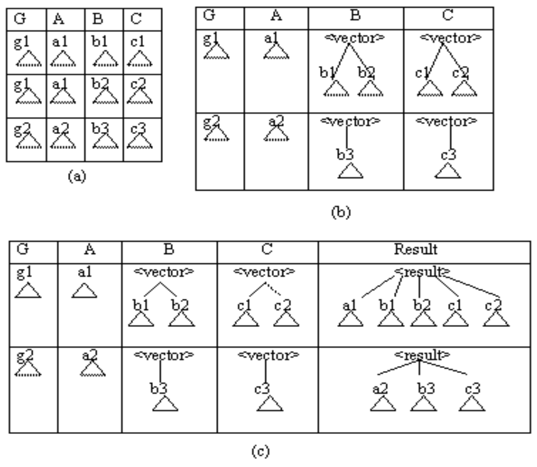


Figure 3: An example of aggregate constructor and element constructor

Hence, while *aggregate constructor* is a vertical concatenation, *element constructor* is a horizontal concatenation.

Figure 3 shows the application of aggregate constructors on each of both columns B and C of the table resulting from grouping on both columns G and A (shown in Figure 3a-b), followed by the application of an element constructor on columns A, B, and C together (shown in Figure 3c). We require that tuples are in the desired order before applying the aggregate constructor.

5. TRANSLATING XQUERY

Given an XQuery Q, our XML query processing framework first canonicalizes it to a query *Q'*, then translates from *Q'* to an extended relational algebra with support for text, using the translation algorithm described below. After obtaining an initial query plan from the translation, we apply query rewritings (see [30]) to optimize it to get a better plan, which is then sent to the underlying extended database management system to be executed. Figure 4 shows the steps of our XML query processing framework.

XQuery canonical form (defined in [29]) restricts the allowable syntactic forms. For example, path expressions can only appear in the for/let clauses; items returned in the return clause must be variables defined before in the for/let clauses, or aggregate functions applied on those variables, or expressions in nested canonical form; only boolean condi-

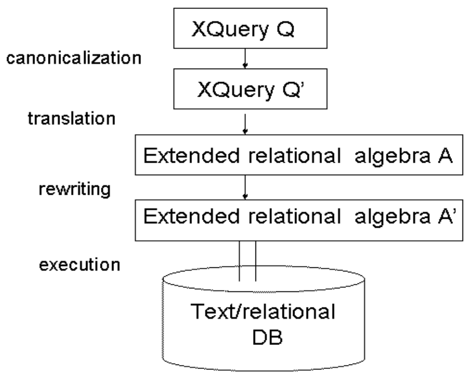
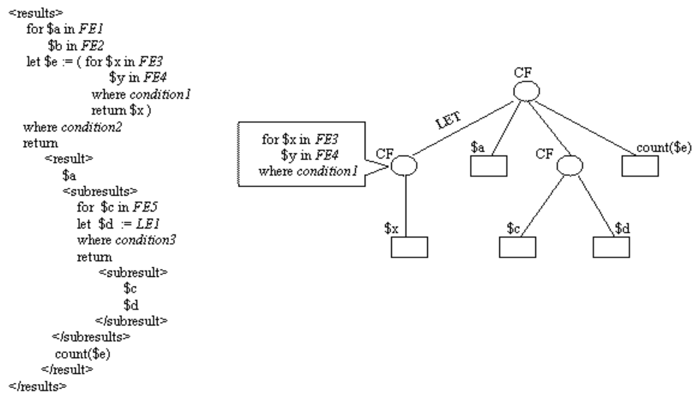


Figure 4: Our XML query processing framework



```
<results>
   for $a in FE1
       $b in FE2
   let $e := ( for $x in FE3
                   $y in FE4
                where condition1
                return $x )
   where condition2
   return
       <result>
          $a
          <subresults>
             for $c in FE5
             let $d := LE1
             where condition3
             return
                <subresult>
                   $c
                   $d
                </subresult>
          </subresults>
          count($e)
       </result>
</results>
```

Figure 5: (a) An example of XQuery canonical form (b) its corresponding query tree

tions can appear in the where clause; and the nested canonical form can only appear in the let clause or return clause. The canonical form also provides a conceptually uniform vision of path expressions, element constructors and FLWOR expressions in XQuery. Therefore it provides a simple way to understand these important features of XQuery. Most FLWOR expressions can be translated to this canonical form through a set of transformation rules [29].

