Unraveling the Duplicate-Elimination Problem in XML-to-SQL Query Translation

Rajasekar Krishnamurthy
University of Wisconsin
sekar@cs.wisc.edu

Raghav Kaushik
Microsoft Corporation
skaushi@microsoft.com

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

ABSTRACT

We consider the scenario where existing relational data is exported as XML. In this context, we look at the problem of translating XML queries into SQL. XML query languages have two different notions of duplicates: node-identity based and value-based. Path expression queries have an implicit node-identity based duplicate elimination built into them. On the other hand, SQL only supports value-based duplicate elimination. In this paper, using a simple path expression query we illustrate the problems that arise when we attempt to simulate the node-identity based duplicate elimination using value-based duplicate elimination in the SQL queries. We show how a general solution for this problem covering the class of views considered in published literature requires a fairly complex mechanism.

1. THE DUPLICATE ELIMINATION PROBLEM

Using a simple example scenario, we first explain why we need duplicate elimination in XML-to-SQL query translation.

Consider the following relational schema for a collection of books.

- **Book** (*id*, title, price, ...)
- **Author** (*name*, bookid, ...)
- **TopSection** (*id*, bookid, title, ...)
- **NestedSection** (*id*, topsectionid, title, ...)

The Book relation has basic information about books and the Author relation has information about authors of each book. Each book has sections and subsections, and the corresponding information is in the TopSection and NestedSection relations respectively.

Consider the XML view $\mathcal{T}_1$ defined over this relational schema shown in Figure 1. We represent the XML view using simple annotations on the nodes and edges of the XML schema. For example, each book tuple creates a new book element. The title of the book is represented as a title subelement. Each section is represented as a subelement and this is captured by the join condition on the edge <2,5>. The rest of the view definition can be understood in a similar fashion.

Seventh International Workshop on the Web and Databases (WebDB 2004), June 17-18, 2004, Paris, France

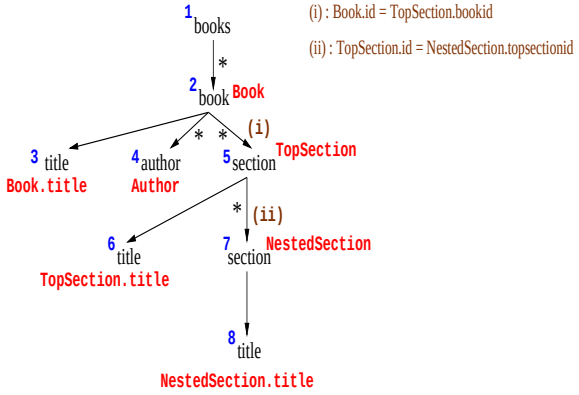

Figure 1: XML view $\mathcal{T}_1$

Suppose we want to retrieve the titles of all sections. One possible XML query is Q = //section//title. XML-to-SQL query translation algorithms proposed in literature, such as [6, 15], work as follows. Logically, the first step is to identify schema nodes that match each step of the query. In this example, there are three matching evaluations for Q, namely S = {<5,6>, <5,8>, <7,8>}. The second step is to generate an SQL query for each matching evaluation in S. The final query is the union of queries generated for all matching evaluations. For Q, the SQL query SQ_1 obtained in this fashion is given below.

```
select TS.title
from   Book B, TopSection TS
where  B.id = TS.bookid
union all
select NS.title
from   Book B, TopSection TS, NestedSection NS
where  B.id = TS.bookid and TS.id = NS.topsectionid
union all
select NS.title
from   Book B, TopSection TS, NestedSection NS
where  B.id = TS.bookid and TS.id = NS.topsectionid
```

In the above query, we see that there are three entries in S, and, as a result, SQ_1 is the union of three queries. On the other hand, looking at the view definition, we notice that there are only two paths that match the query ending at the two schema nodes 6 and 8. The path <1,2,5,7,8> appears twice in S, once as <5,8> and again as <7,8>. This occurs because the *section* step in the query matches both the section elements in the schema. The following *title* step matches the title element (node 8) for each of these evaluations, due to the // axis in the query. As a result, the second and third (sub)queries in SQ_1 are identical and generate duplicate results.

