XML Views as Integrity Constraints and their Use in Query Translation

Rajasekar Krishnamurthy *
IBM Almaden Research Center
rajase@us.ibm.com

Raghav Kaushik *
Microsoft Research
skaushi@microsoft.com

Jeffrey F Naughton
University of Wisconsin-Madison
naughton@cs.wisc.edu

Abstract

The SQL queries produced in XML-to-SQL query translation are often unnecessarily complex, even for simple input XML queries. In this paper we argue that relational systems can do a better job of XML-to-SQL query translation with the addition of a simple new constraint, which we term the "lossless from XML" constraint. Intuitively, this constraint states that a given relational data set resulted from the shredding of an XML document that conformed to a given schema. We illustrate the power of this approach by giving an algorithm that exploits the "lossless from XML" constraint to translate path expression queries into efficient SQL, even in the presence of recursive XML schemas. We argue that this approach is likely to be simpler and more effective than the current state of the art in optimizing XML-to-SQL query translation, which involves identifying and declaring multiple complex relational constraints and then reasoning about relational query containment in the presence of these constraints.

1. Introduction

Using relational database systems to store and query XML documents has received a lot of attention both in the research community and in the leading commercial relational products. Both the research community and the products have explored a number of ways to incorporate support for XML in an RDBMS, but one popular approach that is currently implemented in commercial products (including Oracle XML DB [17], Microsoft SQL Server 2000 SQLXML [18] and IBM DB2 XML Extender [16]) is to "shred" the XML document into relational tuples and to execute XML queries by translating them into SQL. While determining the best way to handle XML in relational database systems is an interesting question, that is not the focus of this paper — we think that this shredding approach has enough advantages and momentum that it will be a commonly provided and perhaps dominant option for the foreseeable future. The goal of this paper is to investigate techniques for improving the quality of the SQL that results from this approach.

As we illustrate with an example in Section 2, when using this shredding approach, sometimes even simple XML path queries get translated into needlessly complex SQL. Intuitively, the reason for this is that the query translator must generate complex queries to ensure that the correct answer to the original XML query is generated for any possible instance of the relational schema. But only a subset of these relational instances could possibly result from shredding documents that conform to the given XML schema, and for these relational instances, in many cases much simple queries can be guaranteed to give the correct answer.

Perhaps in view of this fact, the research literature [4, 10] has explored the idea of improving the generated SQL by identifying relational integrity constraints and using them to minimize the generated SQL queries. This approach is problematic for a number of reasons. First, it is not clear how to automatically deduce and express the relational constraints that are essential for the query minimization process. Second, even if the appropriate relational constraints are discovered and specified, the query minimization problem involves the complex task of reasoning about relational query containment in the presence of integrity constraints.

For this reason, in this paper we argue for a much simpler approach. Instead of "forgetting" the source of the relational data and then patching things up with the addition of relational integrity constraints, we propose the use of one simple new integrity constraint: the "lossless from XML" integrity constraint. Simply put, this constraint says that a given relational data set resulted from the lossless shredding of an XML document that conformed to a given XML schema. We demonstrate the power of this approach by giving algorithms that exploit this "lossless from XML" constraint to translate path expressions into efficient SQL, even in the presence of recursive XML schemas.

Note that we are not claiming any "magic" here — it is possible (even probable) that any optimization that can be done by exploiting the "lossless from XML" constraint could also theoretically be done through the right combina-

* Work done while the authors were students at the University of Wisconsin-Madison

tion of relational integrity constraints and relational query minimization. However, we are arguing that the "from XML" approach is simpler and more direct, and likely to be easier to understand, implement, and extend for other purposes.

We close this introduction by some comments setting this work in the context of other related work. The problem we are addressing in this paper has been called the "XML storage" problem, because it refers to storing XML in a relational system. A closely related problem is XML publishing, in which the data was originally relational but is being exported as XML. The "lossless from XML" integrity constraint is natural for XML storage, but is more problematic in the XML publishing domain, where the data is literally not "lossless from XML." Our previous work has addressed the XML publishing problem [10]. In that paper the key idea was to impose relational constraints on the exported relational tables and then to exploit these constraints during the XML to SQL query translation.

It is possible that in certain circumstances it may make sense to put a "lossless from XML" constraint data that was originally relational, but that would require techniques to verify that the constraint holds (that is, that the data "looks like" it came from an XML document conforming to a specified schema). This is an interesting area for future work, but not the focus of this paper — here we focus on what we regard as the most useful and obvious application of the "lossless from XML" constraint, that is, for data that indeed originated as XML.

The rest of the paper is organized as follows: We present an example in Section 2 to show how more efficient SQL queries can be output during XML-to-SQL query translation. There are many algorithms that could be specified to make use of the "lossless from XML" constraint. For concreteness, in this paper we show how this constraint can be exploited in the context of the XML translation algorithm found in [9]. Towards this purpose, we review the framework and algorithm proposed in [9] in Section 3. We then extend this algorithm for tree XML schemas in Section 4 to exploit the "lossless from XML" constraint. We then present an algorithm that works for recursive XML schema in Section 5.

2. Motivating example

In this section, we present an example scenario to illustrate how knowing that the "lossless from XML" constraint holds can help in generating efficient SQL queries.

Part of the XMark benchmark [15] XML schema is given in Figure 1. One way of mapping the XML schema into relations is given in the figure. We have used a simple annotation mechanism to represent the correspondence between the XML schema and the relational schema.

Consider the evaluation of the following query Q_1, which returns all the item categories: //Item/InCategory/Category. The SQL query output by many published algorithms [5, 8, 13], is SQ_1^1:

```
select   category
from     Site S, Item I, InCat C
where    S.id = I.siteid and I.id = C.itemid and I.continent='africa'
union all ... (6 queries one for each continent except Antarctica)
```

Informally, the translation can be summarized as follows: (i) Identify all paths in the schema that satisfy the query (PathId stage). (ii) For each path, generate a relational query by joining all relations appearing in this path (SQLGen stage). The final query is the union of the queries over all satisfying paths (six paths for Q_1).

Notice how a simple XML query results in a fairly complex SQL query. This happens because the query translator is not taking advantage of the fact that the relational data originated from XML. So, the resulting SQL query has to be a correct translation of Q_1 for every possible instance of the relational schema. This leads to the complex query SQ_1^1.

Now suppose the query translator has the additional information that the relational data satisfies the "lossless from XML" constraint for the given XML view. Then, it can generate the following simpler query SQ_1^2 for XML query Q_1.

```
select   category
from     InCat C
```

The above query is a correct translation for Q_1 because the relational data satisfies the "lossless from XML" constraint. There are six nodes in the schema whose values are stored in the column `InCat.Category` and the query Q_1 selects all of them. Hence, the scan query is a correct translation for Q_1. In comparison to the complex SQL query SQ_1^1, the above query SQ_1^2 is a far more efficient query.