Given an XQuery Q in the canonical form, we represent it as a *query tree* (whose definition will be given shortly), then we translate the query tree to a relational algebraic expression tree. We show that the execution of the resulting algebraic expression tree will produce the same correct query result as the one that would be generated by processing Q directly. Elsewhere we describe possible query rewritings and optimizations that can be subsequently applied [30].

Executing an XQuery FLWOR expression requires the construction of a document fragment that includes the components specified by the return clause. The values for each component are determined by the bindings imposed by the for, let, and where clauses of the expression. Thus we choose to represent a FLWOR expression by a *query tree*, in which the root is annotated by the for, let, and where clauses and has subtrees representing each variable or constant whose value will be returned. Because the canonical form restricts the placement of nested FLWOR expressions to appear in let clauses and return clauses only, a query tree can include query subtrees descending from the root or from any component being returned. In essence, a query tree is a straightforward re-encoding of a canonical XQuery expression, with all the information needed to reconstruct the original canonical query.

Definition 5.1 (Query Tree). A *query tree* has four kinds of nodes: every internal node is called a `CF node` and represents a FLWOR expression or a compound return value; every leaf node represents a simple return value and is either a `V node` to denote a for-variable, `U node` for a let-variable, or `aggU node` for an aggregate function applied to a let-variable. The subtrees of each CF node represent either nested FLWOR expressions that are bound to let-variables or the components of the return clause, in the order of their appearance in the XQuery expression. In the former case, the incoming edge is labeled with 'let' and this CF node is referred to as a `let-CF node`.

Each CF node is annotated with the rest of the query information expressed in the corresponding FLWOR expression: the mapping of for-variables and basic let-variables (i.e., those without nested FLWOR expressions) to path expressions, the contents of the where clause, the *result-tag* (the one surrounding the FLWOR expression, possibly absent), and *return-tag* (the one immediately following the return keyword, possibly absent). Each leaf node is annotated with its respective tag in the return clause.

Figure 5b shows the query tree for the canonical query in Figure 5a, where leaf nodes are represented with rectangles and inner nodes are represented with circles. To avoid cluttering the diagram, only the associated query information for the let-CF node is shown.

We now show how to translate an XQuery in canonical form into a corresponding relational algebra expression. For convenience, we assume there is a two-column table R^0 (which could be a view), with one column for document name and one for document text, serving as a directory to support XQuery's `document()` function. Furthermore, in the explanations below, we describe the operations of the translator *as if it were an interpreter for a relational algebra engine*. Thus, throughout this explanation we say that the translation performs some computation as a shorthand for the translation "produces an operator subtree, which when executed will result in" that computation. By viewing the translation process as interpretive instead of compilatory, we can discuss tables that will not actually be formed until query execution time.

We start by considering a query in *basic XQuery canonical form*, that is, with no nested FLWOR expressions.

Translation 5.1 (`Trans0`: Basic CF Translation). Let Q be a query in basic XQuery canonical form, and QT be the query tree for Q. Note that QT has only one CF node (i.e., root) with some leaf children as returned values. `Trans0` proceeds as follows by considering query information and the returned values associated with the CF node:

1. Translate for clause: extract binding values for each for clause $\$v_i := FE_i$ using the extraction operator χ. FE_i is a path expression beginning with either `document()` or a reference to another variable, which determines the source for the extraction. The rest of

the path expression is translated to a pattern matching string, and supplied as the second parameter to χ.

1.1. If FE_i starts with the `document()` function `Trans0` selects the corresponding row and column of the initial table R^0 to form a new one-row, one-column table R^1 containing the document text *doc*, then extracts $\$v_i$ from R^1 using the tree pattern corresponding to FE_i. When executed, this produces a new column together with a hidden mark column on R^1.

