Efficient XML-to-SQL Query Translation: Where to Add the Intelligence?

Rajasekar Krishnamurthy *
IBM Almaden Research Center
sekar@cs.wisc.edu

Raghav Kaushik *
Microsoft Research
skaushi@microsoft.com

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

Abstract

We consider the efficiency of queries generated by XML to SQL translation. We first show that published XML-to-SQL query translation algorithms are suboptimal in that they often translate simple path expressions into complex SQL queries even when much simpler equivalent SQL queries exist. There are two logical ways to deal with this problem. One could generate suboptimal SQL queries using a fairly naive translation algorithm, and then attempt to optimize the resulting SQL; or one could use a more intelligent translation algorithm with the hopes of generating efficient SQL directly. We show that optimizing the SQL after it is generated is problematic, becoming intractable even in simple scenarios; by contrast, designing a translation algorithm that exploits information readily available at translation time is a promising alternative. To support this claim, we present a translation algorithm that exploits translation time information to generate efficient SQL for path expression queries over tree schemas.

1 Introduction

Exporting XML views of relational data gives rise to the problem of translating XML queries into SQL. To date, the focus of most of the work in the published literature [10, 16, 21] has been on mechanisms for correctly translating complex XML queries into SQL queries, with less emphasis on evaluating the quality of the resulting SQL queries. The efficiency of the SQL queries generated by the translation process is the focus in this paper.

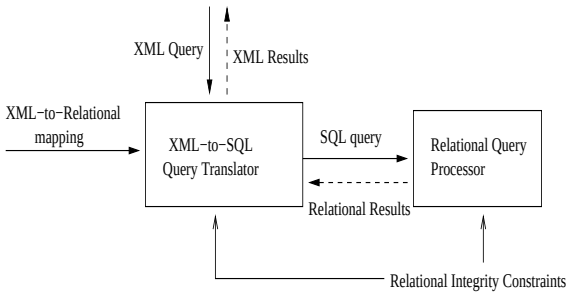

Figure 1: Stages in using an RDBMS to evaluate an XML query

Translating XML queries to SQL involves translating queries over hierarchical schemas into queries over flat relational schemas. This turns out to be problematic — a closer look at the queries generated by the published translation algorithms shows that the hierarchical nature of the exported XML schema is often blindly reflected in the generated SQL query, even when this is clearly not necessary. As a result, in many cases even simple path expression queries result in unnecessarily complex SQL queries. This problem is aggravated when the input XML query includes a traversal of the descendant axis (//), because it does not have a simple equivalent in SQL.

A natural question to ask next is whether the phenomenon of large, complex SQL queries arising from simple XML queries is avoidable, or if it is intrinsic due to the mismatch in data models. We show by example in Section 2.1 that complex SQL is not necessary in many cases — while the SQL generated by published translation algorithms is complex, usually there is a much simpler equivalent SQL query. This observation motivated us to search for techniques that make use of readily available semantic information to improve the quality of the generated SQL.

To understand the alternatives for how we can do this, consider the different stages in the translation process as shown in Figure 1. Given an XML-to-Relational mapping, some relational integrity constraints, and an XML query, the XML-to-SQL query translator generates an equivalent SQL query and hands it over to the relational query processor. The

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

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

relational query processor optimizes and executes the query, and returns the results to the query translator, which adds the appropriate XML tags to the results and returns them to the user. There are two important points to note here: (i) As the XML-to-Relational mapping and relational integrity constraints are valid across multiple query invocations, they are shown separately, and (ii) We have made no assumptions about whether the XML-to-SQL query translator is inside an RDBMS or in middleware. This is the reason for using the term Relational Query Processor instead of RDBMS for the box on the right.

There are two logical extremes in approaches toward obtaining efficient SQL queries for XML workloads. One could generate suboptimal SQL queries using a fairly naive translation algorithm, and then optimize the resulting SQL queries (SQL Optimization); or one could use a more intelligent query translation algorithm and attempt to generate efficient SQL queries directly (Intelligent Query Translation).

In Section 4, we will show that if we take the SQL Optimization approach, then in order to obtain efficient SQL queries we have to solve the relational query minimization problem under bag semantics. The techniques for query minimization in the published literature rely on algorithms for query containment or query equivalence. Unfortunately, these problems become intractable in even simple scenarios, making the SQL Optimization approach impractical. In view of this problem, we need to find a way to generate good SQL queries that does not require the solution of these intractable problems during actual query translation.

In response to this goal, we propose that Intelligent Query Translation should be used instead of SQL Optimization, and propose a translation approach that relies upon three main ideas. First, we identify a class of tree XML-to-relational mappings called *bijective* mappings. Bijective mappings cover a large class of the mappings we have encountered in print, and they have the desirable property that they can be optimized using containment and equivalence algorithms under set semantics instead of multiset semantics.

Second, we observe that for a given XML schema over a given relational schema, the SQL queries generated from XML queries are not arbitrary. That is, the XML-to-Relational mapping determines the class of SQL queries that are likely to be output by the XML-to-SQL query translation algorithm, which in turn fixes the class of queries that need to be minimized. Since the XML-to-Relational mapping and the underlying relational integrity constraints are independent of the query being optimized, we can use them to precompute some useful information, and then use this information during the runtime query translation. This way, we can move the potentially expensive task of reasoning about integrity constraints to the precomputation phase, keeping the run time overhead small.

Third, the conjunctive queries produced by XML to SQL translation are mainly *chain* queries of the form

$$R(x_n) :- R_1(x_1, x_2), R_2(x_2, x_3), \ldots, R_{n-1}(x_{n-1}, x_n)$$

As we will show, in the XML to SQL translation domain, exploiting integrity constraints enables the minimization of such queries by removing a prefix of the relational predicates. We refer to this as *prefix elimination*. This turns out to be more tractable than general conjunctive query minimization.

We show that by exploiting the above three ideas, the XML-to-SQL query translation problem can be solved in polynomial time for path expression queries over bijective tree mappings. Our proof works by presenting a query translation algorithm that solves the problem with the required efficiency. Our algorithm works correctly even over non-bijective mappings; it identifies the bijective portions of the mapping and performs more efficient query translation in those parts. This translation algorithm produces SQL queries that in many cases are far more efficient than those produced by previously published translation algorithms.