According to XPath semantics, the result of a query should not have any duplicates. Here, duplicate-elimination is defined in terms of node-identity. As a result, we need to add a *distinct* clause to SQ_1 to achieve the same effect. We refer to this as the *Duplicate-Elimination* problem. The fact that we need to simulate node-identity based duplicate elimination using the value-based distinct clause in SQL creates several problems and providing a complete solution to this problem is the focus of this paper. Notice that this extra duplicate-elimination step is required to make sure that existing algorithms work correctly for this particular example.

Let us start with the simplest approach to eliminate duplicates from SQ_1. By adding an outer distinct(title) clause, we can eliminate duplicate titles. The corresponding SQL query SQ_1^1 is given below.

```
with Temp(title) as (
  select TS.title
  from   Book B, TopSection TS
  where  B.id = TS.bookid
  union all
  select NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
  union all
  select NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
)
select distinct(title)
from   Temp
```

Notice that the above query does value-based duplication and may eliminate more values than required. For example, duplicates in the XML view that arise in the following ways must be retained in the query result.

1. Two top-level sections have the same title
2. Two nested sections have the same title
3. A top-level section and a nested section have the same title

Since SQ_1^1 applies a distinct clause on title, it eliminates duplicates that arise in the above three contexts. As a result, SQ_1^1 is not a correct query.

Let us next see if using the key column(s) in the relational schema help us solve this problem. Recall that the *id* column is the key for both the TopSection and NestedSection relations. So, by projecting this key column and applying the following distinct clause: "distinct(id,title)", we get the following query SQ_1^2.

```
with Temp(id,title) as (
  select TS.id, TS.title
  from   Book B, TopSection TS
  where  B.id = TS.bookid
  union all
  select NS.id, NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
  union all
  select NS.id, NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
)
select distinct(id, title)
from   Temp
```

While the above query retains the duplicate values corresponding to 1 and 2 above, it does not retain duplicates when a top-level section and a nested section have the same title and the same id as well. This can occur because keys in relational databases are unique only in the context of the corresponding relation.

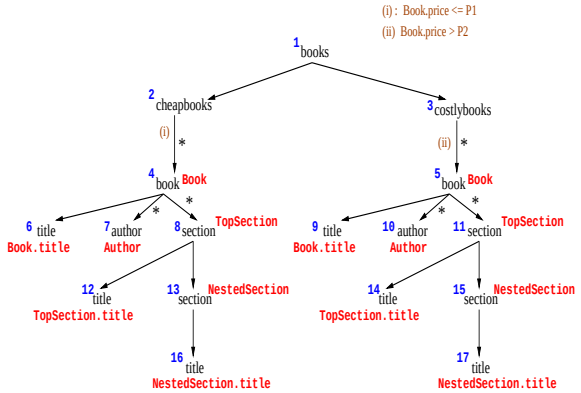

Figure 2: XML view $\mathcal{T}_2$

In order to address this issue, we need to create a key across the two relations. A straightforward way to do this is to combine the name of the relation (or some other identifier) along with the key column(s). This results in the following query SQ_1^3.

```
with Temp(relname, id,title) as (
  select ''TS'', TS.id, TS.title
  from   Book B, TopSection TS
  where  B.id = TS.bookid
  union all
  select ''NS'', NS.id, NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
  union all
  select ''NS'', NS.id, NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
)
select distinct(relname, id, title)
from   Temp
```

Notice how the above approach creates a global key across the entire relational schema. This duplicate-elimination technique is correct for this query, and in fact, it is correct for any query over a class of views that we call non-redundant (see Table 1 in Section 4). Unfortunately this solution is not general enough and is incorrect when parts of the relational data may appear multiple times in the XML view. In the rest of the paper, we identify the techniques required for different class of views ending with a generic solution that is applicable over all views.

2. REDUNDANT XML VIEWS

In this section, we look at some of the simplifying assumptions we implicitly made while generating the correct duplicate-elimination clause for the query Q over view $\mathcal{T}_1$. First, using a slightly modified XML view over the same underlying relational schema we show how the correct solution gets more complex than before. Then, we look at the scenario when the join conditions are not key-foreign key joins.