1.2. If FE_i starts with another variable, `Trans0` must have previously extracted a corresponding column in some table. In this case, extraction starts from that column and forms a new column together with a hidden mark column for that existing table.

Whenever `distinct` is present, duplicate elimination is performed based on value or node identity as desired, depending on the specification of `distinct`.

After this step, each variable corresponds to a column from which the variable takes its binding values, and a hidden mark column indicating where these binding values originate. Since these mark columns are used only by the underlying DBMS, we use the term 'column' to mean 'visible column' unless specified explicitly.

2. Translate let clause: similar to step 1 except that after each extraction, perform a sorting operation on all the remaining columns (except let-columns) based on document order, then perform a partition on all columns except the newly extracted one, followed by an aggregate constructor on the newly extracted column. Thus each let-variable's value is a text tree representing a collection as a vector. To ensure compliance with XQuery semantics, the order of elements in the grouping column list and the sorting column list is the same as in the query.

3. Form a single table: compute the cross product of multiple tables, if any, obtained from the previous steps. Let the resulting table be $R(a_1, a_2, ..., a_n)$.

4. Translate where clause:

4.1. Rewrite the where condition such that variable appearances are replaced by their corresponding column names in R. Denote the rewritten condition as WC.

4.2. Include a selection operator σ with condition WC.

5. Translate return clause:

5.1. Project on columns corresponding to for-variables, let-variables appearing in the return clause, plus those aggregate functions applied on let-columns. If a returned variable has a tag around it, then the projection list includes an element constructor applied to that column.

5.2. Sort the table according to document order or as the query requires. In the case of sorting on document order, the sorting column list is the for-variable list with each for-variable in the order of its appearance in the for clause, and the sorting is performed on the hidden mark column associated with each for-variable. (This corresponds to W3C's specification for ordering.)

5.3. Apply an element constructor using the columns of the previous step. Its parameters are supplied as indicated by the return clause, with the columns in the second parameter (i.e., subelements list) in the order of their corresponding variable appearances in the return clause and the third parameter as *return-tag* if present. Let the resulting column be named a_t.

6. Generate the result: Project on column a_t and apply an aggregate construction operator on a_t treating all rows as a single group, followed by an element constructor adding the tag *result-tag* or *vector* on the aggregated value of a_t. Hence, the result of this step is a one-row, one-column table containing a constructed text tree T.

Note that SQL cannot directly evaluate $agg(x)$ when x is a let-variable binding, since `Trans0` generates a tree for x to encapsulate a group of values. Thus $agg(x)$ can be evaluated by first decomposing this tree, then applying an appropriate partition operation so that for each original tuple before decomposition, there is a group corresponding to it; and then applying *agg* on each group in the usual way. Optimization applied to the resulting query plan can eliminate unnecessary tree creations and decompositions [30].

Given the query in Figure 1b posed on the bib text in Figure 1a, $Trans0$ generates the algebraic expression tree shown on the left side of Figure 6, with the corresponding table generated shown on the right side of Figure 6. To save space, Figure 6 omits the hidden mark columns in the generated tables, and depicts the aggregated values (e.g., let variable $\$a$) as vectors rather than trees. Note that *book'* and *author'* in the algebraic tree refer to the corresponding hidden mark columns, i.e., sorting is based on document order and partition is performed on node id.

LEMMA 5.1. Step 4 in Translation 5.1 keeps a row in table R if and only if the bindings for this row match the selection criteria according to XQuery semantics.

THEOREM 5.1. For any basic XQuery canonical query Q, `Trans0` generates a correct equivalent relational query.

Proof: Steps 1 through 4 ensure that the correct data will be stored in the table, and steps 5 and 6 ensure that the appropriate result is assembled. The complete proof can be found elsewhere [29].