The rest of the paper is organized as follows. In Section 2.1, we present an example scenario to illustrate the problems with published XML-to-Relational translation algorithms. Next we define the query translation problem in Section 3. Then, in Section 4, we present some of the known complexity results we bump into if we attempt to minimize the SQL queries after generating them. We describe our strategy for more intelligent query translation in Section 5. A more formal description of the various components of this approach is presented in Section 6.

1.1 Related Work

Translating XML queries into SQL in an XML Publishing context has been addressed in [9, 10, 12, 16, 17, 20]. Excepting MARS [9], the main focus has been on translating complex XML queries into SQL, and not on the quality of the final SQL query. A more detailed description of the existing published work on XML-to-SQL query translation is given in [15].

In MARS [9], a technique for translating XQuery queries into SQL is given, when both Global-As-View (GAV) and Local-As-View (LAV) views are present. The system achieves the combined effect of rewriting-with-views, composition-with-views, and query minimization under integrity constraints. While this technique has its own advantages, it does not produce efficient SQL queries for simple XML queries that contain the descendant axis (//) (like the example in Section 2.1). The technique in MARS [9] can be viewed as a SQL Optimization technique since the main optimization occurs after the SQL query is generated from the XML query. In our work, we assume a simpler setting of only GAV-style views and show how one can obtain efficient SQL queries by placing the intelligence in the translation process.

2 Motivation

2.1 Translation example

In this section, we present an example recursive query (with the descendant axis //) over a tree XML schema to illustrate that even simple XML queries can give rise to fairly complex SQL queries if we use published translation algorithms.

Part of a sample relational schema for an auction database is shown in Figure 2. The figure also shows one way of exporting this data as XML. The example XML schema is part of the XMark benchmark [23] schema. The associated view definition is easy to construct and is omitted. Each node in the XML schema is annotated with a table name, to indicate the relational table that corresponds to the element represented by the node. Each leaf node has a column name next to it, which indicates the column in which the value of corresponding element is stored.

Consider the evaluation of the following query Q_1, which finds the number of items in a given category:

```
count(/Site/Regions//Item/InCategory[
                    @Category = 'cat1'])
```

Consider the following simple algorithm for handling queries with the descendant axis (//) [12]: Identify all paths in the schema that satisfy the query. For each path, generate a relational query by joining all relations appearing in this path. The final query is the union of the queries over all satisfying paths (six paths for Q_1). This algorithm will result in the following SQL query SQ_1.

```
select count(*)
from   Site S, Item I, InCat C
where  S.id = I.siteid and I.id = C.itemid and
       C.category='cat1' and I.continent='africa'
union all ... (6 queries)
```

Suppose furthermore that the underlying relational schema has the following domain integrity constraint (in addition to the key and foreign key constraints shown in the figure): the column *Item.continent* has only six potential values {asia, africa, australia, europe, namerica, samerica }.

For the above query, we have found through experimentation that the optimizers in current relational systems will use foreign key constraints to eliminate some redundant joins. For instance, the join between Site and Item can be removed. Though the join between Item and InCat is a key-foreign key join, it cannot be removed due to the condition on Item.continent. Thus the query as rewritten by a relational optimizer becomes the new query SQ_1^1:

```
select count(*)
from   Item I, InCat C
where  I.id = C.itemid and C.category='cat1'
       and I.continent = 'africa'
union all ... (6 queries)
```

We have seen that existing commercial RDBMS optimizers convert SQ_1 to SQ_1^1. A reasonable question is whether the XML to SQL translation routines proposed in SilkRoute [10] and Xperanto [21] do better. We find that by merging common subexpressions, they generate a better initial query than SQ_1. But, interestingly, if you feed the queries that they generate to a relational optimizer, the resulting final query is once again SQ_1^1. So, no matter whether we use a naive XML to relational translation, or these more sophisticated translation schemes, in the end the RDBMS will evaluate SQ_1^1.

Another valid question to ask at this point is whether the algorithms for minimizing XML queries, such as in [3, 18], will help in this context. These algorithms remove parts of the XML query that are made redundant by other parts of the query. Notice that the XML query Q_1 has no redundant parts in it, and so XML query minimization will not help in this case.

Unfortunately, SQ_1^1 is far from optimal, since all of these queries are equivalent to the even simpler OQ_1 given below:

```
select count(*)
from   InCat
where  category = 'cat1'
```

The equivalence between the queries SQ_1, SQ_1^1 and OQ_1 holds under the key, foreign key and domain constraints mentioned above. Notice how we are able to replace a query SQ_1^1, which was the union of six queries each with a join, by a single scan query OQ_1.

2.2 Experimental Study

The previous example showed that while published algorithms translate the example XML query into a fairly complex SQL query, there is an equivalent query that looks much simpler. An important question to answer at this point is whether the associated performance gains are substantial. In order to demonstrate that this improvement can be sizable in practice, we performed an experimental study using two datasets: a synthetic ADEX dataset conforming to a standard advertisement schema [1] and a dataset from the XMark Benchmark [23].

The ADEX dataset conforms to the standard DTD being developed by the Newspaper Association of America Classified Advertising Standards Task Force [1]. This standard is intended to pave the way for the aggregation of classified ads among publishers on the Internet, as well as to enhance the development of classified processing systems. We generated synthetic data conforming to the ADEX schema. This generated data consists of 100K advertisements and 200 publications, and is approximately 150 MB. The XMark Benchmark [23] schema contains information about an auction database and we used the standard 100 MB dataset defined in the benchmark. For both scenarios, we built indexes on all columns that appeared in a query. We ran the experiments using the

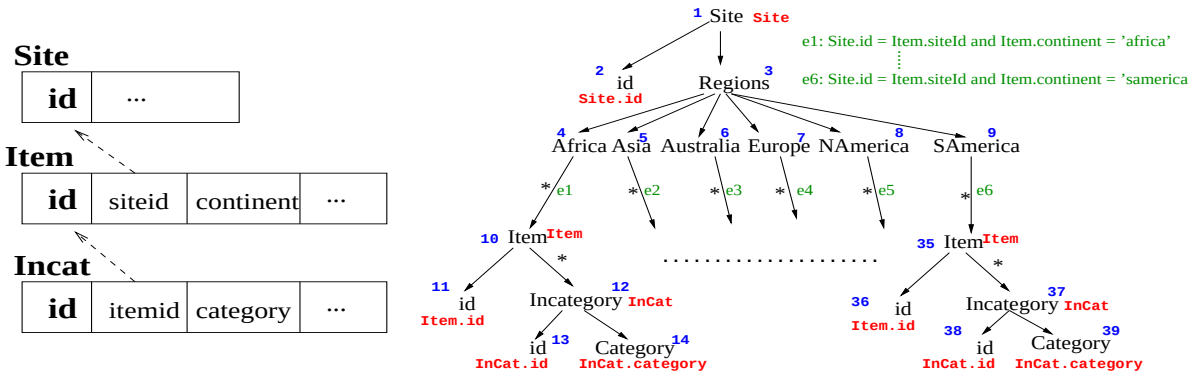

Figure 2: Sample relational schema and corresponding XML view

	Queries	Speedup (Cold buffer)	Speedup (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 addresses of ads in the campus area	27.05	31.1
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
X1	Get the number of items in a particular category	2.69	5.56
X2	For a particular person, get categories of items for which (s)he made a bid	5.35	13.20
X3	For each category, get the number of items in that category	6.40	7.63

