Optimizing Fixed-Schema XML to SQL Query Translation

Rajasekar Krishnamurthy Raghav Kaushik Jeffrey F. Naughton
{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. Motivated by these performance-enhancing opportunities, we present a *constraint-aware* XML to SQL query translation algorithm for path expression queries.

1 Introduction

Currently, there is a lot of interest in supporting XML queries on data that is stored in an RDBMS. To date, there have been two main categories of research relevant to this problem. In the first category, the XML schema is fixed, but the relational schema is not, so the problem is to find a good relational schema for a given XML data set and workload. In the second category, the relational schema is fixed but the XML schema is not, so the problem is how to support (possibly many) user-defined XML views of the relational data. By contrast, in this paper we consider a third category, in which both the XML and relational schemas are fixed. Here the challenge is to translate XML queries over the (pre-defined) XML schema into SQL queries on the (pre-defined) relational schema. Toward this end we show how to exploit constraints on the relational data to improve the translation process. The impact of such constraint-aware translation is non-trivial — in our experiments with a commercial DBMS, our techniques yielded speedups of up to a factor of 90 over the current state of the art.

Our techniques also apply to the previously-studied scenarios in which one of the relational or the XML schema are not fixed. However, we have chosen to focus on the fixed-to-fixed problem for two reasons. First, we expect the fixed-to-fixed schema scenario to arise frequently in practice, perhaps more frequently than either of the "free" schema translation problems. The fixed-to-fixed translation problem arises in any situation in which the relational and XML schemas are defined separately. For example, if an organization has legacy relational data sets that it wants to integrate with an XML schema defined by an external standards body, it will face a fixed-to-fixed query translation problem. Second, the fixed-to-fixed case is in some sense more fundamental, because even in the scenarios in which either the XML or the relational schema can be freely defined, once a good schema has been chosen we are back in the fixed-to-fixed translation scenario.

Our main idea is to use constraints on the relational data in the query translation process. Utilizing constraints during query optimization in relational systems is a well-studied technique [13, 22, 27, 28]. In fact, current commercial relational databases use some simple constraints during query optimization. For example, 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 member has to be associated with a department (through foreign key and not-null constraints on the Faculty.did column). Then the relational optimizer can optimize this query to

```
Select count(*)
From Faculty
```

Commercial relational databases including IBM DB2, Microsoft SQLServer and Oracle perform the above optimization. We observed (through experiments) that current commercial relational optimizers use key and foreign key constraints during query processing to eliminate redundant parts of queries (as illustrated in the above example). One commercial optimizer uses domain constraints if there is an exact syntactic match between the constraint and the SQL query. Beyond that they do not seem to use other constraints during query optimization. This is perhaps a good decision on their part, because (a) the opportunities for using more complex constraints are perhaps not too common in traditional relational workloads and (b) utilizing more complex constraints during SQL query optimization can be expensive [13, 28]. Given (a) and (b), it may not be wise to penalize

the optimization of every query with an expensive constraint analysis phase that is likely to yield little (or no) improvement.

While expanding the use of constraints in query optimization for generic relational queries may not be a good idea, this is not the case when considering translating XML queries into SQL queries. Due to the huge difference between the XML and relational data models and the corresponding query languages (especially the feature of wild cards in XML queries), as we will show, opportunities for constraint-based optimization are very likely to occur in the fixed-schema translation arena. Moreover, the mapping from the XML schema to the relational schema provides hints about the class of queries we can expect to see. Utilizing these hints, it is possible to pre-compute information that makes the optimization cost of using constraints during XML to SQL query translation much more efficient than it is in the generic relational optimization case. In this paper, we follow this approach and present a *constraint-aware* XML to SQL query translation algorithm for path expression queries.

We first illustrate the benefits of performing a *constraint-aware* translation using a sample query on an (industry standard) XML schema [23] in Section 2. Next we present a formal framework for representing the XML to Relational mapping (Section 3). We then present our methodology for capturing the underlying constraints as properties of the XML to Relational mapping (Section 4). These properties are inferred from the constraints in a pre-computation phase. We then present a *constraint-aware* XML to SQL query translation algorithm that uses these properties (Section 5. Since our algorithm only looks at the summarized properties, there is not much overhead added to the actual XML to SQL query translation process. Our experiments (Section 6) show that when the XML query has wild cards in it, this approach yields significant performance speedups (in terms of the execution times of the resulting SQL queries) over current translation algorithms [9, 11, 19]. We then extend the algorithm to handle branching path expression queries in Section 7 and briefly discuss extensions to non-bijective mappings in Section 8.

2 Motivation

Let us consider the example XML schema in Figure 1, which is based on the standard DTD being developed by the Newspaper Association of America Classified Advertising Standards Task Force [23]. 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. The DTD has over 350 elements and several hundred attributes. Each of the elements in the DTD has many attributes and subelements that are not shown in the figure.

Each XML document conforming to this DTD contains information about one or more ad-

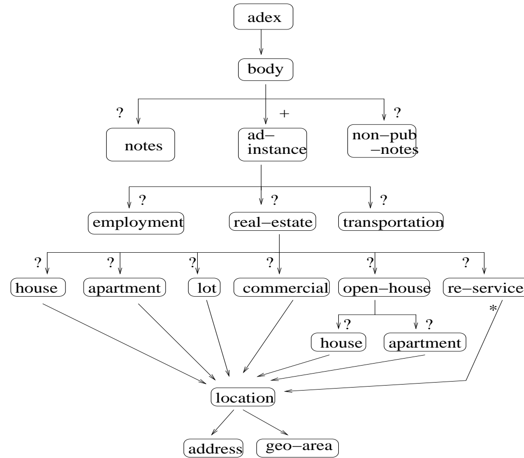
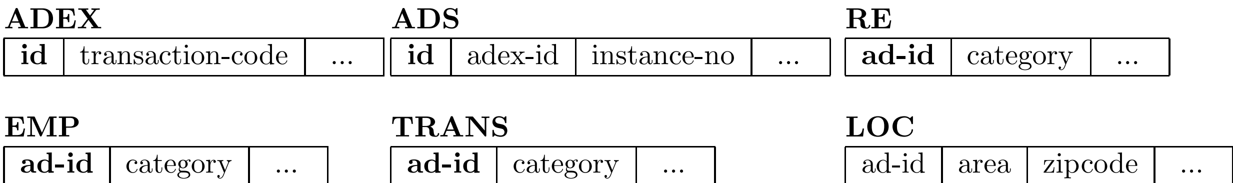


Figure 1: **ADEX DTD for newspaper advertisements**



ADEX

id	transaction-code	...

ADS

id	adex-id	instance-no	...

RE

ad-id	category	...

EMP

ad-id	category	...

TRANS

ad-id	category	...

LOC

ad-id	area	zipcode	...

Figure 2: **Sample Relational schema for advertisements data**

vertisements. 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 belongs 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.

Consider now how data about classified ads might already be represented by a newspaper company. Almost certainly the data is in a relational database; the schema might look like that given in Figure 2. In this relational schema, all ad instances are stored in an *ADS* relation. Information specific to real-estate ads is stored in the *RE* relation. The information separating various categories of real estate is stored in the *category* 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 *LOC* relation.

Suppose that the DBA working for the newspaper knows that the following constraints hold on the relational 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.

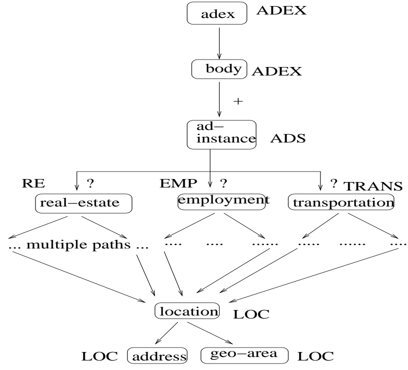


Figure 3: **Sample relational mapping σ for ADEX schema**

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 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, together they exhaust the values in ADS.id.

Consider how the industrial standard XML schema might be mapped to this pre-existing relational schema. One possible mapping from the standard XML schema to this pre-existing relational schema is shown in Figure 3. Each node in the schema has a table name next to it, to indicate the relational table that corresponds to this element. We look at this in more detail in Section 3.

Let us now examine how queries might be translated from XML queries to relational queries in this setup. Consider the following XML query XQ, *Return all addresses in the campus area:*

```
//adinstance//location[geo-area=‘campus’]/address
```

A straightforward approach to do this translation, as described in [11], is to identify all matching paths for the wild cards, issue a separate SQL query for each such path, and take the union of the results. The resulting query SQ is shown below. The 20 unions shown correspond to the 20 matching paths for the adinstance//location wild card (seven through real-estate, four through employment and nine through transportation).

```
select LOC.address
```

```
from   ADEX, ADS, RE, LOC
where  ADEX.id = ADS.adexid and ADS.id = RE.ad-id
       and RE.category = ‘house’ and
       RE.ad-id = LOC.ad-id and LOC.area = ‘campus’
union all ... (20 times)
```

For the above query, the optimizers in current relational systems will use foreign key constraints to eliminate redundant joins whenever possible. For instance, the join between ADEX and ADS can be removed. Furthermore, the join between ADS and RE becomes redundant and can be removed. Thus the query as rewritten by a relational optimizer becomes the new query SQ_1:

```
select LOC.address
from   RE, LOC
where  RE.category = ‘house’ and
       RE.ad-id = LOC.ad-id and LOC.area = ‘campus’
union all ... (20 times)
```

We have seen how existing commercial RDBMS optimizers convert SQ to SQ_1. A reasonable question is whether the more sophisticated XML to SQL translation routines proposed in SilkRoute [15] and in Xperanto [12] can do better. It is true that by merging common subexpressions, they generate a better initial query than SQ. But, interestingly, the final optimized query that results when you feed their initial queries to a relational optimizer is once again SQ_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.

By contrast, the *constraint-aware* algorithm we present in this paper will translate XQ to the SQL query *optQ* given below.

```
select LOC.address
from   LOC
where  LOC.area = ‘campus’
```

We observed from our experiments using DB2 that the query *optQ*, yields a speedup of a factor of 27 over the query SQ_1 (generated by state of the art algorithms) on a sample dataset (see Section 6 for details). This clearly shows that for path expression queries, especially involving wild cards, the performance benefit of a constraint-aware XML to relational query translation algorithm is significant.