An important question to answer at this point is whether the associated performance gains are substantial. In [10], using a synthetic ADEX dataset conforming to a standard advertisement schema [1] and a dataset from the XMark Benchmark [15], we showed that the associated performance benefits are significant (ranging from a relative performance improvement of 1.15 to 93). Many queries had a speed up of an order of magnitude or more. The datasets used in these experiments satisfied the "lossless from XML" constraint; hence, the results also demonstrate the potential benefits that can be reaped by exploiting the "lossless from XML" constraint during query translation.

At this point the reader may wonder if XML query minimization will be helpful in the above scenario. Note that the above path query is already minimal (and so will be all the other example queries we use later in this paper), it is the translation that caused the problem, not the original XML query. All the work on XML query minimization [2, 6, 12] and on containment and equivalence of path

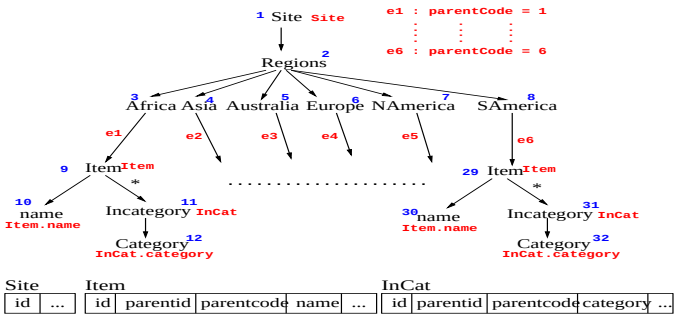

Figure 1. Part of XMark XML benchmark schema and corresponding Relational Schema

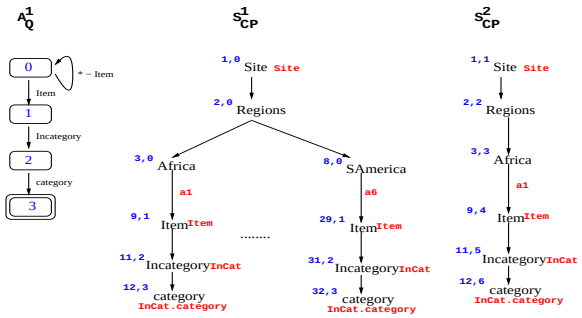

Figure 2. Result of PathId stage for Q_1 and Q_2

expression queries [3, 11, 14] is complementary to the focus of this paper and can be used as the first stage to minimize the input XML queries.

3. Preliminaries

In order to explain how the "lossless from XML" constraint can be exploited in XML-to-SQL query translation, we extend the algorithm proposed in [9] for the XML storage scenario to make use of this constraint. In this section, we first describe a class of XML-to-Relational mappings and then describe when the "lossless from XML" constraint is satisfied. We then briefly explain the algorithm proposed in [9] for translating path expression queries into SQL.

3.1. XML to Relational Mappings

We represent the mapping between XML elements and relational columns through annotations on the schema graph. 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. For ease of exposition, we assume that every non-leaf node has an elemid attribute that uniquely identifies an element within an XML document.

3.2. "Lossless from XML" constraint

While the above description defines the syntax of an XML-to-Relational mapping, we need to define when the relational data satisfies the "lossless from XML" constraint.

A shredding algorithm uses the XML-to-Relational mapping to convert XML data into relational data. We say that the shredding algorithm *respects* a mapping if it satisfies the following properties:

- The shredding algorithm actually shreds the XML data into relations based on the annotations of the mapping.
- All the XML data is completely shredded into relations and no part of the XML data is stored multiple times.
- No data, other than what which is present in the XML document, is inserted into the relations mentioned in the mapping.
- Enough information is maintained in the relational data to enable reconstruction of the original XML data.

Every decomposition scheme we have encountered in the literature satisfies the above properties.

We say that the *lossless from XML* constraint is satisfied if all the relational data was loaded by a shredding algorithm that respects the XML-to-Relational mapping. This implies that the relational data set resulted from the lossless shredding of an XML document that conformed to a given XML schema.

For example, consider the following shredding algorithm A that satisfies the above properties. Given an XML document D_1 conforming to the schema in Figure 1, algorithm A creates (and inserts into the RDBMS) relational tuples in the following fashion. The algorithm processes the elements in the order in which they appear in the document. For the root `Site` element, a tuple is inserted into the `Site` relation. Then, for the `Regions` child element, no action is taken as the node has no annotations. Similarly, no action is taken for the `Africa` child element. Then for each `Item` child element, a tuple is inserted into the `Item` relation with the `parentid` column having the value of the `id` field of the `Site` tuple. Also, the `parentcode` column's value is set to 1 to capture the edge annotation. The value of the *name* column is set to the value of the *name* subelement. Then, for each `Incategory` subelement tuples are inserted in a similar fashion into the `Incategory` relation. Once, the XML subtree rooted under `Africa` has been processed, the subtrees corresponding to the other five continents are processed in a similar fashion.

The above definition of the "lossless from XML" constraint is complete if we formally define the notion of when a shredding algorithm respects a mapping. We introduce some terminology for this purpose.

With every path $p = <n_1, \ldots, n_k>$, we associate an SQL query, $SQL(p)$ obtained by joining all the relations on the path. The annotations on the edges are added as selection conditions. The annotation of node n_k, $Annot(n_k)$, is the value returned by $SQL(p)$. Intuitively, the SQL query retrieves from the relational shredding the information that appeared in portions of the original document that match the 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, $RtoL(l) = \cup_{i=1}^{m} SQL(p_i)$. 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 above schema $RtoL(12)$ is given below.

```
select  IC.category
from    Site S, Item I, InCat IC
where   S.id = I.parentid and I.parentcode = 1 and I.id = IC.parentid
```

For a relational column $R.C$, we define *LeafNodes(R.C)* to be the set of leaf schema nodes annotated with $R.C$.

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.

A shredding algorithm respects an XML-to-Relational mapping,T,if for every collection of XML documents conforming to the given XML schema, the algorithm loads data into the relational database such that the following properties hold.

- P1: For each root-to-leaf path p, $SQL(p)$ returns the values of all elements that satisfy p.
- P2: For every relational column $R.C$ with *LeafNodes(R.C)* $\neq \phi$, let Q be the SQL query: "select $R.C$ from R". Then, $Q = \bigcup_{l \in LeafNodes(R.C)} RtoL(l)$
- P3: For each path $p \in$ T ending in a leaf node, let P denote the set of root-to-leaf paths $p' \in$ T such that the relation names annotating the nodes in p match the annotations for some suffix of p'. Then, $SQL(p) \subseteq \bigcup_{p' \in P} SQL(p')$.