Now we extend the translation of basic XQuery expressions to expressions with or without nesting in return and let clauses. Let the table R in the translation of a basic query be denoted as the *working table*. In the extended translation for general queries, the working table is global to all nested FLWOR expressions.

TRANSLATION 5.2 (`Trans`: XQUERY CF TRANSLATION). Let Q be an XQuery in canonical form, QT be the query tree for Q. For each CF node in the query tree QT, visited in depth-first order, `Trans` applies an extension of the basic translation `Trans0` with the following modifications:

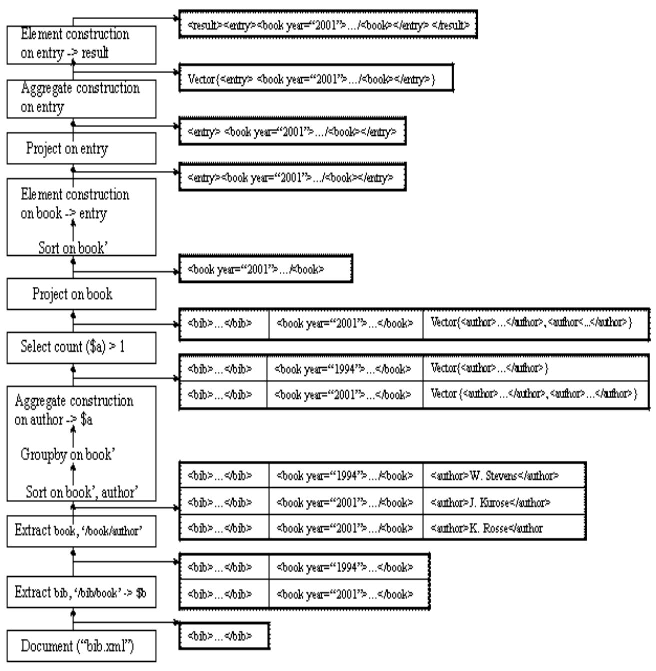

Element construction on entry -> result
Aggregate construction on entry
Project on entry
Element construction on book -> entry
Sort on book'
Project on book
Select count ($a) > 1
Aggregate construction on author -> $a
Groupby on book'
Sort on book', author'
Extract book, '/book/author'
Extract bib, '/bib/book' -> $b
Document ("bib.xml")
<result><entry><book year="2001">.../<book></entry> </result>
Vector{<entry> <book year="2001">.../<book></entry>}
<entry> <book year="2001">.../<book></entry>
<book year="2001">...</book>
<bib>...</bib>
Vector{<author>...</author>, <author<...</author>}
<book year="1994">...</book>
<author>W. Stevens</author>
<author>J. Kurose</author>
<author>K. Rosse</author>

Figure 6: The algebraic expression tree for the XQuery in Figure 1b

- Step 0. Save the current working table R as S.
- Step 2′. Translate let clause. If the let clause is a simple variable binding without nested CF, its translation is the same as that in basic translation. Otherwise, call `Trans` on the nested CF node to create a column containing the sets of values to which the let variable is bound.
- Step 5′. Denote the extracted for- and let-columns for this current CF node as CC, all previously extracted for- and let-columns for nested subqueries as RC, and all previously aggregated columns (including both scalar aggregated columns and aggregated construction columns) for nested subqueries as AC.

5.0a. If there are nested CF nodes in the return clause, call `Trans` on the nested CF nodes.

5.0b. If processing a nested CF node, replace R by the left outer join of S with R. This step is to ensure that empty substructures will be generated, if needed, precisely where they are required by XQuery semantics.

5.1′. Similar to Step 5.1, but with the projection list enlarged to include all columns in RC ∪ AC along with those required in the construction of this CF node.

5.2′. Sort the table as specified by the query, or (if no sorting is specified) sort the table with the sorting column list including all columns in RC ∪ CC (except all let-columns) with each for-variable in the order of its appearance in Q. As before, sorting is performed on the hidden mark column associated with each for-variable to ensure correct document order.