3 Fixed-Schema Mappings

In order to express our translation techniques, we need a representation for XML to Relational mappings. Any reasonable representation would serve our purpose; for concreteness, in this section we present one precise way to represent XML to Relational mappings. This approach is similar to those used in Microsoft SQLServer, where XML views can be created using annotated XDR schemas [24]; and in XML-DBMS [25], a middleware system for transferring data between XML documents and relational databases.

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 labeled with the name of the type and the edges are labeled 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 4. This represents a part of the ADEX DTD from Figure 1. Note that the DTD in Figure 1 corresponds to a single XML document exchanged by two organizations. We have added an advertisements root element in Figure 4 to represent multiple XML documents. We have used integers as vertex labels to represent the types and omitted the multiplicity labels for clarity.

In this paper, we do not consider recursive XML schemas. While recursive schemas are an interesting area for future work, as we will see, even non-recursive schemas present challenges and opportunities for optimization. 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. Note how we obtained the *Tree* schema in Figure 4 from the *DAG* schema in Figure 1 by replicating the *location* subtree. Hence, we consider *tree* schema graphs in this paper.

3.2 XML to Relational Mapping

We represent the mapping between XML elements and relational columns through annotations on the schema graph. For example, the annotations on the schema graph in Figure 4 represent the XML to Relational mapping corresponding to the relational schema 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 advertisements element is the root of the document. For each tuple in the ADEX relation, an adex element is created. This is represented by the annotation on the edge from node 0 to node 1 (called $e_{0,1}$). A body element is created within each adex element. Note the

lack of any annotation on $e_{1,2}$. The ad-instance elements corresponding to the various ad instances are created based on the join condition that annotates $e_{2,3}$. If the current instance corresponds to a *real-estate* ad, a real-estate element is created. This is represented as a join condition on $e_{3,4}$. Furthermore, if it is an ad for a *house*, a house element is created. The corresponding selection condition annotates $e_{4,5}$. The location elements corresponding to this ad are nested within house. For each location, address and geo-area subelements are created. The value of the address element is represented as the node annotation LOC.address.

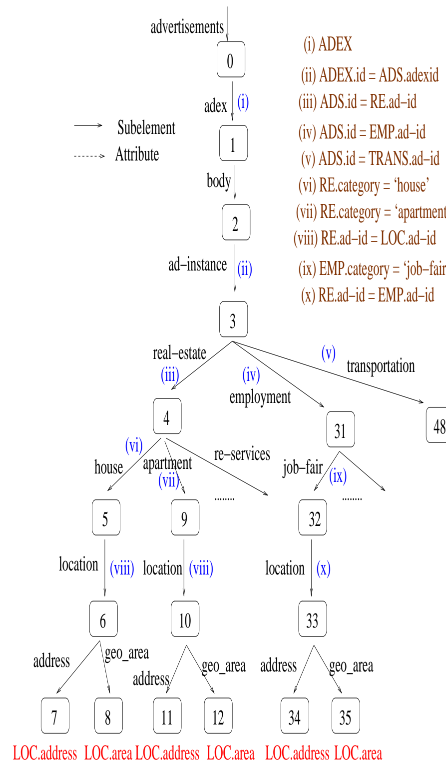


Figure 4: **Annotated schema graph**

More formally, the following edge annotations are allowed. Here, R.C represents column C of relation R.

1. $<R>$: an initial annotation.
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 on 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

1. an initialization condition $<R>$ (here p has to be root(T)), then $Rel(n) = R$.
2. a selection condition involving the column $R.C$, then $Rel(n) = R$. Here we must have that $Rel(p) = R$.
3. a join condition of the form $<R_p.C_p, R_c.C_c>$, then $Rel(n) = R_c$. Here, we must have that $Rel(p) = R_p$.
4. a null annotation, then $Rel(n) = Rel(p)$.

Node annotations are allowed only on leaf nodes. For a node n, $annot(n) = R.C$ is a column in the relation $Rel(n) = R$. This annotation indicates that the values in the column $R.C$ are exported (in the XML view) as values of elements corresponding to the schema node n. We note here that we can extend the above class of mappings to allow extra features such as internal nodes having mixed content. Our techniques extend easily to cover such extensions, although we do not discuss them here due to space constraints.

3.3 Terminology

We now cover some terminology that will be used in the rest of the paper. Since existing XML to SQL query translation techniques in the literature do not use any constraints in the translation process, we refer to them as *constraint-oblivious* algorithms. Often in the following discussion we will need to refer to a relational column $R.C$ and the set of leaf nodes in an XML schema that are annotated with $R.C$. For conciseness, we define *nodes(R.C)* to be this set of leaf nodes annotated with $R.C$. We call two leaf nodes *column-compatible* if they are annotated with the same relational column.

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 $n_k = NS.last$. For any node sequence NS, the relational query Query(NS) is obtained by combining the conditions on the edges of the sequence and projecting $annot(n_k)$. For any node sequence NS, 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.

Let RelSeq(NS) denote the sequence of relations joined in Query(NS) in a bottom-up order. For example, in our running example, for $NS =< 3, 4, 5, 6, 8 >$, RelSeq(NS) = <LOC,RE,ADS>. Notice that successive relations in this sequence are related by a join condition on the path. A relation in this sequence is also associated with a (possibly null) selection condition, which is the conjunct of multiple conditions on the edges. For example, for the above sequence, the relation RE is associated with the condition RE.category = 'house'.

We say that two node sequences NS_1 and NS_2 are *combinable* if the corresponding relation

sequences $RelSeq(NS_1)$ and $RelSeq(NS_2)$ are the same and $NS_1.last$ and $NS_2.last$ are column-compatible. Note that combinability is an equivalence relation. Combinable node sequences are important in deciding when we can rewrite a union query, say
(Query(NS_1) union all Query(NS_2)), as a basic SQL query without unions. This is because if NS_1 and NS_2 are combinable, then Query(NS_1) and Query(NS_2) can be grouped together eliminating the union. This conversion is valid under multi-set semantics if the projected relational column is *bijectively* mapped, as we will see in the next section. In this paper, when we talk about unions, we refer to the union all operation.

4 Utilizing Constraints: Precomputation Phase

Recall that a key idea in our approach to constraint-aware translation is to use the given constraints on the underlying relational data along with the XML to Relational mapping to precompute information that is useful during subsequent query translation. In this section, we describe this summary information. To motivate this summary information, note that we use two main optimizations in our XML to SQL translation: (i) merging queries corresponding to multiple satisfying paths, and (ii) eliminating prefixes of the multi-way joins that result from translating XML path traversals to SQL. To enable the detection of queries that can be merged, we identify in the precomputation phase the *bijective* column mappings. To enable the identification of prefixes that can be eliminated, we precompute the *lowest distinguishing ancestor* information for every pair of column-compatible schema nodes.

4.1 Identifying Bijective Column Mappings

Let us look at how conceptually we can use the complex constraints to optimize SQ_1 and obtain *optQ*. The first step will involve grouping some of the unions in the following manner. There are seven paths through real-estate. The corresponding queries are on the RE and LOC relations. They differ only in the selection condition on RE.category. Since the selection conditions are non-overlapping, the corresponding query results will also be non-overlapping. As a result, these seven paths can be *grouped* into a single SQL query having a disjunctive condition on the RE.category column. By using 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. Let the rewritten query be SQ_2. Parts of SQ_1 and SQ_2 corresponding to the *real-estate* schema fragment are shown below.

SQ_1: SQ_2:

```
select LOC.address            select LOC.address
```

```
from   RE, LOC                 from   RE, LOC
where  RE.category = ’house’  where  RE.ad-id = LOC.ad-id
   and RE.ad-id = LOC.ad-id      and LOC.area = ’campus’
   and LOC.area = ’campus’
 union ... (7 paths)
```

In general, to convert a union query into a disjunction under multiset semantics, one needs to make sure that no result tuple is generated in multiple branches of the union operator. This problem reduces to verifying whether the results of the queries corresponding to any two branches have a null intersection, which is unfortunately an expensive operation to perform as part of each query translation. Thus, it is important to identify opportunities for such simplifications beforehand. Toward this purpose, in the precomputation phase, we identify whether any data is exported multiple times. In the above example, we are able to convert the union into a disjunction since every LOC tuple is exported at most once.

Similarly, suppose that *real-estate* had eight categories instead of seven and that the eighth category were not present in the XML view. In this case, SQ_1 could not be rewritten as SQ_2. While the union could still be converted into a disjunction, the disjunction could not be removed due to the missing eighth category. The optimized query in this case would be the SQL query SQ_3 given below:

```
select LOC.address
from   RE, LOC
where  RE.ad-id = LOC.ad-id and LOC.area = ‘campus’
       and 1 <= RE.category and RE.category <= 7
union all
select LOC.address
from   EMP, LOC
where  EMP.ad-id = LOC.ad-id and LOC.area = ‘campus’
union all
select LOC.address
from   TRANS, LOC
where  TRANS.ad-id = LOC.ad-id and LOC.area = ‘campus’
```

The problem is that in this case, some of the relational data is not exported at all, and this makes a big difference in the fully optimized equivalent SQL query (compare SQ_3 with *optQ*). So, in the precomputation phase we need to identify the situations when some of the relational data is not exported at all.

To summarize, we need to identify parts of the data corresponding to the query result that are exported “exactly once.” In order to formalize the notion of “exactly once”, we associate an

SQL query $fullQuery(n)$ with every leaf node n of the schema graph. This query returns the values of all elements corresponding to schema node n. $fullQuery(n) = \text{Query}(NS)$, where NS is the node-sequence corresponding to the root-to-leaf path in to n. For example, $fullQuery(7)$ in Figure 4 is

```
select LOC.address
from   ADEX, ADS, RE, LOC
where  ADEX.id = ADS.adexid and ADS.id = RE.ad-id
       and RE.ad-id = LOC.ad-id and RE.category = 'house'
```

We also define $fullKeyQuery(n)$ to be keyQuery(NS).