2.1 A Hierarchical XML view example

The XML view $\mathcal{T}_2$, in Figure 2, has created a simple hierarchy, partitioning the books into cheap and costly books by the relationship of their prices to two constants P_1 and P_2.

Let us look at how Q_1 will be translated in this case by some of the existing algorithms [6, 15]. The equivalent SQL query is the union of six queries, three for cheapbooks and

three for costlybooks. Again, the nested section titles occur twice and need to be eliminated through a duplicate elimination operation.

At the end of the previous section, we saw how a distinct clause over the three fields: relname, id and title may suffice (see query SQ_1^3 in Section 1). Applying the same idea here, we obtain the following query, SQ_2^1.

```
with Temp(relname, id,title) as (
  select ''TS'', TS.id, TS.title
  from   Book B, TopSection TS
  where  B.id = TS.bookid and B.price <= P1
  union all
  select ''NS'', NS.id, NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
         and B.price <= P1
  union all
  select ''NS'', NS.id, NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
         and B.price <= P1
  union all
  select ''TS'', TS.id, TS.title
  from   Book B, TopSection TS
  where  B.id = TS.bookid and B.price > P2
  union all
  select ''NS'', NS.id, NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
         and B.price > P2
  union all
  select ''NS'', NS.id, NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
         and B.price > P2
)
select distinct(relname, id, title)
from   Temp
```

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

For the two cases, $P_1 = P_2$ and $P_1 < P_2$, each book appears at most once in the XML view. As a result, the SQL query SQ_2^1 eliminates duplicates correctly. But when $P_1 > P_2$, books in the price range $\{P_2 \ldots P_1\}$ appear twice in the XML view. So, the corresponding section titles must appear twice in the query result. But, since SQ_2^1 applies a distinct clause using the triplet "relname, id, title", it will retain only one copy of each section title and the query result is incorrect.

The main problem in this example scenario is that some parts of the relational data appear multiple times in the XML view. For the XML view $\mathcal{T}_2$, notice that multiple occurrences of the same section title are associated with different schema nodes. One way to obtain the correct result in this case is to keep track of the schema node corresponding to each result tuple. The distinct clause in this case will include the schema node instead of the relation name. The corresponding SQL query SQ_2^2 is shown below.

```
with Temp(nodeid, id,title) as (
  select 12, TS.id, TS.title
  from   Book B, TopSection TS
  where  B.id = TS.bookid and B.price <= P1
  union all
  select 16, NS.id, NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
         and B.price <= P1
  union all
  select 16, NS.id, NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
         and B.price <= P1
  union all
  select 14, TS.id, TS.title
  from   Book B, TopSection TS
  where  B.id = TS.bookid and B.price > P2
  union all
  select 17, NS.id, NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
         and B.price > P2
  union all
  select 17, NS.id, NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
         and B.price > P2
)
select distinct(nodeid, id, title)
from   Temp
```

The above query will be a correct translation for all the three cases: $P_1 = P_2$, $P_1 < P_2$ and $P_1 > P_2$. The above solution is correct for any query on a class of views that we call well-formed (see Table 1 in Section 4).

Notice how simple syntactic restrictions on the view definition language will not allow us to differentiate between views $\mathcal{T}_1$ and $\mathcal{T}_2$. As a result, unless we know that $\mathcal{T}_1$ is a non-redundant view, when we translate Q over $\mathcal{T}_1$, we have to use the schema node number to perform duplicate elimination. In Section 4, we present a way for identifying when XML views are non-redundant.

2.2 Beyond key-foreign key joins

For the two example views, $\mathcal{T}_1$ and $\mathcal{T}_2$, we saw how a correct translation for query Q_1 results in queries SQ_1^3 and SQ_2^2 respectively. The join conditions present in both these view definitions are key-foreign key joins. Under these circumstances, the duplicate elimination technique used is correct. An interesting point to note is that the class of views considered in literature, such as [1, 5, 6, 15], allow the join conditions to be between any two columns (in particular, non key-foreign key joins). Also, excepting [5], the XML-to-SQL query translation algorithms in literature do not know (or use) information about the integrity constraints that hold on the underlying relational schema. In this section, we look at what needs to be done to perform duplicate-elimination correctly when the join conditions are allowed to be over any two columns.