All the above comparisons are under multiset semantics. Every shredding algorithm we have encountered in the literature, including the shredder we presented above, satisfies the above properties.

Notice how the shredding algorithm A needs to be validated just once to make sure that it shreds data respecting any given XML-to-Relational mapping. Once this is done, as long as all the data (in the relations appearing in the mapping) is loaded by the algorithm A , the "lossless from XML" constraint is guaranteed to hold.

3.3. 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. The result of a path expression is the values of the *set* of all nodes that match the query. For non-leaf nodes, we return the value of the corresponding elemid attributes.

3.4. Query Translation Algorithm

The query translation algorithm proposed in [9] has two stages: PathId stage and SQLGen stage. In the PathId stage, the paths in the XML schema that match the query are identified. This is done by treating the schema and the query as automata and constructing the cross-product automaton. For example, consider the query automaton and the corresponding cross-product result shown in Figure 2 for query Q_1. The cross-product schema has been labelled with the pair of schema node and query state number. The result of this stage is a compact representation of all the matching schema paths.

We have also shown the result of the PathId stage for the query Q_2, which returns all the item categories in Africa: /Site/Regions/Africa/Item/Incategory/Category. Notice how there is only one matching schema path for this query.

In the SQLGen stage, the SQL query is generated corresponding to the set of matching paths encoded in the cross-product schema S_{CP}. Informally, the union of all root-to-leaf paths in S_{CP} 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 S_{CP}. While this is a good algorithm when S_{CP} is a tree, it does not suffice when S_{CP} is a DAG (directed acyclic graph) or is recursive. The SQLGen algorithm proposed in [9] uses the *with* clause in SQL99 to handle the DAG and recursive cases.

4. Exploiting the "lossless from XML" constraint for Tree XML Schemas

In this section, we show how we can use the "lossless from XML" constraint while translating queries over a tree schema in order to generate a query that may be more efficient than the corresponding naively generated query. We extend this to directed acyclic graph (DAG) and recursive XML schema in Section 5.

procedure XML_to_SQL_translation(Q, S)
begin
1. Perform PathId using the mapping S and query Q. Let S_{CP} be the resultant cross-product schema.
2. Prune S_{CP} using of the "lossless from XML" constraint .
3. Translate the pruned S_{CP} into SQL.

Figure 3. Query Translation Algorithm using the "lossless from XML" constraint

The outline of this algorithm is given in Figure 3. The output of the PathId stage is pruned making use of the "lossless from XML" constraint and this pruned cross-product schema is the input to the SQLGen stage. The PathId stage is identical to the algorithm presented in [9]. We explain the pruning stage and the SQLGen stage in this section.

4.1. Basic Idea behind the Algorithm

Consider the set of paths P in the schema that end in a node annotated with the column $R.C$. Since the XML-to-Relational mapping satisfies the "lossless from XML" constraint, we know that every tuple in the relation R corresponds to the value of (exactly) one element in the XML document. In other words, each tuple in relation R will appear in the result of the SQL query corresponding to (exactly) one root-to-leaf path in the XML schema. Notice that this gives us two guarantees: (i) No two root-to-leaf paths will have a common tuple in their results and (ii) All the root-to-leaf paths combined together correspond to a scan of the column $R.C$.

Let us look at some example queries to see how we can use the above information to prune the cross-product schema. In the process, we identify two important concepts that form the core of our algorithm. We use the mapping schema in Figure 1 in the following discussion, and will discuss how the algorithm works informally in these specific examples before turning to specifying the full algorithm.

Consider the example query Q_2 from Section 3.4, /Site/Regions/Africa/Item/Incategory/Category. The output of the PathId stage is the cross-product schema S^2_{CP} given in Figure 2. We use the first component of the node identifiers in the cross-product automaton to identify the nodes in the following discussion. There is a single path $p =<1, 2, 3, 9, 11, 12>$ in S^2_{CP} and issuing the query $SQL(p)$ is a correct translation for Q_1. Our goal in this case is to find a shorter suffix p' of p such that $SQL(p)$ is an equivalent translation under the given mapping.

Now, for any suffix p' of p, $SQL(p')$ will certainly contain all the results for Q_1. We also know that the "lossless from XML" constraint is satisfied. So, if we ensure that $SQL(p')$ does not have any results corresponding to some other path in the mapping schema, then we are done. We do this as follows: We first start with the smallest suffix $p' = \{<12>\}$. The equivalent query will be a scan of the InCat.category column. We notice that there is a path $q = \{<32>\}$ in the original schema S that has the same annotation. So, $SQL(p')$ will also return results corresponding to the path q. Since the queries corresponding to the paths p' and q have common results, we say that they are in *conflict* with each other. Now, since q does not have a corresponding path in S^1_{CP}, we know that it is not a part of the query result. This implies that for $p' = \{<12>\}$, $SQL(p')$ will have results corresponding to this path q as well.

So, we go up one level and set $p' = \{<11, 12>\}$. The same conflict persists with the path $q = \{<31, 32>\}$ and we have to increment p' by another level. Repeating this, we get to $p' = \{<3, 9, 11, 12>\}$. Now we see that the corresponding path $q = \{<8, 29, 31, 32>\}$ is not in conflict with p' due to the difference in the parentCode condition (edge annotations e_1 and e_6). In fact, there is no other path in conflict with p' elsewhere in the schema. So, the query $SQL(p')$ will be equivalent to $SQL(p)$ and it is a correct translation for Q_1. This query is given below:

```
select  C.eid
from    Item I, InCat C
where   I.id = C.parentid and I.parentCode=1
```

Contrast this with $SQL(p)$, which will be the relational query output by existing algorithms that do not use the "lossless from XML" constraint. $SQL(p)$ has an additional join with the Site relation, which has been removed using the mapping information. This leads us to the first important concept that we will use in developing an algorithm to exploit the "from XML" constraint while doing path expression to SQL translation.

Concept 1: For every path p in the result automaton S_{CP}, we need to identify a suffix p' that has the following property: SQL(p') will not return results corresponding to any path not appearing in the query result.

Now let us revisit query Q_1 from Section 2. The output of the PathId stage is the cross-product schema S^1_{CP} given in Figure 2. There are six satisfying paths in the schema (we will denote these p_1 to p_6). We need to find the shortest suffix for each of these paths that will together result in a correct SQL query. While we can handle each path independently in a fashion similar to query Q_1, we perform an additional optimization — we combine the queries for different paths whenever possible. In addition to grouping the SQL queries for paths with similar relational joins, this optimization also allows us to eliminate longer prefixes as we will see in this example.

