Contact Author	*Rajasekar Krishnamurthy*
Email	*sekar@cs.wisc.edu*
Address	*Department of Computer Sciences* *University of Wisconsin - Madison* *1210, W. Dayton Street* *Madison, WI 53706*
Phone	*(608) 262 6624*
Paper ID	*518*
Topic Area	*Core Database Technology*
Category	*Research*
Title	*Optimizing Fixed-Schema XML to SQL Query Translation*
Authors	*Rajasekar Krishnamurthy, Raghav Kaushik, Jeffrey F. Naughton*
Topics	*Semi-structured Data, XML* *Optimization and Performance*

Optimizing Fixed-Schema XML to SQL Query Translation

Rajasekar Krishnamurthy   Raghav Kaushik   Jeffrey F. Naughton

University of Wisconsin-Madison
{sekar, raghav, naughton}@cs.wisc.edu

Abstract

Recently, there has been a lot of work on evaluating XML queries over data stored in relational database systems. The vast majority of this work has focused on the cases where either the relational schema is not fixed (so the problem is to find a good relational schema for a given XML workload) or the XML schema is not fixed (so the problem is to develop generic strategies for exporting XML views of relational data). While these cases are interesting, in practice a third scenario, in which both the source relational and target XML schemas are fixed, seems highly relevant. We show that even in this highly constrained environment, there is a lot of freedom in the SQL that can be generated to evaluate a given XML query. Furthermore, we show through experiments with a commercial RDBMS that by exploiting the underlying relational constraints and the properties of a given XML to relational schema mapping, it is possible to generate SQL queries that perform an order of magnitude better than those generated by more naive translations.

1 Introduction

Currently there is a lot of interest in using relational database systems to evaluate XML queries over XML documents. Most of the work to date on this topic falls into one of two categories. In the first category, the XML schema is fixed, but the relational schema is not. For work in this category, the natural focus has been on how to choose a good relational schema corresponding to the XML schema. In the second category, the relational schema is fixed, but the XML schema is not. For work in this category, the natural focus has been on how to support generic XML views of relational data. In this paper we focus on a third category, that in which both the relational and XML schemas are fixed. We expect this scenario to have substantial applicability in practice, as in many cases the relational schema is defined by the organization storing the data, while the XML schema is defined by a standards group. In such a scenario, a new focus emerges: how to generate efficient SQL queries given the (fixed) XML to relational mapping.

Consider a relational schema having Department and Faculty relations. The query "Find the number of faculty associated with a parent department" can be written in SQL as

```
Select count(*)
From Department D, Faculty F
Where D.did = F.did
```

Now suppose that the schema already enforces the condition that each faculty has to be associated with a department (through foreign key and not-null constraints on Faculty.did column). Then the relational optimizer can (and does) optimize this query to "Select count(*) From Faculty". The above example shows how the constraints on the underlying data can be used during query optimization.

In this paper, we show that for XML queries translated to SQL, a larger set of more complex constraints can be used to identify and eliminate redundant computation. The huge difference between the XML and relational data models and the corresponding query languages creates a number of such scenarios. This is accentuated by the presence of wild cards in XML queries. Current relational optimizers, in general, do not use complex constraints during query optimization since such situations are less common under traditional workloads.

**Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002**

One way to use existing relational optimizers is to utilize these constraints while translating XML queries into SQL, instead of while optimizing the SQL queries. In this paper, we consider a large and interesting class of fixed-schema mappings, which we call *well-formed* mappings. In this domain, we show how these complex constraints can be exploited in a natural manner while translating the XML query into SQL to obtain a *constraint-aware* query translation. Our experiments show that this approach yields significant performance speedups over the naive approach of replacing the wildcard with all possible satisfying paths and issuing individual queries for each satisfying path.

To summarize, our contributions are the following.

- A formal framework to identify opportunities for *constraint-aware* XML-SQL translation.
- A *constraint-aware* query translation algorithm for branching path expression queries including those involving wild cards.
- Experiments on two data sets that show significant performance benefits due to our *constraint-aware* translation algorithm.

The rest of the paper is organized as follows: We first illustrate the benefits of performing a *constraint-aware* translation using queries on a sample industry standard XML schema for advertisements [14] in Section 2. Next we present a formal framework for representing the XML-to-relational mapping in Section 3 and describe the class of *well-formed* mappings. We then present the *constraint-aware* query translation procedure for path expression queries in Section 4. We give experiment results, in Section 5, that show the significant performance improvement that can be obtained by using our approach. We discuss related work in Section 6 and present our conclusions in Section 7.

2 Motivation

Let us consider the example XML schema in Figure 1 based on the standard DTD being developed by the Newspaper Association of America Classified Advertising Standards Task Force [14]. This standard is intended to pave the way for aggregation of classified ads among publishers on the Internet, as well as to enhance the development of classified processing systems. The DTD has over 350 elements and several hundred attributes. Each of the elements in the DTD has several attributes and subelements that are not shown in the figure.

Each XML document conforming to this DTD contains information about one or more advertisements. The body element has information about the actual ad. The ad-instance element contains information about an instance of the ad. The ads are classified into three main categories: employment, real-estate and transportation, and each ad will belong to exactly one of these categories. The real-estate element is shown in some detail in Figure 1. Real-estate is classified further into seven categories based on the nature of the property to be rented or sold. Similarly, employment ads are classified into four categories and transportation ads are classified into nine categories.

A sample mapping σ between the XML DTD and the relational schema is given in Figure 2. Each node in the schema has a table name next to it, to indicate the relational table that corresponds to this element. In practice, the mapping information includes other information like the column to which the element is mapped, join condition across relations and so on. We look at this in more detail in Section 3. The ad instances are all stored in an *ADS* relation. Real estate ads are stored in the *RE* relation. The information separating various categories of real estate is stored in the *type* column. The transportation ads and employment ads are stored in the *TRANS* and *EMP* relations respectively. Information about the location of all ads is stored in a single *ADINSTANCE-LOC* relation. The following constraints hold on the data.

1. *Key constraints:* The columns ADEX.id, ADS.id, RE.ad-id, EMP.ad-id and TRANS.ad-id are keys in their respective relations.

2. *Foreign key constraints:* The column ADS.adexid is a foreign key pointing to ADEX.id. The columns RE.ad-id, EMP.ad-id and TRANS.ad-id are foreign keys pointing to ADS.id. Similarly, the ADINSTANCE-LOC.ad-id column points to ADS.id, and is non-nullable.

3. *Domain constraints:* The *category* column of the RE, EMP and TRANS tables can take one of seven, four and nine values respectively.

4. *Multi-relation constraints:* Each ad is exactly one of Real-Estate, Employment or Transportation. This means that the values in the columns RE.ad-id, EMP.ad-id and TRANS.ad-id are unique, not just within the same table, but *across* all the three. Moreover, they together exhaust the values in ADS.id.

