Recursive XML Schemas, Recursive XML Queries, and Relational Storage: XML-to-SQL Query Translation

Rajasekar Krishnamurthy Venkatesan T. Chakaravarthy Raghav Kaushik Jeffrey F. Naughton
University of Wisconsin-Madison
{sekar,venkat,raghav,naughton}@cs.wisc.edu

Abstract

We consider the problem of translating XML queries into SQL when XML documents have been stored in an RDBMS using a schema-based relational decomposition. Surprisingly, there is no published XML-to-SQL query translation algorithm for this scenario that handles recursive XML schemas. We present a generic algorithm to translate path expression queries into SQL in the presence of recursion in the schema and queries. This algorithm handles a general class of XML-to-Relational mappings, which includes all techniques proposed in literature. Some of the salient features of this algorithm are: (i) It translates a path expression query into a single SQL query, irrespective of how complex the XML schema is, (ii) It uses the “with” clause in SQL99 to handle recursive queries even over non-recursive schemas, (iii) It reconstructs recursive XML subtrees with a single SQL query and (iv) It shows that the support for linear recursion in SQL99 is sufficient for handling path expression queries over arbitrarily complex recursive XML schema.

1. Introduction

This paper is the first to present a generic algorithm that translates path expression queries to SQL in the presence of recursion in the schema in the context of schema-based XML Storage shredding of XML into relations. Here, we refer to path expression queries having the descendant axis (//) as recursive XML queries and to techniques that store XML data into an RDBMS based on an XML schema (or DTD) as schema-based XML Storage techniques.

The reader may justifiably be skeptical of this claim. After all, there have been many schema-based techniques proposed for shredding XML data into relations [3, 18, 20, 28]. There has also been a lot of work on schema-oblivious shredding of XML into relations [9, 15, 24], where the target relational schema is fixed oblivious to the XML schema. Moreover, there has been a great deal of work on translating XML queries into SQL [4, 12, 16, 21, 25, 27] in the context of publishing existing relational data as XML (the “XML Publishing” scenario). It seems plausible that somewhere in all this work must lie the solution to the problem we claim to solve in this paper. Unfortunately, that is not the case — none of this previous work solves the query translation problem for schema-based shredding in the presence of recursion in the XML schema.

Firstly, while [20, 28] propose schema-based XML shredding methods applicable over recursive XML schemas, there has been no published work presenting algorithms for translating XML queries into SQL in this context. Secondly, while the schema-oblivious methods (for example, the Edge approach [15]) can handle recursive schemas and recursive queries with ease, the query translation algorithms for these approaches are not applicable in the context of schema-based shredding. Finally, in the “Publishing” domain the class of XML schemas that have been considered includes only (non-recursive) tree schemas.

At this point the reader may be wondering if this gap in the literature exists because the problem is not well motivated. We think that is not the case. In a recent study of real-world DTDs [6], out of the 60 DTDs analyzed, more than half (35) of them were recursive, which suggests that recursive XML schemas are common in practice. Furthermore, recursion is ubiquitous in XML queries, as it appears in any path expression that uses the descendant axis (//). Finally, there is a growing body of work suggesting that for many query workloads, schema-based shredding approaches yield far better performance than schema-oblivious shredding [29, 30].

We present a generic algorithm that translates path expression queries to SQL in the presence of recursion in the XML schema and queries in the context of schema-based shredding of the XML into relations. This algorithm always outputs an SQL query of size polynomial in size of the input XML-to-Relational mapping and the XML query. An interesting aspect of this is that we need the SQL99 with
We represent the mapping between XML elements and relational columns through annotations on the schema graph. For example, one way of mapping the XML schema in Figure 1 into relations results in the following relational schema.

- **Book** $(id, \text{title}, \ldots)$
- **Author** $(id, \text{parentid}, \ldots)$
- **Section** $(id, \text{parentid}, \text{parentcode}, \text{title}, \ldots)$
- **Para** $(id, \text{parentid}, \ldots)$
- **Figure** $(id, \text{parentid}, \text{caption}, \text{image}, \ldots)$

The annotations on the schema graph in Figure 1 correspond to this decomposition. Each non-leaf (internal) node in the schema is associated with a relation name (shown next to the node). Each leaf node is associated with a column name as well. The relational schema into which we shred the XML data is the set of relations that occur in the node annotations. Each relation has an id field, which is the primary key. In addition, parentid and parentcode fields are included as required to preserve document structure.

A node annotation for a leaf node $n$, $Annot(n)$, is of the form $R.C$, where $R$ denotes a relation and $C$ denotes a column in $R$. A node annotation for a non-leaf node $n$, $Annot(n)$, is of the form $R$ indicating a relation name $R$. If a node $n$ in the schema has multiple in-coming edges, then each of these edges is annotated with a condition of the form $\text{parentcode} = val$, indicating a code for the parent of an element matching $n$ in the document. For a relational column $R.C$, we define $\text{LeafNodes}(R.C)$ to be the set of leaf schema nodes annotated with $R.C$.

We now discuss what properties we expect from an XML-to-Relational mapping.

- The decomposition algorithm that actually shreds the XML data into relations must respect the mapping.
- All the XML data must be completely shredded into relations and no part of the XML data must be stored multiple times.
- There must be no data in the relations other than that which is present in the XML document.
• Enough information must be maintained in the relational data to enable reconstruction of the original XML data.

Every lossless decomposition scheme we have encountered in the literature satisfies the above properties. We formalize these as follows.