Suppose id is not a key for the Book, TopSection and NestedSection relations. Then, the join between Book.id and TopSection.bookid in the view definition $\mathcal{T}_2$ is not a key-foreign key join. Similarly, the join between TopSection.id and NestedSection.topsectionid is also not a key-foreign key join.

What happens in this case is that some parts of the relational data may appear in the XML view multiple times. For example, part of an instance of the relational data is shown in Figure 3.

Suppose the XML view $\mathcal{T}_2$ was defined with $P_1 = 65$ and $P_2 = 65$. For the above data instance, the corresponding view will have three book elements. Since two of the books have the same value for the id column, the sections of each of these two books will be repeated under both of them. For example, the Introduction and Motivation sections will appear twice in the XML view, once for each of the two books with id 1. But, both occurrences of these section titles correspond to the same schema node (node 12 in Figure 2). As a result, our earlier technique using schema node numbers does not work. Note how the XML view has redundant data

Book

id	title	price	....
1	ABC	50	
1	EFG	60	
2	EFG	70	

TopSection

id	bookid	title	....
1	1	Introduction	
2	1	Motivation	
⋮	⋮	⋮	

Figure 3: Sample data instance

even when $P_1 = P_2$.

Previously, we just had to keep track of some information concerning the properties of the query result schema nodes. Once the joins are allowed to be general joins, we need to keep track of the actual relational tuples that contribute to the result tuple. The distinct clause needs to trace the lineage of the resulting value.

One way to do this is as follows: project the values of the key fields for each relation in the query. Also project the node identifier of the return schema node. Finally, the distinct clause is applied across these fields. If the number of relations is different across various paths (like our example, in which retrieving NestedSection titles requires two joins while retrieving TopSection titles requires only a single join), we add appropriate number of null columns as required. The SQL query in this case, SQ_2^3 is given below. Here we assume that $R.key$ is the key field for relation R.

```
with Temp(nodeid, key1, key2, key3, title) as (
  select 12, B.key, TS.key, null, TS.title
  from   Book B, TopSection TS
  where  B.id = TS.bookid and B.price <= P1
  union all
  select 16, B.key, TS.key, NS.key, NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
         and B.price <= P1
  union all
  select 16, B.key, TS.key, NS.key, NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
         and B.price <= P1
  union all
  select 14, B.key, TS.key, null, TS.title
  from   Book B, TopSection TS
  where  B.id = TS.bookid and B.price > P2
  union all
  select 17, B.key, TS.key, NS.key, NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
         and B.price > P2
  union all
  select 17, B.key, TS.key, NS.key, NS.title
  from   Book B, TopSection TS, NestedSection NS
  where  B.id = TS.bookid and TS.id = NS.topsectionid
         and B.price > P2
)
select distinct(nodeid, key1, key2, key3, title)
from   Temp
```

The above technique will work when $\mathcal{T}$ is a tree XML view as the schema node id for a node n uniquely identifies the root-to-leaf path to n. This, in turn, identifies the name of the relations that appear on the root-to-leaf path and the corresponding schema node ids.

We would like to emphasize the fact that we are not arguing that scenarios like the above one are likely to be common. But, since existing XML-to-SQL query translation techniques do not place any restrictions on the class of allowable views and also are not aware of what relational integrity constraints hold over the underlying data, the above relational schema is a valid input and we need the above techniques to obtain the correct results.