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 $\bigcup_{n \in nodes(R.C)}$ fullKeyQuery(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, the containment operations are assumed to use multi-set semantics.

As an example of the application of these terms, in our running real estate ad example, the LOC.address column is bijectively mapped.

4.2 Lowest Distinguishing Ancestor

We have seen that in the adex schema in Figure 4, the relational column LOC.address is bijectively mapped. Consider the XML query XQ_1, //real-estate/house/location/address, which returns addresses of all houses available for rent. All elements corresponding to schema node 7 will appear in this query result. An equivalent SQL query is $fullQuery(7)$. The question arises whether we need to issue the full query corresponding to node 7. For this example, using the constraints on the data, we can see that Query($< 4, 5, 6, 7 >$) is also a valid SQL translation. To identify such opportunities for query simplification, we introduce the notion of *lowest distinguishing ancestor*.

To do the preceding simplification, we need to identify an ancestor n of node 7 such that Query($< n, \ldots, 7 >$) is a correct translation. Now, Query($< n, \ldots, 7 >$) will certainly contain all the results that we want, with their correct multiplicity since LOC.address is bijectively mapped. However, it could potentially contain additional values that are not to be returned, because we

have omitted a prefix of the join. Thus, for correctness, we need to find an ancestor n that *distinguishes* 7 from all other column-compatible leaf nodes. For this, we need to check if any LOC.address value that appears in the result of the query Query($< n, \ldots, 7 >$) also appears in the result of $fullQuery(n')$, for some column-compatible $n' \neq n$. Since LOC.address may have duplicate values, we need to add the key column(s) of LOC to the SQL queries. This ancestor will be the lowest distinguishing ancestor; we make all this precise with the following definitions.

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 $n_k = u$, be the node-sequence representing 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 fullKeyQuery(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 7 with respect to every other *column-compatible* node. Observe that the *distinguishing ancestor* relation is not a symmetric relation. For example, in the annotated schema graph shown in Figure 4, consider schema node 34, an address element under employment. Now, $7 \mid^5 34$ is true. Notice that node 5 is an ancestor of node 7 but not an ancestor of node 34. So, $34 \mid^5 7$ is false.

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

We represent this as $w = lda(u, v)$ or $w = u \mid\mid v$. The *lda* relation is not symmetric. For example, $7 \mid\mid 34 = 5 \neq 34 \mid\mid 7$.

Using these definitions, and our previously defined notion of a a bijective 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.*

4.3 Computing Summary Information from the Constraints

Given an XML schema, a relational schema, constraints on the relational data, and an XML to Relational mapping, 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 $v \mid\mid u$.

```
procedure IsColumnBijective(R.C)
begin
    Let S ← nodes(R.C)
    For each node u ∈ S,
        If fullKeyQuery(u) can have duplicates
            return false
    For each pair of nodes u, v ∈ S
        If fullKeyQuery(u) ∩ fullKeyQuery(v) ≠ φ
            return false
    If keyProject(R.C) ⊆ NodeKeyProject(R.C)
        return false
    return true
end
```

Figure 5: **Algorithm to check if a relational column is bijectively mapped**

4.3.1 Identifying Bijective Column Mappings

To check if a relational column $R.C$ is bijectively mapped, we need to check if $R.C$ is both At-least-once and At-most-once mapped.

Let us first look at the At-least-once property. Recall from Definition 1 that for this we need to check if

$KeyProject(R.C) \subseteq NodeKeyProject(R.C)$ under multiset semantics. Since the key columns are also projected, the left hand side query has no duplicates. So, performing conjunctive query containment under set semantics will suffice.

Similarly, to check whether the At-most-once property is satisfied by $R.C$, we need to check if $KeyProject(R.C) \supseteq NodeKeyProject(R.C)$ under multiset semantics. Notice that this containment holds trivially under set semantics. Moreover, since the left hand side query has no duplicates, it suffices to check if the right hand side query also has no duplicates. Recall that $NodeKeyQuery(R.C) =$

$\bigcup_{u \in nodes(R.C)} fullKeyQuery(u)$. Duplicates can be produced either in a single component of the union or across two different components. For the former case, we check if, for each $u \in nodes(R.C)$, any result tuple in $fullKeyQuery(u)$ has two or more valuations. For the latter case, we need to check if $fullKeyQuery(u)$ and $fullKeyQuery(v)$ have a non-null intersection, for all node pairs $u, v \in nodes(R.C)$.

The algorithm to determine if a relational column is bijectively mapped is given in Figure 5.

4.3.2 Computing least distinguishing ancestor information

The other summary information that we compute is the *lda* information. If we have a subroutine that verifies whether or not $u \mid^{w} v$ holds, then we can process the ancestors of u in a bottom-up

```
procedure ldaComputation(u,v)
begin
  Let currAncestor = u
  while (true) do
    Let NS =<n_j, ..., n_k = u>
    If keyQuery(NS) ∩ fullKeyQuery(v) = φ
      return currAncestor
    If (currAncestor = root(T))
      return null
        // this occurs if At-most-once condition is violated
    currAncestor = parent(currAncestor)
end
```

Figure 6: **Algorithm to compute** *lda***information**

fashion, and obtain $u \mid\mid v$. By definition, to check if $u \mid^{w} v$ is true we need to check if the intersection of the results of the queries keyQuery($<n_j, \ldots, n_k = u>$) and $fullKeyQuery(v)$ is empty, where $n_j = w$. The algorithm is given in Figure 6.

In the above algorithms, we have used procedures for solving the following problems on conjunctive queries in the presence of constraints.

QC: 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?

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

We adapt the chase and query containment algorithms proposed in [2] to solve the above problems. The procedures we obtain have one-sided errors, where a "yes" answer to any of the above questions is always correct. In other words, we might miss out on identifying certain bijective column mappings or identify a higher node as *lda*for a given pair of nodes, thus missing certain optimization opportunities. But, our algorithm will always produce a correct summary information. We briefly present the details of our procedures in Appendix A.

5 Run-Time Query Translation

In this section, we present our *constraint-aware* query translation algorithm. We focus here on the class of simple path expression queries, i.e., path expressions without branching. We then extend our techniques to handle branching path expression queries in Section 7.

```
procedure NodeId(Q)
begin
   Let Q = s1 l1 s2 l2 ... sk lk.
   CurrentNodes = {root(T)}.
   NextNodes = {}.
   For i = 1 to k
      For each node n ∈ CurrentNodes
         If si is /
            Add all child nodes of n with element
            name li to NextNodes.
         If si is //
            Add all descendant nodes of n with
            element name li to NextNodes.
      CurrentNodes = NextNodes.
      NextNodes = {}.
   Return CurrentNodes.
end
```

Figure 7: *NodeId* **algorithm for simple path expressions**

5.1 Translating 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.) The result of such a path expression is the set of all nodes that match the path expression query.

Evaluating a path expression query over an XML schema that has been mapped to a relational schema can be viewed as a two stage process: (i) using the XML query to identify the vertices in the XML schema graph that satisfy the query, and (ii) using the annotations from the XML to relational mapping of the XML schema to construct an equivalent relational query. We refer to these stages as the *NodeId* and *RelationalQueryGeneration* stages respectively.

In the *NodeId* stage, we execute the path expression query on a schema graph and identify the satisfying nodes. The algorithm is given in Figure 7. Let S be the set of schema nodes returned by this stage. For purposes of exposition, in this section we assume that S consists only of leaf nodes.

A simple *constraint-oblivious* algorithm to generate an SQL query corresponding to a given path expression is to return $\bigcup_{n \in S} fullQuery(n)$. We call this the *naive* algorithm. As we have seen, this query, while correct, can be very inefficient. In our *constraint-aware* algorithm, we exploit situations where a relational column labeling the nodes in S is bijectively mapped, utilizing available relational constraints to perform a more efficient translation in these cases. As we have mentioned, the main optimizations we perform can be divided into two techniques: eliminating unnecessary path prefixes (*Prefix-Elimination*) and grouping the resulting queries whenever they are on the same set of relations (*Grouping*). The algorithm to do this is given in Figure 8. We explain the

```
procedure constraint-aware-translation(Q)
begin
    Let S ← NodeId(Q)
    For each node n ∈ S,
      If annot(n) is bijective
        S_bij = S_bij ⋃{n}
      Else
        S_nonbij = S_nonbij ⋃{n}
    Prefix-Elimination(S_bij)
    SQL_bij = Grouping(S_bij)
    SQL_nonbij = ⋃_{n∈S_nonbij} fullQuery(n)
    Return SQL_bij ⋃ SQL_nonbij
end
```

Figure 8: *constraint-aware* **query translation algorithm for simple path expressions**

```
procedure Prefix-Elimination(S)
begin
  for each node n ∈ S do
    Let Schema(n) denote the set of schema nodes mapped to the same column as n
    Let Conflict(n) ← Schema(n) − S
    Let LDA_Set(n) denote set of n || x for every node x in Conflict(n)
    Conflict_lda(n) = highest node in LDA_Set(n)
  While true do
    If (∃n1, n2 ∈ S), such that
      RelSeq(Conflict_lda(n1),n1) and RelSeq(Conflict_lda(n2),n2) are not combinable
      and n1 || n2 is a strict ancestor of Conflict_lda(n1)
    Then
      Conflict_lda(n1) = n1 || n2
    Else
      Break
end
```

Figure 9: **Prefix-Elimination for simple path expressions**

prefix-elimination and grouping stages in more detail in the next two sections.

5.1.1 Eliminating Redundant Prefixes

The *Prefix-Elimination* algorithm is given in Figure 9. The input to this algorithm is a set of schema nodes returned by the *NodeId* phase. 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 for each node n in this set. Thus, for each node n in S_{bij}, we want to find the lowest ancestor a such that $Query(< a, \ldots, n >)$ returns the correct answer, that is, where the prefix of $fullQuery(n)$ from root($\mathcal{T}$) to a can be safely eliminated. Now, a must distinguish n from all nodes not in S_{bij}. Hence, we first identify the highest among *lda*'s of n against all other column-compatible nodes not in S_{bij}. Let us call this Conflict_lda(n). This computation corresponds to the

for loop in Figure 9. We use the pre-computed information about least distinguishing ancestors in this computation.