With every path \( p = <n_1, \ldots, n_k> \), we associate an SQL query, \( SQL(p) \) as given in Figure 2. Intuitively, the SQL query retrieves from the relational shredding the information that appeared in portions of the original document that match the path \( p \).

```
procedure SQL(path p)
begin
    add Annot(n1) to the From clause
    for (i from 2 to k) do
        if (Annot(n_i) is different from Annot(n_{i-1})) then
            add Annot(n_i) to the From clause
            add Annot(n_{i-1}).id = Annot(n_i).parentid
                to the Where clause
        else if (Annot(e) is of the form parentcode = val) then
            add Annot(n_i) to the From clause
            if (Annot(e) is of the form C = val) then
                /* C may be parentcode */
                for (i from 2 to k) do
                    add Annot(n_i) to the Where clause
                    add R.C = val to the Where clause
                    Add Annot(n_k) = R.C to the Select clause
                /* if there are multiple instances of the relation R, 
                use the last instance */
        end
    end
end
```

**Figure 2. Query associated with a path \( p \)**

With a leaf (schema) node \( l \), we associate a root-to-leaf SQL query, \( RtoL(l) \) as follows. Let the root-to-leaf paths to \( l \) be \( p_1, \ldots, p_m \). Then, \( RtoL(l) = \bigcup_{i=1}^{m} SQL(p_i) \). The union operation here preserves duplicates. If the mapping schema is recursive, the number of root-to-leaf paths will be infinite for certain leaf nodes and the \( RtoL \) query for such nodes is the union of infinitely many queries.

For example, for the schema in Figure 1, \( RtoL(9) \) is given below.

```
select S2.title
from Book B, Section S1, Section S2
where B.id = S1.parentid and S1.parentcode = 1
    and S1.id = S2.parentid and S2.parentcode = 2
```

Again, intuitively, \( RtoL(l) \) retrieves from the relations the information that would be found in the original XML document by starting at the route and traversing all paths that match \( l \). If the XML-to-Relational mapping satisfies the properties mentioned above, then the following properties also hold.

1. For each root-to-leaf path \( p \), \( SQL(p) \) returns the values of all elements that satisfy \( p \). This is under multiset semantics.
2. For each leaf node \( l \) in the schema, \( RtoL(l) \) returns the values of all elements (attributes) associated with \( l \). This is under multiset semantics.
3. For every relational column \( R.C \) with \( LeafNodes(R.C) \neq \emptyset \), let \( Q \) be the SQL query: “select \( R.C \) from \( R \)”. Then,

\[
Q = \bigcup_{l \in LeafNodes(R.C)} RtoL(l)
\]

4. Consider edge \( e = <n_i, n_j> \) where \( Annot(n_i) = R_i \) and \( Annot(n_j) = R_j \) or \( R_j.C \). If \( R_i \neq R_j \), then \( e \) must be annotated with the condition \( parentcode = i \).

The final condition ensures that if a relation \( R_j \) “points to” more than one relation, then by examining the parent code, we can find out which relation is being pointed to by a given tuple. This information is needed to reconstruct the original XML data. We note here that if \( R_j \) points to only one relation in the entire mapping, then this annotation can be omitted.

We refer to the annotated schema graph illustrating the XML-to-Relational mapping as the *mapping schema graph*. Any mapping schema graph that satisfies the above mentioned properties is a *valid* mapping schema.

In general, we allow two additional features in the mapping: selection conditions as edge annotations and presence of dummy nodes. Any edge \( e \) from \( n_1 \) to \( n_2 \) may have an optional annotation of the form \( C = val \), where \( C \) is a column in the relation \( Annot(n_1) \). In XML documents, certain elements may be introduced just to group elements that appear beneath them. We refer to such schema nodes as *dummy* nodes. For example, we could have a dummy Section node in-between nodes 1 and 4 to group together all the sections in a book. An algorithm that shreds this document into relations need not take any action on finding a dummy node. We can detect that a shredding algorithm has considered a node \( n \) to be a dummy node by the fact that (1) \( n \) is a non-leaf node, (2) \( n \) is annotated with the same relation as its parent, (3) each in-coming edge is labeled \( e \) and, (4) each in-coming edge has a null annotation. For ease of exposition, we assume that any non-leaf node that is not a dummy node has an *elemid* attribute that uniquely identifies an element within an XML document.

### 2.2.1. Path Expression Queries

A simple path expression (SPE) can be denoted as “\( s_1 \ l_1 \ s_2 \ l_2 \ldots s_k \ l_k \)”, where each of the \( l_i \) is a tag name and each of the \( s_i \) is either / (denoting a parent-child traversal) or // (denoting an ancestor-descendant traversal). Each \( s_i \ l_i \)
pair is a navigation step of the path expression and $k$ is the number of steps in the query.

A generalized simple path expression (GSPE) can be denoted as $p_{1} p_{2} \ldots p_{k}$ where each $p_{i}$ is of the form $p_{i}^{1} p_{i}^{2} \ldots p_{i}^{k_{i}}$ ($k_{i} \geq 1$). Here, each $p_{i}^{j}$ is a simple path expression. Each $p_{i}$ thus denotes a disjunction of simple path expressions. Also, the special tag name "*" matches any tag name in a GSPE query.

The result of a generalized path expression is the set of all nodes that match the path expression query. This is similar to XPath semantics, where the result is an ordered list of matching nodes. We assume an unordered data model in this paper and discuss how our algorithm can be modified to work in an ordered model in Section 4.3. There are two possible ways to return the set of matching nodes:

- **Select** mode: For leaf nodes, this corresponds to returning the values of the elements. For non-leaf nodes, we return the value of the corresponding `elemid` attribute.
- **Reconstruct** mode: For leaf nodes, this corresponds to returning the values of the elements. For non-leaf nodes, we reconstruct the subtree rooted at the element.