Consider the path $p_1 =<1, 2, 3, 9, 11, 12>$. We start with the suffixes $p'_1 = \{<12>\}$ for this path. The path $q = \{<1, 2, 8, 29, 31, 32>\}$ is in conflict with p'_1. So, $SQL(p'_1)$ will have results corresponding to the path q. But this time q appears in S^1_{CP}, which means that it is a part of the query result (categories of South American items are a part of the query result). The corresponding suffix

is $q'_1 = \{<32>\}$. At this point, if we issue SQL queries for the two paths p'_1 and q'_1 separately, then we will get duplicate results. All the item categories will be returned twice. So, we need to go further up the tree for both the paths. On the other hand, if we issue a common query for the two paths, then we need not worry about the common results across these paths. In this case, since the two paths p'_1 and q'_1 have the same relation sequence (scan of the `InCat.Category column`), we can combine the queries for these two paths. Similarly, the other four schema nodes that have the annotation `InCat.category` are also part of the query result. So, the suffix $p'_1 = \{<12>\}$ suffices for the path p_1. We reach a similar decision for the other five paths. Finally, combining the queries for the six paths, we obtain the final relational query SQ_1^2 (see Section 2) that is a scan of the `InCat.Category` column.

Notice how by using the "lossless from XML" constraint, we are able to replace a query SQ_1^1 that was the union of six queries, each with 2 joins, by a simpler scan query SQ_1^2.

Concept 2: Suppose we are considering suffixes p' and q' for paths p and q respectively. We need not worry about the queries corresponding to the two suffixes generating common results as long as we issue a combined single SQL query for them.

From the above discussion, we see that by making sure that the above two concepts are satisfied we can find the required prefix for every path in the cross-product schema. Notice that we are able to do this only because the "lossless from XML" constraint holds. This constraint implies that there is a "one-to-one" correspondence between the relational data and the XML data. So, if we know that for a path p and a suffix p', SQL(p') satisfies concept 1, we are then guaranteed that it does not return any extra results. This holds because the mapping completely captures the relational data, i.e., all the root-to-leaf queries together represent the entire relational data. Similarly, concept 2 holds because the relational data stores values corresponding to each XML element separately, i.e., the result of no two root-to-leaf queries will have the value of the same XML element.

On the other hand, if we did not use the fact that the "lossless from XML" constraint holds for this instance, things will be a lot different. For example, suppose the join between `Item` and `InCat` relations was not a key foreign-key join. Then, the "lossless from XML" constraint may no longer be valid — for example, $RtoL(12)$ may return some tuples in the `InCat` relation multiple times as they join with several tuples in the `Item` relation. So, SQ_1^2 will no longer be a correct translation for query Q_1. The fact that the "lossless from XML" constraint is satisfied makes it a lot simpler to design a good query translation algorithm.

In the above examples, notice that we used the notions of combinability and conflict in the context of two paths in the mapping. We formally define these notions and then describe the pruning andSQLGen stages of the algorithm.

4.2 Terminology

We next introduce some terminology that will be used in the full specification of the translation algorithm. Whenever we refer to paths, we mean paths in the schema graph that end in leaf nodes.

Let $p = <n_1, \ldots, n_k>$ be a path in the schema graph, ending in a leaf node. We refer to n_k as $p.last$. Let RelSeq(p) denote the sequence of relations joined in SQL(p) in a top-down order. For example, for the path $p = <1, 2, 3, 9, 11, 12>$, RelSeq(p) = <Site,Item,InCat>.

We say that two paths p_1 and p_2 are *combinable* if the corresponding relation sequences $RelSeq(p_1)$ and $RelSeq(p_2)$ are the same and $Annot(p_1.last)$ and $Annot(p_2.last)$ are the same. Note that combinability is an equivalence relation. Combinable paths are useful in identifying when we can rewrite a union query, say (SQL(p_1) union all SQL(p_2)), as a SQL query without unions. For example, let $p' = <1, 2, 8, 29, 31, 32>$. Then the paths p and p' are combinable. From the "lossless from XML" constraint we know that the resulting query does not have to retain any duplicate results. This allows us to combine the two queries even if they have overlapping results.

Given two paths p_1 and p_2, we define when the two paths are in *conflict*. Intuitively, the two paths are in conflict if the result of SQL(p_1) and SQL(p_2) will have common results. Here, by common results, we refer to the two queries returning the value of a common element in the original XML document. For example, the paths p and p' are not in conflict as they return the categories of Africa items and South America items respectively. So, while the results of SQL(p) and SQL(p') may have common values, these will be the values of different elements in the original XML document. On the other hand, consider $p'' = <29, 31, 32>$. The two paths p and p'' are in conflict as the corresponding SQL queries overlap: the categories of africa items are common to the two query results.

Given two paths, p_1 and p_2, we say that they are in conflict if the following conditions hold.

- $RelSeq(p_1)$ is a suffix of $RelSeq(p_2)$ or vice versa.
- Without loss of generality, let $RelSeq(p_1)$ be a suffix of $RelSeq(p_2)$. Let p_3 be the longest suffix of p_2 such that $RelSeq(p_1) = RelSeq(p_3)$. Let $RelSeq(p_1) = RelSeq(p_3) = <R_1, \ldots, R_k>$. Then,
  there is no column $R_i.C$ such that $SQL(p_1)$ has a selection $R_i.C = val_1$, and $SQL(p_3)$ has a selection $R_i.C = val_3$, where $val_1 \neq val_3$.

The former condition checks if the two paths differ in the sequence of relations joined. If each sequence has a join not present in the other, they will not generate common results.

procedure Pruning(S_{CP},S)
begin
1. Let PathSet = $\{<n> | n$ is a leaf node in $S_{CP}\}$.
2. do
3. foreach ($p \in$ PathSet)
4. Let Conflict(p) denote the set of root-to-leaf paths in S that are in conflict with CurrPath(p)
5. If ($\exists p' \in Conflict(p)$ that does not match the query)
6. Increment p by one level
7. endFor
8. while (some path was modified in the previous iteration)
9. do
10. Let p and q be two paths in PathSet that are in conflict
11. If p and q are not combinable
12. Let RelSeq(p) be no longer than RelSeq(q)
13. Increment p by one level
14. while (some path was modified in the previous iteration)
15. Return $PathSet$
end

Figure 4. Pruning stage for tree mappings

We know this from the "lossless from XML" constraint. The latter condition checks if a conflicting edge annotation is present on any relation across the two paths; if so they will not generate common results and are not in conflict.

If p_1 is not in conflict with p_2, then we say that p_1 is *safe* from p_2. In the presence of the "lossless from XML" constraint, we know that no two root-to-leaf paths p_1 and p_2 have common results. In other words, root-to-leaf paths are always safe from each other. Observe that this may not be true if the integrity constraint is not satisfied.

4.3. The Pruning Stage

We present the pruning algorithm for translating path expression queries in Figure 4. Recall that the result of the PathId stage is S_{CP}, which represents all the matching paths in S. For each path, instead of constructing the SQL query from the root of the schema, our goal is to start bottom-up and stop at the lowest possible level.

