A General Technique for Querying XML Documents using a Relational Database System

Jayavel Shanmugasundaram, Eugene Shekita, Jerry Kiernan
IBM Almaden Research Center
San Jose, CA 95120
{shanmuga, kiernan, shekita}
@almaden.ibm.com

Rajasekar Krishnamurthy, Efstratios Viglas, Jeffrey Naughton
University of Wisconsin
Madison, WI 53706
{sekar, stratis, naughton}
@cs.wisc.edu

Igor Tatarinov
University of Washington
Seattle, WA 98195
igor@cs.washington.edu

ABSTRACT

There has been recent interest in using relational database systems to store and query XML documents. Each of the techniques proposed in this context works by (a) creating tables for the purpose of storing XML documents (also called *relational schema generation*), (b) storing XML documents by shredding them into rows in the created tables, and (c) converting queries over XML documents into SQL queries over the created tables. Since relational schema generation is a physical database design issue – dependent on factors such as the nature of the data, the query workload and availability of schemas – there have been many techniques proposed for this purpose. Currently, each relational schema generation technique requires its own query processor to efficiently convert queries over XML documents into SQL queries over the created tables. In this paper, we present an efficient technique whereby the *same* query-processor can be used for *all* such relational schema generation techniques. This greatly simplifies the task of relational schema generation by eliminating the need to write a special-purpose query processor for each new solution to the problem. In addition, our proposed technique enables users to query seamlessly across relational data and XML documents. This provides users with unified access to both relational and XML data without them having to deal with separate databases.

1. INTRODUCTION

XML [14] has emerged as the dominant standard for representing and exchanging data over the Internet. Its nested, self-describing structure provides a simple yet flexible means for applications to model and exchange data. For example, a business can easily model complex structures such as purchase orders in XML form. As another example, all of Shakespeare's plays can be marked up and stored as XML documents.

With a large amount of data represented as XML documents, it becomes necessary to store and query these XML documents. To address this problem, there has been work done on building native XML database systems [6][9]. These database systems are built from scratch for the specific purpose of storing and querying XML documents. This approach, however, has two potential disadvantages. Firstly, native XML database systems do not harness the sophisticated storage and query capability already provided by existing relational database systems. Secondly, native XML database systems do not allow users to query seamlessly across XML documents and other data stored in relational database systems.

To overcome the first of the above limitations, there have been techniques proposed for storing and querying XML documents using relational database systems [3][5][11]. These approaches work as follows. The first step is *relational schema generation*, where relational tables are created for the purpose of storing XML documents. The next step is XML document *shredding*, where XML documents are "stored" by shredding them into rows of the tables that were created in the first step. The final step is XML query processing, where XML queries over the "stored" XML documents are converted into SQL queries over the created tables. The SQL query results are then tagged to produce the desired XML result.

The wealth of literature in this field [3][5][10][11] makes it clear that there are many possible approaches for relational schema generation. This is because the appropriate relational schema for a given application depends on many factors such as the nature of the data, the query workload, and availability of XML schemas. Currently, each relational schema generation technique has its own query processor for translating XML queries into SQL queries. This is because there was no known way of providing a general and efficient query capability for all relational schema generation techniques. (As will be discussed in more detail in the section on related work, previous techniques only provided partial solutions to this problem.)

In this paper, we present an efficient technique whereby the *same* query processor can be used with *all* relational schema generation techniques as long as they satisfy the following two conditions. Firstly, they should provide a lossless mapping from XML to relations; i.e., there should be sufficient structural information in the created relational tables to reconstruct the shredded XML documents. Secondly, they should map XML element and attribute names/values to relational column values (so that XML queries can be translated into SQL queries over columns).

All relational schema generation techniques that we are aware of satisfy the above two conditions, and consequently, can use the same query processor. This greatly simplifies the task of relational schema generation by eliminating the need to write a special-purpose query processor for each new solution to the problem. We also show how the query processor used for these relational schema generation techniques can be the same as the query processor used for querying XML views of existing relational data. Therefore, this query processor can also be used to process queries that span XML documents and (XML views of) relational data.

We have implemented the above technique in the context of the XPERANTO system [1][13]. Our implementation is extensible and new relational schema generation techniques can be easily added to the system. We have used this extensibility feature to implement two relational schema generation techniques published in the literature [5][11].