### 3. Query Translation Over Recursive XML schemas

In this section, we present an XML-to-SQL query translation algorithm over recursive mapping schemas for the class of generalized simple path expression (GSPE) queries defined in Section 2.2.1. We will assume the Select mode in this section and present our solution for the Reconstruct mode in Section 4.2.

Evaluating a path expression query over an XML-to-Relational mapping can be viewed as a two stage process: (i) use the XML query to identify the paths in the XML schema graph that satisfy the query, and (ii) use the annotations from the XML-to-Relational mapping to construct an equivalent relational query. We refer to these stages as the PathId and SQLGen stages respectively. We explain the two stages in the next two subsections.

#### 3.1. PathId stage

In the PathId stage, we execute the GSPE query $Q = p_{1} \ldots p_{k}$ on a schema graph and identify the satisfying paths in the schema graph. Since the mapping schema may be recursive, the number of paths may be infinite, so we cannot enumerate all the possible matching paths. Even when the mapping schema is non-recursive, for DAG schema graphs, it is possible for the number of matching paths to be exponential in the size of the mapping schema and the query. So, we should not attempt to enumerate all complete paths.

Instead, just like the DAG schema graph represents shared information across multiple paths in a compact fashion, we represent the matching paths as a graph. This will allow us to handle recursive and non-recursive mapping schemas in a unified fashion. As an added benefit, we shall see later how preserving the relationship across multiple paths that existed in the original mapping schema will help us in the SQLGen stage.

Consider the evaluation of a query $Q$ over a mapping schema $S$. We treat the mapping schema as an automaton $AS$ and the query as an automaton $AQ$. We construct the cross-product automaton $ASQ$ from $AS$ and $AQ$. We eliminate all the dead-states in $ASQ$ and the resulting automaton has all the matching paths in it. This approach is similar to the one proposed in [14] for evaluating regular path queries over graph schemas. We illustrate the main idea with an example and explain the parts where our algorithm differs from the one in [14]. The reader is referred to [14] for more details.

Consider the schema $S$ given in Figure 3, which is a part of the schema in Figure 1. The corresponding automaton $AS$ is shown next to it. Similarly, the query $Q = /book/section/title$ is translated into the automaton $AQ$, where state 3 corresponding to the `title` element in $Q$ is the accepting state. We construct the cross-product automaton $ASQ$ and remove the dead states. The resulting automaton $ASQ$ is shown in the figure. A state with number $(i, j)$ in $ASQ$ represents a combination of state $i$ in $AS$ with state $j$ in $AQ$. Since state 3 in $AQ$ is an accepting state, all states with state number $(i, 3)$ are accepting states in $ASQ$ (in this case just $(5, 3)$). Notice how $ASQ$ has simulated the query over the mapping schema and identified the single matching path. The state numbers in $ASQ$ illustrate exactly how each path matched the query. In general, $AS$ and $AQ$ are non-deterministic, and as a result $ASQ$ is also a non-deterministic automaton. This cross-product automaton can then be viewed as a mapping schema $S_{SQ}$. The node (edge) annotations for $S_{SQ}$ are the same as the underlying annotations in $S$.

The PathId stage for the query $Q_1 = /book/section/title$ is also shown in the figure. Notice how the // operation in the query translates into a self-loop on node 2 in $AQ$. Also, there are two matching paths in the schema for this query. So, there are two root-to-leaf paths in the cross-product automaton $ASQ_1$.

For purposes of exposition, we assume that all accepting states in $S_{SQ}$ correspond to a leaf node in the original schema. If an accepting state $s \in S_{SQ}$ corresponds to a non-leaf node $n \in S$, we add the state corresponding to the `elemid` child of $n$ as a final state in $S_{SQ}$ (instead of $s$). Informally, this corresponds to returning the `elemid`’s of non-leaf nodes as the result of the query. This corresponds to the...
Select mode described in Section 2.2.1. We will present our solution for the Reconstruct mode in Section 4.2.

3.1.1. Handling XPath semantics

According to XPath semantics, the result of a path expression query is a duplicate-eliminated sequence of nodes. So, even if an element has multiple derivations with respect to the query, it should appear in the query result only once. For example, consider the evaluation of query \( Q_2 = //section//title \). The cross-product automaton \( A_{SQ} \) for this query is given in Figure 4. Notice how there are two matching paths in \( A_{SQ} \) for the title node under the second-level section (node 9). This is due to the fact that either of the section nodes in \( S \) (4 or 7), can match the //section part of \( Q_1 \). For both the cases, the //title part of the query is matched by the title node (node 9) in \( S \). As a result, the //section//section//title path in the schema is replicated twice in \( A_{SQ} \). So, if we construct a SQL query based on this cross-product automaton, we may get duplicate results. But, according to XPath semantics, we should return each satisfying element exactly once. In this section, we explain our approach to handling this issue.

We first examine what the primary reason for the presence of duplicate paths in \( A_{SQ} \) is and how we can avoid it. Going back to the above example, we see that the two paths for book/section/section/title have the following property: the first component of the nodes occurring in the two paths are identical, while the second component differs in (at least) one place. In other words, a single path in the schema gets duplicated, once with each of the two different derivations for the query. So, if the query automaton is a DFA, then the cross-product automaton will not have any schema path duplicated.