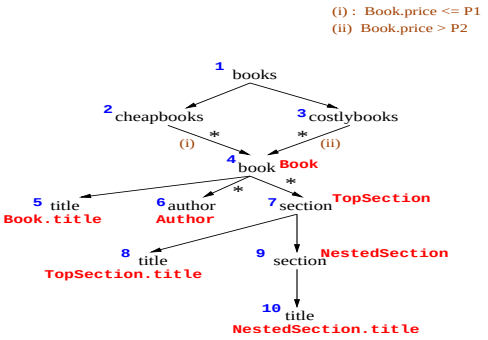

Figure 4: DAG XML view $\mathcal{T}_3$

3. DAG XML VIEWS

If the schema for the XML view is a directed acyclic graph (DAG), the solution for duplication-elimination becomes more complex. Recall that the complete solution for tree XML views (in Section 2.2) is to project the schema node id of the projected element along with the key fields of the joining tuples. The underlying idea was that the schema node id uniquely identifies the root-to-leaf path up to the projected element, which in turn identifies the names of the corresponding relations. For DAG XML views, the node id of the projected node no longer uniquely identifies the root-to-leaf path. So, we need a slightly more complex mechanism to achieve the same goal.

For example, consider the XML view $\mathcal{T}_3$ in Figure 4, which is similar to the XML view $\mathcal{T}_2$ in Figure 2. The only difference between these two views is that while $\mathcal{T}_2$ is a tree view, $\mathcal{T}_3$ is a DAG view. So, while $\mathcal{T}_3$ is a more compact schema compared to $\mathcal{T}_2$, both the views will generate identical XML documents from any given relational instance.

In this case, notice how the section titles for the cheap books and costly books correspond to the same schema node. So, just projecting the node id of top level and nested section titles will not suffice. This happens because there are multiple paths from the root to the title nodes. We need to identify the complete path and apply the distinct clause over this path identifier. One way to do this is to associate a pathid with respect to each root-to-leaf path in the XML view and use this path id to eliminate duplicates. A simple pathid is the concatenation of the node ids of all nodes appearing on the path. The SQL query obtained SQ_3^1 is similar to the query SQ_2^3 in Section 2.2 with the following difference: the nodeids 12, 16, 14 and 17 will be replaced by pathids 1.2.4.5, 1.2.4.7.8, 1.3.4.5 and 1.3.4.7.8 respectively.

A point to note here is that if the DAG view has two nodes u and v such that there are two edges from $u \rightarrow v$, then the pathid needs to include edge identifiers as well.

4. A GENERAL SOLUTION

From the previous examples, we see that a generic solution for the simple class of views considered in this paper requires a fairly complex mechanism. On the other hand, if we know something about the properties of the XML view much simpler solutions suffice. In this section, we categorize the solutions based on the properties of the XML views.

4.1 Definitions

We use the simple approach of defining an XML view with annotations on the XML schema nodes and edges. A non-leaf node may be annotated with a relation name, while a

Table 1: Summary of duplicate-elimination techniques

XML View Type	Duplicate elimination mechanism
all views	Use key field(s) for each relation in the query + identifier for the root-to-leaf path
well-formed views	Use key field(s) corresponding to projected value + identifier for the root-to-leaf path
non-redundant views	Use key field(s) corresponding to projected value + Relation name

leaf node is annotated with the name of a relational column. Each edge may be annotated with a join condition and/or a selection condition.

We define an XML view to be a *well-formed* XML view if each join condition involved in an edge annotation is a key-foreign key join. The relation corresponding to the parent node should be the relation with the key field in this join condition.

We define a relational column $R.C$ to be *non-redundantly* mapped in the XML view if for every valid data instance, no tuple value in this column $R.C$ will appear more than once in the XML view. Otherwise $R.C$ is redundantly mapped. Note that two different tuples t_1 and t_2 in R may have the same value for this column and, in turn, create two different XML elements.

For example, every column in the view $\mathcal{T}_1$ (Figure 1) is non-redundantly mapped. On the other hand, for the view $\mathcal{T}_2$ (Figure 2), if $P_1 > P_2$, then the columns TopSection.title and NestedSection.title are redundantly mapped.

An XML view $\mathcal{T}$ is non-redundantly mapped if each of the relational columns annotating some leaf node in $\mathcal{T}$ is non-redundantly mapped. Otherwise, the view is redundantly mapped.

Finally, $\mathcal{T}$ is a tree XML view if the corresponding XML schema is a tree schema. If the XML schema is a DAG schema, then $\mathcal{T}$ is a DAG XML view. If the XML schema is recursive, then $\mathcal{T}$ is a recursive XML view.