Table 1: **Relative performance improvement obtained by using constraint information**

IBM DB2 database on a Linux workstation with an Intel 800 MHZ Pentium processor and 256 MB of main memory. The buffer pool was set to 32 MB. A complete description of the experimental setup is given in [14].

We compare the execution times we measured for the queries in Table 1. The queries labeled Ai are on the advertisement dataset, while those labeled Xi are on the XMark dataset. Note that query X1 in this table is the example query we considered in the previous section. For each XML query, we generated relational queries using several prior published algorithms and used the best timing for comparison with our approach, where we use the constraint information as well. The speedups obtained in execution times are given in the table.

The relative improvement in performance ranges from 1.15 to 93. In general, by using the constraint information we do no worse than any of the prior strategies; so the relative performance is always greater than or equal to 1. We found that the actual performance improvement depends on two main factors: (i) number of satisfying paths that can be merged together due to the fact that they have the same relation sequence and (ii) the length of the prefix that can be eliminated.

For example, the wild card in query A1 had two 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 wild card increases, the benefit obtained by our approach also increases considerably. The above three queries have a selection condition on the geographic area of an advertisement. Queries A4, A5 and A6 compute information across all areas. For example, A4 gets the number of ads for each area. Even for these queries, we observed significant speedups when constraint information was used to generate optimized SQL queries.

Similarly, queries X1, X2 and X3 on the XMark dataset also had significant speedups ranging from a factor of 2.7 to a factor of 13.2. The speedup was smaller in these cases relative to the ADEX dataset as the maximum number of satisfying paths for a wild card is only six for the XMark schema.

2.3 Observations

The above experimental results show that by using constraint information it is possible to obtain significant speedups in SQL query execution times in a number of cases. This improvement is markedly higher when the XML query has wild cards in it and the constraints on the data allow several of these branches to be merged. Opportunities for such optimizations occur when we build a hierarchy in the XML view from flat relational data. For example, in the XMark schema in Figure 2 a hierarchy was created by partitioning items based on the continents to which they belong to.

In the rest of the paper, we look at two different ways of attempting to automatically generate these better queries: SQL Optimization, and Intelligent Query Translation. In the former approach, SQL queries are generated in a straightforward fashion

and then optimized using the relational integrity constraints. In the latter approach, we use the constraint information during the XML-to-SQL query translation process itself.

3 Problem Definition

In this section, we present a formal description of the XML-to-SQL query translation problem.

For concreteness, we need to provide some mechanism for representing how an XML schema is mapped to a relational schema. In this paper, we use the simple approach of defining an XML view with annotations on the XML schema nodes and edges. A non-leaf node is annotated with a relation name, while a leaf node is annotated with the name of a relational column. Each edge $e = (u \rightarrow v)$ is annotated with a conjunctive query, where the relations allowed in the query are the relational annotations of nodes on the path from the root of the graph to node v. A *simple* XML view is one in which each of the edge annotations involves at most one join condition.

We illustrate this approach to defining views with an example[1]. Consider the relational schema and the corresponding XML view definition in Figure 2. Consider a top-down traversal of this schema, which illustrates how an XML document can be constructed from underlying relational data. The Site element is the root of the document and it has an id child whose value is the value of Site.id attribute. A Regions child element is created within the Site element and six subelements are created within Regions, one for each continent. Within each continent element, the information about the items in that continent are exported. For example, consider the element Africa. The annotation on the outgoing edge $(4, 10)$ indicates that for each tuple in the Item relation corresponding to this continent and satisfying the join condition, an Item subelement is created. For each such item, its id is exported as an id child element and the categories to which the item belongs is represented as incategory subelements. The annotation on edge $(10, 12)$ is a join condition Item.id = InCategory.itemId (not shown in figure). This view definition is an example of a *simple* view definition as each edge annotation has at most one join condition.

In this paper we focus on a simple but useful class of queries: simple path expressions. A simple path expression 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).

For a path expression query over a tree XML view, the equivalent relational query output by published translation algorithms can be viewed as the union of several conjunctive queries. So, we consider this class of queries with a simple extension: a disjunction of selection conditions is allowed for each conjunctive query.

[1]See [14] for a formal description of the view definition

For concreteness, we need to define what is meant by a translation of an XML query to a SQL query. That is, we need to define when a SQL query is considered a correct translation for a given path expression query Q under a mapping $\mathcal{T}$. Our approach to defining these semantics is to present a straightforward translation algorithm that returns the query *baseline*(Q). Any SQL query SQ that is equivalent to *baseline*(Q) under the given relational integrity constraints is a correct SQL translation for the XML query Q.

For a leaf node n in the schema, we define the root-to-leaf query $rtol(n)$ as the SQL query obtained by (conjunctively) combining the annotations on the edges of the root-to-leaf path of n and projecting the annotation of node n. For example, $rtol(14)$ is the query

```
select C.category
from   Site S, Item I, InCat C
where  S.id = I.siteid and I.id = C.itemid
       and I.continent='africa'
```

Given a tree XML-to-Relational mapping $\mathcal{T}$ and a simple path expression query Q, let $S = \{n_1, n_2, \ldots, n_k\}$ denote the set of nodes in $\mathcal{T}$ that match the query Q. Then a *baseline* query translation algorithm is to return the SQL query $\bigcup_{n \in S} rtol(n)$. Let *baseline* (Q) denote this query.

Finally, we are able to define what we mean by the XML-to-SQL translation problem: Given an XML-to-Relational mapping $\mathcal{T}$, a simple path expression query Q and integrity constraints on the underlying relational schema, find the equivalent SQL query with minimum cost.