The remainder of this paper is organized as follows. In Section 2, we present our proposed technique, and in Sections 3 and 4, we illustrate this using two different relational schema generation techniques published in the literature. In Section 5, we discuss related work, and in Section 6, we present our conclusions.

2. THE PROPOSED TECHNIQUE

In the proposed technique, whenever an XML document repository is to be created over a relational database system, one of possibly many relational schema generation techniques is used to automatically create relational tables for storing XML documents. Inserted XML documents are then shredded and "stored" as rows in these tables. In addition, a *reconstruction XML view* is created over the created relational tables, which (virtually) reconstructs the "stored" XML documents from the shredded rows. The reconstruction XML view is specified just like a regular XML view of relational data [1][2][4]. Queries over the stored XML documents are then treated as queries over the reconstruction XML view. This is shown in Figure 1.

The key observation here is that a reconstruction XML view makes it possible to treat XML documents as though they are an XML view of relational data. As a result, a query

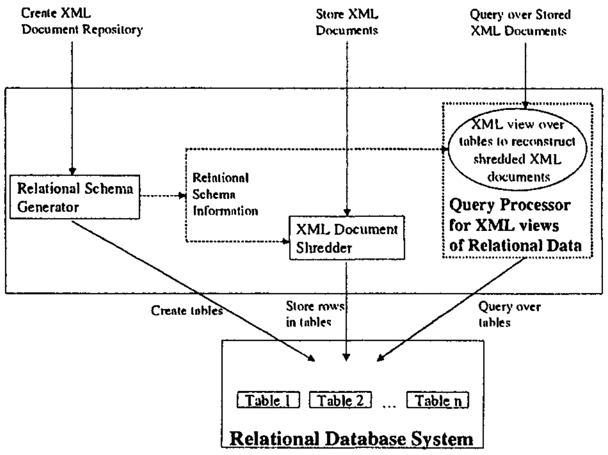

Create XML Document Repository
Store XML Documents
Query over Stored XML Documents
XML view over tables to reconstruct shredded XML documents
Relational Schema Generator
Relational Schema Information
XML Document Shredder
Query Processor for XML views of Relational Data
Create tables
Store rows in tables
Query over tables
Table 1
Table 2
...
Table n
Relational Database System

Figure 1: High-level Architecture

over XML documents can be processed as a query over the reconstruction XML view. This in turn can be efficiently handled by a query processor used for processing queries over XML views of relational data [1][2][4]. Thus, a single query processor is sufficient to provide a general XML query capability over XML documents, regardless of the relational schema generation technique. Further, this query-processor can process queries over XML documents and XML views of existing relational data, because they are all just XML views of relational data. This makes it possible to query seamlessly over XML documents *and* relational data.

As mentioned earlier, the proposed technique is general enough to support many mechanisms for relational schema generation. This is because, for a given mechanism, only a program stub that does the following is required. When the stub is invoked (possibly with the schema of the XML documents to be stored), it does the following.

1) Generates the desired relational schema for storing XML documents.
2) Produces an XML shredder object that can take in XML documents and shred them into rows in the tables of the generated relational schema.
3) Creates a reconstruction XML view over the generated relational schema that indicates how shredded XML documents are to be (virtually) re-constructed.

As shown in Figure 1, the above three components are sufficient to provide a general query capability over XML documents using any relational schema generation technique. It is important to note that (1) and (2) have to be written regardless of whether the proposed technique is used. However, using the proposed technique, it is sufficient to just generate a reconstruction XML view (3) instead of writing a full-blown XML query processor. The former is probably an order of magnitude easier to accomplish than the latter. Thus, the proposed technique greatly simplifies the task of relational schema generation.

```
 1. Create XML Document Repository PurchaseOrder using DTD
 2.  <!ELEMENT PurchaseOrder (ItemsBought, Payments)>
 3.  <!ATTLIST  PurchaseOrder  BuyerName  CDATA #REQUIRED
 4.                            Date       CDATA #REQUIRED
 5.
 6.  <!ELEMENT ItemsBought     (Item)*>
 7.
 8.  <!ELEMENT Item            EMPTY>
 9.  <!ATTLIST   Item          PartId     CDATA #REQUIRED
10.                            Cost       CDATA #REQUIRED>
11.
12.  <!ELEMENT Payments        (Payment)* >
13.
14.  <!ELEMENT Payment         EMPTY>
15.  <!ATTLIST   Payment       CreditCard  CDATA #REQUIRED
16.                            ChargeAmt   CDATA #REQUIRED>
```

Figure 2: Creating an XML Document Repository

For the remainder of this paper, we illustrate how the reconstruction XML view is relatively easy to generate for widely different relational schema generation techniques. In order to do this, we use two relational schema generation techniques published in the literature – one that uses XML schema information, and one that does not.

