XQuery Translation to Sem-SQL[*]

Li Yang
Department of Computer Science
State University of West Georgia
Carrollton, GA 30118, U.S.A.

Naphtali Rishe
High Performance Database Research Center
School of Computer Science
Florida International University
Miami, FL 33199, U.S.A.

Abstract *XML query translation is an inevitable step involved in using non-XML databases storing XML data. In this paper, we address the XQuery to Sem-SQL translation issue, part of the XML storage and retrieval using the Semantic Binary Object-Oriented Database System (Sem-ODB) project, by providing a high-level description of the translation scheme between XQuery and Sem-SQL. Our translation scheme utilizes the navigation-oriented paradigm of Sem-SQL in translating XQuery path expressions to avoid excessive joins. Our approach can deal with all recursive path expressions without requiring the underlying database system to support recursive queries. We also extend the sorted outer union approach with consideration on user-defined orderings.*

Keywords: XQuery, Sem-SQL, XML query translation, sorted outer union approach

1.0 Introduction

In recent years, as its popularity growing rapidly, the Extensive Markup Language (XML) has been widely adopted in applications like E-commerce, and B2B processes. As the amount of XML data grows, so does the need for storing and retrieving it efficiently. Different approaches have been proposed to store and query XML documents [5, 11, 4, 3, 14, 15, 12]. Among them, a great deal of effort has been put on using relational databases as XML repositories because of its mature technology. However, because of the mismatch between the two-dimensional table-column-based relational data model and the graph-based XML data model, fragmentation is inevitable when mapping an XML schema into a relational one [11]. Consequently, *path expressions,* one of the basic expressions in any XML query language, can only be translated as joins in SQL.

To avoid the above problems and efficiently store and retrieve XML, we are building a system using the Semantic Binary Object-Oriented Database System (Sem-ODB) [13]. Sem-ODB is based on a conceptual data model, the Semantic Binary Object-Oriented Data Model (Sem-ODM [8]). Sem-ODM has the features of Object-Oriented data models, such as inheritance, surrogates (object ids), explicit description of relationships and a high-level data model. The Sem-ODB has been successfully deployed for highly complex applications such as applications intended for storage and processing of a large number of earth science observations and the Terrafly Geographic Information System (GIS) [7]. The query language that is supported by the Sem-ODB is the Semantic SQL (Sem-SQL) [9]. It is compatible with the traditional SQL92 with some enhanced semantic interpretation to reduce the complexity in designing a query. To store an XML document, the system first maps an XML DTD (Document Type Definition) into a Sem-ODM semantic schema, and loads the XML data into the Sem-ODB created according to the mapping, and then an XQuery query is translated into a Sem-SQL query and posed against the Sem-ODB database, and finally the results will be tagged and returned as XML data. In this paper, we will only focus on the XQuery to Sem-SQL query translation part. Interested readers are referred to [13] for more details.

* This research was supported in part by NSF grants EIA-0320956, EIA-0220562, and HRD-0317692.

In this paper, we address the XQuery to Sem-SQL translation issue by providing a high-level description of the translation scheme between XQuery and Sem-SQL. Our translation scheme utilizes the navigation-oriented paradigm of Sem-SQL in translating XQuery path expressions. Thus, it avoids the common problem of the join number being equal to the length of a path expression when translating an XML query into a SQL query in relational approaches [11]. Our approach can deal with all recursive path expressions without requiring the underlying database system (Sem-ODB in our case) to support recursive queries. We also extend the *sorted outer union* approach [10] with consideration on user-defined orderings.

The rest of the paper is organized as follows. Section 2 presents related work. A brief overview of the Sem-ODM, Sem-SQL, DTD and XQuery is given in section 3. Section 4 describes the translation scheme from XQuery to Sem-SQL. Conclusion and future work are presented in section 5.

2.0 Related Work

The issue of XML query translation to SQL has been addressed in several research projects recently. Yoshikawa et al. [15] propose an approach called XRel for storage and retrieval of XML documents using relational databases. Their work only focuses on path expressions, which are a subset of XQuery and relatively easier to translate than some complex XQuery expressions such as FLWOR. Our work distinguishes from theirs by offering a more complete picture on the correspondence between XQuery and Sem-SQL. Furthermore, because path expressions do not contain any ordering expressions, their approach always generates the data in document order. We consider queries with document order as well as user-defined order. Similar to XRel, another work done by Tatarinov et al. [14] offers algorithms for translating XPath path expressions into SQL queries. Their work focuses on evaluating the performance of different order encoding methods. They also consider translating insert/delete queries into SQL, however, since these update-related features are not specified in the current version of XQuery specifications, we do not consider them in this paper.

Manolescu et al [6] address the XML query translation to SQL issue in a data integration context. They provide an analysis of XML query languages and describe the translation of different features to SQL. Our work has been influenced by theirs. However, our work is focused on translating XQuery to Sem-SQL, while theirs is on Quilt to SQL. The differences between XQuery and Quilt, and Sem-SQL and SQL, result in different query translation schemes. For instance, id-based queries can be translated into Sem-SQL since surrogates are supported in Sem-SQL, while it is not possible in SQL. Besides the major XQuery expressions, we also describe in our work how to deal with FLWOR expressions (i.e., FLWR with an order by clause), which is introduced in the latest W3C XQuery specifications.