So, for every path $p \in S_{CP}$, we start with just the leaf node and keep going up until we find a suffix p' that satisfies the following property:
For every path p'' in conflict with p'

- p'' appears in the cross-product schema S_{CP} and
- The queries corresponding to p' and p'' are combinable.

For every path p, steps 2-8 make sure that conflicts with paths not in the query result are resolved. Steps 9-14 make sure that duplicate results are not produced, i.e, conflicts with paths appearing in the query result are resolved, if the corresponding relation sequences are not combinable. We do the above computation for all paths in S_{CP} and stop when we have found the required suffixes for all of them. Since, each iteration in both the while loops will increment the length of the path, they will terminate in at most d iterations (combined), where d is the length of the longest path in S_{CP}.

The result of the pruning stage is a set of paths, `PathSet`, in the cross-product schema S_{CP}. We know that the set S_{CP} has all the schema paths matching the query. So, the union of the queries corresponding to all root-to-leaf paths in S_{CP} is a correct SQL translation. In the pruning stage, for each path p, we have removed some prefix of p and have the suffix path p' in PathSet instead of p. For correctness, we need to argue that the same results are returned if we use p' instead of p. Since the "lossless from XML" constraint is satisfied, it can be easily shown that every suffix p'' of p will return a superset of results, i.e., $SQL(p) \subseteq SQL(p'')$. So, we need to make sure that extra results are not returned. In the above algorithm, through additional checks we make sure that this does not happen, i.e., tuples corresponding to schema paths not matching the query are not returned and tuples corresponding to schema paths matching the query are returned exactly once. The first while loop in the algorithm takes care of the former and the second while loop takes care of the latter. These checks make use of the fact that the "lossless from XML" constraint is satisfied.

While incrementing a path by one level, we make the following optimization. If the edge we traverse to go up one level has an edge annotation on it, we split the operation of going up one level into two parts: (1) use the edge annotation to see if that suffices and (2) go up to the parent node, if necessary. By doing this optimization, we may be able to save a join operation. We will later show an example in Section SQLGen that uses this optimization.

4.4. The SQLGen Stage

The result of the pruning stage is a set of paths, `PathSet` in the cross-product schema S_{CP}. We partition this set based on the relation sequence and issue a SQL query for each equivalence class created. The final query is the union of the queries corresponding to all the partitions.

Suppose a single equivalence class C had two paths p_1 and p_2 in it. Since p_1 and p_2 are combinable, SQL(p_1) and SQL(p_2) involve the same relations. Thus, the from clause of the grouped query $SQL(C)$ contains these relations. Let C_{common} denote the set of conditions that are common to both SQL(p_1) and SQL(p_2). Let C_i denote the conditions corresponding to p_i that are not present in C_{common}. The where clause has the condition (C_{common} and (C_1 or C_2)). This solution generalizes when we have to combine more than two paths.

In the example queries we saw earlier, the second `while` loop is trivially satisfied — for Q_1 all the six paths in PathSet were combinable, while for Q_2 there was only one path

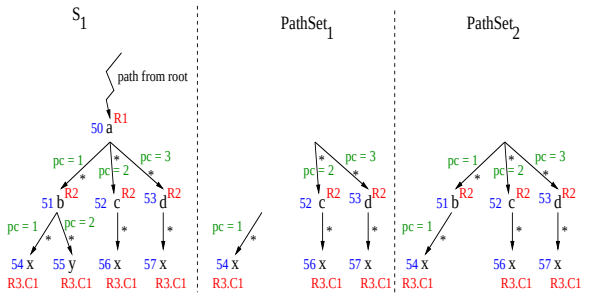

Figure 5. Example mapping S_1

in PathSet. We next present an example scenario where we find two paths that appear in the result, but are not combinable.

We use the example XML schema and mapping in Figure 5 in the following discussion. All `a` elements are stored in relation `R1`. The child elements are all stored in relation `R2`, with the `pc` column distinguishing between `b`, `c` and `d` children. Similarly, the children of `b` elements are stored in relation `R3`, with the `pc` column distinguishing between `x` and `y` children. The children of `c` and `d` elements are all stored in relation `R3`. Since all the children are `x` elements in these two cases, the `pc` column is not specified in these cases. An important point to note here is that since the value of the column is not specified for tuples corresponding to schema nodes 56 and 57, any value in the corresponding domain (including 1,2 and null) is allowed. As we will see later, this difference in the information available about the column `R3.pc` needs to be handled correctly when we attempt to find the correct suffix for each path.

Notice that we are only concerned with the subtree rooted at element `a` and this is only a portion of the entire schema. Let us assume that the element name `x` does not occur anywhere else in the XML schema. Also, no other leaf node in the schema is annotated with the column `R3.c1`.

Consider the evaluation of query Q_3 that returns all `x` elements, //x. The PathId stage will identify the three paths p_1, p_2 and p_3 ending in nodes 54,56 and 57 respectively. Suppose we execute the algorithm in Figure 4. After the first while loop (steps 2-8), we observe that the suffixes are $p'_1 = <51, 54>, p'_2 = <50, 52, 56>, p'_3 = <50, 53, 57>$. The only conflicting path p_4 ends in node 55. For p_1, the annotation on edge $<51, 54>$ makes it safe from p_4. So, going up one level was sufficient. Note that the edge annotation was sufficient in this case, and we did not need the join with relation R_1 (*annot*(51)). For p_2 and p_3, there was no annotation on the edges $<52, 56>$ and $<53, 57>$. So, we had to go up one more level till node 51. The annotation on edges $<50, 52>$ and $<50, 53>$ are different from the one on edge $<50, 51>$. Hence by going up two levels, p'_2 and p'_3 become safe from p_4.

Let us observe what happens if we skip the next while loop. The resulting PathSet is shown in Figure 5 as PathSet_1 and the corresponding SQL query SQ_3^1 is

```
select   R3.C1
from     R3
where    R3.pc = 1
union all
select   R3.C1
from     R2,R3
where    R2.id = R3.parentid and R2.pc IN {2,3}
```

Notice how values of `x` elements corresponding to schema nodes 56 and 57 may appear twice in the query result. This happens when the `pc` column of these tuples has a value of 1, which is valid as the XML-to-Relational mapping conveys no information about the value of the `pc` column of these tuples. So, the above query is not a correct translation for Q_1. While it returns the correct set of results, it may return duplicates.

In order to avoid generating duplicate results, we go up the schema further (steps 9-14 in algorithm 4) resulting in the set of paths PathSet_2 shown in Figure 5. This will result in the following correct SQL query SQ_3^2:

```
select   R3.C1
from     R2,R3
where    R2.id = R3.parentid and (R2.pc IN {2,3} or
             (R2.pc = 1 and R3.pc = 1))
```

The above example illustrates the point that we should make sure that two paths p' and q' that appear in the query result are not in conflict, if they are not combinable (i.e., if we are going to construct the queries for them separately). The second while loop (steps 9-14) in the algorithm takes care of this.