3. CASE STUDY 1

In this section, we show how a reconstruction XML view can be generated for the *shared* relational schema generation technique proposed in [11]. We begin by briefly describing this relational schema generation technique.

3.1 Relational Schema Generation and XML Document Shredding

The *shared* relational schema generation technique uses XML schema information (DTDs [14]) to create the appropriate relational tables. To illustrate how the technique works, consider the XML document repository definition shown in Figure 2. The body of the definition specifies the DTD of the XML documents to be stored. A description of the DTD specification is provided for readers unfamiliar with DTDs. The top-level element is called "PurchaseOrder" (lines 2-4). Each purchase order element has two sub-elements, namely "ItemsBought" and "Payments" (line 2). Each purchase order element also has two attributes, namely "BuyerName" and "Date" (lines 3-4). Each "ItemsBought" element has zero or more "Item" elements (line 6), and each "Item" element in turn has two attributes (lines 9-10) but no sub-elements (line 8). "Payments" elements are defined similarly.

Given the DTD of the XML documents to be stored, the relational schema generation works as follows. First, a structure called the DTD graph that mirrors the structure of the DTD is created. The DTD graph for our example is shown in Figure 3. As can be seen, each node in the graph represents an XML element, an XML attribute or an

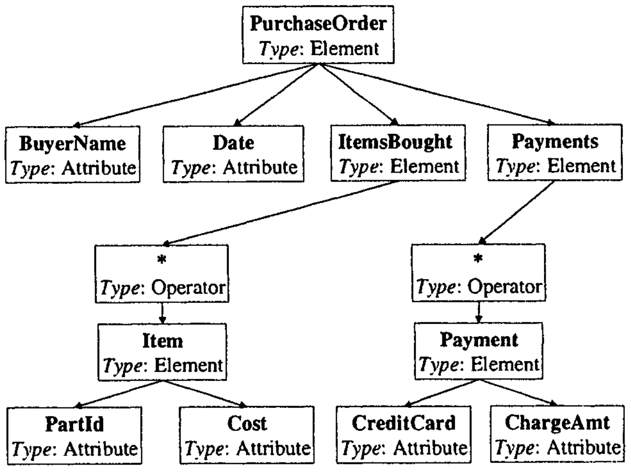


Figure 3: A DTD Graph

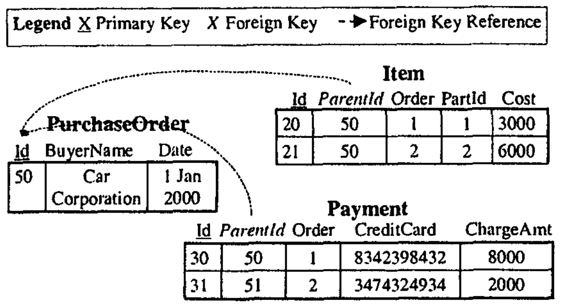
"operator". The "*" operator is used to identify "set" sub-elements, i.e., those that can occur many times under a parent element.

Once the DTD graph is created, it is traversed to construct the desired relational schema. This is done by creating a relation for the root element of the DTD graph ("PurchaseOrder" in our example). All children of an element are represented in the same relation as the element except if the child is a "*" node. In this case, because it corresponds to a "set" child, and because regular relations cannot capture set-valued attributes, the child of the "*" node is represented in a separate relation. Thus separate relations are created for the "Item" and "Payment" elements in our example. The relational schema generated for our example DTD graph is shown in Figure 4. Note that all relations have an "Id" field, which serves as the primary key. In addition, all relations corresponding to non-root elements ("Item", "Payment") also have a "ParentId" field, which is a foreign key reference to its parent "PurchaseOrder". This links a child element to its parent element. Each relation corresponding to a non-root element also has an order field, which specifies the element's relative order among its siblings.