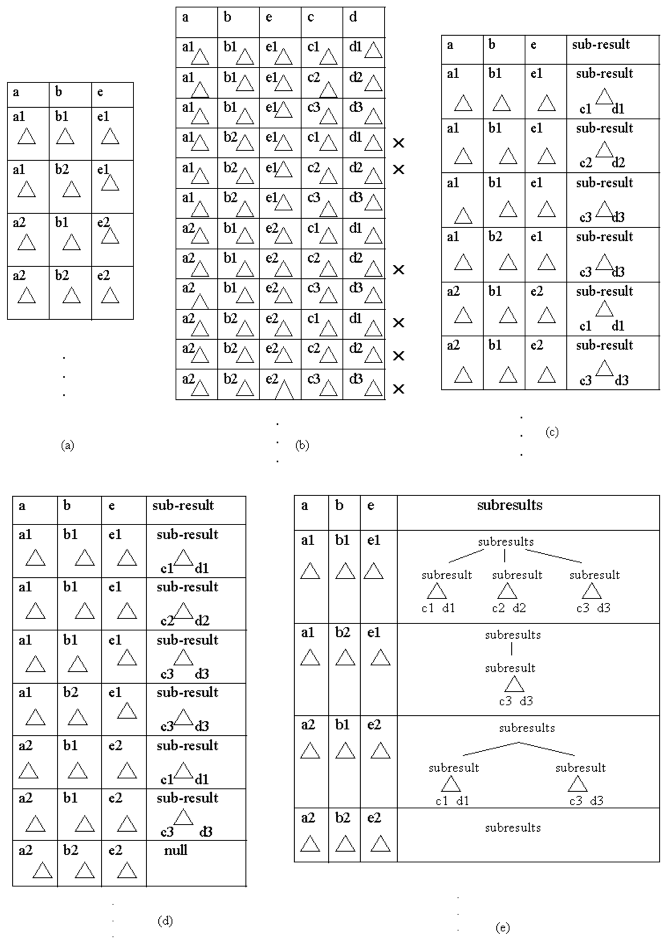


Figure 7: Translating the nested CF node of query in Figure 5

5.3′. Same as step 5.3.

- Step 6′. Partition on all columns in RC ∪ AC, construct the result of this CF node in the same way as Step 6, and put the result into a new column in table R.

To demonstrate how the code produced by `Trans` operates, consider the nested CF node in the query in Figure 5. Because of the bindings for variables $\$a$, $\$b$, and $\$e$ from the root, we start with a table such as shown in Figure 7a. This is table S described in Step 0 for `Trans`. Executing the translation of the nested for and let clauses results in a table such as shown in Figure 7b, from which the translated where clause will remove several rows as indicated. We assume that the application of the where condition eliminates all the tuples with $(\$a, \$b, \$e) = (a_2, b_2, e_2)$. The nested return clause corresponds to code that aggregates *subresult* as in Figure 7c. The left outer join of S and R generates the table shown in Figure 7d (including a null sub-result for the empty match), and finally the code that aggregates *subresults* causes the element construction shown in Figure 7e.

Again we can prove that `Trans` preserves the semantics of Q [29].

Lemma 5.2. As `Trans` is invoked on any CF node n in the query tree QT, there is a bijection from the columns of working table R to the union of (1) the variables bound in all ancestors of n; and (2) previously constructed return nodes, including each aggU node and sibling located to the left of n and to the left of each of its ancestor nodes. After `Trans` is invoked on n, there is one more column, corresponding to the constructed result for n.

Theorem 5.2. For any CF node *cf* in the query tree QT, and each valid binding for variables that are defined before *cf*, the invocation of `Trans` on *cf* generates one and only one semantically correct result corresponding to *cf* for that binding.

Theorem 5.3. `Trans` generates the semantically correct result for an expression Q in XQuery canonical form.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we propose to query XML documents by dynamic shredding. This approach does not require any specific mapping between XML documents and relational tables. It converts subtexts to relational fields as needed dynamically in response to user queries, and keeps the original XML text untouched. Hence, the problem of reconstructing XML data from relational tables, which is tedious and inefficient for alternative approaches, does not exist. Meanwhile, there is no undesired duplication of XML data needed to keep the document as a whole while providing access to various components. The approach is applicable for managing any stored XML documents.