The above definition is precise modulo the interpretation of the phrase "minimum cost." Different problems will result with different cost metrics. A reasonable cost metric is the traditional metric for conjunctive query minimization — that is, the cost of a query is the number of relational conjuncts. Let us denote this metric *RelCount*.

4 The SQL Optimization approach

The scenario we presented in Section 2.1 showed that query minimization is a core issue in generating efficient SQL queries for XML workloads. For the class of path expression queries over a tree XML-to-Relational mapping, recall that *baseline* (Q) is a union of conjunctive queries. Hence, we need to minimize a union of conjunctive queries under multiset semantics in the presence of relational integrity constraints. In this section, we first discuss prior work on relational query minimization and then discuss the impact on the SQL Optimization approach.

4.1 Previous work on Relational Query Minimization

Most, if not all, techniques in the published literature for minimizing relational queries are based on algorithms for query containment or query equivalence.

We next present some known results about the complexity of these problems.

- The containment, equivalence and minimization problems for conjunctive queries under set semantics are NP-complete [2, 5].
- The containment problem for conjunctive queries under multiset semantics is π_2^P-hard [6].
- The equivalence problem for conjunctive queries under multiset semantics is same as graph isomorphism [6].
- The containment and equivalence problems for monotonic relational expressions under set semantics is π_2^P-complete [19].
- The containment problem for union of conjunctive queries is undecidable under multiset semantics [11].

There has also been a lot of work on the use of constraints in query optimization of relational queries [7, 13, 25]. In [13], the query containment problem under functional dependencies and inclusion dependencies is studied. In [22], a scheme for utilizing semantic integrity constraints in query optimization, using a graph theoretic approach, is presented. In [24], a necessary and sufficient condition for the IC-RFT problem (does a conjunctive query always produce an empty result under a given set of implication constraints) is presented and in [25] the results are extended when referential constraints are also allowed. Polynomial equivalence to other problems like the query containment problem are also proved.

More recently, the chase and backchase algorithm (c&b) was introduced in [7] motivated by logical redundancy and physical independence in mediator-like components. This approach brings together use of indexes, use of materialized views, semantic optimization and join/scan minimization and allows non-trivial use of indexes and materialized views through the use of semantic constraints. In [8], the authors present a generalization of the classical chase algorithm for embedded dependencies [4] to a richer class of constraints known as Disjunctive Embedded Dependencies (DEDs).

4.2 Impact on the SQL Optimization approach

While a lot of research has been done on relational query optimization in the presence of constraints, there are some mismatches with what we need in the XML-to-SQL query translation scenario.

- Most of the prior work is on reasoning under set semantics. On the other hand, we need to optimize relational queries under multi-set semantics. We are not aware of any published algorithm for minimizing union of conjunctive queries under multi-set semantics (both in the absence and presence of integrity constraints).
- Even under set semantics, the running time of these algorithms are exponential in the size of the input (relational schema, constraints and query). Incurring this overhead on a per-query basis may be expensive in practice.
- The class of constraints handled by different approaches vary considerably and no single technique dominates the others.

By a simple reduction, we have the following result.

PROPOSITION **1** *Solving the XML-to-SQL Query Translation problem using the SQL Optimization approach for a simple tree XML view under the metric* RelCount *is at least as hard as minimizing a union of conjunctive queries under multiset semantics.*

5 Intelligent Query Translation

In this section, we present our approach to generating SQL queries that are often more efficient than those generated by existing translation algorithms. We are able to do so by focusing on a tractable yet important subpart of the problem space. This section is somewhat complex; we begin with an overview of our approach and then explain the main components of our approach. A more formal description is presented in the following section (Section 6).

5.1 Outline of our approach

As we saw in the previous section, the SQL Optimization approach has three main problems: (i) lack of techniques for query minimization under multi-set semantics, (ii) high overhead for reasoning using constraints even under set semantics and (iii) variety of techniques for different class of constraints. In the Intelligent Query Translation approach, we circumvent each of these problems in the following fashion.

While reasoning about query minimization under multi-set semantics can be a lot different from reasoning under set semantics, there are scenarios where the two notions are similar. In our approach, we identify a class of views that, informally speaking, have the property that the target relational data is exported *exactly* once in the XML view. We refer to such views as *bijective*, and describe this concept in more detail in Section 5.2. Such mappings have the desirable property that they can be optimized using containment and equivalence algorithms under set semantics instead of multiset semantics. In our approach, we identify parts of the mapping that are bijective and apply our optimizations to those parts.

In order to address problems (ii) and (iii), we adopt the following strategy. By observing that the XML-to-Relational mapping and the underlying relational integrity constraints remain constant across multiple query invocations, we compute some summary information in a precomputation phase. In this precomputation phase, we make use of an algorithm for reasoning about conjunctive query containment under set

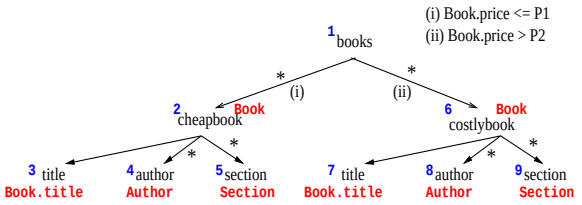

Figure 3: Sample mapping

semantics (say $\mathcal{A}$). Then, when we need to translate an XML query into SQL, we use this summary information in the run-time query translation phase. This way, the potentially expensive part of reasoning using integrity constraints is moved to a (offline) phase and the run-time overhead is kept small. In addition, we can make use of different algorithms for $\mathcal{A}$ that work for varying classes of relational integrity constraints. This is especially useful as we can choose algorithm $\mathcal{A}$ based on the class of relational integrity constraints that are applicable for the current relational schema.

Since, we are going to use some summary information during the run-time query translation process, we need to relax the optimality metric that we hope to achieve. As we have seen in Section 2.1, optimizing SQL generated by XML to SQL translation frequently involves eliminating unnecessary prefixes in the SQL queries. Motivated by this observation, we define a different notion of minimality for generated SQL queries — one where we would like to maximize the length of the prefix eliminated for each matching path in the schema. We define this metric, *PrefixMetric* in Section 5.3.

Using the above techniques, we developed a *constraint-aware* approach to efficiently translate path expression queries into SQL. We describe the main components of our approach informally in the following subsections. A more formal description of our approach is presented in Section 6.

5.2 Bijective mappings

Consider the XML schema shown in Figure 3, which represents information about a collection of books. The XML view has created a simple hierarchy, partitioning the books into cheap and costly books by the relationship of their prices to two constants P_1 and P_2.