5. Exploiting the "lossless from XML" constraint for complex XML Schemas

In this section, we extend the techniques for exploiting the "lossless from XML" constraint to more complex XML schemas: directed acyclic graph (DAG) and recursive schemas. The outline of the algorithm remains the same as for tree XML schema (Figure 3). We need to augment the pruning stage to address some additional challenges that arise for complex XML schemas such as extending the notion of combinability to handle more complex XML schemas.

5.1. Combinability for Complex Schema

Consider the mapping shown in Figure 6. Suppose the query result is the set of paths shown in the figure. First of all, as shown in [9], enumerating all the matching paths may be expensive, as the number of paths may be exponential in the size of the DAG schema. So, the algorithm in [9] uses the *with* clause in SQL99 to represent common computation present in the DAG schema.

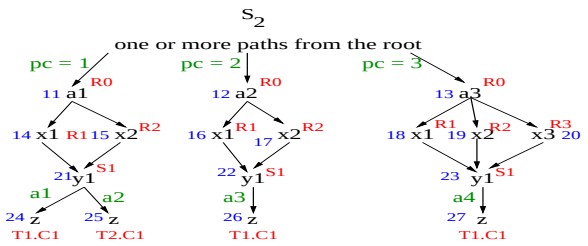

Figure 6. Example mapping S_2

For example, the paths $<11, 14, 21, 24>$, $<11, 14, 21, 25>$, $<11, 15, 21, 24>$ and $<11, 15, 21, 25>$ share some common computation as represented by the common schema nodes across them. So, the SQL fragment generated will be

```
with temp_21 as (
    select S1.*
    from  R0,R1,S1
    where R0.id = R1.parentid and R0.parentcode = 1
          and R1.id = S1.parentid
    union all
    select S1.*
    from  R0,R2,S1
    where R0.id = R2.parentid and R0.parentcode = 1
          and R2.id = S1.parentid
)
select   T1.C1
from     temp_21, T1
where    temp_21.id = T1.parentid and a1
union all
select   T2.C1
from     temp_21, T2
where    temp_21.id = T2.parentid and a2
```

Notice how the *with* clause is used to group together paths that have common computation. In a similar fashion, the *with* clause is used to handle the other two subtrees rooted at nodes 12 and 13 also. The final query result is the union of the three queries.

Let us revisit the definition of combinability of paths in this context. Suppose we want to combine the SQL queries corresponding to the subtrees rooted at nodes 11 and 12. This implies that we will have a single SQL query for the six paths. The following problem arises: while the structure of the *with* clause ($R0 \bowtie R1 \bowtie S1 \bigcup R0 \bowtie R2 \bowtie S1$) is common for the two subtrees, the subtree rooted at node 11 has two suffixes, edges $<21, 24>$ and $<21, 25>$. On the other hand, the subtree rooted at node 12 has only one suffix, edge $<22, 26>$. So, if we combine the *with* clause for the two subtrees in a simple fashion, we will generate two spurious paths $<12, 16, 22, 25>$ and $<12, 17, 22, 25>$. In order to avoid this, we need to somehow differentiate between tuples corresponding to nodes 21 and 22. One way to do this is to use the annotations on the incoming edges to the nodes 11 and 12. This helps us in differentiating between tuples corresponding to nodes 21 and 22 and the resulting query is:

```
with temp_21_22 as (
    select S1.*,R0.parentcode
    from  R0,R1,S1
    where R0.id = R1.parentid and R0.parentcode = 1
          and R1.id = S1.parentid
    union all
    select S1.*,R0.parentcode
    from  R0,R2,S1
    where R0.id = R2.parentid and R0.parentcode = 1
          and R2.id = S1.parentid
)
select   T1.C1
from     temp_21_22, T1
where    temp_21_22.id = T1.parentid and
           ((a1 and temp_21_22.parentcode = 1)
             or (a3 and temp_21_22.parentcode = 2))
union all
select   T2.C1
from     temp_21_22, T2
where    temp_21_22.id = T2.parentid and a2
           and temp_21_22.parentcode = 1
```

Notice how if the annotations `a1` and `a3` are different, we again need to differentiate between tuples corresponding to schema node 21 and 22.

Suppose we want to combine the SQL queries corresponding to the subtrees rooted at nodes 12 and 13 instead. In this case, notice how the path $<13, 20, 23, 27>$ has no equivalent path in the other subtree. So, we need to be careful and ensure that we do not create a spurious path $<13, 20, 23, 26>$ (through the join $R0 \bowtie R3 \bowtie S1 \bowtie T1$).

To summarize, from the above discussion we observe that for DAG schemas, we have a lot of options in defining when and how we combine the queries corresponding to different subtrees. Notice that whenever we have subtrees that have a different set of relation sequences (such as subtrees rooted at nodes 11,12 and 13), we have a choice: either combine the queries by maintaining some more state information or construct the queries separately. Whenever we need to maintain state information, we are also making the *where* clause of the SQL query complex. More importantly the relational query optimizer may have problems in optimizing the final SQL query efficiently, as it has no way of interpreting the semantics of this additional state information.

In this paper, we use a simple definition for combinability over complex schemas: one that requires minimal additional state information. Intuitively, we combine two subtrees if they are similar: the joins involved in each part of the resultant query are identical (the selection conditions can be different). This definition of combinability extends naturally to recursive schema also. Notice that this is a generalization of the definition of combinability in Section 4.2 for single paths. We next define this formally as follows.

A graph path gp corresponds to a subgraph of the schema. It is a concise way of representing a large number (possibly infinite) paths. Let Paths(gp) denote the set of

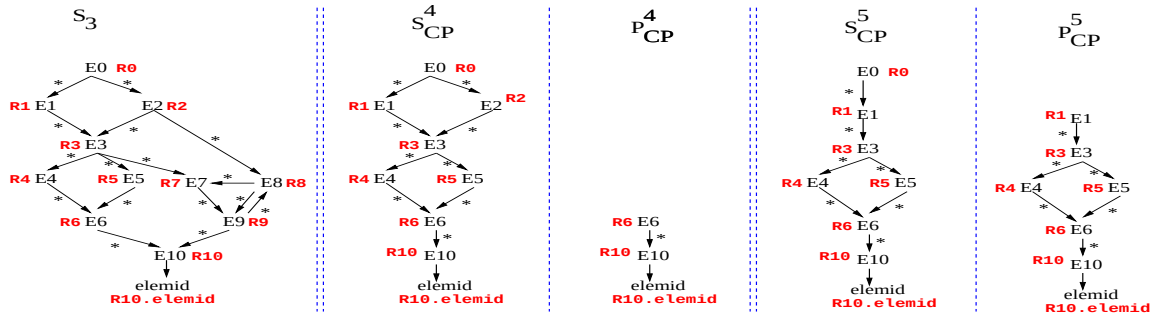

Figure 7. Example queries over recursive schema S_3

paths represented by the graph path.