The XML document shredder uses the DTD graph to shred XML documents as rows in the generated tables. Figure 4 shows the rows obtained by shredding the XML document in Figure 5.

3.2 Reconstruction XML View Generation

We now show how the reconstruction XML view can be generated for the technique described in the previous section. Recall that a reconstruction XML view is defined over the tables used to store shredded XML documents. It is used to reconstruct the original XML documents. This enables queries over XML documents to be treated as queries over the reconstruction XML view.



PurchaseOrder

Id	BuyerName	Date
50	Car Corporation	1 Jan 2000

Item

Id	ParentId	Order	PartId	Cost
20	50	1	1	3000
21	50	2	2	6000

Payment

Id	ParentId	Order	CreditCard	ChargeAmt
30	50	1	8342398432	8000
31	51	2	3474324934	2000

Figure 4: Generated Relational Schema

```
<PurchaseOrder BuyerName="Car Corporation" Date="1 Jan 2000">
  <ItemsBought>
    <Item PartId="1" Cost= "3000"/>
    <Item PartId= 2" Cost="6000"/>
  </ItemsBought>
  <Payments>
    <Payment CreditCard="8342398432" ChargeAmt="8000.00"/>
    <Payment CreditCard="3474324934" ChargeAmt="2000.00"/>
  </Payments>
</PurchaseOrder>
```

Figure 5: Purchase Order XML Document

```
<db>
    <PurchaseOrder>
        <row> <Id>50 </Id> <BuyerName> Car Corporation </BuyerName> <Date> 1 Jan 2000 </Date> </row>
    </PurchaseOrder>
    <Item>
        <row> <Id> 20 </Id> <ParentId> 50 </ParentId> <Order> 1 </Order> <PartId> 1 </PartId> <Cost> 3000 </Cost> </row>
        <row> <Id> 21 </Id> <ParentId> 50 </ParentId> <Order> 2 </Order> <PartId> 2 </PartId> <Cost> 6000 </Cost> </row>
    </Item>
    <Payment>
    ... similar to <PurchaseOrder> and <Item>
    </Payment>
</db>
```

Figure 6: The Default XML View

```
For $PurchaseOrder in view("default")/PurchaseOrder/row
Return <PurchaseOrder BuyerName=$PurchaseOrder/BuyerName Date=$PurchaseOrder/Date>
            <ItemsBought>
                For $Item in view("default")/Item/row[ParentId = $PurchaseOrder/Id]
                Return <Item PartId=$Item/PartId Cost=$Item/Cost/>
                Sortby ($Item/Order)
            </ItemsBought>
            <Payments>
                For $Payment in view("default")/Payment/row[ParentId = $PurchaseOrder/Id]
                Return <Payment CreditCard=$Payment/CreditCard
                                ChargeAmt=$Payment/ChargeAmt/>
                Sortby ($Payment/Order)
            </Payments>
        </PurchaseOrder>
```

Figure 7: A Reconstruction XML View

Since a reconstruction XML view is just an XML view of relational data, it can be processed in the same way. We illustrate this in the context of the XPERANTO system [1][13]. The technique described should also work with other systems that support the creation and querying of XML views of relational data [2][4].

As a starting point, XPERANTO automatically creates a *default XML view,* which is a low-level XML view of the underlying relational database. Users can then define their own XML views on top of the default view using the XQuery query language [15]. Figure 6 shows the default XML view for the relational database shown in Figure 4. As shown, top-level elements correspond to tables with table names appearing as tags. Row elements are nested under the table elements. Within a row element, column names appear as tags of sub-elements, and column values appear as text.