Consider the following XML query *XQ*.

```
Find the number of advertisements
corresponding to the campus area
 count(//adinstance[//geographicarea='campus'])
```

Current techniques, in systems like Xperanto [12], identify all matching paths for the wild cards, issue a separate SQL query for each such path and union the results. The resulting query *SQ* is shown below. The 20 unions shown correspond to the 20 matching paths (seven through real-estate, four through employment and nine through transportation).

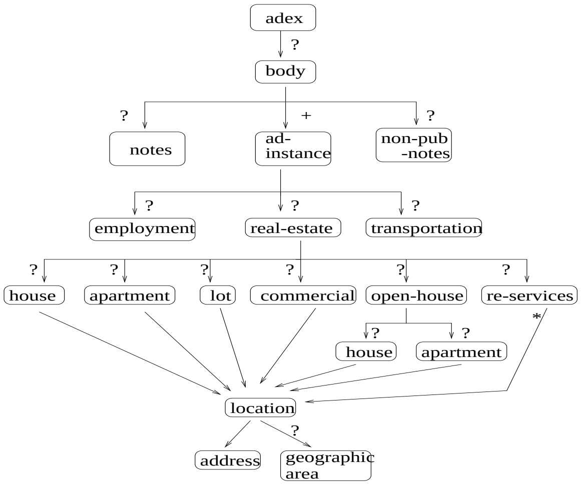

Figure 1: **ADEX DTD for newspaper advertisements**

```
with tmp(loc) as
  (select ADINSTANCE-LOC.ad-id
   from   ADEX, ADS, RE, ADINSTANCE-LOC
   where  ADEX.id = ADS.adexid
          and ADS.id = RE.ad-id
          and RE.category = 'house'
          and RE.ad-id = ADINSTANCE-LOC.ad-id
          and ADINSTANCE-LOC.area = 'campus'
   union ... (20 times)
  ) select count(*) from tmp
```

The relational optimizer uses foreign key constraints to eliminate redundant joins whenever possible. In this case, for instance, the join between ADEX and ADS can be removed. Furthermore, the join between ADS and RE becomes redundant and can be removed. The rewritten query SQ_1 is

```
with tmp(loc) as
  (select ADINSTANCE-LOC.ad-id
   from   RE, ADINSTANCE-LOC
   where  RE.category = 'house'
          and RE.ad-id = ADINSTANCE-LOC.ad-id
          and ADINSTANCE-LOC.area = 'campus'
   union ... (20 times)
  ) select count(*) from tmp
```

However, we can optimize this query further by grouping some of the unions in the following manner. There are seven paths through real-estate. The corresponding queries are on RE and ADINSTANCE-LOC relations. They differ only in the selection condition on RE.category. These seven paths can be *grouped* into a single SQL query having a disjunctive condition on the RE.category column. By using the domain constraint on this column, we can remove this (disjunctive) condition since it exhausts the domain. Similar optimizations can be performed for paths going through employment and transportation. This rewritten query SQ_2 is shown below.

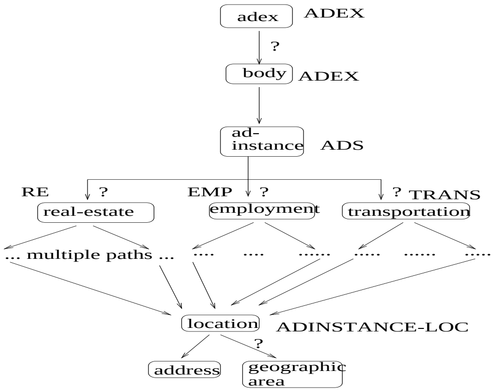

Figure 2: **Relational mapping σ for different location elements**

```
with tmp(loc) as
  (select ADINSTANCE-LOC.ad-id
   from   RE, ADINSTANCE-LOC
   where  RE.ad-id = ADINSTANCE-LOC.ad-id
          and ADINSTANCE-LOC.area = 'campus'
   union
   select ADINSTANCE-LOC.ad-id
   from   EMP, ADINSTANCE-LOC
   where  EMP.ad-id = ADINSTANCE-LOC.ad-id
          and ADINSTANCE-LOC.area = 'campus'
   union
   select ADINSTANCE-LOC.ad-id
   from   TRANS, ADINSTANCE-LOC
   where  TRANS.ad-id = ADINSTANCE-LOC.ad-id
          and ADINSTANCE-LOC.area = 'campus'
  ) select count(*) from tmp
```

We can actually do even better for this query. The multi-relation constraint together with the foreign key constraint involving the ADINSTANCE-LOC.ad-id column means that this column points to exactly one of the three ad-id columns in the tables EMP, RE and TRANS. Thus, we obtain the query *optQ* given below.

```
select count(*)
from   ADINSTANCE-LOC
where  ADINSTANCE-LOC.area = 'campus'
```

We observe from our experiments using DB2 that the optimizer generates a plan corresponding to the query SQ_1 after using the foreign key constraints alone. We also observe that the query *optQ* yields a speedup of a factor of 42 over this plan as we show in Section 5. This clearly shows that for path expression queries, especially involving wild cards, the performance benefit obtained by exploiting more complex constraints, such as the multi-relation constraint above, is significant.

There are two ways to do this. First, we could increase the set of constraints supported by RDBMS, and extend their optimizers to optimize the generated SQL using these constraints. Second, we could make use of this information outside of the RDBMS in order to generate the optimized SQL. The first approach adds significant complexity to already complex systems, so the second is an attractive option. However, our techniques and results apply in both approaches. In this paper, we show how these complex constraints can be exploited in a natural manner while translating the XML queries into SQL. For the above query, using our *constraint-aware* algorithm results in *optQ*. Since current query translation techniques do not use any constraints in the translation, we refer to them as *constraint-oblivious* algorithms.

3 Formal Model for fixed-schema mappings

Here, we provide a formal model to describe the class of fixed-schema mappings between XML and RDBMS considered in this paper.

3.1 XML Schema graph

An XML schema can be viewed as a directed graph $\mathcal{T} = (V, E)$, where V is the set of vertices and E is the set of edges. The vertices correspond to the types of elements and attributes and the edges represent containment(parent-child) relationships. The vertices are labelled with the name of the type and the edges are labelled with the name of the element or attribute. The edges have an additional multiplicity label that can take a value from $\{?, *, +, \epsilon\}$. A sample schema graph is given in Figure 3. Here the vertex labels are omitted for clarity. In this paper, we do not consider recursion in the schema. Thus, our schema graphs are directed acyclic graphs (DAG schema graph).