Let us now consider three possible scenarios: $P_1 = P_2$, $P_1 < P_2$ and $P_1 > P_2$. If $P_1 = P_2$, then the XML view has information about all the books exactly once, while if $P_1 < P_2$ the XML view has information about only certain books. On the other hand, when $P_1 > P_2$, the XML view has information about the books in the price range $\{P_2 - P_1\}$ twice.

The scenario when $P_1 = P_2$ corresponds to an interesting and common class of mappings, one in which there is a one-to-one correspondence between the XML view data and the underlying relational data. We refer to this class of mappings as *bijective*. These mappings have the property that the query results of two root-to-leaf path queries do not have any common results, so the corresponding SQL queries can be merged without worrying about preserving counts of duplicates.

For example, the *rtol* queries for nodes 3 and 7 returning the titles of cheapbooks and costlybooks will not have any common results when the mapping is bijective. This simple observation makes the query minimization process a lot simpler as we can use algorithms for query minimization under set semantics instead of multiset semantics.

Notice that whether an XML view definition is *bijective* or not is a property of the view, and that one cannot determine if an XML view definition is bijective by simply examining the relational schema without the mapping. So, while one can easily use this information during the query translation process (where we know about the XML view), in order to perform similar optimizations after the SQL query has been generated, the appropriate module (be it the relational optimizer or some other module) needs to know about properties of the XML view. This means that if existing relational optimizers are to be extended to handle optimizations based upon bijective views, they need to be extended to understand XML views, which is not very attractive.

5.3 Prefix Elimination Optimality

We define the cost metric $PrefixMetric(SQ, \mathcal{T})$ to be the number of nodes in the XML-to-Relational mapping $\mathcal{T}$ that correspond to the SQL query SQ. For example, consider the query SQ_1 in Section 2.1. The fragment of this query identifying items in Africa corresponds to the sequence of nodes $<1, 3, 4, 10, 12, 14>$, and so the cost is six. Since there are six such fragments in SQ_1, the total cost $PrefixMetric(SQ_1, \mathcal{T})$ is 36. Similarly, the cost for each fragment of query SQ_1^1 is four and the total cost $PrefixMetric(SQ_1^1, \mathcal{T})$ is 24. For query OQ_1, the total cost $PrefixMetric(OQ_1, \mathcal{T})$ is six.

By definition, the cost of any SQL query that does not correspond to a path in the mapping is undefined.

Notice that the definition of the *PrefixMetric* metric restricts the class of equivalent SQL queries considered. For example, we are only interested in finding equivalent queries that are in some sense "syntactically" contained in some conjunctive query fragment in *baseline* (Q). While this misses opportunities to find equivalent queries that involve materialized views or cached query results or eliminating intermediate relations in the conjunctive query, it is still general enough to cover a large number of interesting scenarios.

5.4 The query translation algorithm

In this section, we briefly explain the main components of our query translation algorithm using examples. The algorithm has two parts: an (offline) precomputation phase, in which summary information is computed; and a run time phase when the actual query translation occurs.

5.4.1 Precomputation Phase

Here, we make use of the fact that the XML-to-Relational mapping and the relational integrity constraints are valid across multiple queries and use them to precompute some summary information. The information that we precompute is related to properties of the root-to-leaf queries we discussed in connection with the semantics of translation in Section 3.

For a given node in the XML schema, it may be possible to eliminate a prefix of its corresponding root to leaf query. The actual prefix that can be eliminated for a leaf node varies depending on the subset of schema nodes selected by the query. We define the notion of **L**east **D**istinguishing **A**ncestors (LDAs) to capture this. For each pair of leaf nodes (u, v), we compute $\text{LDA}(u, v) = w$. Intuitively, w is the lowest ancestor of u such that if node u matches a given XML query, it is sufficient to issue the query from $w - u$ (instead of the root to leaf query for u) without returning any results corresponding to node v. In order to create the query for a node u, it suffices to pick the highest ancestor among $\text{LDA}(u, v)$ over all leaf nodes v not matching the query.

For example, for the schema in Figure 2, $\text{LDA}(14, 39) = 4$ and $\text{LDA}(39, 14) = 9$. In other words, if node 14 matches a query and node 39 does not, then it suffices to issue the query corresponding to the path $\{4, 10, 12, 14\}$ in order to return the results corresponding to node 14. This query is shown below.

```
select IC.category
from Item I, Incat IC
where I.id = IC.itemid and I.continent = 'africa'
```

In our precomputation phase, for every pair of non-leaf nodes u, v that have the same annotation, we compute $\text{LDA}(u, v)$. In addition, we identify the parts of the XML view definition that are bijective. In our running example, the entire XML view is bijective.

5.4.2 Run-time Query Translation

We use the following query on the mapping schema in Figure 2 to illustrate the translation algorithm.

Q: //Item/InCategory/Category

We first execute Q on the schema graph and identify the satisfying nodes: $S = \{14, 19, 24, 29, 34, 39\}$. For each node $n \in S$, issuing $rtol(n)$ is a correct translation. Our goal is to find the smallest suffix of each such query. Consider the leaf node $n_1 = 14$. We need to identify the lowest ancestor a_1 of n_1 such that it suffices to output the query for the path $<a_1, \ldots, n_1>$. In order to find a_1, we look at the other nodes with the same annotation, namely $C = \{19, 24, 29, 34, 39\}$ and compute $LDA(14, x), \forall x \in (C - S)$. The highest node among these corresponds to a_1. In this particular case, $(C - S)$ is empty, so we do not have to look at the LDA values. As a result, for node 14 it suffices to issue the scan query corresponding to the leaf node. We obtain a similar scan query for the other five schema nodes in S. Since the six scan queries are on the same relation, we merge them and issue a single query OQ given below:

```
select C.eid from InCat C
```

On the other hand, using existing algorithms we would have obtained a relational query SQ that is the union of six queries, each with two joins (similar to the example query SQ_1 in Section 2.1).

5.5 Analysis

The proofs of the theorems in this section are omitted due to lack of space and are presented in [14].

THEOREM **1** *Given a tree XML-to-Relational mapping $\mathcal{T}$ along with the integrity constraints that hold on the underlying relational schema, and a path expression query P, the constraint-aware algorithm outputs a correct equivalent SQL query in polynomial time.*

We would like to point out that our algorithm performs XML-to-SQL query translation correctly even when part of the mapping is not bijective or when the conjunctive query containment algorithm $\mathcal{A}$ is sound but not complete for the class of relational constraints that are applicable. Note that the running time of algorithm $\mathcal{A}$ does not impact the complexity of the translation algorithm, since $\mathcal{A}$ is run once as a precomputation step, not on a per-query basis during translation.