For an SPE query with \( k \) steps, we have an algorithm to construct an equivalent DFA with \( k + 1 \) states. We explain this algorithm using the query \( Q_2 \). The resulting deterministic automaton \( A_{Q_2} \), is shown in Figure 4. We partition the query into blocks such that each block has a leading // and there is no occurrence of // in that block. In this case, there are two blocks, one each for //section and //title. Then, we process these blocks from left to right. For the first block //section, we create a start state (state 0) and add a transition to state 1 on the label section. For any other label, since there is a leading //, we add a transition into state 0.
procedure PathId(Q,S)
begin
1. Let $A_S$ be the automaton corresponding to $S$
2. If ($Q$ is an SPE query) then
3. Let $A_{Q_D}$ be the DFA corresponding to $Q$
4. return the cross-product automaton $A_{SQ_D}$
5. Else // $Q$ is a GSPE query
6. Let $A_Q$ be the NFA corresponding to $Q$
7. Let $A_{SQ}^2$ be the automaton that accepts all strings with two or more accepting paths in $A_Q$
8. Compute the cross-product automaton $A_{SQ}^2$
9. If ($A_{SQ}^2$ is empty) then
10. return the cross-product automaton $A_{SQ}$
11. Else
12. Convert $A_Q$ into a DFA $A_{Q_D}$
13. If ($A_{Q_D}$ does not have an exponential increase in size) return the cross-product automaton $A_{SQ_D}$
14. Else
15. return the cross-product automaton $A_{SQ}$
16. // a distinct clause needs to be added in this case
17. // to the final SQL query
end

Figure 5. PathId stage

itself (the start state of the current block). In general, when there are multiple steps in a block, there may be a partial match with the current string and we may have to transition not to the start state but to some intermediate state. This can be found by identifying the longest suffix that matches the current set of labels and is similar to the Knuth-Morris-Pratt string matching algorithm [7]. State 1 is the final state for this block and will act as the start state for the next block. We repeat the process for //title and add state 2 and a transition from 1 to 2 on title. We also add transitions on other labels for state 1. Since this is the last block, we also compute the transitions from state 2 and set state 2 as the final state for $A_{Q_D}$. The general algorithm is omitted for want of space.

**Lemma 1** For an SPE query $Q$ having $k$ steps, the equivalent DFA having $k + 1$ states can be computed in $O(k^2)$ time.

On the other hand, for GSPE queries, there are scenarios when the smallest equivalent DFA is exponential in the size of the query. In this case, we use the following approach. Let $A_Q$ denote the NFA corresponding to the query $Q$. We first compute an NFA $A_{Q}^2$ that accepts all input strings that have two or more accepting paths in $A_Q$. Then, we compute the cross-product automaton $A_{SQ}^2$ between $A_S$ and $A_{Q}^2$. If this automaton is empty, then it means that the cross-product automaton $A_{SQ}$ obtained from the original query and schema automata ($A_Q$ and $A_S$ respectively) will not have any duplicate schema paths and we use $A_{SQ}$ as the output of the PathId stage. On the other hand, if $A_{SQ}^2$ is not empty, then we have two options: (1) convert $A_Q$ into a DFA $A_{Q_D}$ and compute cross-product between $A_S$ and $A_{Q_D}$ or (2) apply a distinct clause for the query obtained from $A_{SQ}$. We choose one of these options based on whether there is a size explosion when we convert $A_Q$ into a DFA.

The PathId algorithm along with the above modifications to handle the semantics of XPath is given in Figure 5.

### 3.1.2. Analysis of PathId stage

In this section, we present an analysis of the number of states in the resulting cross-product automaton and the running time of the above algorithm. We omit the proofs due to lack of space.

Let $s$ and $e$ be the number of nodes in the schema and $k$ be the number of steps in the query. Then the number of states in $A_S$ is $n_s = s + 1$ and the number of states in $A_Q$ is $n_q = k + 1$. For an SPE query, the number of states in $A_{Q_D} = k + 1$.

**Lemma 2** The number of states in the cross-product automaton $A_{SQ}$ is no greater than $n_s * n_q$.

**Lemma 3** If $S$ is a Tree schema and $Q$ is an SPE query, then the number of states in $A_{SQ_D}$ is no greater than $n_s$.

For an SPE query $Q$, for every label $x$, let $\text{ChildOccur}(x, Q)$ denote the number of occurrences of the pattern $/x$ in $Q$. For example, for the query $Q = /section//title$, ChildOccur(section, Q) = 1 and ChildOccur(title, Q) = 0. Let $\text{MaxChildOccur}(Q)$ denote the maximum across all values for ChildOccur($x, Q$) over all labels. In this case, $\text{MaxChildOccur}(Q) = 1$.

Let $\text{DescendantSteps}(Q)$ denote the number of // steps in $Q$. For the above example, $\text{DescendantSteps}(Q) = 2$. Notice how $\text{DescendantSteps}(Q) + \text{MaxChildOccur}(Q) \leq n_q$.

**Lemma 4** For an SPE query $Q$, the number of states in the cross-product automaton $A_{SQ_D}$ is no greater than $n_s * (\text{DescendantSteps}(Q) + \text{MaxChildOccur}(Q))$.

Let us now consider the running time of the various steps in the PathId stage.

From Lemma 1, we see that the DFA corresponding to a query $Q$ can be computed in time $O(n_q^2)$.

**Lemma 5** The cross-product automaton of two state machines with $n_1$ and $n_2$ states respectively can be computed in $O(n_1^2 * n_2^2)$.

**Lemma 6** For a query $Q$, the automaton $A_{Q_D}$ can be computed in $O(n_q^4)$.

---

1This can be achieved by placing a bound on the number of states explored in the NFA-to-DFA conversion.
procedure SQLGen(SSQ)
begin
1. Identify strongly connected components (SCCs) in SSQ
2. Let C be the set of SCCs
3. Merge adjacent components in C that are acyclic
   if one of them dominates the other
4. foreach (c ∈ C in top-down topological order) do
   if (c is not recursive) then
      generate the query for c using SQLForDAG(c)
   else
      generate the query for c using SQLForRecursive(c)
      // a relational query T(n) is associated with
      // each leaf node n now
   end
9. Let finalQ be \( \bigcup_c \text{select distinct}(* \text{from } T(n)) \)
10. If (duplicate elimination is required) then
11. Output the query “select distinct(*) from finalQ”
end

**Figure 6. SQLGen Algorithm for recursive mapping schemas**

**Theorem 1** The running time of the PathId stage for an SPE query is \( O(n_c^2 + n_q^2) \), while for a GSPE query, the running time is \( O(n_c^2 + n_q^2) \).