If the schema graph is a tree, then we call it a *Tree* schema graph. A DAG schema graph can be converted into a *Tree* schema graph by replicating subtrees rooted at vertices having multiple incoming edges. Hence, without loss of generality, we consider *tree* schema graphs. Note that the above conversion could increase the size of the resulting tree schema graph. We discuss this issue in Section 4.3.1.

3.2 XML-to-relational mapping

The mapping between the XML elements and the relational columns is represented through annotations on the schema graph. This is one way of defining an XML view of the RDBMS. A similar approach is being used in Microsoft SQLServer where XML views can be created using annotated XDR schemas [15] and in XML-DBMS [16], a middleware system for transferring data between XML documents and relational databases.

To formally describe these schema annotations, consider the following way of defining XML views over relational databases. In this method, proposed in [10], a default view is defined over the relational database and the XML view is defined as an XML query (like XQuery) over the default view. The default view maps each relation R to an element with name R. Each row of the relation R is mapped to a row element under element R. This row element has one child element for each column of relation R.

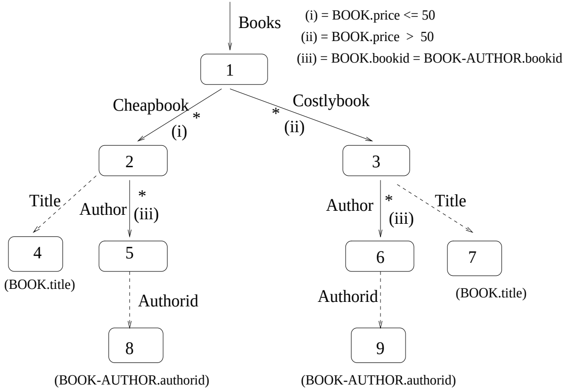

Figure 3: **Sample schema graph**

For example, consider the annotations on the schema graph in Figure 3. It represents a set of books and the corresponding authorids nested under each book. Let the underlying relational schema contain the BOOK(bookid,title,price) and BOOK-AUTHOR(bookid,authorid) relations. Here bookid is the key for the BOOK relation and (bookid,authorid) is the key for the BOOK-AUTHOR relation. The corresponding XQuery (over the default view) is shown in Figure 4.

```
create view  book-db as (
<Books>
  for $book in  view("default")/BOOK/row
  where $book/price ≤ 50
  return
     <Cheapbook Title = $book/title>
        for $author in  view("default")/
                              BOOK-AUTHOR/row
        where $book/bookid = $author/bookid
        return
           <author Authorid=$author/authorid/ >
     < /Cheapbook>
  for $book in  view("default")/BOOK/row
  where $book/price > 50
  return....
  .                       .
  .                       .
  .                       .
</Books> )
```

Figure 4: **Mapping expressed as an XQuery over default view**

Consider a top-down evaluation of this query. A Cheapbook element is created for every tuple of the BOOK relation that satisfies the condition price ≤ 50. We represent this selection condition as an annotation on the edge from node 1 to node 2 (called $e_{1,2}$) in Figure 3. A similar condition appears as an annotation on the edge $e_{1,3}$. The title column of such a book tuple is exported as an attribute. This is represented by storing the column name (of course, along with the relation name) as an annotation on node 4. In the view definition, for every Cheapbook element, the set of its authorids is exported by joining the BOOK and BOOK-AUTHOR tables. This join condition annotates $e_{2,5}$. The rest of the annotations are identified in a similar fashion.

The notation R.C is used to represent column C of relation R. More formally, the following edge annotations are allowed.

1. *Initial annotations:* An edge between the root of the schema and its children can have the annotation $<R>$. This indicates that there is an element in the XML view for each tuple in the relation R.
2. $<R.C$ op value$>$: a selection condition on column $R.C$
3. $<R_p.C_p, R_c.C_c>$: an equijoin between the two columns.
4. Null annotation (to allow dummy elements).

The annotation of an edge e is called *annot*(e). The relations in the edge annotations are not allowed to be arbitrary. We associate a relation $Rel(n)$ with every schema node n in a top down fashion as follows. Let the (unique) in-coming edge into n be $e_{p,n}$. If $annot(e_{p,n})$ is a (1) initialization condition $<R>$(here p has to be the root(T)), then $Rel(n) = R$, (2) selection condition involving the column $R.C$, then $Rel(n) = R$, (3) a join condition of the form $<R_p.C_p, R_c.C_c>$, then $Rel(n) = R_c$ and (4) a null annotation, then $Rel(n) = Rel(p)$.

The relations associated with the schema nodes and the edge annotations together must satisfy the following constraints. For an edge $e_{p,n}$, if $annot(e_{p,n})$ is (1) a selection condition involving the column $R.C$, then $Rel(p) = R$, (2) a join condition of the form $<R_p.C_p, R_c.C_c>$, then $Rel(p) = R_p$.

Node annotations are allowed only on leaf nodes. For a node n, $annot(n)$ is a column in the relation $Rel(n)$ and indicates that the values in this column are exported (in the XML view) as values of elements that correspond to the schema node n. We note here that we can extend the above class of mappings to allow extra features like, for instance, internal nodes having mixed content. Our techniques extend easily to cover such extensions. We do not discuss them for ease of exposition.

Consider the query //Author/Authorid. A naive translation will yield the following SQL query

```
select BA.authorid
from BOOK-AUTHOR BA, BOOK B
where B.bookid = BA.bookid and B.price <= 50
union
select BA.authorid
from BOOK-AUTHOR BA, BOOK B
where B.bookid = BA.bookid and B.price > 50
```

Given that BOOK-AUTHOR.bookid is a non-nullable foreign key pointing to BOOK.bookid, and bookid is a key for relation BOOK, a simpler equivalent SQL query is

```
select authorid
from BOOK-AUTHOR
```

The above optimization can be done in three steps: (i) the two branches are grouped into one by changing the union into a disjunctive condition (price ≤ 50) *vee* (price > 50), (ii) since this condition is redundant, it is removed and (iii) the join between BOOK and BOOK-AUTHOR is redundant due to the foreign key constraint and so, is removed. Current relational optimizers do not make the above optimization, for a valid reason — to group the two branches of the union query into a disjunctive condition, one needs to make sure that a single BOOK-AUTHOR tuple will contribute to the query result in at most one branch. This reduces to verifying whether the results of the two branches have a null intersection, which is an expensive operation to perform at query optimization time. We note that in the absence of the above foreign key constraint, the simplified query is an incorrect translation. Thus, it is important to identify opportunities for such simplifications before hand. We do this by defining the notion of *well-formed* mappings.