Let Q_1 be a conjunctive query and Q_2 be a union of conjunctive queries. Let UQC denote the problem: is $Q_1 \subseteq Q_2$ under set semantics? and DUP denote the problem: Are the results of Q_1 duplicate-free?

Suppose $\mathcal{C}$ is the class of integrity constraints that hold on the relational schema and $\mathcal{A}$ and $\mathcal{A}'$ are sound and complete algorithms for the UQC and DUP problems over this class of constraints. Examples of such algorithms and a description of the corresponding class of integrity constraints can be found in [7, 25]. In such cases, our algorithm actually outputs the optimal query under metric *PrefixMetric*.

THEOREM **2** *Given sound and complete algorithms $\mathcal{A}$ and $\mathcal{A}'$ for the UQC and DUP problems over the class $\mathcal{C}$, the XML-to-SQL Query Translation problem for a bijective tree XML view under metric* PrefixMetric *can be solved in polynomial time.*

6 The *constraint-aware* approach

In this section, we formally describe the various components of our approach. We start by describing some terminology used in the formalization followed by the actual description of the two main components: precomputation phase and run-time query translation phase.

6.1 Terminology

Most of the properties we talk about address leaf nodes in the schema that are annotated with the same relational column. We define *nodes(R.C)* to be the set of leaf nodes annotated with $R.C$. We call two leaf nodes *column-compatible* if they are annotated with the same relational column. We refer to the annotation of node n as $annot(n)$.

Recall that we defined $rtol(n)$ to be the root-to-leaf query for a node n. We generalize this notion to an arbitrary sequence of nodes as follows. A *node sequence* $NS =< n_1, n_2, \ldots, n_k >$ is a sequence of nodes in the schema graph that corresponds to a path starting from the node $n_1 = NS.first$ and terminating in the leaf node $n_k = NS.last$. The relational query Query(NS) is obtained by combining the conditions on the edges of the sequence and projecting $annot(n_k)$. The relational query keyQuery(NS) is the same as Query(NS), except that the key column(s) of $Rel(n_k)$ is (are) also projected. Query(NS) and keyQuery(NS) are always conjunctive queries. Just like Query(NS) corresponds to $rtol(n)$, we refer to keyQuery(NS) as $keyrtol(n)$.

Let RelSeq(NS) denote the sequence of relations joined in Query(NS), in a bottom-up order. For example, for $NS =<1, 3, 4, 10, 12>$, RelSeq(NS) = <InCat,Item,Site>.

Two node sequences NS_1 and NS_2 are said to be *combinable* if the corresponding relation sequences $RelSeq(NS_1)$ and $RelSeq(NS_2)$ are the same, the join conditions are on the same set of columns for each pair of relations, and $NS_1.last$ and $NS_2.last$ are column-compatible. In other words, the two relation sequences are identical modulo the selection conditions.

6.2 Precomputation Phase

Recall that, in the precomputation phase we identify the parts of the XML-to-Relational mapping that are bijective and also compute LDA information. We formally define these two notions in the next two subsections and then describe how we compute this information.

6.2.1 Bijective column mappings

For a relational column $R.C$, let KeyProject(R.C) denote the query "select R.key, R.C from R" and NodeKeyProject(R.C) denote the query $\bigcup_{n \in nodes(R.C)}$ keyrtol(n). Here, $R.key$ denotes the key column(s) of R. We will make use of the following definitions:

DEFINITION **1** *For a relational column R.C,*

- *If* $KeyProject(R.C) \subseteq NodeKeyProject(R.C)$, *then R.C is At-least-once mapped*
- *If* $KeyProject(R.C) \supseteq NodeKeyProject(R.C)$, *then R.C is At-most-once mapped*
- *If* $KeyProject(R.C) = NodeKeyProject(R.C)$, *then R.C is bijectively mapped*

In the preceding definition, the containment operations are under multi-set semantics.

Informally, if all the values in the column $R.C$ appear in the XML view "exactly once", then the relational column is bijectively mapped. In order to check this under multi-set semantics, we use the key field(s) of the relation R.

An XML-to-Relational mapping $\mathcal{T}$ is bijectively mapped if each of the relational columns annotating some leaf node in $\mathcal{T}$ is bijectively mapped.

6.2.2 Lowest Distinguishing Ancestor

Let u and v be two column-compatible leaf nodes in the schema. Let node sequence $NS =< n_1, n_2, \ldots, n_k >$, where $n_1 = \text{root}(\mathcal{T})$ and $n_k = u$, represent the root-to-leaf path in to u.

DEFINITION **2** *The node* n_j *is a* distinguishing ancestor *for* u *with respect to* v *if the intersection of the results of the two queries,* $keyQuery(< n_j, \ldots, n_k >)$ *and* $keyrtol(v)$, *is empty.*

If n_j is a *distinguishing ancestor* for u with respect to v, then we write $u \mid^{n_j} v$. Thus, for the above example, 4 is a distinguishing ancestor of 14 with respect to every other *column-compatible* node. In other words, issuing the query from 4 to 14, we will obtain all the results corresponding to node 14 and no result corresponding to any other column-compatible node (such as node 39).

Observe that the *distinguishing ancestor* relation is not a symmetric relation. For example, in the annotated schema graph shown in Figure 2, consider schema nodes 14 and 39, which are column-compatible. Now, $14 \mid^4 39$ is true. Notice that node 4 is an ancestor of node 14 but not an ancestor of node 39. So, $39 \mid^4 14$ is false.

DEFINITION **3** *The* lowest distinguishing ancestor *for* u *with respect to* v, $u \parallel v$, *is the lowest ancestor* w *of* u *such that* $u \mid^w v$.

We represent this as $w = lda(u, v)$ or $w = u \parallel v$. The lda relation is not symmetric. For example, $14 \parallel 39 = 4 \neq 39 \parallel 14$.