By issuing the prefix-eliminated query Query(<Conflict_lda(n),...,n>) instead of $fullQuery(n)$, we get the *set* of results we want. However, by eliminating the prefix up to $Conflict_lda(n)$ independently for each node n, the resulting query may return an answer multiset with the incorrect multiplicity of some result tuples. The problem arises when some tuple appears in the results of the prefix-eliminated queries associated with n_1 and n_2 in S_{bij} when it should really appear in the result of only one of them, say n_1. To keep this from occurring, we ensure that for nodes n_1, n_2 in S_{bij}, the corresponding prefix eliminated queries can either be grouped into a single query or their results do not have any common tuples. In other words, if the node sequences <Conflict_lda(n_1), ..., n_1> and <Conflict_lda(n_2), ..., n_2> are not combinable, then we check if the intersection of the results of the corresponding queries is guaranteed to be empty. If not, we increment Conflict_lda of n_1 and n_2 appropriately. This is an iterative process that will terminate in at most $(k * d)$ iterations, where $k = |S_{bij}|$ and d is the maximum depth (in the XML schema) among all nodes in S_{bij}. This can be seen by noting that in each iteration, the Conflict_lda of at least one node in S_{bij} incremented by one level. At the end of this process we have the prefix eliminated node sequence for every node in S_{bij}.

5.1.2 Grouping Multiple Paths

We next construct the optimized SQL query by taking the result of the previous stage and grouping multiple paths that involve the same sequence of relations. Let $\mathcal{NS} = \{$<Conflict_lda(n),...,n >: $n \in NodeId(Q)\}$. Recall that combinability of node sequences is an equivalence relation. We partition $\mathcal{NS}$ based on combinability and issue a basic SQL query for each equivalence class created. Let NS_1 and NS_2 be two combinable node sequences. Query(NS_1) and Query(NS_2) are combined to yield a single query as follows. Because NS_1 and NS_2 are combinable, Query(NS_1) and Query(NS_2) involve the same relations. Thus, the from clause of the grouped query contains these relations. Let C_{common} denote the set of conditions that are common to both Query(NS_1) and Query(NS_2). Let C_i denote the conditions corresponding to NS_i that are not present in C_{common}. The where clause has the condition (C_{common} and (C_1 or C_2)). This solution generalizes when we have to combine more than two node sequences. This operation is correct under multi-set semantics because it is only applied to columns that are bijectively mapped. The overall algorithm for grouping the node sequences together and obtaining the final SQL query is given in Figure 10.

Recall that the output of the *naive* algorithm on a query Q is $\bigcup_{n \in S} fullQuery(n)$, where $S = NodeId(Q)$.

THEOREM **1** *Let P be an XML path expression query. Let Q_1 be the output of naive query trans-*

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

Figure 10: **Grouping phase for simple path expressions**

lation algorithm and Q_2 be the output of the constraint-aware query translation algorithm. Then, for all database instances D satisfying the given set of constraints, $Q_1(D) = Q_2(D)$ under multiset semantics.

Proof : For a given path expression query Q, let Q_1 be the SQL query obtained by applying our *constraint-aware*-translation algorithm. Let Q_2 be the SQL query obtained by applying the simple *constraint-oblivious* translation algorithm. Recall that $Q_2 = \bigcup_{n \in S} fullQuery(n)$, where $S = NodeId(Q)$. We need to show that $Q_1 = Q_2$ under multiset semantics.

From the algorithm in Figure 8, we have

$$Q_1 = SQL_{bij} \bigcup SQL_{nonbij}$$

Similarly, Q_2 can be written as

$$Q_2 = \bigcup_{n \in S_{bij}} fullQuery(n) \quad \bigcup \quad \bigcup_{n \in S_{nonbij}} fullQuery(n)$$

By definition, we have

$$SQL_{nonbij} = \bigcup_{n \in S_{nonbij}} fullQuery(n)$$

So, we need to show that

$$SQL_{bij} = \bigcup_{n \in S_{bij}} fullQuery(n)$$

Since each relational column being projected in these two queries is *bijectively* mapped, it will suffice if we prove that

$$keySQL_{bij} = \bigcup_{n \in S_{bij}} fullKeyQuery(n)$$

Here $keySQL_{bij}$ refers to the query SQL_{bij} augmented with the appropriate key column(s) in the projection clause.

$keySQL_{bij} \supseteq \bigcup_{n \in S_{bij}} fullKeyQuery(n)$**:** We have to show that if a tuple t appears in the result

of the RHS query, then it appears in the result of the LHS query as well. Let t belong to relation R and $R.C$ be the *bijective* column being projected. Let n_1 be the schema node, such that, t occurs in the result of $fullKeyQuery(n_1)$. Consider the same node n_1 in the LHS query, $keySQL_{bij}$. Since node $n_1 \in S_{bij}$, at the end of the Prefix-Elimination stage in our *constraint-aware* algorithm, we have a node sequence $NS =< Conflict_lda(n_1), \ldots, n_1 >$ corresponding to node n_1. In the Grouping phase, we create a basic SQL query (say Q_3) corresponding to RelSeq(NS). It can be seen that tuple t occurs in the result of query Q_3. This is due to the fact that the relation sequence corresponding to Q_3 is a suffix of the relation sequence corresponding to $fullKeyQuery(n_1)$. Moreover, the where clause in Q_3 is of the form C_{NS} *or* (conditions corresponding to other nodes with same relation sequence as n_1). Notice that C_{NS} is a subset of the where clause of $fullKeyQuery(n_1)$. As a result, tuple t occurs in the result of query Q_3. This implies that t occurs in the result of $keySQL_{bij}$ (by construction). Hence, we have the required containment result under set semantics.

Since $R.C$ is *bijectively* mapped, t appears exactly once in the result of the RHS query. So, we have the containment result under multiset semantics.

$keySQL_{bij} \subseteq \bigcup_{n \in S_{bij}} fullKeyQuery(n)$**:** Here, we have to show that if a tuple t occurs in the result of the LHS query, it also occurs in the result of the RHS query. Since the column under consideration is *bijectively* mapped, there are no duplicates in the result of the RHS query. So, we also have to show that there are no duplicates in the LHS query.

Let us first show that if a tuple t occurs in the result of the LHS query, it also occurs in the result of the RHS query. Since the corresponding column is *bijectively* mapped, there is some schema node n, such that, $fullKeyQuery(n)$ has t in its result set. We next show that $n \in S_{bij}$. Let us assume that the tuple is produced by a basic SQL query Q_4 in $keySQL_{bij}$. Let the where clause of Q_4 be of the form C_{common} and (C_{n_1} or C_{n_2} or $\ldots or$ or C_{n_k}). We can identify a condition C_{n_i} that produces tuple t in the result. Let n_i be the corresponding schema node. Now t appears in the results of both the prefix-eliminated query for n_i and $fullKeyQuery(n)$. This implies that either $n_i = n$ or $n \in S_{bij}$ and the prefix-eliminated queries of n_i and n are combineable. Otherwise, the *lda* for n_i would have been a higher ancestor. In either case, we have that $n \in S_{bij}$. Combining this with the fact that t appears in the result of $fullKeyQuery(n)$, we have that t appears in the result of $\bigcup_{n \in S_{bij}} fullKeyQuery(n)$.

We next show that there are no duplicates in the LHS query. Assume the contrary. Suppose a tuple t appears more than once in the result. Since the relational column being projected is *bijectively* mapped, a single basic SQL query in $keySQL_{bij}$ will not produce duplicates. So, there are two basic SQL queries in $keySQL_{bij}$ that have t in their result set. Let these queries be Q_5

and Q_6. As in the previous case, we can find the corresponding schema nodes n_5 and n_6 that cause t to appear in the result of Q_5 and Q_6 respectively. Let NS_5 and NS_6 be the corresponding prefix-eliminated node sequences. Since the column is *bijectively* mapped, there is some schema node n, such that, $fullKeyQuery(n)$ has t in its result set. Using the same argument as in the previous case, we can show that $n \in S_{bij}$. Let NS be the prefix-eliminated node sequence for n. We can show that NS and NS_5 are combineable. Similarly, we can show that NS and NS_6 are also combineable. Since combinability is an equivalence relation, this implies that NS_5 and NS_6 are also combineable. This contradicts the assumption that n_5 and n_6 belong to two different basic SQL queries. So, there are no duplicates in the LHS query.

5.2 Example Query Translations

In this section, we illustrate our translation algorithm for some simple path expression queries over our classified ad schemas.

First consider the translation of the query
Q_1 = //adinstance//location/address. The result of the *NodeId* phase is $S = \{7, 11, 15, \ldots\}$, a set of 20 schema nodes corresponding to the 20 simple paths to an *address* element. All the 20 nodes are mapped to the same relational column LOC.address, which is *bijectively* mapped. So, $S_{bij} = S$. There is no schema node that is mapped to LOC.address but is not present in S. Hence, in the prefix elimination stage, $\forall n \in S, Conflict_lda(n) = n$. As a result, all the 20 node sequences are on the same relation LOC and they are combinable. The grouping phase combines them into a single scan query on LOC.address. The final SQL query generated is:

```
select LOC.address
from LOC
```

We next consider the translation of the query
Q_2 = //adinstance//real-estate//location/address. The result of the *NodeId* phase for this query is $S = \{7, 11, 15, \ldots\}$, a set of 7 schema nodes corresponding to the seven categories of *real-estate*. Again, since LOC.address is bijectively mapped, $S_{bij} = S$. In the Prefix-Elimination stage, we compute Conflict_lda for each node in S_{bij}. Some values so computed are: Conflict_lda(7) = 5 and Conflict_lda(11) = 9. The relation sequence for every node in S_{bij} is the same, ($< LOC, RE >$), so all seven node sequences are grouped into a single query. The common condition for all the seven node sequences is LOC.ad-id = RE.ad-id (annotation (viii) in Figure 4). The node sequences do not have any additional conditions. So, we get the following SQL query:

```
select LOC.address
from LOC, RE
```

```
where LOC.ad-id = RE.ad-id
```

Since the schema node $n = 7$ appears in the result of the *NodeId* phase for both queries Q_1 and Q_2, it is interesting to ask why the corresponding SQL has different prefixes eliminated from the join. The answer is that this is because $Conflict_lda(7)$ is different for the two queries.