The XML query over the default XML view that defines the reconstruction XML view for our example is shown in Figure 7. It reconstructs the XML document of Figure 5 from the rows in Figure 4. As shown, the query loops over all rows in the PurchaseOrder table to (re)construct the top-level "PurchaseOrder" XML elements. Nested queries are used to (re)construct "Item" and "Payment" sub-elements. Note that a sortby clause appears in the nested queries so that the sub-elements appear in the same order as they appeared in the original XML document. Given this reconstruction XML view, queries over the stored XML documents can be processed by the XPERANTO query-processor as queries over the reconstruction XML view.

```
Algorithm buildReconstructionQuery (DTDGraphNode node, String parentTableRowVariable)
   returns query
 1. // First identify the type of the DTD Graph node
 2. if (node is of type Element) then
 3.    // Check whether a separate table is created for this element node
 4.    if (node is stored in separate table from parent) then
 5.       // Create a new variable that ranges over the rows of the table
 6.       QueryString = "For $" + node.name + " in view(DefaultView)/" + node.name + "/row
 7.       // Join on parentId if this is not the root element
 8.       if (not node.isRoot) then
 9.          QueryString += "[parentId = " + parentTableRowVariable + "/Id]"
10.       endif
11.       QueryString += "Return"
12.       currTableRowVariable = "$" + node.name
13.    else
14.       // Child is represented in the same table as the parent.
15.       currTableRowVariable = parentTableRowVariable;
16.    endif
17.    // Construct the element template by recursing on the attributes and sub-elements
18.    QueryString += "<" + node.name + " "
19.    for (all attributes A of node) do
20.       QueryString += buildReconstructionQuery(A, currTableRowVariable)
21.    endfor
22.    QueryString += ">"
23.    for (all sub-element, operator and text node children E of node) do
24.       QueryString += buildReconstructionQuery(E, currTableRowVariable)
25.    endfor
26.    QueryString += "</" + node.name + ">"
27.    // If this element node is joined with its parent, then sort by the Order field
28.    if (node stored in separate table from.parent and not node.isRoot) then
29.       QueryString += "sortby (" + currTableRowVariable + "/Order)"
30.    endif
31. else if (node is of type Attribute) then
32.    // Attributes are always in the same relation as the parent. So, just create attribute
33.    QueryString = node.name + " = " + parentTableRowVariable + "/" + node.name
34. else if (node is of type Text) then
35.    // Add the text value of the node to the query
36.    QueryString = parentTableRowVariable + "/" + node.name
37. else // Node is of type Operator
38.    // Simply recurse on child
39.    QueryString = buildReconstructionQuery(node.child, parentTableRowVariable)
40. endif
41. // Return the query string built
42. return QueryString
```

Figure 8: Algorithm to generate Reconstruction XML Views

Figure 8 presents the algorithm for creating a reconstruction XML view given an arbitrary DTD graph[1]. The algorithm works by recursively traversing the DTD graph. Let us walk through the algorithm using the DTD graph in Figure 3 as an example. The algorithm is invoked with the root node of the DTD graph (PurchaseOrder in our example). Since the root node has no parents, parentTableRowVariable is set to null. Since the PurchaseOrder node is of type "Element" and a new table is created for this element during relational table creation, an XQuery "For" clause that binds the variable $PurchaseOrder to the rows of the PurchaseOrderTable is created (line 6 in Figure 8 generating line 1 in Figure 7). Then the PurchaseOrder XML element tag is created (line 18 in Figure 8) and the algorithm is invoked recursively on the child attribute (lines 19-21), sub-element, operator and text (lines 23-25) nodes to create the XQuery fragments to reconstruct these nodes.

[1] This algorithm does not handle recursive DTD graphs. Although we have a general algorithm that handles recursion, we do not present it here because the details are not particularly illuminating in the current context.

During the recursion, parentTableRowVariable is set to the value "$PurchaseOrder" so that children can refer to rows in the parent table. Constructing XQuery query fragments for attribute nodes (lines 31-33) simply assigns the attribute name to the appropriate attribute value using the parent table row variable because attributes are always represented in the same table as their parent elements. This generates the attribute construction fragments in lines 2, 5, 10 and 11 in Figure 7. Constructing XQuery query fragments for text nodes of elements is done similarly (lines 34-36).

Constructing XQuery query fragments for operator nodes (lines 37-39) is achieved by recursing on the child of the operator node. Constructing XQuery fragments for sub-element nodes is similar to that of the root node, except that a join condition is needed to relate it to its parent (lines 8-10 in Figure 8 generating lines 4 and 9 in Figure 7). Also, a sortby clause is needed to order the sub-elements in the same way as they appear in the original XML document (lines 28-30 in Figure 8 generating lines 6 and 12 in Figure 7). If a separate relation has been created for a node in the relational schema, a new (sub-)query is generated. In our example, separate queries are created for PurchaseOrder, Item and Payment nodes. Nested queries are related to the parent query using the parentId field.