For a graph path gp, let Template(gp) denote the corresponding graph constructed based on the relational annotations. The template graph represents the structure of the SQL query corresponding to the subtree. For example, Template(11) will be a graph with five nodes, one for each node in the subtree (since each edge traversal corresponds to a join operation). Each node will be labelled with the name of the corresponding relation. In this case, the nodes will be labelled with relation names R0,R1,R2,T1 and T2. For each edge $e \in gp$ that corresponds to a join between two relations, we add a corresponding edge in the template graph. Note that the template graph is similar to the original subtree; if the latter is recursive the template graph is recursive as well.

We say that two graph paths gp_1 and gp_2 are combinable if the corresponding templates are isomorphic.

Just like the tree schema case, the above definition makes use of the fact that as the "lossless from XML" constraint is satisfied, we can combine any two graph paths even if they have overlapping results. Since the result of a path expression query returns the values of all matching XML elements (exactly once), and there is a "one-to-one" correspondence between XML elements and relational tuples, this implies that no relational tuple will appear multiple times in the result of any equivalent SQL query.

5.2. The Pruning Stage

The pruning algorithm for recursive mappings is shown in Figure 8. While it looks very similar to the tree schema case, there are some important differences. These include the definition of combinability and conflict, and how we keep track of all the matching paths in PathSet.

We say that a path p is in *conflict* with a graph path gp if p is in conflict with some path $q \in$ Paths(gp).

Another important difference is that the set of matching paths is maintained as subgraphs of the cross-product schema (as there may be infinitely many of them if we enumerate them). Contrast this with the algorithm for tree schema where we explicitly keep track of all the paths. We

procedure Pruning(S_{CP},S)
begin
1. Let PathSet = $\{<n> | n$ is a leaf node in $S_{CP}\}$.
2. do
3. foreach ($p \in$ PathSet)
4. Let Conflict(p) denote the set of root-to-leaf paths in S that are in conflict with p
5. If ($\exists p' \in Conflict(p)$ that does not match the query)
6. Increment p by one level
7. endFor
8. while (some path was modified in the previous iteration)
9. do
10. Let p and q be two (graph) paths in S_{CP} that are not combinable
11. If $\exists$ path $q' \in q$ such that q' is in conflict with p
12. Let p' be the path $\in p$ that is in conflict with q'
13. Increment p' by one level
14. while (some path was modified in the previous iteration)
15. Return $PathSet$
end

Figure 8. Pruning stage for recursive mappings

describe how we increment the graph paths by one level (steps 6 and 13) through some examples later in this section.

Also note that in order to check the condition in Steps 4-5, we need to enumerate all the paths that do not appear in the query result. Since the query automaton for simple path expression queries is a deterministic finite automaton as constructed in [9], we can do this efficiently.

The SQLGen stage is similar to the original algorithm proposed in [9], with a slight modification. We combine the queries corresponding to two different graph paths in PathSet, if they are combinable. To do this, we partition the graph paths based on combinability (similar to Section 4.4) and construct the SQL query for each equivalence class in a fashion similar to [9].

We now explain some example query evaluations over the XML schema S_3 in Figure 7.

Consider query Q_4 =/E0//E6/E10/elemid. After the PathId stage, we obtain the cross product mapping S^4_{CP} shown in Figure 7. If we directly translate this into SQL, we will get a complex query involving two *with* clauses,

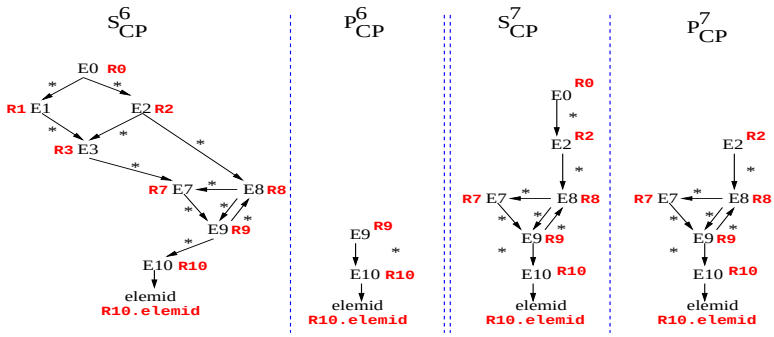

Figure 9. Example queries over recursive schema S_3

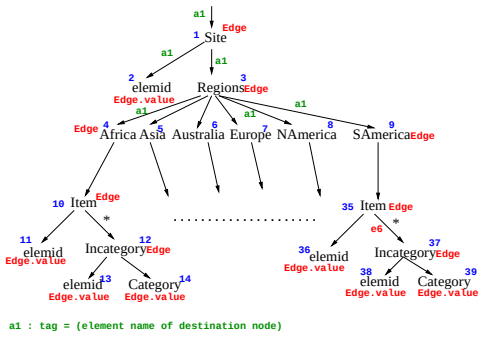

Figure 10. XMark schema mapped to the Edge relation

corresponding to elements `E3` and `E6`. On the other hand, by using the pruning algorithm in Figure 8, we obtain the pruned mapping P^4_{CP}. The corresponding SQL query is fairly simple. Let us look at how the algorithm worked in this case. We start with the single path p = `elemid` in PathSet. Since, path p_1 =<E0,E2,E3,E7,E9,E10,elemid> does not appear in the query result and is in conflict with p, we increment p by one level. The same conflict persists and so we go up one more level. Now, p =<E6,E10,elemid> and p_1 is no longer in conflict with p (they have different relation sequences now). So, we have completed steps 2-8 of the algorithm. Now since there is only one path left, steps 9-14 can be skipped and we return PathSet as the result.

Let us now consider Q_5 =/E0//E1//E6/E10/elemid. The result of the PathId stage, S^5_{CP}, is shown in the figure. Notice that there are two satisfying paths. Let us see what happens in the pruning stage. We start with the single path p = `elemid` in PathSet. Just like the previous query, we need to go two levels higher and p =<E6,E10,elemid>. Notice how while p_1 is no longer in conflict with p, the path p_2 =<E0,E2,E3,E4,E6,E10,elemid> is in conflict with p. Also, p_2 is not in the query result. So, we need to increment p by one more level. Element E6 has two parent nodes and so we go up along both paths. In the process, a single graph path p gets split into two graph paths p' and p'', rooted at nodes E4 and E5 respectively. p' is still in conflict with p_2, while p'' is in conflict with p_3 =<E0,E2,E3,E5,E6,E10,elemid>. So, we increment the paths, p' and p'', by one level each. The two paths are merged into one (say p rooted at E3), but the conflict with p_2 and p_3 persist. Finally, when we increment p by one more level to get the graph P^5_{CP} in the figure. This graph path is safe from both p_2 and p_3 (join with relation R1, instead of relation R2). Also, there are no other conflicting paths. Hence, P^5_{CP} is the result of the pruning stage.