6 Experiments

In this section we explore the impact that our *constraint-aware* translation algorithm can have on the performance of the SQL queries that result from XML queries. We report results from two datasets: a synthetic ADEX dataset conforming to the standard advertisement schema we have been discussing throughout this paper [23] and a dataset from the XMark Benchmark [18]. We ran the experiments using the 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.

We generated synthetic data for the ADEX dataset. This generated data consists of 100K advertisements and 200 publications, and is approximately 92 MB. We briefly describe the structure of the advertisement data. Each advertisement has a buyer and 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. These main categories are further classified into subcategories. For example, real-estate can be one of seven categories. We used a uniform distribution 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 1. 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 a relational schema for XMark that is relevant for our experiments is given in Table 2. We built indices on all columns that appeared in a query.

We compare the execution times we measured for the queries in Table 3. The queries labeled Ai are on the advertisement dataset, while those labeled Xi are on the XMark dataset. 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 speedups obtained in execution times are given in Table 3.

The relative improvement in performance ranges from 1.15 to 93. In general, by using the constraint information we do no worse than any *constraint-oblivious* strategy so the relative im-

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
LOC	Location information for all ads
JOBCAT	Category info for employment ads
JOBREF	Reference info for employment ads

Table 1: **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

Table 2: **Part of Relational Schema for XMark dataset**

	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 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 3: **Relative performance improvement for queries on ADEX schema**

provement is always greater than or equal to 1.

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

$$(\mathsf{ADEX} \bowtie \mathsf{ADS} \bowtie \mathsf{RE} \bowtie \mathsf{LOC}) \bigcup (\mathsf{ADEX} \bowtie \mathsf{ADS} \bowtie \mathsf{RE} \bowtie \mathsf{LOC})$$

The DB2 optimizer utilizes the specified integrity constraints (key-foreign key and not null) and rewrites A_1^1 as

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

Our translation algorithm will result in the query A_1^2,($\mathsf{RE} \bowtie \mathsf{LOC}$). 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 1.22 and 1.15 in the cold and warm timings respectively.

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

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

Even in this case, the DB2 optimizer identifies that joins with the ADEX and ADS relations are redundant and simplifies the query to

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

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, for this query our translation will produce a single join between RE and LOC relations. So, the relative performance improves by a factor of 2.73 and 3.25 in the cold and warm buffer scenarios respectively.

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 27 with a cold buffer pool and a factor of 31 with a warm buffer pool.

Notice that 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. 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 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 comparatively smaller in these cases as the maximum number of satisfying paths for a wild card is 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 wild cards in it and the constraints on the data allow several of these branches to be merged.

7 Handling Branching Path Expressions

We next illustrate the *constraint-aware* query translation for branching path expression queries. We first present our translation algorithm for queries with a single predicate. Then we discuss how constraints can be used in additional ways for branching path expression queries.

A branching path expression is one of the form "$s_1\ l_1\ \{Pred_1\}\ s_2\ l_2\{Pred_2\} \ldots s_k\ l_k\{Pred_k\}$" where each $Pred_i$ is an optional predicate, each l_i is a tag name and each s_i is either / or // denoting respectively parent-child and ancestor-descendant traversal. A predicate is either of the form Q or of the form Q op value, where Q is a simple path expression.

We present our translation algorithm for branching path queries with a single predicate. So, the query is of the form $Q =$ "$s_1\ l_1\ s_2\ l_2 \ldots s_i\ l_i$ [Q_p op value] $s_{i+1}\ l_{i+1} \ldots s_k\ l_k$". Q_p is of the form

"$s_1^{'}\ l_1^{'}\ s_2^{'}\ l_2^{'} \ldots s_{k_1}^{'}\ l_{k_1}^{'}$"

We use the query Q_1 = //adinstance[id = 'value']//address to illustrate our algorithm. This query returns the address corresponding to the adinstance with a particular id.

7.1 *NodeId* phase

First, in the *NodeId* stage we identify the schema nodes that satisfy the query. Recall that for simple path expressions, the result of this stage is a set of satisfying schema nodes corresponding to the address element. For branching path expressions, each entry in the result of the *NodeId* phase is a triplet of nodes: a branching node, a result node and a condition node. For the query Q_1 the branching nodes are adinstance nodes, the result nodes are address nodes, and the condition nodes are id nodes. Evaluating Q_1 on the schema graph in Figure 4 returns the set $S = \{< 3, 7, 87 >, < 3, 11, 87 >, \ldots\}$, a set with 20 elements. Let Branching(S), Result(S) and Condition(S) denote the set of branching nodes, result nodes and condition nodes in S respectively.

The *NodeId* algorithm for branching path expressions is given in Figure 11. In this algorithm, *NodeId*(Q, n) refers to the algorithm in Figure 7, where CurrentNodes is initialized to $\{n\}$ instead of $\{\text{root}(\mathcal{T})\}$. It corresponds to executing the path expression from node n instead of the entire schema. A single pair of result and condition nodes in the schema may appear multiple times in the result of the *NodeId* algorithm, with different branching nodes. Any pair of these branching nodes will have an ancestor-descendant relationship. Since according to path expression semantics in XQuery the result is a set of nodes, we need not keep track of the number of possible evaluations for each result node. As a result, it suffices to keep the highest branching node, among all satisfying triplets, for a given pair of result and condition nodes.

A branching node sequence is a 4-tuple of schema nodes $T =< n_1, n_2, n_3, n_4 >$, where n_1 is an ancestor of n_2, which is an ancestor of the other two nodes. Let $NS_1 =< n_1, \ldots, n_2 >$, $NS_2 =< n_2, \ldots, n_3 >$ and $NS_3 =< n_2, \ldots, n_4 >$. The relational query Query(T) is obtained by joining the three queries, Query(NS_1), Query(NS_2) and Query(NS_3). The relation $Rel(n_2)$ occurs in the three queries and is included only once in Query(T). $annot(()n_3)$ is the projected column and $Projection(T)$ refers to this column. For example, Query($< 3, 6, 7, 8 >$) is

```
select L1.address
from   ADS, RE, LOC L1
where  ADS.id = RE.ad-id and RE.category = 'house' and RE.ad-id = L1.ad-id
```

The above query returns all addresses of houses available for rent that have a geographic area. Any condition on the value of the area would appear in the selection condition as well. Similarly, the query for the branching node sequence $< 3, 4, 7, 11 >$ is

```
select L1.address
```

```
procedure NodeId-BPE(Q)
begin
  Output = {}.
  Let Q_b = "s_1 l_1 s_2 l_2 ... s_i l_i .
  BranchingNodes = NodeId(Q_b, root(T)).
  Let Q_r = "s_{i+1} l_{i+1} ... s_k l_k".
  For each node n^b ∈ BranchingNodes do
    ResultNodes = NodeId(Q_r, n^b).
    ConditionNodes = NodeId(Q_p, n^b).
    For each n^r ∈ ResultNodes do
      For each n^c ∈ ConditionNodes do
        Add < n^b, n^r, n^c > to Output.
  If < n^{b_1}, n^r, n^c > and < n^{b_2}, n^r, n^c > ∈ Output, and b_2 is a descendant of b_1,
    remove < n^{b_2}, n^r, n^c > from Output.
  Return Output.
```

Figure 11: *NodeId* **algorithm for branching path expressions**

```
from   ADS, RE, LOC L1, LOC L2
where  ADS.id = RE.ad-id and RE.category = 'house' and RE.ad-id = L1.ad-id
       and RE.category = 'apartment' and RE.ad-id = L2.ad-id
```

Note that the above query always has an empty result due to the two contradicting conditions on the RE.category column. We will use this fact in Section 7.6 to improve our *NodeId* algorithm.

Note that, for a branching node sequence $< n_1, n_2, n_3, n_4 >$ if the paths $< n_2, n_3 >$ and $< n_2, n_4 >$ overlap, duplicate copies of the relations on the overlapping region will be present in Query(T), except for $Rel(n_2)$. For example, for the 4-tuple $< 3, 4, 7, 8 >$, the path $< 4, 5, 6 >$ is common to the two sequences $< 4, 7 >$ and $< 4, 8 >$. So, duplicate copies of the LOC relation will be present in Query($< 3, 4, 7, 8 >$).

Let BranchingRelSeq(T) denote the sequence of relations joined in Query(T) in the order RelSeq(NS_1) followed by RelSeq(NS_2) followed by RelSeq(NS_3). For $T =< 3, 4, 7, 12 >$, BranchingRelSeq(T) = <ADS,RE,LOC,LOC>. Notice how the relation RE is not repeated thrice in the relation sequence.

Two branching node sequences T_1 and T_2 are said to be *combinable* if the corresponding relation sequences $BranchingRelSeq(T_1)$ and $BranchingRelSeq(T_2)$ are the same and, $Projection(T_1) = Projection(T_2)$.

7.2 A *constraint-oblivious* translation algorithm

Let us next look at a simple algorithm for generating the SQL query from the set S. In this algorithm, we generate a basic SQL query BasicQuery(n^b, n^r, n^c) for each triple $< n^b, n^r, n^c > \in S$.

The final SQL query is the union of all possible basic queries, $\bigcup_{<n^b,n^r,n^c>\in S}$ BasicQuery(n^b, n^r, n^c).

For a single pair, BasicQuery(n^b, n^r, n^c) is Query($<$ root($\mathcal{T}$)$, n^b, n^r, n^c >$). If the branching predicate in the path expression query has a value condition, then the appropriate value condition on $annot(()n^c)$ is also added.

Note that since the condition is an existential condition, this translation may introduce duplicate results in some cases. We discuss how this is handled in Section 7.5. We now show how we can use the constraint information and remove implied prefixes and group multiple satisfying paths, whenever possible. Just like in Section 5.1, we perform the following optimizations only for bijectively mapped columns.