4.2 The solutions

The different duplicate elimination techniques presented in this paper are summarized in Table 1. For a generic solution over all XML views, the correct solution is to use the key field(s) of all the joining relations along with a pathid identifying the root-to-leaf path. While in our examples, we used one column per key field in the select clause and a single column for the pathid, other alternatives are also correct. For example, we can create a single value by concatenating all the key field(s) and the pathid, and apply a distinct clause on this value.

If the input XML view is well-formed, then the key field(s) corresponding to the projected schema node suffices (instead of key field(s) of all joining relations).

Furthermore, if we know that the XML view is non-redundantly mapped, then we can replace the schema node id with the relation name. If the query results are from a single relation, we can just use the key of this relation. We no longer need the relation name in this case.

Recall that for DAG XML views, a string representing the entire root-to-leaf path is a valid pathid. For tree XML schemas, the node id of the projected schema node uniquely identifies the entire root-to-leaf path. So, just the node id is a valid pathid for tree XML views.

Going a step further, let us look at what more needs to be done for recursive XML views. The main difference in this case is that the actual depth of the XML document cannot be determined at query translation time. So, this means that the path id cannot be hard-coded into the query, but needs to be constructed as part of the query. Similarly, the number of iterations in the recursion will depend on the actual data. As a result, the key field(s) of all joining relations need to be represented as a string field, which will be constructed as part of the query.

4.3 Identifying non-redundant views

We briefly outline a technique for identifying when a tree XML view is non-redundant. For a leaf node n in the schema, we define the root-to-leaf query $Query(n)$ as the SQL query obtained by (conjunctively) combining the annotations on the edges of the root-to-leaf path of n and projecting the annotation of node n, along with the key field(s) of the corresponding relation. For example, $Query(16)$ in Figure 2 is the query shown below.

```
select NS.id, NS.title
from   Book B, TopSection TS, NestedSection NS
where  B.id = TS.bookid and TS.id = NS.topsectionid
       and B.price <= P1
```

In order to determine if a relational column $R.C$ is non-redundantly mapped, we need to check if for every pair of schema nodes u, v annotated with $R.C$, $Query(u) \wedge Query(v)$ is empty. If so, then the view is non-redundantly mapped. We can check this by using the techniques proposed in literature for solving conjunctive query containment, such as [4, 18].

Notice that the above solution can be applied for any DAG XML view by constructing an equivalent tree XML view from the DAG view through unrolling of the DAG schema. Extending the same techniques to identify non-redundant recursive XML views is future work.

An important class of non-redundant XML views arise in the context of XML Storage. Here, an RDBMS is used to store and query XML data. As discussed in [16], it is possible to use techniques from the XML publishing domain in the XML storage domain. To see this, notice that once XML data is shredded into relations, we can view the resulting data as if it were pre-existing relational data. Now by defining a reconstruction view that mirrors the XML-to-relational mapping used to shred the data, the query translation algorithms in the XML publishing domain are directly applicable. Most of the techniques in published literature [2, 9, 12, 13, 17] result in reconstruction XML views that are non-redundant.

In general, the problem of checking if a given XML view is non-redundant is closely related to the classical query containment problem. While the general problem over recursive XML views is undecidable, investigating the complexity of this problem for different classes of XML views and identifying when it is tractable is future work.

5. DISCUSSION

We have seen how even for a very simple class of XML views, a fairly complex duplicate-elimination technique is required. In this section, we first look at how the solutions proposed in this paper extend to more complex view definition mechanisms proposed in literature. We then look at whether we can avoid the duplicate elimination problem by not generating duplicates in the first place during XML-to-SQL query translation.

5.1 Complex view definitions

Consider a more general class of annotation-style XML views, where each edge annotation is allowed to be a conjunctive query. We refer to this class of views as generalized views. The duplicate-elimination techniques proposed in Section 4 for well-formed views and non-redundant views are directly applicable for their generalized counterparts. This is due to the fact that the solution for these two classes of views requires just the key field(s) of the relation corresponding to the projected element. The solution for an arbitrary generalized view needs a minor extension though: we need to use the key field(s) for all relations occurring in the conjunctive queries annotating each of the edges from the root to the projected element.