Using these definitions, and our previously defined notion of a a bijective column mapping, we have the following lemma that aids in the identification of lowest distinguishing ancestors:

LEMMA **1** *Let* u *and* v *be two column-compatible nodes in the schema graph* $\mathcal{T}$, *where* $annot(u) = annot(v) =$ *R.C and R.C is bijectively mapped. Then* $u \mid^{root(\mathcal{T})} v$ *holds.*

6.2.3 Computing Summary Information from the Constraints

Given an XML-to-Relational mapping $\mathcal{T}$ and the integrity constraints that hold on the underlying relational schema, we precompute the following information

- For each relational column $R.C$, is $R.C$ *bijective*?
- For every pair of column-compatible nodes (u, v), $u \mid\mid v$ and $v \mid\mid u$.

In this computation, we use procedures for solving the following problems on conjunctive queries in the presence of constraints.

UQC: Given a conjunctive query Q_1 and a union of conjunctive queries Q_2, is $Q_1 \subseteq Q_2$ under set semantics?

EQI: Is the intersection of two given conjunctive queries empty?

DUP: Are the results of a given conjunctive query duplicate-free?

We have developed procedures for these three problems by adapting the chase and query containment algorithms proposed in [8]. We have also designed an alternate solution using the algorithm proposed in [25]. In general, any algorithm for conjunctive query containment under set semantics can be used to develop procedures for the above three problems. The details of the above procedures and how we use them to precompute the required summary information are omitted due to lack of space. These details can be found in [14].

6.3 Run-Time Query Translation Algorithm

The run-time query translation algorithm is outlined in Figure 4. Given a path expression query Q, we first identify the parts of the schema that match the query. Let S denote the set of matching schema nodes. For purposes of exposition, we assume that S consists only of leaf nodes, leaving the handling of non-leaf nodes to Section 6.4.1.

We then partition the set S into two sets based on whether the corresponding relational column is bijectively mapped. For the set S_{nonbij}, we construct the root-to-leaf queries just like prior algorithms. On the other hand, for the set S_{bij} we utilize the summary information to eliminate parts of the query that are redundant. This is a two stage process: first we find the longest prefix that can be eliminated for each node $n \in S_{bij}$ (*Prefix-Elimination*); then we construct the SQL query using the prefix-eliminated set of nodes (*SQLGen*). Finally, we union the queries corresponding to the bijective and non-bijective nodes.

We next describe the prefix-elimination and *SQLGen* stages.

6.3.1 Eliminating Redundant Prefixes

The *Prefix-Elimination* algorithm is given in Figure 5. We use the pre-computed information about least distinguishing ancestors in this computation. Instead of taking the naive approach of issuing the full query for each of these nodes and taking their union, we wish, at the very least, to be able to issue a smaller query

procedure *constraint-aware*(Q)
begin
  Let $S \leftarrow NodeId(Q)$
  Partition S into S_{bij} and S_{nonbij}
    based on whether $annot(n)$ is *bijective*
  $SQL_{nonbij} = \bigcup_{n \in S_{nonbij}} rtol(n)$
  Prefix-Elimination(S_{bij})
  $SQL_{bij} = SQLGen(S_{bij})$
  Return $SQL_{bij} \bigcup SQL_{nonbij}$
end

Figure 4: *constraint-aware* **query translation algorithm for path expression queries**

procedure Prefix-Elimination(S)
begin
  for each node $n \in S$ do
    Let $Schema(n)$ denote the set of schema nodes mapped to the same column as n
    Let $Conflict(n) \leftarrow Schema(n) - S$
    Let LDA_Set(n) denote set of $n \mid\mid x$ for every node x in Conflict(n)
    C_lda(n) = highest node in LDA_Set(n)
  While true do
    If $(\exists n, n_1 \in S)$, such that
      RelSeq(C_lda(n),n) and RelSeq(C_lda(n_1),n_1) are not combinable **and**
      $n \mid\mid n_1$ is a strict ancestor of C_lda(n)
    Then
      C_lda(n) = $n \mid\mid n_1$
    Else
      Break
end

Figure 5: **Prefix-Elimination phase**

for each node $n \in S$. Thus, we want to find the lowest ancestor a such that $Query(< a, \ldots, n >)$ returns the correct answer, that is, where the prefix of $rtol(n)$ from root($\mathcal{T}$) to a can be safely eliminated. There are two conditions to check here:

- a must distinguish n from all column-compatible nodes not in S. This computation corresponds to the *for* loop in Figure 5.
- For each column-compatible node $n_1 \in S$, either the two queries are combinable or a distinguishes n from n_1. This corresponds to the *while* loop in Figure 5.

The *while* loop is an iterative process that will terminate in at most $(k * d)$ iterations, where $k = |S|$ and d is the maximum depth (in the XML schema) among all nodes in S. At the end of this process we have the prefix eliminated node sequence for every node in S.

6.3.2 *SQLGen* Stage

We next construct the optimized SQL query by taking the prefix-eliminated set of nodes and grouping multiple paths that involve the same sequence of relations.

Let $\mathcal{NS} = \{<\text{C\_lda}(n),\ldots,n>: n \in \mathit{NodeId}(Q)\}$. Notice that combinability of node sequences is an equivalence relation. We partition $\mathcal{NS}$ based on combinability and construct a SQL query for each equivalence class created. The final SQL query is the union of the queries across all equivalence classes. Notice that all the queries in an equivalence class have the same relation sequence and differ only in the selection conditions. This operation is correct under multi-set semantics because it is only applied to columns that are bijectively mapped.

6.4 Extensions to More General Cases

In this section, we discuss how the methods discussed to up to this point extend to more general situations. Note that our optimization techniques will never generate an incorrect query — they will either not apply (in which case we will generate the naive query) or they will apply and will generate a query expected to be more efficient than the naive query. Hence the discussion here outlines techniques that allow us to apply optimizations to more queries.

6.4.1 Path Expression Queries Involving Non-Leaf Nodes

In our discussion in Section 6.3 on translating path expression queries, we assumed that the query matches a set of leaf nodes in the schema. If the result includes non-leaf nodes as well, then there are two alternative ways of returning the resulting XML elements corresponding to the non-leaf nodes.

1. For each non-leaf element, we can return an identifier or representative subelement. In this case, each non-leaf node n in the schema is associated with a child leaf node n_c. If n appears in a query result, then the corresponding n_c elements are returned instead. For example, we can associate the key field(s) of the corresponding relations with each non-leaf node.

2. For each non-leaf element, we can return the entire subtree rooted at this element. The problem of efficiently constructing entire subtrees of XML documents has been considered in [10, 21]. We leave the interesting problem of combining our algorithm with one of these algorithms for future work.