7.3 Prefix-Elimination

We perform prefix elimination in a manner similar to the prefix elimination stage for simple path expressions. First, we compute the Conflict_lda for each pair of result and condition nodes. Looking at how we generate a SQL query for a branching node sequence, we notice that the branching node is the link between the three queries. So, we cannot eliminate any prefix below the branching node. So for a triple $< n^b, n^r, n^c >\in S$, we compute Conflict_lda(n^r) and Conflict_lda(n^c) as in the algorithm for simple path expressions (Figure 9). Then we pick the highest among these two nodes and n^b.

Next we ensure that for any two node triples s_1 and s_2 in S, one of the following conditions hold:

1. The corresponding branching node sequences are combinable.
2. The result nodes in the two pairs are the same.
3. The Conflict_lda of the corresponding result nodes is below the appropriate *lda*.
4. The Conflict_lda of the corresponding condition nodes is below the appropriate *lda*.

This is similar to the iterative stage in the prefix-elimination stage for simple path expressions. One main difference is the addition of the second condition. If two different node pairs have the same result node, we do not look at the *lda* of the corresponding condition nodes. Due to this, we may get a single result value multiple times. How we handle this is explained in Section 7.5.

The above verification is an iterative process and will terminate in at most $(k * d)$ iterations, where $k = |S|$ and d is the maximum depth (in the XML schema) among all the condition and result nodes in S.

The complete algorithm is given in Figure 12.

procedure Branching-Prefix-Elimination(S)

begin

For each pair $s =< n^r, n^c >\in S$ do

Let $Schema(n^r)$ denote the set of schema nodes mapped to the same column as n^r

Let $Conflict(n^r) \leftarrow Schema(n^r) - Result(S)$

Let LDA_Set(n^r) denote set of $n^r \mid\mid x$ for every node x in Conflict(n^r)

Conflict_lda(n^r) = highest node in LDA_Set(n^r)

Let $Schema(n^c)$ denote the set of schema nodes mapped to the same column as n^c

Let $Conflict(n^c) \leftarrow Schema(n^c) - Condition(S)$

Let LDA_Set(n^c) denote set of $n^c \mid\mid x$ for every node x in Conflict(n^c)

Conflict_lda(n^c) = highest node in LDA_Set(n^c)

Let n^a be the least common ancestor of n^r and n^c.

Conflict_lda(s) = highest node in {Conflict_lda(n^r), Conflict_lda(n^c), n^a}.

notCompleted = true.

While (notCompleted) do

notCompleted = false

If ($\exists s_1 =< n_1^r, n_1^c >, s_2 =< n_2^r, n_2^c >\in S$), such that, $n_1^r \neq n_2^r$ and

BranchingRelSeq($<$Conflict_lda(s_1),n_1^r,$n_1^c >$) and BranchingRelSeq($<$Conflict_lda(s_2),n_2^r,$n_2^c >$) are not combinable

Then

If lda(n_1^r,n_2^r) is a strict ancestor of Conflict_lda(s_1)

Then

Conflict_lda(s_1) = lda(n_1^r, n_2^r)

notCompleted = true

If lda(n_1^c,n_2^c) is a strict ancestor of Conflict_lda(s_1)

Then

Conflict_lda(s_1) = lda(n_1^c, n_2^c)

notCompleted = true

end

Figure 12: **Prefix-Elimination for branching path expressions**

7.4 Grouping Multiple Paths

We next group the prefix-eliminated node sequences together by modifying the algorithm in Section 5.1.2 to handle branching node sequences and generate the final SQL query. The modifications are straightforward and we omit the details. The final SQL query for the above example query Q_1 is

```
select LOC.address
from   ADS, RE, LOC
where  RE.ad-id = LOC.ad-id and ADS.id = RE.ad-id and ADS.id = ‘value’
union all
select LOC.address
from   ADS, EMP, LOC
where  EMP.ad-id = LOC.ad-id and ADS.id = EMP.ad-id and ADS.id = ‘value’
union all
select LOC.address
from   ADS, TRANS, LOC
where  TRANS.ad-id = LOC.ad-id and ADS.id = TRANS.ad-id and ADS.id = ‘value’
```

7.5 Handling duplicates in the query result

Duplicate results may be produced in two different ways, either by the SQL fragments corresponding to a single node pair $s \in S$ or by two different node pairs $s_1, s_2 \in S$ having the same result nodes.

Consider a single node pair $s_1 =< n_1^r, n_1^c >$. Let n_1^a denote the least common ancestor of n_1^r and n_1^c. The SQL query corresponding to just this pair, without any optimizations, is Query$<$ root$(\mathcal{T}), n_1^r, n_1^c >$. Even this query may produce duplicate results, i.e., a single result tuple may occur in the result multiple times. This occurs due to the fact that a single element corresponding to the node n_1^a may have multiple (say k) descendant elements corresponding to the node n_1^c. So, for a single result tuple will appear k times in the result set (if all the k condition elements satisfy the predicate). For example, for the query $//adinstance[//address < \text{‘}value']/id$, a single id may be projected multiple times as a single adinstance may have multiple address descendants satisfying the value predicate.

In a similar fashion, duplicates may also be produced from two different node pairs in S. So, we have to ensure that each result element occurs exactly once in the result. It can be shown that the above algorithm will generate the correct set of results. To ensure a multiplicity of one for each result element, we use keyQuery(s) in place of Query(s). If columns belonging to multiple relations appear in the projection column of the SQL query, we add an identifier for the relation name as well. This is required to provide a key across relations. We finally add a distinct clause to the entire result. The final SQL query with these modifications is given below. Here, we assume that

(ad-id, address) is a key for the LOC relation.

```
with temp(ad-id,address) as (
   select LOC.ad-id, LOC.address
   from   ADS, RE, LOC
   where  RE.ad-id = LOC.ad-id and ADS.id = RE.ad-id and ADS.id = ‘value’
   union all
   select LOC.ad-id, LOC.address
   from   ADS, EMP, LOC
   where  EMP.ad-id = LOC.ad-id and ADS.id = EMP.ad-id and ADS.id = ‘value’
   union all
   select LOC.ad-id, LOC.address
   from   ADS, TRANS, LOC
   where  TRANS.ad-id = LOC.ad-id and ADS.id = TRANS.ad-id and ADS.id = ‘value’
)
select distinct(ad-id, address)
from temp
```

In the above translation, we can use the constraint information to identify situations when duplicates will never be produced. In such scenarios, we do not have to explicitly remove duplicate results by projecting the key column(s) and by adding a distinct clause. This optimization is discussed in Section 7.7.

An alternative strategy is to translate the existential condition as a nested subquery. In this case, we have to redefine Query(t) to correspond to a nested subquery. For example, Query($< 3, 7, 12 >$) is

```
select L1.address
from   ADS, RE, LOC L1
where  ADS.id = RE.ad-id and RE.category = ‘house’ and RE.ad-id = L1.ad-id
       and not empty
           ( select *
             from LOC L2
             where RE.category = ‘apartment’ and RE.ad-id = L2.ad-id
           )
```

The techniques presented above can be modified appropriately to generate SQL with nested subqueries.

7.6 Optimizations in the *NodeId* Stage for Branching Path Expressions

In the case of branching path expressions, we can utilize constraint information during the *NodeId* stage also. We illustrate this with an example.

Consider the query XQ presented in Section 2.

```
//adinstance//location[geo-area='campus']/address
```

The //adinstance//location tag in the query matches a set of 20 schema nodes, $S_1 = \{6, 10, 14, \ldots\}$. Let us consider further processing for one branching node, say 6. The corresponding sets of result nodes and condition nodes are $\{7\}$ and $\{8\}$ respectively. So, the result of the *NodeId* stage is a set $S = \{< 7, 8 >, < 11, 12 >, \ldots\}$ of 20 entries.

Now consider the query XQ_1 given below.

```
//adinstance[//geo-area='campus']//address
```

In this case, the //adinstance tag in the query matches a single node $S_1 = \{3\}$. The set of result nodes for this branching node is ResultNodes $= \{7, 11, 15, \ldots\}$, a set of 20 schema nodes. Similarly, the set of condition nodes is ConditionNodes $= \{8, 12, 16, \ldots\}$, a set of 20 schema nodes. So, the output of the *NodeId* phase is a set $S = \{< 7, 8 >, < 7, 12 >, \ldots, < 11, 8 >, < 11, 12 >, \ldots\}$ of 400 entries.

If we look at the constraints on the underlying data, it can be seen for each result node, only one condition node will produce results. The other 19 condition nodes will not produce any results. For example, the node pair $(7, 8)$ will generate result tuples, while the pair $(7, 12)$ will always generate an empty result. This intuitively corresponds to the fact that each ad is exactly one of the 20 categories (7 Real-estate, 4 Employment and 9 Transportation). So, we need to look at corresponding address and area elements only.

To utilize the constraint information during *NodeId*, we compute another kind of summary information during the pre-computation stage. We call this the *lowest common branching ancestor* information (*lcba*) for every pair of schema nodes. Intuitively, for two schema nodes u and v, *lcba*(u,v) is a schema node w such that a single XML element n_w may have two descendant elements n_u and n_v corresponding to u and v respectively. The *lcba* computation is very similar to *lda* computation and we omit the details.

With the *lcba* information, we can eliminate node pairs like $(7, 12)$ from the result of the *NodeId* phase. For example, $lcba(7, 12) = 2$. So, we can identify that $(7, 12)$ will have no results and remove it from the result set S.

7.7 Utilizing Constraint Information in Duplicate Handling

In this section, we briefly describe how constraint information can be used to identify situations when duplicates will not be produced in the SQL query. In this case, we need not apply the duplicate elimination methods presented in Section 7.5.

In Section 7.5, we saw that a single node pair $s =< n^r, n^c >$ may produce duplicate results. For each node pair, we check whether the corresponding conjunctive query will produce duplicate results (i.e., we check if a single result can have multiple evaluations). If not, then that node pair is guaranteed to have no duplicates.