Note how the algorithm of Figure 8 eliminates the need for a special-purpose query-processor for this relational schema generation technique. In addition, it enables seamless query capability over XML documents and relational data.

4. CASE STUDY 2

We now show how a reconstruction XML view can be generated for the *edge* relational schema generation technique proposed by Florescu and Kossman [5]. This technique, unlike the previous one, does not make use of XML schema information. We first briefly describe the technique, and then show how the reconstruction XML view can be generated.

4.1 Relational Schema Generation and XML Document Shredding

The basic idea behind this approach to relational schema generation is to view an XML document as a graph. The nodes of the graph are XML elements and attributes, and the edges of the graph represent containment relationships[2]. Each edge of this graph is then stored in a relational table called the Edge table. Figure 9 shows the Edge table populated with the edges of the XML document in Figure 5.

As shown, each edge is uniquely identified by the id fields of the source and destination nodes (the sid and did fields).

[2] Florescu and Kossmann [5] do not distinguish between attributes and sub-elements. However, since these are distinguished in the XML model, we treat them separately.

Did	Sid	Ordinal	Name	Value	Type
1	0	0	PurchaseOrder	null	Element
2	1	null	BuyerName	Car Corp	Attribute
3	1	null	Date	1 Jan 00	Attribute
4	1	0	ItemsBought	null	Element
5	1	1	Payments	null	Element
6	4	0	Item	null	Element
...	...	...	...	...	...

Figure 9: The Edge Table

Each edge also contains the name, value, and type information about its destination node. The order among sibling sub-elements is captured using the ordinal field. In our example, the edge pointing to the root XML element ("PurchaseOrder") is mapped to the first row. Its sid field is 0, which represents the id of the document root. The edges pointing to the BuyerName and Date attributes of the "PurchaseOrder" element are mapped to the second and third row, respectively. Note that these are related to the purchase order did field using the sid field. Similarly, the "ItemsBought" and "Payments" sub-elements of the "PurchaseOrder" element are represented by the fourth and fifth row respectively. The ordinal field captures their order. The other edges of the document are stored similarly.

4.2 Reconstruction XML View

Figure 10 shows the XQuery query used to define the reconstruction XML view for the above relational schema generation approach. The query first determines the edge pointing to the root element and invokes a function called buildElement to construct the root element (lines 14-16). The buildElement function (lines 1-13) is recursive and builds up document fragments rooted at a given element. It first creates an element with the appropriate tags (line 3). It then produces the character values associated with an element (line 4). A nested sub-query is then used to determine the edges pointing to the attributes of the element (lines 5-7), and the attributes are then created using the XQuery built-in function *attribute* (line 7). Finally, another nested sub-query is used to determine the edges pointing to the sub-elements of the element (lines 8-9), and these are then created by recursively invoking the buildElement function (line 10). The sub-elements are then ordered by their ordinal position (line 11).

Note that since XML schema information is not used in this technique, the same reconstruction XML view can be used for any XML document. Given this reconstruction XML view, queries over XML documents can be processed using a query-processor for XML views of relational data. Again, the need for a special-purpose query engine is made unnecessary because of the reconstruction XML view.

```
Function buildElement ($id integer, $name string,
                              $value string) returns element {
  <$name>
     $value,
     For     $att in view("default")/Edge/row
    Where  $att/sid = $id and $att/type = "Attribute"
     Return  attribute($att/name, $att/value),
     For     $subelem in view("default")/Edge/row
    Where $subelem/sid = $id and $att/type = "Element"
    Return buildElement($subelem/did,
                                  $subelem/name,$subelem/value)
    Sort by $subelem/ordinal
  </$name>}
for $root in view("default")/Edge/row
where $root/sid = 0
return buildElement($root/did, $root/name, $root/value)
```

Figure 10: Reconstruction XML View

5. RELATED WORK

Deutsch et al. [3] present a technique whereby different relational schema generation techniques can be specified using a declarative language called STORED. They also present algorithms for translating semi-structured queries into SQL queries, given a STORED specification. However, the expressive power of the STORED language is rather limited, and it cannot handle relational schema generation techniques involving recursion (such as *edge* [5] and *shared* [11]). Further, unlike our technique, their query processor cannot query across relational data and XML documents.