In addition, we define an extended relational algebra to operate with structured text. We also present an algorithm to translate from an XQuery canonical form into that algebra, and we prove its correctness. This algorithm can be used as the basis of a sound translation from XQuery to SQL, and the starting point for query optimization, which is required for XML to be supported by relational database technology. The algebra and its translation provide a simple, but powerful, mechanism to support XQuery processing on a commercial relational database management system.

Currently we are building a prototype XML query processor and optimizer based on this study, on top of the Text and Relational Database Management System (T/RDBMS) developed between 1994 and 1997 at the University of Waterloo. On one hand, relational queries involving join, grouping, aggregation, set operations, and sorting are supported directly in SQL; on the other hand, full-text search queries or XPath-like queries are naturally supported by using text ADT operators and the tree pattern matching sub-language. Moreover, since T/RDBMS seamlessly integrates a text ADT with SQL, queries involving both traditional relations and structured text manipulation can be easily accommodated, and preliminary experience has shown that the text manipulation described in this paper can be performed efficiently (see http://db.uwaterloo.ca/trdbms/tpcd/).

We claim that our approach is particularly suitable to implement an XQuery processor for the following reasons:

- Storing whole documents in a column of text rather than chopping them into pieces to be mapped to relational tables and columns preserves the original XML data. (If a document is often updated, it may be desirable to partition the text into several "update units," such as chapters, that can be stored and modified individually, rather than editing and re-indexing the whole. However, the document need not be as highly fragmented as is typical when using static shredding.) Note that RDBMS can not only be used as a place to store the unstructured data, but its power can also be harnessed to manipulate components of that data. However, this approach is only feasible if appropriate structured text operators are supported.
- Path expressions in an XQuery expression are easily translated into tree pattern matching operators, which provide simultaneous access to related components. We anticipate extensive opportunities for query optimization through appropriate choices of tree patterns that cover multiple XPath expressions spanning one or more XQuery expressions.
- Text ADT operations such as *extract_subtexts()* isolate desired text fragments but also remember the context within which they occur. This provides a means for isolating substructures to be evaluated using the full power of SQL, while retaining their origins so that the results of the evaluations can be carried back into the original contexts.
- Because it is designed to work with SQL-92, T/RDBMS support is readily available to be included in an XQuery processor built on any of today's commercial or academic systems. If the underlying engine is upgraded to SQL-99 or beyond, extra features are immediately also available for XQuery support.

In the future, we plan to conduct experiments on large amount of XML documents to study execution performance.

Acknowledgments

We gratefully acknowledge financial assistance from the University of Waterloo, Bell University Labs, and the Natural Sciences and Engineering Research Council of Canada. The foundations for dynamic shredding arose from our previous work [8].

7. REFERENCES

[1] XML path language (XPath) 2.0. In *http://www.w3.org/TR/xpath20.*

[2] XQuery 1.0: An XML query language. In *http://www.w3.org/TR/xquery.*

[3] XQuery 1.0 and XPath 2.0 formal semantics. In *http://www.w3.org/TR/query-semantics.*

[4] XQuery 1.0 and XPath2.0 data model. In *http://www.w3.org/TR/query-datamodel.*

[5] XQuery implementation. In *http://www.w3.org/XML/Query#implementations.*

[6] C. Beeri and Y. Tzaban. SAL: An algebra for semi-structured data and XML. In *Proc. of the 2nd Workshop on the Web and Databases*, pages 37–42, Philadelphia, June 1999.

[7] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML schema to relations: A cost-based approach to XML storage. In *Proc. of the 19th Int. Conf. on Data Engineering*, pages 64–75, San Jose, Feb. 2002.

[8] L. J. Brown, M. P. Consens, I. J. Davis, C. R. Palmer, and F. W. Tompa. A structured text ADT for object-relational databases. *Theory and Practice of Object Systems (TAPOS)*, 4(4):227–244, 1998.