### 3.2. SQLGen stage

Once we have identified all matching paths in the schema S corresponding to query Q, we have a cross-product schema SSQ with all the matching paths encoded in it. Informally, the union of all root-to-leaf paths in SSQ corresponds to the query result. A simple algorithm to generate an SQL query corresponding to Q is to return \( RQ = \bigcup RtoL(l) \) over all leaf nodes in SSQ. While this is a good algorithm when SSQ is a tree, it does not suffice when SSQ is a DAG or is recursive. If SSQ is a DAG, then the number of matching paths may be exponential. Moreover, by unfolding a DAG we may also be missing shared computation in the form of common subexpressions in the final SQL query. So, we need to somehow reflect the DAG structure of SSQ in the SQL query. Similarly, if SSQ is recursive, then RQ is the union of infinite queries. In this section, we show how using the support for linear recursion in SQL99 (with operator) along with the outer union approach, we can construct the equivalent (finite!) SQL query for a recursive cross-product schema.

In order to illustrate our algorithm for handling complex cross-product schema, we use the schema graph S in Figure 7. Notice how this schema has a DAG part and a recursive part. The edge annotations are omitted for clarity. We use the shorthand \( i \) to denote the schema node corresponding to element \( E_i \) and refer to the \textit{elemid} node as node 11.

We explain the algorithm by running through the evaluation of the query \( Q = /E0//E10 \) on the schema graph S in Figure 7. The PathId stage will result in a cross-product schema SSQ identical to S. Notice how since E10 is a leaf node, we add the \textit{elemid} attribute node to SSQ and make that the accepting state.

The outline of the algorithm is given in Figure 6. We first identify the components in SSQ that are recursive. The rest of the nodes are grouped into a set of non-recursive components. We perform this computation by first identifying the strongly connected components in SSQ (step 1) and then merging adjacent non-recursive components whenever possible (step 3). Recall that a component \( c_1 \) dominates component \( c_2 \) if every path from the root to a node in \( c_2 \) passes through some node in \( c_1 \). After the first 3 steps in Figure 6, there are three components in C. They are \( c_1 = \{0, 1, 2, 3, 4, 5, 6\}, c_2 = \{7, 8, 9\} \) and \( c_3 = \{10, 11\} \) (Here node 11 refers to the \textit{elemid} node). We then process these components in top-down topological order, namely \( c_1 \) followed by \( c_2 \) followed by \( c_3 \). For each component, we generate the appropriate relational queries. In the process, we associate a temporary relation \( T(n) \) with every schema node n that is either a leaf node or has a child node in a different component. Once we have processed all components, we generate the final relational query in steps 10-12 using the temporary relations defined earlier.

The algorithm for generating SQL queries corresponding to a non-recursive and a recursive component are given in Figures 8 and 9 respectively. We discuss these in the next two subsections.

#### 3.2.1. Handling a non-recursive component

For non-recursive components, a straightforward approach is to translate each path in the DAG component into a SQL query and take the union of all these queries. However, the number of paths can be exponential in the size of the component. The question arises whether there is any way in
procedure SQLFromDAG(c)
begin
1. Let \( N \) be the set of nodes in \( c \) with either
   a parent or a child in a different component
2. Add any node in \( c \) to \( N \) if it corresponds
   to a leaf node in \( S \).
3. Add all nodes in \( c \) with \( > 1 \) incoming edge to \( N \)
4. Add all nodes in \( c \) with \( > 1 \) outgoing edge to \( N \)
5. With each node \( n \in N \), associate a unique temporary
   relation \( T(n) \)
6. foreach (\( n \in N \) in top-down topological order) do
7.  //generate SQL fragment to populate \( T(n) \)
8.  foreach (incoming edge \( e \) into \( n \)) do
9.    Backtrack along \( e \) till either a node \( m \in N \)
    or a node \( m \notin c \) is obtained.
10. Let the unique \( m \) to \( n \) path be \( p \)
11. Generate SQL(\( p \)) using \( T(m) \) as the relation
    corresponding to \( m \)
12. //Other node and edge annotations in \( S_{SQ} \) are
    //same as underlying ones in the mapping \( S \)
13. Call this query SQL(e)
14. \( T(n) \) is defined as the union of all the SQL(\( e \))
15. endFor
end

Figure 8. SQLGen Algorithm for DAG component

which we can at least guarantee a query that is polynomial in
the size of the DAG component. We show that this is im-
possible if we only consider relational queries involving the
select, project, join and union operations (SPIU queries).
We formalize this claim as follows. Let \( C_1 \) denote the class
of relational queries whose relational algebra expression has
the select, project, join and union operators. Let the size of
a query \( SQ \in C_1 \), RelInst(\( SQ \)), be the number of relation
instances in the relational algebra expression. Then we have
the following result.

**THEOREM 2** There is a family of mapping schemas \( SG \)
such that, for each schema \( S_i \in SG \), there is a simple path
expression query \( p_i \) that has the following property. No re-
lation query \( SQ \in C_1 \), whose size is polynomial in the
size of \( S_i \) and \( p_i \), is a correct translation for \( p_i \).

The proof for the above theorem is based on the fact that
there are instances of acyclic Deterministic Finite Automata
(DFA) whose minimum equivalent regular expression is of
length \( O(\alpha(n)^n) \) [11].