The idea is to check whether all the relational data is exported "exactly once". Without loss of generality, we can assume that if a relational column occurs in an edge annotation, then its value is explicitly exported as an attribute or leaf element. If not, we add a dummy attribute to achieve this purpose. For example, in Figure 3, the BOOK.price column appears in an edge annotation, but is not explicitly exported. To handle this, we add a dummy price attribute to the Cheapbook and Costlybook elements. These dummy attributes are used only to check well-formedness of a mapping and can be safely ignored during query translation. In a similar fashion, we add dummy attributes corresponding to relational columns (and tables) that do not appear in any annotation.

In order to formalize the notion of "exactly once", we associate an SQL query $SQL(n)$ with every leaf node n of the schema graph. This is the query obtained by joining the relations associated with the nodes on the root-leaf path to n, using the edge annotations as the "where" clause. The column *annot*(n) is projected. For example, $SQL(node(8))$ in Figure 3, is

```
select BA.authorid
from BOOK-AUTHOR BA, BOOK B
where B.price <= 50 and BA.bookid = B.bookid
```

For a relational column $R.C$, let $nodes(R.C) = \{n : n$ is a leaf node and $annot$(n) $= R.C\}$.

DEFINITION 1 *A mapping is said to be* well-formed *if for each relational column R.C:*

- *At-least-once mapping:*
  "select C from R" $\subseteq \cup_{n \in nodes(R.C)} SQL(n)$. *We say that the set of nodes, $nodes(R.C)$, covers R.C.*

- *At-most-once mapping:*
  $\cup_{n \in nodes(R.C)} SQL(n) \subseteq$ *"select C from R"*

The containment operations are under multi-set semantics.

If a column $R.C$ is non-nullable and is covered by a set of nodes S, then S is said to cover the relation R.

3.3 Verifying whether given mapping is *well-formed*

In this section, we discuss how to check if a mapping is *well-formed*. Notice that this check needs to be performed only once, when the mapping is defined, as a preprocessing step.

3.3.1 At-Most-once mapping

Notice that checking whether the mapping satisfies the at-most-once condition is equivalent to the following. For each column $R.C$, for any two nodes $n_1, n_2 \in nodes(R.C)$, the intersection of the results of SQL'(n_1) and SQL'(n_2) is empty. Here $SQL'(n)$ is the same as $SQL(n)$ except that the key column(s) of R is (are) also projected. In Section 4.1.1, we give a simple algorithm to check this condition with a one-sided error (i.e., if the intersection is not empty, the algorithm will always identify it correctly). As a result, we identify mappings that do not satisfy the "at-most-once" property correctly.

3.3.2 At-Least-once mapping

We next see how we can verify whether the given mapping satisfies the at-least-once constraint. We illustrate our algorithm with an example, and omit the details as they are straightforward but tedious.

Let us assume that advertisement data was stored in the relational database as shown in Figure 2. The ad-instance elements correspond to the ADS relation and real-estate elements correspond to the RE relation. Let us assume that we know that the ad-instance node covers the ADS relation. The join condition on the edge between the ad-instance and real-estate elements is "ADS.ad-id = RE.ad-id" (not shown in figure). Since a key-foreign key constraint has been specified on these columns, it implies that tuples in the RE relation, with a non-null value for RE.ad-id, are all exported as real-estate elements. Furthermore, since the RE.ad-id column is designated to be a "non-nullable" column, this implies that the RE relation is covered by the real-estate node. In a similar fashion, we can verify that the EMP and TRANS relations are also covered.

Let us now look at the subtree below the real-estate element. A real-estate ad can be one of seven types and the condition on each of the edges from real-estate will be of the form "RE.category = value". Since the domain of the RE.category column is defined to be these seven values, we can infer that the node set S_1, comprising the nodes corresponding to house, apartment, ..., re-services, covers the RE relation. Similarly, the node set S_2, comprising the four different categories of employment ads, covers the EMP relation and the nodeset S_3, corresponding to the nine categories of transportation ads, covers the TRANS relation.

Now for the location subelement, which is mapped to the ADINSTANCE-LOC relation, the join condition on all its incoming edges will be of the form "R.ad-id = ADINSTANCE-LOC.ad-id". Here R is one of RE, EMP or TRANS. Utilizing the multi-relation constraint that we mentioned in Section 2, we identify the fact that the nodeset $S = (S_1 \cup S_2 \cup S_3)$ covers the ADS relation. Combining this with the foreign key relationship between ADINSTANCE-LOC.ad-id and ADS.id, and the non-nullable condition on ADINSTANCE-LOC.ad-id, we infer that ADINSTANCE-LOC relation is covered by S.

4 Query Translation

In this section, we present our *constraint-aware* query translation algorithm for the class of branching path expression queries, when the XML-to-relational mapping is *well-formed*. We first discuss the algorithm for simple path expressions (i.e., no branches) and then extend it to the case of more complicated branching path expressions. We then discuss extensions to our algorithms in a more general setting.

4.1 Simple Path Expressions

A simple path expression is one of the form "s_1 l_1 s_2 $l_2 \ldots s_k$ l_k" where each l_i is a tag name and each s_i is either / or // denoting respectively parent-child and ancestor-descendant traversal. The semantics are the XPath [3] semantics.

Evaluating a path expression query can be viewed as a two stage process: (i) using the XML query to identify the vertices in the schema graph that satisfy the query and (ii) using the annotations to construct the equivalent relational query. We refer to these stages as the *NodeIdentification* and *RelationalQueryGeneration* stages respectively.

The *NodeIdentification* stage can be implemented by any vanilla algorithm. Let S be the set of schema nodes returned by this stage. For purposes of exposition, we assume that S consists of leaf nodes. The discussion naturally extends to the case when S contains some non-leaf nodes.

A simple *constraint-oblivious* algorithm is to return the SQL query, $\bigcup_{n \in S} SQL(n)$. In our *constraint-aware* algorithm, we utilize the relational constraints to perform a more efficient translation in the *RelationalQueryGeneration* stage. This consists of two broad parts: eliminating unnecessary path prefixes and grouping the resulting queries whenever they are on the same set of relations. The algorithm is given below. We explain the prefix-elimination and grouping stages in the next two sections.

```
procedure QueryTranslation(Q)
begin
    Let $S \leftarrow NodeIdentification(Q)$
    Prefix-Elimination($S$)
    Return Grouping($S$)
end
```

4.1.1 Eliminating Redundant Prefixes

A node sequence NS is a sequence of nodes in the schema graph that correspond to a path in the schema graph starting from the node $NS.first$ and terminating in the leaf node $n = NS.last$. The relational query, Query(NS) is obtained by combining the conditions on the edges of the sequence and projecting *annot*(n) and the key of *Rel*(n). The relational query Query'(NS) is the same as Query(NS), except that the key column(s) of *Rel*(n) is (are) also projected. Query(NS) and Query'(NS) are always conjunctive queries.