[9] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmungasundaram, D. Shekita, and S. Subramanian. XPERANTO: Publishing object-relational data as XML. In *Proc. of the 3rd Workshop on the Web and Databases*, Dallas, Texas, May 2000.

[10] V. Christophides, S. Cluet, and J. Simeon. On wrapping query languages and efficient XML integration. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 141–152, Dallas, Texas, May 2000.

[11] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proc. of the 13th Int. Conf. on Very Large Data Bases*, pages 197–208, Brighton, England, Sept. 1987.

[12] D. DeHaan, D. Toman, M. P. Consens, and T. Özsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, San Diego, CA, June 2003.

[13] A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 431–442, Philadelphia, June 1999.

[14] M. EI-Sayed, K. Dimitrova, and E. A. Rundensteiner. Efficiently supporting order in XML query processing. In *Proc. of the fifth ACM Workshop on Web Information and Data Management (WIDM)*, pages 147–154, New orleans, Louisiana, Nov. 2003.

[15] M. Fernandez, J. Simeon, and P. Wadler. An algebra for XML query. In *Proc. of the 12th Conf. on the Foundations of Software Technology and Theoretical Computer Science*, Delhi, Dec. 2000.

[16] M. Fernandez, W. C. Tan, and D. Suciu. SilkRoute: Trading between relational and XML. In *Proc. of the 9th Int. World Wide Web Conf.*, Amsterdam, Netherlands, May 2000.

[17] T. Fiebig and G. Moerkotte. Algebraic XML construction and its optimization in Natix. *World Wide Web Journal*, 4(3):167–187, 2001.

[18] D. Florescu and D. Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, Sept. 1999.

[19] G. H. Gonnet and F. W. Tompa. Mind your grammar: A new approach to modeling text. In *Proc. of the 13th Int. Conf. on Very Large Data Bases*, pages 339–346, Brighton, England, Sept. 1987.

[20] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In *Proc. of the 8th Int. Workshop on Database Programming Languages*, Rome, Sept. 2001.

[21] Y. Kadiyska and D. Suciu. Mixed XML/relational data processing. In *Informal Proc. of the workshop on programming language technologies for XML (PLAN-X 2004)*, pages 73–82, Venice, Italy, Jan. 2004.

[22] C. C. Kanne and G. Moerkotte. Efficient storage of XML data. In *Proc. of the 17th Int. Conf. on Data Engineering*, page 198, San Diego, CA, March 2000.

[23] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries on heterogeneous data sources. In *Proc. of the 27th Int. Conf. on Very Large Data Bases*, pages 241–250, Rome, Sept. 2001.

[24] G. N. Paulley. *Exploiting functional dependence in query optimization.* PhD thesis, Dept. of Computer Science, University of Waterloo, Ontario, Canada, April 2000.

[25] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 39–48, San Diego, June 1992.

[26] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *Proc. of the 27th Int. Conf. on Very Large Data Bases*, pages 261–270, Rome, Sept. 2001.

[27] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of the 25th Int. Conf. on Very Large Data Bases*, pages 302–314, Edinburgh, Scotland, 1999.

[28] X. Wang, J. Luan, and Y. Dong. An adaptive and adjustable mapping from XML data to tables in RDB. In *Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web, VLDB 2002 Workshop EEXTT and CAiSE 2002 Workshop DIWeb, S. Bressan and A.B. Chaudhri and M.L. Lee and J.X. Yu and Z. Lacroix (eds.)*, pages 117–130. Springer-Verlag, Berlin Heidelberg, 2003.

[29] H. Zhang. *XML query processing and optimization.* PhD thesis, School of Computer Science, University of Waterloo, Ontario, Canada, March 2003.

[30] H. Zhang and F. W. Tompa. XQuery rewriting at the relational algebra level. In Trends in XML technology for the global information infrastructure, a special issue of. *Int. Journal of Computer Systems, Science, and Engineering*, 18(5):241–262, Sept. 2003.