6.4.2 Beyond Path Expressions

Our techniques can be extended in a straightforward way to handle branching path expression queries (as we show in [14]); because that extension does not provide any additional insight, and due to space constraints, we do not discuss that extension here.

We now briefly describe how to extend *constraint-aware* translation to more general queries. A path expression query corresponds to a single For clause in XQuery. Consider an XQuery that has several of these For clauses and (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 XQ involving two path expressions p_1 and p_2 with a join condition between them. We apply our *constraint-aware* translation on p_1 and p_2 individually to obtain relational queries Q_1 and Q_2 respectively. Note that Q_1 and Q_2 are the union of k_1 and k_2 queries respectively. We generate the query $Q = Q_1 \bowtie Q_2$ as the SQL query corresponding to XQ. If $k_1 > 1$ or $k_2 > 1$, then we could have generated the final SQL query in a number of other ways. For example, we could have distributed the unions over the join and generated the query Q' that is the union of $k_1 * k_2$ queries. Choosing the best query from amongst these (possibly exponential) alternatives is also an interesting area for future work.

6.4.3 Beyond Bijective Mappings

Recall that our technique optimizes the SQL query corresponding to bijective parts of the mapping. It constructs the *baseline* query for the non-bijective parts of the mapping. While we expect bijectively mapped columns to be common, we have extended our algorithm to perform efficient XML to SQL query translation when either the At-least-once or the At-most-once condition is satisfied. We outline the main ideas here with an example and omit the details due to lack of space.

Let us look at the scenario when a relational column $R.C$ satisfies the At-most-once condition but violates the At-least-once condition. For example, consider the example in Figure 2. While the XMark XML schema contains information about items in six continents, in reality, there is actually a seventh continent (Antarctica). So, it is reasonable to assume that the relational schema has an integrity constraint on *Item.continent* allowing seven potential values. In this case, parts of the relational data are not present in the XML view, namely the items corresponding to Antarctica. Now while SQ_1 and SQ_1^1 are correct SQL queries for Q_1, OQ_1 is not. The best query in this scenario will be a variation of SQ_1^1 that combines all the six queries into one, since they are on the same sequence of relations. This query is given below.

```
select count(*)
from   Item I, InCat C
where  I.id = C.itemid and C.category='cat1'
       and I.continent IN {'africa',...,'samerica'}
```

Notice how we were able to group together the six paths corresponding to different continents. This was possible due to the fact that InCat.category satisfied the At-most-Once condition. As a result, the rtol queries corresponding to any two column-compatible schema nodes mapped to InCat.category will not have any common results. So, we can translate the unions

to a disjunction. In other words, we can perform the *SQLGen* phase without any change.

On the other hand, we need to be careful in the prefix-elimination stage. We cannot eliminate any prefix below the continent nodes due to one missing continent in the XML schema. To account for this fact, we have to augment the prefix-elimination stage. We do this as follows: $S = nodes(InCat.category) = \{14, 19, 24, 29, 34, 39\}$. For each schema node $n \in S$, we compute the lowest schema node below which the prefix cannot be eliminated (since the column is not completely exported). Let us call this *lowest required ancestor* (lra(n)). For example, lra(14) = 4 and lra(39) = 9. This ensures that the selection condition on Item.continent is always present in the query.

The lra computation is another summary information that we precompute for schema nodes corresponding to relational columns that violate the At-least-once condition.

7 Conclusion

We have considered the problem of generating efficient SQL queries for XML workloads and showed that published translation algorithms can generate SQL queries that are suboptimal. We consider the problem of where to add the intelligence in order to obtain optimized SQL queries using integrity constraint information. Our results argue that the quality of the resulting SQL should be a concern of the translation algorithm itself, rather being left in the hands of a traditional relational optimizer. This is because many "easy" opportunities for optimization are apparent only when the XML view definition and relational integrity constraints are considered simultaneously. These opportunities vanish by the time the relational optimizer is presented with SQL.

A number of directions for future research exist. Extending our approach to a more general class of XML-to-Relational mappings (including recursive mappings) is an interesting problem. Similarly, looking at a larger class of input XML queries gives rise to other interesting problems. In a different direction, the XML-to-SQL query translation problem also arises in another context: XML Storage, where data that was originally XML is to be stored and queried in an RDBMS (as opposed to the case considered here, where data that was originally relational is to be viewed and queried as XML.) The class of XML-to-Relational mappings produced by existing techniques for XML storage are bijective and there may be alternative ways of computing the summary information without even resorting to relational integrity constraints. Exploring this variant of the problem is another open 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] A. Aho, Y. Sagiv, and J. Ullman. Equivalence among relational expressions. *SIAM J. Comput.*, 8(2), 1979.

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

[4] C. Beeri and M. Vardi. A proof procedure for data dependencies. *Journal of the ACM*, 31(4), 1984.

[5] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *ACM STOC*, 1977.

[6] S. Chaudhuri and M. Y. Vardi. Optimization of real conjunctive queries. In *PODS*, 1993.

[7] A. Deutsch, L. Popa, and V. Tannen. Physical data independence, constraints and optimization with universal plans. In *VLDB*, 1999.

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

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

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

[11] Y. E. Ioannidis and R. Ramakrishnan. Containment of conjunctive queries: beyond relations as sets. *ACM TODS*, 20(3), 1995.

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

[13] D. S. Johnson and A. C. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. In *PODS*, 1982.

[14] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. Efficient XML-to-SQL Query Translation: Where to Add the Intelligence? (full version). `http://www.cs.wisc.edu/ sekar/publications.html`.

[15] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. XML-SQL Query Translation Literature: The State of the Art and Open Problems. In *XML Database Symposium*, 2003.

[16] C. Li, P. Bohannon, H. Korth, and P.P.S. Narayan. Composing XSL Transformations with XML Publishing Views. In *SIGMOD*, 2003.

[17] I. Manolescu, D. Florescu, and D. Kossman. Answering XML queries over heterogeneous data sources. In *VLDB*, 2001.

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

[19] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM (JACM)*, 27(4), 1980.

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

[21] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *VLDB*, 2000.

[22] S. T. Shenoy and Z. M. Ozsoyoglu. A system for semantic query optimization. In *SIGMOD*, 1987.

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

[24] X. Zhang and Z. M. Ozsoyoglu. On efficient reasoning with implication constraints. In *DOOD*, 1993.

[25] X. Zhang and Z. M. Ozsoyoglu. Implication and referential constraints: A new formal reasoning. *TKDE*, 9(6), 1997.