Let u and v be two leaf nodes in the schema such that *annot*(u) = *annot*(v) = $R.C$. Let node sequence NS = <root= $n_1, n_2, \ldots, n_k = u$> be the node-sequence representing the root-leaf path to u.

Definition 2 *The node n_j is a distinguishing ancestor for u against v if the intersection of the results of queries Query'(<$n_j, \ldots, n_k = u$>) and SQL'(v) is empty — we then say that $da(u, v, n_j)$ is true.*

Definition 3 *The lowest distinguishing ancestor, lda(u,v) is the lowest ancestor w of u such that $da(u, v, w)$ holds.*

Notice that for a well-formed mapping and any two leaf nodes, u and v, $da(u, v, root(\mathcal{T}))$ always holds. We also have the following lemma. We omit the proof for lack of space.

Lemma 1 *Let w be the least common ancestor of two leaf nodes u and v in the schema graph. Then, $da(u, v, w)$ holds.*

For node sequence NS, let RelSeq(NS) denote the sequence of relations that are involved in Query'(NS) in a bottom-up order. For example, for NS_1 =<1,2,5,8>, RelSeq(NS_1) = <BOOK-AUTHOR,BOOK>. Notice that successive relations, R_1 and R_2, in this sequence are related by a join condition on the path, say Condition(R_1, R_2). A relation R in this sequence is also associated with a (possibly null) selection condition, Condition(R), which is the conjunct of multiple conditions on the edges. For example, BOOK is associated with the condition BOOK.price $\leq$ 50. For $S = RelSeq(NS)$, we use Query(S) as a shorthand for Query(NS), when the context is clear.

We now discuss our algorithm for finding *lda*(u,v). The algorithm is shown in Figure 5. The idea is to process the ancestors of u in a bottom-up order. For each ancestor curr-anc, we check whether the "prefix" of SQL'(u) ending in curr-anc can be eliminated. This can be done if the intersection of results of Query'(curr-seq) and SQL'(v) is empty. Here curr-seq is the node sequence from curr-anc to u. This condition is tested by assuming the contrary and attempting to arrive at a contradiction by chasing the functional dependencies and join conditions [7] and remembering the selection conditions. At the end, we check if any of the constraints are violated. If so, then the intersection is indeed empty and curr-anc is the least ancestor of u when this happens. Otherwise, we move on to the parent of curr-anc. Since the least common ancestor w of u and v satisfies $da(u, v, w)$, this process will return w or one of its descendants.

```
procedure lda(u,v)
begin
   Let NS-v← node sequence for root-v path
   Let $SSeq \leftarrow RelSeq$(NS-v) =<$S_1, \ldots, S_l$>
   Let curr-anc ← $u$
   Let curr-seq ←<$u$>
   while true do
      Let $RSeq \leftarrow RelSeq$(curr-seq) =<$R_1, \ldots, R_k$>
      //Note that $R_1 = S_1$
      //check whether Query'(curr-seq) and Query'(NS-v)
      //    have empty intersection
      Equate the projected columns of $R_1$ and $S_1$
      Chase functional dependencies in $R_1$ and $S_1$
      Associate selection conditions in Condition($R_1$)
         with the corresponding columns of $R_1$
      Do similarly for $S_1$
      for $r = 2$ to $k$ do
         equate the variables corresponding to joining
            columns of $R_{r-1}$ and $R_r$
         chase the fds in $R_r$
         associate appropriate selection conditions
            to columns in $R_r$
      do similarly for $S_2, \ldots, S_l$
      if any constraints are violated return curr-anc
      if curr-anc = root(T) return null
      curr-anc ← parent of curr-anc
      prepend curr-anc to curr-seq
end
```

Figure 5: **Algorithm for determining *lda*(u,v)**

Some ways in which constraints can be violated are:

1. A key constraint is violated, for example, when a relation has the same key value in the two node sequences but the selection conditions associated with some column place conflicting conditions.

2. A multi-relation constraint is violated, for example, when the columns of two relations are constrained to be disjoint, but the chase implies that they are equal.

For example, consider the computation of *lda*(8,9), for the schema in Figure 3, as given in Figure 5. In this case, NS-v = <1,3,6,9> and S =<BOOK-AUTHOR,BOOK>. We start with curr-seq = <8> and check if we can identify any contradiction with the assumption that the result sets of Query'(curr-seq) and Query'(NS-v) intersect. In this case, there is no contradiction. In other words, Query'(curr-seq) selects all the authors and Query'(NS-v) selects authors of CostlyBooks. So, their intersection is not empty. So, we go up one level and consider curr-seq = <5,8> and then curr-seq = <2,5,8>. There are no contradictions in these situations also. We next consider curr-seq = <1,2,5,8>. In this case, we identify a contradiction as follows.

Recall that BOOK(bookid,title,price) and BOOK-AUTHOR(bookid,authorid) is the relational schema. The column bookid is the key for the BOOK relation and (bookid,authorid) is the key for the BOOK-AUTHOR relation. BOOK-AUTHOR.bookid is a foreign key to BOOK.bookid. We start with the relational sequences RSeq = $<BA_1, B_1>$ and SSeq = $<BA_2, B_2>$. We have $Query(RSeq) = BA_1(x_1, x_2)B_1(x_3, x_4, x_5)$ and $Query(SSeq) = BA_2(y_1, y_2)B_2(y_3, y_4, y_5)$. We are using a conjunctive query notation to represent the queries. Equating the key columns and projected columns of BA_1 and BA_2, we get $Query(RSeq) = BA_1(x_1, x_2)B_1(x_3, x_4, x_5)$ and $Query(SSeq) = BA_2(x_1, x_2)B_2(y_3, y_4, y_5)$. There are no functional dependencies or selection conditions on BA_1. So, we apply the join condition between BA_1 and B_1 to get $Query(RSeq) = BA_1(x_1, x_2)B_1(x_1, x_4, x_5)$. The selection condition, BOOK.price $\leq$ 50 is associated with $B_1.x_5$. Similarly, we apply the join condition between BA_2 and B_2 to obtain $Query(SSeq) = BA_2(x_1, x_2)B_2(x_1, y_4, y_5)$. The selection condition, BOOK.price $>$ 50 is associated with $B_2.y_5$. We now check if any of the constraints on the relational data are violated. BOOK.bookid is a key for the BOOK relation. So, $B_2(x_1, y_4, y_5)$ is modified to $B_2(x_1, x_4, x_5)$. Now, $B_1.x_5$ and $B_2.x_5$ have two conflicting conditions, price ≤ 50 and price > 50, both of which cannot hold simultaneously. This contradicts our original assumption that the intersection of results of Query'(curr-seq) and Query'(NS-v) is non-empty. So, node 1 is the *lda*(8,9).