Similarly, duplicate results may be produced by two node pairs $s_1 =< n^r, n^c_1 >$ and $s_2 =< n^r, n^c_2 >$. Testing this condition corresponds to testing if the results of the corresponding conjunctive queries have an empty intersection. If so, then the two node pairs will not produce any duplicate entries.

We presented solutions for the above problems on conjunctive queries in Section 4.3. Incorporating the above information in our algorithm is straightforward and we omit the details. One main change needs to be made in the iterative part of the prefix-elimination stage. The second condition in Section 7.3 has to be removed and result nodes and condition nodes treated in a similar fashion.

7.8 On the impact of our Optimizations

The experimental results in the previous section showed how our *constraint-aware* translation can give significant performance improvements, when the fixed schema mapping allows prefix-elimination and grouping optimizations to be performed. In this section, we first describe why we feel that such opportunities are likely to be useful in a wide variety of fixed schema scenarios. We then explain briefly why we introduced the notion of bijective column mappings and present some extensions to handle non-bijective mappings.

From the results in [7], a study about how real DTDs look like, we see that about $\frac{2}{3}$rd's of the nonrecursive DTDs (considered in that paper) have simple paths of length ≥ 5, and the longest simple path is of length 20. So, we see that prefix-elimination can be quite useful in many situations.

The other optimization involves grouping multiple paths and is applicable when the XML path expression query has multiple satisfying paths in the schema graph.

This can occur in the following two ways: (i) The query has one of the following operators in it: descendant operator (//), wildcards (*) or the disjunction operator (|). (ii) In the context where XML data is stored in RDBMS, certain relational decompositions handle disjunctions in a manner such that, even a simple path expression like a/b may produce multiple satisfying paths. For example, in [5], the authors propose a technique called *Union Factorization/Distribution.* Using this technique, the content model a,(b|c) can be modified into the content model (a,b)|(a,c). Then the query *a* will have two satisfying schema nodes [1].

[1]Note that our description of the XML schema graph in Section 3.1 can be augmented to support disjunctions in a straightforward manner. Appropriate changes have to be made to the *NodeId* stage in Figure 7.

Consider the case when the query has a descendant operator in it (say //author). The query may have multiple satisfying paths due to two reasons: (i) presence of multiple elements in the schema with name author and (ii) presence of an element called author in the schema that has multiple paths from the root of the schema. Let us consider the latter scenario. This may occur if an author schema node has some ancestor with multiple parents. The location element in Figure 1 is an example. These nodes are referred to as *hubs* in [7] and the number of parents a node has is called the fan-in of the node. The authors analyzed 60 DTDs and found that *hubs* existed in most of the DTDs. In some cases, DTDs contained multiple hubs. Over 10 of the DTDs had nodes with fan-in ≥ 20 and the maximum fan-in was 120. Notice how if a node n is a hub, then every node in the subtree of n has a number of incoming paths from the root.

Now that we have argued that hubs occur frequently in XML schemas, let us now look at how the hubs in the XML schema are mapped onto relational columns. Let us consider the two cases (i) when XML data is stored in an RDBMS and (ii) existing relational data is exported as XML. In the former case, some algorithms in literature, like the Shared algorithm in [21], create a single relation for the hub and store all occurences of the hub element in that relation. If this technique is followed, then grouping multiple satisfying paths becomes viable and important. The Hybrid algorithm [21] also creates relational mappings where grouping becomes important. In the latter case, existing relational data is exported as XML. For this scenario, if we look at the XML schemas that are available online (like the XML.org DTD repository [26]), we notice that many of these schemas make heavy use of horizontal partitioning. For example, the ADEX DTD (see Figure 1) has multiple paths to the location created by partitioning of the data based on the category of the advertisement. A similar feature is also present in the XMark benchmark [18], where we see a paritioning of items based on the continent where they are available. We believe that this tendency to move (relational) data into (XML) metadata and build a hierarchy from flat relational data will create a lot of opportunities for the kind of optimizations discussed in this paper.

8 Bijective Mappings and Beyond

In this paper, we introduced the notion of bijective column mappings (in Section 4) and applied our optimization techniques (like prefix elimination and grouping multiple paths) to the queries that returned elements corresponding to bijective columns. We next briefly describe the reason behind introducing the notion of bijective mappings.

Earlier in the paper, we pointed out that utilizing constraints in the SQL optimization phase can be expensive[13, 28] (exponential). We moved the expensive computation to a pre-computation phase and modified the XML to SQL query translation phase to use some summary information.

To ensure that the running time of the query translation phase increases only by a polynomial time, we identify bijective column mappings as a subset of the entire fixed-schema scenario where we can exploit the constraint information efficiently. We made this choice mainly based on two observations:

- When we consider the scenario where XML data is stored in an RDBMS, all the techniques proposed in current literature result in an XML to Relational mapping that is bijective. So, we believe that bijective column mappings is a useful class of XML to Relational mappings.
- A lot of work has been done in relational database theory on the problem of query containment (with or without constraints on the relational schema). Most of this work has been concentrated on containment under set semantics. In Section 4.3, we use this body of work for computing the necessary summary information. Since current results on relational conjunctive query containment in the presence of constraints are mainly under set semantics, we introduce the notion of bijective column mappings so as to use the existing literature under multiset semantics as well. In other words, we convert the problem of verifying whether $Q_1 \subseteq Q_2$ under multiset semantics into performing the containment check under set semantics and also verifying that Q_1 does not have any duplicates.

For non-bijective column mappings, we have extended our algorithm to perform efficient XML to SQL query translation in many situations. We outline the main ideas with an example.

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, suppose that real-estate had eight categories instead of seven and that the eighth category were not present in the XML view. Consider the XML query XQ from Section 2.

```
//adinstance//location[geo-area='campus']/address
```

The best relational translation for this query will be

```
select LOC.address
from   RE, LOC
where  RE.ad-id = LOC.ad-id and LOC.area = 'campus'
       and 1 <= RE.category and RE.category <= 7
union all
select LOC.address
from   EMP, LOC
where  EMP.ad-id = LOC.ad-id and LOC.area = 'campus'
union all
```

```
select LOC.address
from   TRANS, LOC
where  TRANS.ad-id = LOC.ad-id and LOC.area = 'campus'
```

Notice how we were able to group together the seven paths corresponding to real-estate ads. This was possible due to the fact that LOC.address satisfied the At-most-Once condition. As a result, the fullKeyQueries corresponding to any two column-compatible schema nodes mapped to LOC.address will not have any common results. So, we can translate the unions to disjunctions. In other words, we can perform the Grouping phase without any change.

On the other hand, for real-estate ads, we need the selection condition on RE.category to filter out advertisements belonging to category 8. Due to this fact, the join between EMP and LOC relations is required to identify employment ads and the join between TRANS and LOC relations is required to identify transportation ads. To account for these facts, we have to augment the prefix-elimination stage. We do this as follows: $S = nodes(LOC.address) = \{7, 11, 15, \ldots\}$, a set of 20 schema nodes. 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(7) = 4 and lra(11) = 4. This ensures that the selection condition on RE.category is present in the query. On the other hand, lra(34) = 32. So, though the join between the EMP and LOC relations is required, the selection condition on EMP.category is not needed. The lra computation is another summary information that we precompute for schema nodes corresponding to relational columns that violate the At-least-once condition.

9 Related Work

The Xperanto [8, 20, 19] and SilkRoute [16, 9] 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 underlying relational constraints during query translation. In this paper, we use constraints on the underlying relational database during query translation to eliminate redundant predicates and also group multiple satisfying paths for wild card XML queries, whenever possible.

The Agora[14] project deals with translating XML queries into SQL in the context of integrating heterogeneous data sources using a global XML schema. In their 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 the relational schema and the class of XML queries allowed. On the other hand, we handle the case of fixed-schema mappings and do not place any restrictions on the

relational schema. Moreover, in [14], the XML to SQL query translation is done regardless of the data mapping and underlying constraints, while we use the mapping information along with the constraints placed on the underlying relational data to obtain an efficient SQL query.

In [11], the authors present an algorithm for translating XSLT programs into efficient SQL queries. The main focus of the paper is on bridging the gap between XSLT's functional, recursive paradigm, and SQL's declarative paradigm. They also identify a new class of optimizations that need to be done either by the translator, or by the relational engine, in order to optimize the kind of SQL queries that result from such a translation. For XSLT programs that resemble simple path expression queries, they allude to a possible optimization to remove redundant joins called QTree reductions. This optimization uses some annotations on the XML schema that describe when some paths are redundant and how they can be removed. They also describe how the XML schema information can be used in two other scenarios. So, their algorithm can be viewed as *partially constraint-aware*. On the other hand, we present a query translation algorithm using additional constraints on the underlying data as well.

There has been some work on optimizing queries in a semi-structured framework[6, 10, 17] using graph schemas. These papers dealt with the *NodeId* stage of query translation, while the focus of our work is on using constraints on the relational data during the *RelationalQueryGeneration* stage.

Recall that we defined three problems based on conjunctive queries in Section 4.3 and used them in our precomputation phase. There has been a lot of work on the use of constraints in query optimization of relational queries [1, 13, 22, 27, 28] that is related to these problems. 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 [27], 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?) was presented and in [28] the results were extended when referential constraints were also allowed. Polynomial equivalence to other problems like the query containment problem were also proved. The results in [28] provide an alternative way of solving the three problems discussed in 4.3.

More recently, the chase and backchase algorithm (c&b) was introduced in [1] motivated by logical redundancy and physical independence in mediator-like components. Their 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. They used this work in [3, 2] to study the problem of query containment of XPath expressions both in the absence and presence of constraints. In this paper, we use their algorithm for conjunctive query containment under set semantics and their chase algorithm during the pre-computation

phase. Our work differs from theirs in that we are looking at XML to SQL query translation in a fixed schema scenario, where containment and minimization of unions of conjunctive queries under multiset semantics needs to be solved. We do this through the notion of bijective column mappings. Moreover, in our approach we utilize the constraints in a pre-computation phase and use the summary information (and not the actual constraints) during query optimization time.