It turns out that we can use the with clause to solve this
problem. Even though the with clause was primarily intro-
duced for supporting recursive queries, it also provides
us with a mechanism for creating temporary relations in a
SQL query. So, whenever there is some computation that
can be shared by multiple paths, we create a temporary re-
lation corresponding to this shared computation, which can
be used repeatedly in the rest of the query. Notice how cre-
ating temporary relations in the query allows us to reduce
the size of the generated SQL query from (potentially) ex-
ponential in the size of the component to a guaranteed poly-
nomial bound.

Component \( c_1 \) is non-recursive and an example of a DAG
component. We use the algorithm for generating the rela-
tional query corresponding to a DAG component given in
Figure 8. We associate temporary relations with any
node that is either a leaf node (part of the final query re-
sult), has a parent or child in a different component, or
represents shared computation (multiple incoming/outgoing
edges). For component \( c_1 \), the set \( N = \{ 2, 3, 6 \} \). So,
we generate SQL with clauses for three temporary relations
corresponding to \( T(2), T(3) \) and \( T(6) \) in that order. The
query corresponding to \( T(3) \) is given below.

```
with T3 as (  
  select R3.*  
  from R0, R1, R3  
  where R0.id = R1.parentid and  
  R1.id=R3.parentid and R3.parentcode=1  
  union all  
  select R3.*  
  from T2, R3  
  where T2.id=R3.parentid and R3.parentcode=2  
)
```

Notice how the query is the union of two subqueries, one
corresponding to each in-coming edge into node 3. Also
note how we use \( T2 \) in the definition of \( T3 \), as node \( 2 \in N \)
and has a temporary relation associated with it. In a similar
fashion, the query for \( T6 \) will have \( T3 \) in it. This illustrates
how shared computation can be efficiently reflected in the
relational query. We would like to point out that the use of
the with clause has two benefits. Firstly, it avoids the po-
tential size blowup for complex DAG schema. Secondly, it
represents the shared computation across different root-to-
leaf paths explicitly. The relational optimizer can choose
from the two options of either sharing computation across
different fragments in the final execution plan or unfolding
the with clause into the union of several conjunctive queries.
In fact, we know of one commercial RDBMS whose optimi-
zer does this exploration. On the other hand, if we did not
use the with clause in the SQL query, then the relational
optimizer has the additional task of finding common subex-
pressions, which is known to be a difficult task.

### 3.2.2. Handling a recursive component

Let us now look at how to generate the relational query for
a recursive component. This algorithm is given in Figure 9.
For each recursive component \( c \), we associate a temporary
relation \( T_R \) whose schema is the outer-union of the schemas
of relations annotating some node in \( T_R \) and generate a re-
cursive query for \( T_R \) as follows. A recursive query has two

```
Let $Q_1$ be the query:

```sql
select R2.*, id(n)
from $T(n)$ R1, Annot(n') R2
where Annot(e) and R2.parentid = R1.id''
```

5. Null pad $Q_e$ appropriately to reflect outer-union schema

6. Let $Q_{init}$ be $\cup Q_e$ over all in-coming edges $e$

//Construct the initialization part for $T_R$

7. foreach (edge $e$ with both end-points in $c$) do

8. Let $e$ be from $n_1$ to $n_2$

9. if ($e$ corresponds to a join edge) then

10. Let $Q_e$ be the query:

```sql
select R2.*, id(n_2)
from $T_R$ R1, Annot(n_2) R2
where R1.schemanode = id(n_1) and
R2.parentid = R1.id and Annot(e)
```

else

// $e$ corresponds to a selection or $n_2$ is a dummy node

12. Let $Q_e$ be the query:

```sql
select R1.*, id(n_2)
from $T_R$ R1
where R1.schemanode = id(n_1) and Annot(e)
```

13. Null pad $Q_e$ appropriately

14. Let $Q_{rec}$ be $\cup Q_e$ where the union is taken over edges $e$ with both end-points in $c$.

$T_R$ is a recursive query defined with $Q_{init}$ as the initialization condition and $Q_{rec}$ as the recursive component.

16. With each node $n \in c$ we associate the query $T(n)$:

```sql
select *, id(n)
from $T_R$ where schemanode = id(n)
```

Figure 9. SQLGen Algorithm for recursive component

parts, an initialization part and a recursive part. The initialization part for the query defining $T_R$ (steps 2-6) captures all incoming edges into $c$ from a different component. For the component $c_2$, there are two such edges $(2, 8)$ and $(3, 7)$ and the initialization part will be the union of two conjunctive queries, one for each incoming edge. The recursion in the component $c$ is captured by the recursive part of the definition of $T_R$. Each edge in $c$ is translated into a query as shown in steps 8-13 and the recursive part of the query defining $T_R$ is the union across all edges within the component. For the component $c_2$, there are four edges and so the recursive query $Q_{rec}$ is the union of four recursive queries. In this case, all four edges are join edges. For example, the edge $(8, 7)$ will translate to the following query:

```sql
select R7.*, id(7)
from TR, R7
where R7.parentid=TR.id and TR.schemanode=id(8)
and R7.parentcode=8
```

Notice how the condition TR.schemanode ensures that the parent tuple corresponds to schema node 8 and the other conditions capture the annotations on the edge. By projecting the id of the child node, we ensure that the queries for outgoing edges from node 7 can be correctly constructed.

Returning to the example query, finally, component $c_3$ is non-recursive and we generate the equivalent relational query using the algorithm in Figure 8.

If $S_{SQ}$ has two root-to-leaf paths matching the same path in $S$, then duplicate elimination is required and we add the distinct clause (recall discussion in previous section, step 16 in Figure 5). In such a scenario, for each leaf node $n \in S_{SQ}$, the id of $n$ and the key column of $R$, where Annot(n) = R.C also have to be projected along with Annot(n) while creating the temporary relations $T(n)$.

For a recursive component $C$, let $N_C$ denote the number of columns in the outer union schema for $C$. This is the sum of the number of columns over all relations annotating some node in $C$. For a mapping schema $S$, let $N_{C_{max}}(S)$ denote the maximum across all values for $N_C$ over all recursive components in $S$.

**Theorem 3** For a mapping schema $S$ and query $Q$, let the output of the PathId stage, $A_{SQ}$ have $V$ nodes and $E$ edges. The equivalent SQL query can be obtained using the SQL-Gen algorithm in $O((V + E) \times N_{C_{max}}(A_{SQ}))$ time.

The proof is omitted due to lack of space.

### 3.3. Preliminary Evaluation of Running Time

We implemented the above algorithm for evaluating SPE queries over a generic XML-to-Relational mapping. Using the XMark benchmark schema [31] and SPE query fragments that appear in the associated query test suite, we evaluated the equivalent SQL queries using the above query translation algorithm. The XML-SQL query translation process took less than 6ms for each SPE query. The XML-to-Relational mapping schema has 101 nodes. We also observed that in all cases, the size of the cross-product schema was less than 100 nodes (the size of the schema).

In order to test the running time of the algorithm under extreme scenarios, when the cross-product schema may have $n_s \times n_q$ states, we used a more complex XML schema. This mapping schema was a complete graph of $n$ nodes and all transitions were on a single label $x$. We then measured the running time of the query translation for the query /x/x/x/x/x/x/x, which has 5 steps. The cross-product automaton has approximately $4n^2$ states and $4n^3$ transitions.
Table 1. Execution time of translation algorithm

<table>
<thead>
<tr>
<th>Clique size (n)</th>
<th>Time Taken (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>80</td>
</tr>
</tbody>
</table>

The running time for different values of \( n \) are given in Table 1.

Notice how while the running time shows a quadratic growth due to the quadratic increase in the number of transitions in the schema, it is still small for reasonable clique sizes. The size of every recursive component that we have seen in real-world DTDs has been less than 10. So we believe that the running time of our translation algorithm will be small in practice.

4. Extensions to more complex path expressions

In this section, we briefly describe our approach to handle branching path expression queries and reconstruct XML subtrees.

4.1. Branching path expression queries

Let us first consider the class of branching path expression (BPE) queries that have a single predicate at the end of the path expression. These queries are of the form \( p_1[p_2]\) or \( p_1[p_2 \text{ op value}] \), where \( p_1 \) and \( p_2 \) are GSPE queries.

The \textit{PathId} stage works as follows. First, we compute the set of all satisfying paths for the GSPE query \( p_1 \). Let \( A_{SQ} \) be the resulting cross-product automaton and let \( F \) be the set of final states. For each \( f \in F \), we compute an auxiliary automaton, \( A_{\text{pred}}(f) \), that corresponds to evaluating the path expression query \( p_2 \) with \( f \) as the start state.

In the \textit{SQLGen} stage, we first compute the SQL query corresponding to \( A_{SQ} \) without the predicates. Now, for each final state \( f \in F \), we add the SQL fragment corresponding to the predicate as follows. Let \( T(f) \) be the temporary relation corresponding to the state \( f \). Then we add a \textit{with} clause of the form