In this example, *lda*(8,9) happens to be the least common ancestor of the two nodes and the root of the schema graph, as well. In general, the resulting node will be a descendant of the least common ancestor.

Note that we can use the above algorithm to check whether the mapping is *well-formed*. To do this, for each relational column $R.C$, $\forall n_1, n_2 \in nodes(R.C)$, we check if *lda*($n_1$,$n_2$) is non-null. If this holds always, then the mapping is *well-formed*.

We precompute the lda information for all pairs of nodes mapped to the same column and store it in an array. Using this array at query translation time, we perform prefix elimination as shown below. At the end of this stage, for query Q, we know for each node n in $NodeIdentification(Q) = S$, the lowest ancestor a of n such that the prefix of SQL(n) from the root to a can be safely eliminated.

```
procedure Prefix-Elimination(Q)
begin
  Let $S \leftarrow NodeIdentification(Q)$
  for each $n \in S$ do
    Let $Conflict_n \leftarrow \{x : annot(x) = annot(n) \text{ and } x \notin S\}$
    Let LDA $\leftarrow \{y : y = lda(n, x), x \in Conflict_n\}$
    $lda$(n,S) = highest node in LDA
end
```

One valid translation of query Q is SQL-PE(Q) = $\bigcup_{n \in S} Query$(PE-NS(n)) where PE-NS(n) is the node sequence from *lda*(n,S) to n.

4.1.2 Grouping multiple paths

We further optimize SQL-PE(Q) by grouping multiple paths that involve the same sequence of relations.

Two node sequences NS_1, NS_2 are said to be *combinable* if the corresponding relation sequences $RelSeq(NS_1), RelSeq(NS_2)$ are the same and $annot(NS_1.last) = annot(NS_2.last)$. In this case, Query($NS_1$) and Query($NS_2$) can be grouped together into a single basic SQL query (i.e., without unions). Note that this converts unions into disjunctions, which is valid under multi-set semantics due to the well-formedness of the mapping.

The above definition of combinability is an equivalence relation. Let NS= { PE-NS(n): $n \in NodeIdentification(Q)$}. We partition NS based on combinability and issue a basic SQL query for each equivalence class created. The algorithm for grouping the node sequences together and obtaining the final SQL query is given below.

```
procedure Grouping($S$)
begin
  Let NS= { PE-NS(n): $n \in S$}
  Partition NS based on the combinability of
    the node sequences
  Create a basic SQL query for each equivalence class
  Return the union of the basic SQL queries
end
```

We now briefly explain how we generate a basic SQL query, SQL(NS), for a set $\mathcal{NS}$ of combinable node sequences. Let $\mathcal{NS} = \{NS_1, NS_2, \ldots, NS_k\}$ denote the set of combinable node sequences. Let $R = RelSeq(NS_1) = \{R_1, R_2, \ldots, R_l\}$. Note that all the node sequences in NS have the same relational sequence. Let C_0 denote the set of conditions that are common across all the node sequences, NS_i. Let C_i denote the conditions corresponding to NS_i, not present in C_0. The from clause of SQL(NS) is the set of relations R and the column $annot(NS_1.first)$ is projected. The where clause has the condition $(C_0 \cap (C_1 \vee C_2 \vee \ldots C_k))$.

4.2 Handling Branching path expressions

We next illustrate the *constraint-aware* query translation for branching path expression queries with an example. The details are omitted for lack of space. We use the sample schema given in Figure 6 that represents a part of the advertisement data.

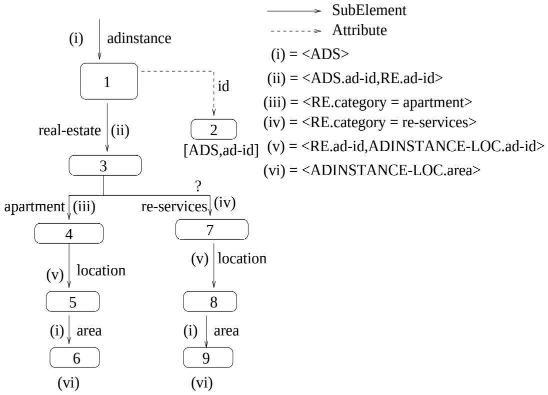

Figure 6: **Sample schema graph**

Consider the path expression query $Q =$ adinstance[id = 'value']// area. This query returns the geographic area corresponding to all adinstances with a particular id. In the *NodeIdentification* stage, the schema nodes that satisfy the query are identified. Recall that for simple path expressions, the result was a set of satisfying schema nodes corresponding to the area element. For branching path expressions, each entry in the result is a 3-tuple: the result node, the branching node and a set of condition nodes. In the above example, the result nodes are area nodes, branching nodes are adinstance nodes and condition nodes are id nodes. Evaluating Q on the schema graph in Figure 6 returns the set $S = \{<6,1,\{2\}>, <9,1,\{2\}>\}$. Let S' denote the set of result nodes. In our example, $S' = \{6, 9\}$. For a result node, let Branching(n) denote the corresponding branching node and Condition(n) denote the set of condition nodes.

A naive SQL translation is to generate an SQL query for each result node, condition node pair that have a common branching node and union all such queries. For a single pair (n, c), the SQL query is SQL(n) joined with Query(NS), where NS is the node sequence from the node Branching(n) till the node c. Note that since the condition is an existential condition, this translation may introduce duplicate results in some cases and we discuss how this is handled later in this section. In our translation algorithm, we use the constraint information and remove implied prefixes and group multiple satisfying paths, whenever possible.

For each result node n, we compute the prefix that can be eliminated. We first compute n_1 =*lda*(n,S') as defined in Section 4.1.1. In our example, *lda*(6,S') = 3 and *lda*(9,S') = 3. For each pair of result and condition nodes, (n, c), we compute n_2 =*lda*(c,Condition(n)). We set the lowest required ancestor, *lra*(n,c), to the highest node among n_1, n_2 and Branching(n). This denotes the node till which we have to issue an SQL query and its ancestors can be safely eliminated from the query. In our example, *lra*(6,2) = 1 and *lra*(9,2) = 1. For each (n, c) pair, we now have a branching node sequence that needs to be translated into SQL — this comprises of two branches corresponding to the *lra*(n,c)-n path and *lra*(n,c)-c path. Both the branches meet at the branching node, Branching(n), and the query terminates at *lra*(n.c).