10 Conclusions and Future Work

In this paper we considered the problem of translating XML queries into SQL when both the XML and the SQL schemas are fixed. We showed that *constraint-oblivious* translation, which ignores constraints on the underlying relational data, often produces inefficient queries. 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 existing more naive translation algorithms. Our *constraint-aware* algorithm uses the fact that the schemas and constraints are known ahead of query translation time to move a substantial amount of the work required for constraint-aware translation to an offline precomputation phase.

In this paper we have certainly not exhausted all opportunities for constraint-aware query translation. By considering more complex classes of constraints and/or more complex fragments of XML to relational schema mappings, it may be possible to specify more elaborate constraint-aware optimizations, albeit at the expense of increasing the complexity of both the runtime and precomputation phases of the translation algorithm. Exploring these opportunities, and resolving the tradeoff between translation complexity and runtime SQL query performance is a promising area for future research.

Finally, as we have noted in the introduction, our work for the query translation problem for the fixed-schema case also applies to the situation in which one of the schemas can be varied. For example, consider a case in which we want to use an RDBMS to store and query existing XML data. In such a scenario, the XML schema is fixed, but the relational schema is not. Here there are two main subproblems: (i) Given an XML schema, choose a good relational schema and a mapping between the two, and (ii) Given the XML schema and this mapping to the new relational schema, translate XML queries to SQL over this relational schema.

Once subproblem (i) is solved, problem (ii) becomes an instance of the fixed-schema translation problem addressed in this paper. Moreover, most (if not all) of the techniques proposed in current literature for choosing a good relational schema result in an XML to Relational mapping that is bijective. So, the techniques presented in this paper are applicable directly. Interestingly, though, we now have a "chicken and egg" problem, because the opportunities for constraint-aware query

translation will depend upon the details of the chosen relational schema; conversely, what is deemed a "good" relational schema should take into account the query translations that will be generated by a constraint-aware translation algorithm. Handling this interaction between constraint-aware query translation algorithms and the problem of choosing a good schema is a complex but interesting open problem.

References

[1] Alin Deutsch and Lucian Popa and Val Tannen. Physical Data Independence, Constraints and Optimization with Universal Plans. In *International Conference on Very Large Databases (VLDB)*, 1999.

[2] Alin Deutsch and Val Tannen. Containment for Classes of XPath Expressions Under Integrity Constraints. Technical Report MS-CIS-01-21, Department of Computer and Information Science, University of Pennsylvania, 2001.

[3] Alin Deutsch and Val Tannen. Optimization Properties for Classes of Conjunctive Regular Path Queries. In *DBPL'01*, 2001.

[4] Catriel Beeri and Moshe Y. Vardi. A proof procedure for data dependencies. *JACM*, 31(4):718–741, 1984.

[5] Philip Bohannon, Juliana Freire, Prasan Roy, and Jerome Simeon. From XML schema to relations: A cost-based approach to XML storage. In *ICDE*, 2002.

[6] Peter Buneman, Susan B. Davidson, Mary F. Fernandez, and Dan Suciu. Adding structure to unstructured data. In Foto N. Afrati and Phokion Kolaitis, editors, *Database Theory—ICDT'97, 6th International Conference*, volume 1186, pages 336–350, Delphi, Greece, 8–10 1997. Springer.

[7] Byron Choi. What Are Real DTDs Like. Technical Report MS-CIS-02-05, Department of Computer and Information Science, University of Pennsylvania, 2002.

[8] Michael J. Carey, Daniela Florescu, Zachary G. Ives, Ying Lu, Jayavel Shanmugasundaram, Eugene J. Shekita, and Subbu N. Subramanian. XPERANTO: Publishing object-relational data as XML. In *WebDB (Informal Proceedings)*, pages 105–110, 2000.

[9] Mary F. Fernandez, Atsuyuki Morishima, and Dan Suciu. Efficient evaluation of XML middleware queries. In *SIGMOD Conference*, 2001.

[10] Mary F. Fernandez and Dan Suciu. Optimizing regular path expressions using graph schemas. In *ICDE*, pages 14–23, 1998.

[11] Sushant Jain, Ratul Mahajan, and Dan Suciu. Translating xslt programs to efficient sql queries. In *Proceedings of the eleventh international conference on World Wide Web*, pages 616–626. ACM Press, 2002.

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

[13] David S. Johnson and Anthony C. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. In *Proceedings of the ACM Symposium on Principles of Database Systems, March 29-31, 1982, Los Angeles, California*, pages 164–169. ACM, 1982.

[14] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering xml queries over heterogeneous data sources. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 241–250. Morgan Kaufmann, 2001.

[15] Mary F. Fernandez. Personal Communication, May 2002.

[16] Mary Fernandez and Wang-Chiew Tan and Dan Suciu. SilkRoute: Trading between Relations and XML. In *WWW9*, May 2000.

[17] Jason McHugh and Jennifer Widom. Compile-time path expansion in lore. In *Workshop on Query Processing for SemiStructured Data and Non-Standard Data Formats*, January 1999.

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

[19] Jayavel Shanmugasundaram, Jerry Kiernan, Eugene J. Shekita, Catalina Fan, and John Funderburk. Querying xml views of relational data. In *Proceedings of 27th International Conference on Very Large Data Bases*, pages 261–170. Morgan Kaufmann, 2001.

[20] Jayavel Shanmugasundaram, Eugene J. Shekita, Rimon Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently publishing relational data as xml documents. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 65–76. Morgan Kaufmann, 2000.

[21] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and

opportunities. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 302–314. Morgan Kaufmann, 1999.

[22] Sreekumar T. Shenoy and Z. Meral Özsoyoglu. A system for semantic query optimization. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 181–195. ACM Press, 1987.

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

[24] Sql server. `http://msdn.microsoft.com/library`.

[25] Xml-dbms: Middleware for transferring data between xml documents and relational databases. `http://www.rpbourret.com/xmldbms`.

[26] Xml.org registry. `http://www.xml.org/xml/registry.jsp`.

[27] Xubo Zhang and Z. Meral Özsoyoglu. On efficient reasoning with implication constraints. In *Deductive and Object-Oriented Databases, Third International Conference, DOOD'93, Phoenix, Arizona, USA, December 6-8, 1993, Proceedings*, volume 760 of *Lecture Notes in Computer Science*, pages 236–252. Springer, 1993.

[28] Xubo Zhang and Z. Meral Özsoyoglu. Implication and referential constraints: A new formal reasoning. *TKDE*, 9(6):894–910, 1997.

A Computing Summary Information

Recall that we need procedures for the problems QC, EQI and DUPL defined in Section 4.3. In order to solve these problems, we use the results from [2]. We first review their results. They present an algorithm for conjunctive query containment (under set semantics) in the presence of a class of constraints known as **D**isjunctive **E**mbedded **D**ependencies (DEDs).

DEFINITION **4** *The general form of Disjunctive Embedded Dependencies (DEDs) is the following*

$$\forall x_1 \ldots x_n[\phi(x_1,\ldots,x_n) \rightarrow \\ \vee_{i=1}^{l} \exists z_{i,1},\ldots,z_{i,k_i}\psi_i(x_1,\ldots,x_n,z_{i,1},\ldots,z_{i,k_i})]$$

where ϕ, ψ_i are conjunctions of relational atoms of the form $R(w_1,\ldots,w_l)$ and equality atoms of the form $w = w'$, where $w_1,\ldots,w_l,w,w'$ are variables.

The class of DEDs covers a rich set of constraints including functional dependencies, inclusion dependencies, multi-relational inclusion dependencies and uniqueness constraints, and a restricted class of domain constraints. For example, the key condition on the ADS relation in our example in Section 2, can be expressed as

$$\forall x_1, x_2, x_3, x_4, x_5 [ADS(x_1, x_2, x_3) \wedge ADS(x_1, x_4, x_5) \rightarrow (x_2 = x_4) \wedge (x_3 = x_5)]$$

Similarly, the multi-relation condition on ADS can be expressed as the following constraint

$$\forall x_1, x_2, x_3 [ADS[x_1, x_2, x_3) \rightarrow (\exists z_1\ RE(x_1, z_1)) \vee (\exists z_2\ EMP(x_1, z_2)) \vee (\exists z_3\ TRANS(x_1, z_3))]$$

The paper presents the following results:

1. They present a generalization of the classical chase algorithm for embedded dependencies [4] to the class of DEDs. While the result of the classical chase procedure is a chase sequence, in the presence of DEDs, the result is a chase *tree*. More details can be found in [2].

2. Given conjunctive queries Q_1, Q_2, and the set D of DEDs, assume that the chase of Q_1 with D terminates. Then we have (1) Q_1 is equivalent to the union of the leaves of the chase tree, and (2) Q_1 is contained in Q_2 under D if and only if there is a containment mapping from Q_2 into every leaf $L \in chase_D(Q_1)$.

We now briefly explain our procedures for the QC, EQI and DUPL problems.

QC: We need to verify whether $Q_1 \subseteq Q_2$, where Q_1 is a conjunctive query and $Q_2 = \bigcup Q_2^i$. We then have the following result: assume that the chase of Q_1 with D terminates. Then we have that Q_1 is contained in Q_2 under D if for every leaf $L \in chase_D(Q_1)$, there is a containment mapping from some Q_2^i into L.

EQI: In order to check if the intersection of two conjunctive queries Q_1 and Q_2 is empty, we need to check if the query $Q = Q_1 \wedge Q_2$ has an empty result, i.e., $Q_1 \wedge Q_2 \subseteq \phi$. Here the distinguished (i.e., the projected) variables of Q_1 and Q_2 are equated and become the distinguished variables of Q.

DUPL: In order to check whether a conjunctive query Q produces duplicate results, we take two copies of Q, Q_1 and Q_2 and construct the query $Q^{'} = Q_1 \wedge Q_2$. As above, the distinguished (i.e., the projected) variables of Q_1 and Q_2 are equated and become the distinguished variables of $Q^{'}$. Then we compute $chase(Q^{'})$ and check if for every conjunct in $Q^{'}$ that was originally from Q_1, the values in the key columns are the same as the values in the corresponding conjunct from Q_2. If so, multiple evaluations for a single tuple are not possible.