```sql
with T_Pred as ( 
    select * 
    from T 
    where exists (PQ)
)
```

where \( PQ \) is the query corresponding to the predicate automaton \( A_{\text{pred}}(f) \). The final query is the union of all the \( T_{\text{Pred}} \) relations.

Let us now consider a more general BPE query with one or more predicates occurring in any step of the path expression. This query is of the form \( p_1\{Pred_1\} \cdots p_k\{Pred_k\} \), where each \( p_i \) is a GSPE query and each \( Pred_i \) is of the form \( p_j \) or \( p_j \text{ op value} \). An example query is \( Q = //\text{section}[//\text{caption} = \text{v1}]/\text{title} \). Our algorithm proceeds as follows:

In the \textit{PathId} stage, we first apply the above procedure to evaluate \( p_1\{Pred_1\} \). Let \( F \) be the set of final states in the resulting automaton. Then with \( F \) as the set of start states, we compute \( p_2\{Pred_2\} \). We continue this process \( k \) times to obtain the automaton for the entire query. The \textit{SQLGen} stage is also extended in a similar manner to process the resulting automaton.

Let \textit{PrimaryPath}(\( Q \)) be the query \( PP(\( Q \)) = p_1p_2\cdots p_k \).

If the cross-product automaton obtained from \( A_{SQ} \) and \( A_{\text{PP}}(\( Q \)) \) is not empty, then the above algorithm may generate duplicate results and we add a \textit{distinct} clause to the final SQL query. We illustrate why this is needed with an example. Consider the evaluation of \( Q \) over the mapping schema in Figure 1. The section nodes 4 and 7 match the path expression //section. In other words, any instance element \( e \) in an XML document corresponding to either of these schema nodes will match this path expression. On the other hand, whether \( e \) will match //section//caption = \text{v1} //section is going to depend on whether \( e \) satisfies the predicate. Notice how while (incoming) structural conditions can be verified during query translation time (without looking at the data), predicate conditions depend on the data. So, given a pair of parent-child section elements, \( e_1 \) and \( e_2 \), matching nodes 4 and 7, while we can be sure that both match //section, we cannot be sure of whether one or both of them will match the predicate at query translation time. So, the SQL query has to handle all possible cases and as a result, it may produce duplicate results (when both of them satisfy the predicate, then the child of \( e_2 \) will appear twice in the result).

4.2. Reconstructing XML subtrees

In [12, 26], algorithms were presented for reconstructing XML subtrees when the mapping schema is a tree. In this section, we describe how to handle the reconstruction of a recursive component and a DAG component.

Notice that the \textit{SQLGen} algorithm for handling a recursive component in Figure 9 actually reconstructs the XML data corresponding to the entire recursive component. But, what is missing in order to reconstruct the XML subtree is structural information about the different elements. Recall that in [12, 26], this could be determined statically as for a tree XML schema, the number of distinct root-to-leaf paths is fixed. On the other hand, for recursive components, we need to construct the root-to-leaf path dynamically. Notice that the \textit{schemaNode} column in relation \( T_R \) keeps track of
the schema node corresponding to the tuple. We maintain an additional rtol column that keeps track of the path from the root of the subtree being constructed. This is similar to the approach proposed in [29] for constructing dewey numbers dynamically.

In order to handle a DAG component, we have two options. We could either unroll the DAG into a tree and apply prior techniques. If this may lead to a size explosion, we could reconstruct a DAG directly by keeping track of the root-to-leaf path as mentioned above.

4.3. Handling order in XPath semantics

According to XPath semantics, the results of a path expression query have to be returned in document order. In order to support this, the schema-based shredding of XML into relations will need to maintain the relative position among sibling XML elements in some form. This was the primary focus of [29], where solutions were proposed to handle order in XML for an arbitrary query translation algorithm. Hence, in particular, their techniques can be integrated with our algorithm in a straightforward fashion.

5. Related Work

The prior literature on XML-to-SQL query translation can be broadly classified into three areas: (1) schema-based XML storage, (2) schema-oblivious XML storage and (3) XML Publishing.

A number of approaches have been proposed for using an RDBMS to store and query XML data in a schema-based fashion [3, 18, 20, 28, 29]. The main focus of [3, 18, 20, 28] was defining a “good” relational schema for the given XML schema. In [28] the general approach to translating XML queries into SQL is illustrated with examples without any algorithmic details. In [29], the focus is on supporting order-based queries. The authors give an algorithm for the schema-oblivious scenario and briefly mention how the ideas for adding support to order-based queries can be applied with any existing schema-based approach. We are not aware of any published XML-to-SQL query translation algorithm in this scenario.

Several techniques have been proposed for the schema-oblivious XML storage scenario [9, 15, 24, 32] approaches. Each of these approaches (except [9]) proposed a fixed relational schema for storing the XML data and algorithms were presented for translating path expression queries into SQL. In [9], the relational schema is decided based on the XML data. Since the techniques are schema-oblivious, they are applicable irrespective of whether the XML schema is recursive or not. The techniques also consider the presence of the // axis in path expression queries. In [8], an algorithm for translating more general XQuery queries into SQL is presented. Our work is complementary to these techniques because we consider the query translation problem in the schema-based XML storage scenario. In [29, 30], it was shown that for many query workloads over non-recursive XML schema, schema-based shredding approaches yield far better performance than schema-oblivious shredding. So, it is conceivable that the same result holds in a number of scenarios even when the XML schema and/or query is recursive.

In the XML Publishing scenario, there has been a lot of work on translating complex XML queries into SQL [4, 10, 12, 13, 16, 19, 21, 25, 26]. While the class of XML queries considered are fairly complex in these approaches (a significant subset of XQuery/XSLT), the focus is on (non-recursive) tree XML schemas. In contrast, our focus is on XML-to-SQL query translation over recursive XML schema. While some of the above techniques handle // in the XML query by enumerating all satisfying paths, we present a different solution that represents all satisfying paths in a more compact manner.

In [2], an algorithm for reconstructing a recursive XML view was presented. Their solution does not use the support for recursion in SQL and simulates the recursion in middleware instead. In contrast, we show how we can use the support for linear recursion in SQL99 and by combining it with the “outer union” approach construct a single SQL query to reconstruct a recursive XML subtree.

A more detailed description of the existing published work on XML-to-SQL query translation in the above three scenarios is given in [17].

There has been some work on optimizing queries in a semi-structured framework [5, 14, 22] using graph schemas. These techniques are similar to the PathId stage of query translation, and we adapted the cross-product automaton technique proposed in [14], for the PathId algorithm in Section 3.1.

In [1, 23], algorithms for minimizing tree pattern queries, both in the presence and absence of XML schema information, are presented. These algorithms remove redundant parts of the XML query that are implied by either other parts of the query or by the XML schema or a combination of both. These algorithms are complementary to our algorithm and can be used as the first stage to minimize the input XML queries.

6. Conclusions

We presented a generic algorithm to translate path expression queries to SQL in the presence of recursion in the XML schema and queries. This algorithm is applicable over a wide class of techniques for schema-based shredding of XML into relations. We also showed how the with clause in SQL99 is useful in XML-to-SQL query translation over
DAG XML schema and how the support for linear recursion in SQL:99 is sufficient for translating path expression queries into a single SQL query over an arbitrary (recursive) XML-to-Relational mapping.

The algorithm presented in this paper for translating path expression queries into SQL can be adapted to the “Publishing” domain as well. The details of the algorithm will change based on the view definition language, but the main ideas about how to handle recursive schemas, DAG schemas and recursive queries remain the same.

There are a number of avenues for future research. Extending the work in this paper to perform XML-to-SQL query translation for more complex FLWOR XQuery queries, when the XML schema is recursive, is open. Comparing the schema-based and schema-oblivious solutions for XML storage in the presence of recursive XML schema is another important area for future research. Similarly, combining the interval-based techniques used in XML-to-SQL query translation in the schema-oblivious scenario along with the techniques proposed in this paper is another interesting avenue for future work.

Acknowledgement: This work was supported in part by NSF grants ITR-0086002 and CCR-0208013, and a Microsoft fellowship.

References