We then group the prefix-eliminated node sequences together by modifying the algorithm in Section 4.1.2 to handle branching node sequences and generate the final SQL query. The query for the above example is

```
select al.area
from ADINSTANCE-LOC al, RE re, ADS ads
where re.ad-id = ads.ad-id
      and ((re.category = 'apartment') or
           (re.category = 're-services'))
      and al.ad-id = re.ad-id
      and ads.ad-id = 'value'
```

The condition in the path expression is an existential condition. So, if a result node n had several corresponding condition nodes, then the above translation may return a tuple corresponding to node n in the result multiple times. To solve this problem, we project the key of the relation in *annot*(n) and the node identifier of node n, as well. We also add a distinct clause to the query. An alternative strategy is to translate the existential condition as a nested subquery. We omit the details of our algorithm for this case.

4.3 Extensions to more general cases

4.3.1 DAG schema graphs

If the input XML schema graph is a DAG schema graph, one option is to unfold it into a tree schema graph, as explained earlier in Section 3.1. However, this may cause an exponential blowup in the worst case. Note that if we exported the relational data using XQuery (without functions), then the query defining the view will have a similar blowup. Using functions in the view definition, this blowup can be avoided, but in this scenario, optimizing queries will become difficult. We have a modified algorithm that will operate directly on the DAG schema graph. Due to lack of space, we do not present the algorithm.

4.3.2 Beyond path expressions

We considered *constraint-aware* translation of path expression queries. We shall briefly describe how to extend this to more general queries. A path expression query corresponds to the For clause in XQuery. Let us consider an XQuery that has several of these For clauses, along with some optional Where clauses. A natural way of applying our techniques is to perform *constraint-aware* translation for each of the individual path expressions, and then combine the resulting queries with appropriate join conditions. For example, consider a query Q involving two path expressions p_1 and p_2. A *constraint-aware* translation may produce SQL queries that are union of n_1 and n_2 queries respectively. We can combine them in a (exponential) number of ways. In our experiments, we use the approach where the SQL query is expressed as the join of two union queries.

5 Experiments

We conducted our experiments on two datasets: the ADEX dataset we generated conforming to the standard advertisement schema [14] and the dataset from the XMark Benchmark [13]. We ran the experiments using the IBM DB2 database on an Intel 800 MHZ Pentium processor with 256 MB of main memory running Linux. The buffer pool was set to 32 MB.

We generated synthetic data for the ADEX dataset. The data consists of 100K advertisements and 200 publications. The data size is approximately 92 MB. We briefly describe the structure of the advertisement data. Each advertisement has a buyer, one or more proof-readers and is printed in one or more publications. Each ad is equally likely to be a real-estate, employment or transportation ad. The main categories are further classified into subcategories. For example, real-estate can be one of seven categories. We assume uniform distributions while choosing the subcategories. The ads are classified into commercial and personal ads. Personal ads have one location, while commercial ads have up to four locations. The geographic area of the locations is equally likely to be one of twenty values. Employment ads have category information denoting areas of interest. Some subcategories of employment ads also have information about references. A brief description of the relations in the relational schema is given in Table 7. We built indices on all columns that appeared in a query.

For the XMark dataset, we used the standard 100 MB XML dataset. A brief description of the part of the relational schema we used, that is relevant for our experiments, is given in Table 8. We built indices on all columns that appeared in a query.

We compared the execution times for the queries in Figure 1. The queries labelled Ai are on the advertisement dataset, while those labelled Bi are on the XMark dataset. For each query, we generated relational queries using *constraint-oblivious* translation algorithms and using the *constraint-aware* algorithm presented in this paper. For each XML query, we generated relational queries using several *constraint-oblivious* translations and used the best timing for comparison with our *constraint-aware* approach. The relative improvements obtained in execution times are given in Table 1.

The relative improvement in performance ranges from 15% to a factor of 422. In general, by using the constraint information we do no worse than any *constraint-oblivious* strategy and so, the relative improvement is always >= 1.

For Query A1, the naive translation results in a query A_1^1 of the form,

$$\begin{aligned}&(\mathsf{ADEX} \bowtie \mathsf{ADS} \bowtie \mathsf{RE} \bowtie \mathsf{ADINSTANCELOC})\\ \bigcup\ &(\mathsf{ADEX} \bowtie \mathsf{ADS} \bowtie \mathsf{RE} \bowtie \mathsf{ADINSTANCELOC})\end{aligned}$$

The relational optimizer utilizes the specified integrity constraints (key-foreign key and not null) and rewrites A_1^1 as

$$(\mathsf{RE} \bowtie \mathsf{ADINSTANCELOC}) \bigcup (\mathsf{RE} \bowtie \mathsf{ADINSTANCELOC}).$$

Our translation algorithm will result in the query A_1^2, $(\mathsf{RE} \bowtie \mathsf{ADINSTANCELOC})$. A_1^2 is better than the rewritten A_1^1 as it has translated the union into a disjunctive condition on category. So, we obtain a speedup of 22% and 15% in the cold and warm timings.

For Query A2, the naive translation results in an SQL query of the form

$$\bigcup_{i=1}^{7} (\mathsf{ADEX} \bowtie \mathsf{ADS} \bowtie \mathsf{RE} \bowtie \mathsf{ADINSTANCELOC}).$$

Even in this case, the relational optimizer identifies that joins with the ADEX and ADS relations are redundant and simplifies the query to

$$\bigcup_{i=1}^{7} (\mathsf{RE} \bowtie \mathsf{ADINSTANCELOC}).$$

But, it does not merge the 7 queries into one. It also does not use the constraint that the category column can take only one of 7 values, which are all covered by this query. On the other hand, our translation will produce a single join between RE and ADINSTANCELOC relations. So, the relative performance improves by a factor of 2.73 and 3.25 in the cold and warm buffer scenarios.

	Queries	Cold buffer	Warm buffer
A1	Get the number of open-house ads in the campus area	1.22	1.15
A2	Get the number of real-estate ads in the campus area	2.73	3.25
A3	Get the number of ads in the campus area	42.04	422
A4	For each geographic area, get the number of ads in that area	51.34	92.96
A5	For each person, get the number of times (s)he is a reference	12.82	29.79
A6	For each job category, get the number of people interested in that category	6.11	34.13
B1	Get the number of items in a particular category	2.69	5.56
B2	For a particular person, get the categories of items for which (s)he has made a bid	5.35	13.20
B3	For each category, get the number of items in that category	6.40	7.63

Table 1: **Relative performance improvement for queries on ADEX schema**

ADEX	Advertisements
ADCONTACTS	Contact People appearing in ads
ADS	Instance of a particular ad
RE	Information about real-estate ads
TRANS	Information about transportation ads
EMP	Information about employment ads
ADINSTANCELOC	Location information for all ads
JOBCATEGORY	Category info for employment ads
JOBREFERENCE	Reference info for employment ads

Figure 7: **Part of Relational Schema for ADEX dataset**

ITEM	Items available for auction
INCATEGORY	List of categories for each item available for auction
CATEGORY	Information about all categories
OPENAUCTION	Information about currently active auctions
BIDDER	Bidder information for open auctions

Figure 8: **Part of Relational Schema for XMark dataset**

For query A3, as we saw in Section 2, a *constraint-oblivious* translation results in query SQ, which is optimized by the relational optimizer to the query SQ_1. Our *constraint-aware* algorithm results in the query *optQ*. The resulting performance improvement is a factor of 42 with a cold buffer pool. Since our query is an index lookup on a single relation, in a warm buffer pool, the query execution time reduces from 230ms to 20ms. But the optimized *constraint-oblivious* query is the union of 20 queries, each having a join. So, the response time only improves marginally from 9.67s to 8.44s. As a result, the relative speedup increases by a factor of 10 to 422.