We now look at the various view definition languages proposed in literature. In Silkroute [7], Relational to XML Transformation language (RXL) was introduced to define XML views over relational data. In [8], XQuery was used as the view definition language. In both these cases, the internal representation used by the Silkroute system for the XML view definition was a view forest. The view query specification used in [3] (schema tree query), and in [10] (View Tree) are adapted from view forests. The core representation of a view forest is identical to a generalized tree XML view. So, the techniques proposed in this paper are directly applicable for a large class of view forest views.

In [1], a view definition language based on Attribute Translation Grammars is presented. Here, a DTD is extended by associating semantic rules in the form of SQL queries. The semantic rules are logically equivalent to the annotations described in this paper. The class of ATG views defined with conjunctive queries corresponds to generalized XML views discussed above.

In MARS [5], the view definition mechanism uses skolem functions to generate unique identifiers for each XML element in the XML view. While it is true that in the presence of such skolem functions the duplicate elimination problem can be trivially handled, it should be noted that all the techniques presented in this paper are applicable in the design of skolem functions.

In XPeranto [15], XQuery is used as the view definition language and XML Query Graph Model (XQGM) is used as the intermediate representation. Similarly, in [5, 14] LAV-style (Local-As-View) views are allowed. In all these cases, the view definition mechanisms are a lot different from the class of annotation-style views considered in this paper. Adapting the duplicate-elimination techniques proposed in this paper to these techniques is future work.

5.2 Avoid generating duplicates: An alternative to duplicate elimination

For the simple query we considered in this paper, there is an alternative way to avoid the duplicate-elimination problem. Recall that the first step in XML-to-SQL query translation is to find the schema nodes that match the various elements in the query. By adding a duplicate elimination step that removes duplicate root-to-leaf paths at this stage, it is possible to avoid generating duplicate results in the first place. In [11] we presented an XML-to-SQL query translation algorithm for path expression queries that follows this methodology. For the class of path expression queries without predicates, this technique avoids generating duplicate results completely. On the other hand, for branching path expression queries, this algorithm does not eliminate duplicate paths completely and may generate duplicate results. The duplicate elimination techniques presented in this paper are applicable in this context.

Most of the other published XML-to-SQL query translation algorithms [5, 6, 15] do not explicitly keep track of the actual matching paths. As a result, it is not obvious as to how these techniques can be augmented to avoid generating duplicate results in the first place. In these cases, the techniques proposed in this paper are necessary to perform duplicate-elimination correctly.

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

6. REFERENCES

[1] M. Benedikt, C. Y. Chan, W. Fan, R. Rastogi, S. Zheng, and A. Zhou. DTD-Directed Publishing with Attribute Translation Grammars. In *VLDB*, 2002.

[2] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML schema to relations: A cost-based approach to XML storage. In *ICDE*, 2002.

[3] P. Bohannon, H. Korth, P.P.S. Narayan, S. Ganguly, and P. Shenoy. Optimizing view queries in ROLEX to support navigable tree results. In *VLDB*, 2002.

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

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

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

[7] M. Fernández, D. Suciu, and W.C. Tan. SilkRoute: Trading Between Relations and XML. *WWW9*, 2000.

[8] M. F. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W. C. Tan. SilkRoute: A framework for publishing relational data in XML. *ACM Trans. Database Syst.*, 27(4), 2002.

[9] S. Hongwei, Z. Shusheng, Z. Jingtao, and W. Jing. Constraints-Preserving Mapping Algorithm from XML-Schema to Relational Schema. In *EDCIS*, 2002.

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

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

[12] D. Lee and W.W. Chu. Constraints-preserving Transformation from XML Document Type Definition to Relational Schema. In *ER*, 2000.

[13] M. Mani and D. Lee. XML to Relational Conversion using Theory of Regular Tree Grammars. In *VLDB Workshop on EEXTT*, 2002.

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

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

[16] J. Shanmugasundaram, E. Shekita, J. Kiernan, R. Krishnamurthy, S. D. Viglas, J. Naughton, and I. Tatarinov. A General Technique for Querying XML Documents using a Relational Database System. *SIGMOD Record*, 30(3), 2001.

[17] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*, 1999.

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