We now consider some example queries that match recursive parts of the schema. Consider query Q_6 = /E0//E9//E10/elemid. The set of matching paths S^6_{CP} is shown in Figure 9. The result of the pruning stage P^6_{CP} is shown in the figure. Notice how the join between relations $R9$ and $R10$ suffices to make the path safe from all the paths not satisfying the query.

We consider query Q_7 = /E0/E2/E8//E10/elemid to illustrate what happens when we need to go up the schema and enter a recursive component. The result of PathId S^7_{CP} is shown in the figure. Notice how the edge <E3,E7> does not match the query. So, query results corresponding to all the paths that pass through this edge need to be avoided in the final SQL query. The pruned schema P^7_{CP} is shown in the figure. Notice how we have to go up the recursive component during step 6 of the algorithm. We start with `elemid` and go up two levels till E9. The next time we have to increment a level, we enter the recursive component (comprising of nodes E7,E8 and E9). Here, we use a simple algorithm to go up the schema: include the entire recursive component in one step. Finally, when we add the element E2, we can stop. In this particular case, we managed to save a single join operation with relation R0.

5.3. Schema-Oblivious Storage

The examples in the preceding sections may give the erroneous assumption that the optimizations discussed in this paper depend somehow upon the relational schema into which the documents are shredded reflecting a good deal of the XML schema for the document being shredded. In this section we show that this is not true — in fact, the "lossless from XML" constraint is useful even when the relational schema is generic and reflects nothing of the XML schema (a scenario we term "schema-oblivous storage.")

In schema-oblivious XML storage, the relational schema is fixed independent of the XML schema. This option may be chosen either because the XML schema may is unavailable during data load time or due to the fact that the XML schema changes frequently.

The Edge approach [7] is one example of schema-oblivious storage. Here, the input XML document is viewed

as a graph and each edge of the graph is represented as a tuple in a single table. This *Edge* relation has 5 columns, ID, PARENTID, TAG, ORDER AND VALUE.

During query translation time, let us assume that an XML schema is either given or has been inferred from the XML documents loaded into the system. For example, a sample XML-to-Relational mapping is shown in Figure 10. All the nodes are annotated with the same relation name `Edge`. All the edges have similar annotations. For example, an edge $e = u \rightarrow v$ has the annotation "tag = element_name(v)". Notice each edge traversal will translate into a join operation.

Since this input scenario satisfies the "lossless from XML" constraint, the query translation algorithms presented earlier in this paper are applicable. For example, the query Q_8 =/Site//Item//Category will translate into the following two-way join query over the `Edge` relation.

```
select  E2.value
from    Edge E1, Edge E2
where   E1.tag = 'InCategory' and E2.tag = 'Category'
        and E1.id = E2.parentid
```

The above query was obtained by exploiting the "lossless from XML" constraint. In contrast, if we use just the XML schema information, we will identify the six matching paths and the equivalent SQL query will be the union of six queries, one corresponding to each matching schema path (similar to query SQ_1^1 in Section 2). Each of these queries will be a join between six copies of the Edge relation. If we do not use the XML schema information at all (like the algorithm proposed in [7]), the resulting SQL query will be a recursive SQL query (due to the // in the XML query).

The above example illustrates how the "lossless from XML" constraint can be used to generate efficient SQL queries in a wide spectrum of scenarios: ranging from schema-based to schema-oblivious techniques.

6. Summary and Future Work

We considered the problem of generating efficient SQL queries from queries expressed over XML views and showed how the presence of a simple "lossless from XML" integrity constraint can help us in improving the quality of the final SQL queries produced. We then presented an algorithm to translate simple path expression queries over recursive XML schema by exploiting this single constraint. While some techniques have been proposed previously to achieve a similar goal for tree XML schema, they rely on using the underlying relational integrity constraints to reason about how we can simplify the SQL query. In contrast, the technique proposed in this paper is simple and does not rely on solutions to complex problems such as relational query containment and equivalence. Moreover it naturally extends to complex XML schema.

A number of avenues exist for future research including extending our techniques to more general class of XML queries and looking at alternate definitions of combinability for complex XML schema. Designing efficient data structures to speedup the various steps involved in our algorithm is important future work and a study of real-world schema will go a long way in helping this process. Also, looking at how relational schema design for XML storage can be more tightly coupled with query translation is an interesting problem.

Acknowledgement: This work was supported in part by NSF grant ITR-0086002.

References

[1] Naa classified advertising standards task force. `http://www.naa.org/technology/clsstdtf/`.

[2] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *SIGMOD*, 2001.

[3] A. Deutsch and V. Tannen. Containment and Integrity Constraints for XPath Fragments. In *KRDB*, 2001.

[4] A. Deutsch and V. Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *VLDB*, 2003.

[5] M. Fernandez, A. Morishima, and D. Suciu. Efficient Evaluation of XML Middle-ware Queries. In *SIGMOD*, 2002.

[6] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of Xpath queries. In *VLDB*, 2003.

[7] D. Florescu and D. Kossman. Storing and Querying XML Data using an RDBMS. *Data Engineering Bulletin*, 22(3), 1999.

[8] S. Jain, R. Mahajan, and D. Suciu. Translating XSLT Programs to Efficient SQL Queries. In *WWW*, 2002.

[9] R. Krishnamurthy, V. Chakaravarthy, R. Kaushik, and J. F. Naughton. Recursive XML Schemas, Recursive XML Queries, and Relational Storage: XML-to-SQL Query Translation. In *ICDE*, 2004.

[10] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. Efficient XML-to-SQL Query Translation: Where to Add the Intelligence? In *VLDB*, pages 144–155, 2004.

[11] G. Miklau and D. Suciu. Containment and equivalence for an xpath fragment. In *PODS*, 2002.

[12] P. Ramanan. Efficient algorithms for minimizing tree pattern queries. In *SIGMOD*, 2002.

[13] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML Views of Relational Data. In *VLDB*, 2001.

[14] P. T. Wood. Containment for XPath Fragments under DTD Constraints. In *ICDT*, 2003.

[15] Xmark: The xml benchmark project. http://monetdb.cwi.nl/xml/index.html.

[16] DB2 XML Extender. http://www-3.ibm.com/software/data/db2/extenders/xmlext/index.html.

[17] Oracle XML DB. http://otn.oracle.com/tech/xml/xmldb.

[18] SQLXML and XML Mapping Technologies. http://msdn.microsoft.com/sqlxml/default.asp.