Notice that the wildcard in query A1 had 2 satisfying paths, while that in A2 and A3 had seven and twenty satisfying paths respectively. The response times show that as the number of satisfying paths for a wildcard increases, the benefit obtained by our approach also increases considerably. The above three queries have a selection condition on the geographic area. Queries A4, A5 and A6 compute information for entire sets. For example, A4 gets the number of ads for each area. Even for these queries, we observed significant speedups when our translation algorithm was used.

Similarly, queries B1, B2 and B3 on the XMark dataset also had significant speedups ranging from a factor of 2.7 to a factor of 13.2. The speedup was comparatively smaller in this case as the maximum number of satisfying paths for a wildcard was only six for the XMark schema.

To summarize, we see that using a *constraint-aware* translation algorithm, we get significant performance benefits. This improvement is markedly higher when the XML query has wildcards in it and the mapping constraints allow several of these branches to be merged.

6 Related Work

The Xperanto [2, 10, 11] and SilkRoute [5, 6] projects showed ways of efficiently exporting XML views of relational databases. Answering XML queries over these views by pushing most of the computation to the RDBMS was the main focus. These methods do not use the underlying relational constraints during query translation. In Xperanto [12], a wildcard in the XML query is replaced with all matching paths and the resulting query is the union of the SQL translations for each of these paths. The SilkRoute project do not specify how they handle wild cards in the XML queries. In this paper, we use the constraints on the underlying relational database during query translation and also group multiple satisfying paths for wild card XML queries, whenever possible.

The Agora[8] project deals with translating XML queries into SQL in the context of integrating heterogeneous data sources using a global XML schema. The local (relational) schemas are defined as views over a fixed global schema. The XML query is translated into SQL over the global schema and the resulting SQL query is rewriten using the view definition of the local schema. A transitive closure relation is used to answer wildcard queries involving “//”. In this approach, the translation of the SQL query on the global schema into an SQL query on the actual relational schema reduces to the well-known problem of answering queries only using a set of views. This places some implicit restrictions on the nature of relational schema and the class of XML queries allowed. For example, for the relational schema in Figure 3, all the author information is stored in a single BOOK-AUTHOR relation. The XML query Books/CheapBook/Author/Authorid cannot be translated into SQL by this approach. On the other hand, we handle the case of fixed-schema mappings and do not place any restrictions on the relational schema. Moreover, in [8], the XML to SQL query translation is done regardless of the data mapping, while we use the mapping information along with the constraints placed on the underlying relational data to obtain an efficient SQL query.

There has been some work on optimizing queries in a semi-structured framework[1, 4, 9] using graph schemas. These papers dealt with the *NodeIdentification* stage of query translation, while the focus of our work is on using constraints on the relational data during the *RelationalQueryGeneration* stage.

7 Conclusions and Future Work

In this paper we considered the problem of translating XML queries into SQL, in the fixed schema scenario. We showed that *constraint-oblivious* translation, which is correct irrespective of the constraints on the data, is sub-optimal. We then presented a *constraint-aware* algorithm for path expression queries and showed through experiments on two datasets that our algorithm results in significant performance improvement over a *constraint-oblivious* translation.

We described several general relational constraints that are relevant in the optimization of SQL queries, especially when these queries are obtained from an XML to SQL translation. We proposed an approach where we used these constraints during the XML to SQL query translation itself. The alternative approach of enhancing the relational optimizer to exploit these constraints efficiently is an area for future work.

We presented a translation algorithm that identifies optimization opportunities when the mapping is *well-formed*. Extending our translation algorithms to work when the "at-most once" and "at-least once" conditions are relaxed, and to work for recursive schema graphs, provides another interesting direction for future work.

References

[1] P. Buneman, S. Davidson, M. Fernandez, D. Suciu, "Adding Structure to Unstructured Data", ICDT 6th International Conference, January 1997.

[2] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, S. Subramanian, "XPERANTO: Publishing Object-Relational Data as XML", Workshop on the Web and Databases (WebDB), May 2000.

[3] J. Clark and S. DeRose, XML path language (XPath) 1.0. W3C recommendation, World Wide Web Consortium, http://www.w3.org/TR/xpath, Nov.1999.

[4] M. Fernandez, D. Suciu, "Optimizing Regular Path Expressions Using Graph Schemas", Proceedings of the 14th ICDE Conference, Orlando, Florida, February 1998.

[5] M. Fernandez, W. C. Tan, D. Suciu, "SilkRoute: trading between relations and XML", Proceedings of the 9th World Wide Web Conference, 2000.

[6] M. Fernandez, A. Morishima, D. Suciu, "Efficient Evaluation of XML Middle-ware Queries", Proceedings of the ACM SIGMOD Conference on Management of Data, 2001.

[7] D. Maier, A. O. Mendelzon, Y. Sagiv, "Testing Implications of Data Dependencies", TODS 4(4): 455-469 (1979)

[8] I. Manolescu, D. Florescu, D. Kossman, "Answering XML queries over heterogeneous data sources", Proceedings of the VLDB Conference, 2001.

[9] J. McHugh, J. Widom, "Compile-Time Path Expansion in Lore", Workshop on Query Processing for SemiStructured Data and Non-Standard Data Formats, Jerusalen, Israel, January 1999.

[10] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, B. Reinwald, "Efficiently Publishing Relational Data as XML Documents", Proceedings of the VLDB Conference, Egypt, September 2000.

[11] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, J. Funderburk, "Querying XML views of relational data", Proceedings of the VLDB Conference, 2001.

[12] J. Shanmugasundaram, Personal Communication, November 2001.

[13] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, R. Busse, "The XML Benchmark Project", Technical Report INS-R0103, CWI, April 2001

[14] NAA Classified Advertising Standards Task Force, "http://www.naa.org/technology/clsstdtf/"

[15] SQL Server, "http://msdn.microsoft.com/library"

[16] XML-DBMS: Middleware for Transferring Data between XML Documents and Relational Databases, "http://www.rpbourret.com/xmldbms"