In parallel work, Manolescu et al. [8] describe a query processor that works for different relational schema generation techniques. Their work is done in the context of data integration, and the tables generated by each relational schema generation technique are specified as materialized views over a virtual global schema. Materialized view matching algorithms are then used to rewrite XML queries into SQL queries. However, these materialized view matching algorithms are NP-Hard even for simple conjunctive queries [7][8] (approximations *cannot* be used because materialized views have to be matched for queries to run). Further, their approach does not currently handle complex SQL constructs such as arbitrarily nested queries. In contrast, the approach presented in this paper relies on more efficient view/query composition [2][4][13] over the reconstruction XML view, and can handle arbitrarily complex query constructs [13].

6. CONCLUSION

We have presented a technique for querying XML documents using a relational database system, which (a) enables the same query processor to be used with most relational schema generation techniques, and (b) allows users to query seamlessly across relational data and XML documents. The proposed technique can thus serve as the infrastructure for investigating issues common to different relational schema generation techniques, such as document caching, updates, and information retrieval style querying.

A potential cause for concern is that our general technique may be less efficient than a special-purpose query processor for translating XML queries into SQL queries. However, based on our prototype implementation in Java, we have found that it only takes about 100-300 milliseconds to translate complex XML queries into SQL queries [13]. Most of the time is actually spent in SQL query execution, which typically takes on the order of seconds. For SQL query execution, we use the sorted outer union technique, which has been shown to be both stable and efficient [12].

7. REFERENCES

[1] M. Carey, et al., "XPERANTO: Publishing Object-Relational Data as XML", Workshop on the Web and Databases (WebDB), Dallas, Texas, May 2000.

[2] V. Christophides, S. Cluet, J. Simeon, "On Wrapping Query Languages and Efficient XML Integration", SIGMOD Conference, Dallas, Texas, June 2000.

[3] A. Deutsch, M. Fernandez, D. Suciu, "Storing Semi-structured Data with STORED", SIGMOD Conference, Philadelphia, Pennsylvania, May 1999.

[4] M. Fernandez, W. Tan, D. Suciu, "SilkRoute: Trading Between Relations and XML", World Wide Web Conference, Toronto, Canada, May 1999.

[5] D. Florescu, D. Kossman, "Storing and Querying XML Data using an RDBMS", IEEE Data Engineering Bulletin, 22(3), pp. 27-34, 1999.

[6] R. Goldman, et al., "From Semi-structured Data to XML: Migrating the Lore Data Model and Query Language", WebDB Workshop, Philadelphia, Pennsylvania, June 1999.

[7] A. Levy, A. Mendelzon, Y. Sagiv, D. Srivastava, "Answering Queries using Views", PODS Conference, San Jose, California, 1995.

[8] I. Manolescu, D. Florescu, D. Kossman, "Answering XML Queries over Heterogenous Data Sources", VLDB Conference, Rome, Italy, September 2001 (to appear).

[9] J. Naughton, et al., "The Niagara Internet Query System", IEEE Data Engineering Bulletin, Vol. 24, No. 2, 2001.

[10] A. Schmidt, et al., "Efficient Relational Storage and Retrieval of XML Documents", Workshop on the Web and Databases (WebDB), Dallas, Texas, May 2000.

[11] J. Shanmugasundaram, et al., "Relational Databases for Querying XML Documents: Limitations and Opportunities", VLDB Conference, Edinburgh, Scotland, September 1999.

[12] J. Shanmugasundaram, et al., "Efficiently Publishing Relational Data as XML Documents", VLDB Conference, Cairo, Egypt, September 2000.

[13] J. Shanmugasundaram, et al., "Querying XML Views of Relational Data", VLDB Conference, Rome, Italy, September 2001 (to appear).

[14] World Wide Web Consortium, "Extensible Markup Language (XML) 1.0 (Second Edition)", W3C Recommendation, October 2000.

[15] World Wide Web Consortium, "XQuery: A Query Language for XML", W3C Working Draft, February 2000.