3.0 Overview of Sem-ODM, Sem-SQL, DTD and XQuery

Sem-ODM: The Sem-ODM, a conceptual, high level data model, is the underlying data model of Sem-ODB. Two constructs, *category* and *relation*, are used to describe a Sem-ODM. Categories are like *Entities* in the Entity Relationship (ER) model, except that the *Attributes* in the ER model are represented as relations in Sem-ODM. A Category can either be a *Concrete Category* or an *Abstract Category*. Concrete Categories are categories like String, Number, and Boolean, etc. Abstract Categories are categories composed of abstract objects, such as books. There can be binary relations from an abstract category, which is called the *Domain* of the relation, to another category, which is called the *Range* of the relation, in a Semantic Schema.

Sem-SQL: The Semantic SQL (Sem-SQL) was adopted from traditional SQL92. Compared to the traditional SQL, the Sem-SQL incorporates some advanced concepts, such as the navigation operator and inverse relation operator, which significantly reduce the length of a complex query and provide an easier query facility. In the Sem-SQL, double underscores represent the navigation operator. It provides a way to query relationships directly, and allows navigation from one category *A* to categories that are reachable from *A* in a semantic schema via the relations along the way. In this way, some queries involving different categories do not have to be joined, thus reducing the overhead of join operations. Sem-SQL's navigation-oriented query paradigm is one of the most salient features that distinguish it from the traditional SQL. The inverse relation operator is represented as one underscore in the Sem-SQL. It makes it possible to traverse from the range of a relation *R* to the domain of *R* in a query. It also helps one to navigate in a schema graph without worrying about join operations.

DTD: Document Type Definition (DTD) [1], a schema language that can be used to specify the basic building blocks as well as the structure and semantic constraints of XML documents. Figure 1 (a) shows a DTD example that is extracted from [11] and slightly modified and used as the DTD running example throughout this paper ; (b) shows a tree representation of an example XML document conforming to (a). The shaded nodes represent text nodes, while the black node is the document node, and triangles represent attributes.

```
<!ELEMENT publication (article| monograph)*>
<!ELEMENT article (title, author*, contactauthor)>
<!ELEMENT title  (#PCDATA)>
<!ELEMENT contactauthor EMPTY>
<!ATTLIST contactauthor authorID IDREF #IMPLIED>
<!ELEMENT monograph (title, author, editor)>
<!ELEMENT editor (monograph*)>
<!ATTLIST editor name CDATA #REQUIRED>
<!ELEMENT author (name, addr)>
<!ATTLIST author id ID #REQUIRED>
<!ELEMENT name (first?, last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT last  (#PCDATA)>
<!ELEMENT addr ANY>
```

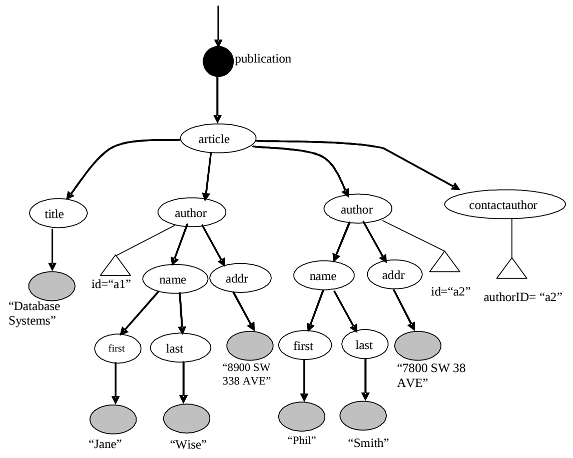

Figure 1 (a) DTD Example (b) Example XML Document Tree

XQuery: The XQuery [2] is the W3C XML query language proposal. It is a functional language, in which *expressions* are the building blocks of a query and they can be nested with full generality as in OQL. Literals, variables and function calls are primitives of the language, and they can be used to construct other more complex expressions such as path expressions, element constructors, FLWOR expressions, conditional expressions, and others, which are the principal expressions of XQuery.

4.0 XQuery to Sem-SQL Translation

Before translating an XQuery query into a SQL query, we first simplify the XQuery query using some of the normalization rules proposed in [6] to reduce its complexity and to ease the Sem-SQL translation process. Due to the space constraint, we do not describe the rules here. For more details, please refer to [6, 13]. The result of the simplification process is a simplified XQuery query. For instance, a complex FLWOR expression where FLWOR expressions nested in the For-clause of another FLWOR expression can be shortened to one single FLWOR expression.

To translate the simplified XQuery, we consider the translation schemes on the principal expressions of XQuery, including path expressions, FLWOR expressions, functions, and comparison expressions. In the following sections, we only illustrate the translation schemes on path expressions and FLWOR expressions.

4.1 Path Expression Translation Scheme

Path expressions such as */a//b* are the fundamental building blocks of an XQuery query with which traversal over the tree structure of XML documents is made possible. To better explain the translation scheme, we assume *a* and *b* are XML elements instead of step expressions. A path expression may consist of relative path delimiters '//' or wildcards such as '*' or recursive queries, in which case, recursive views have to be supported by the underlying database to translate such an expression into a SQL query. Unfortunately, SQL92 doesn't support such views, neither does Sem-SQL. In our work, we adopt the idea of [15] to build a Path Table *PT* to alleviate the difficulty. Each row in *PT* represents a distinct path from the root node to the current node. Each path is denoted as a concatenation of the nodes in the path from the root to the node currently being parsed. The delimiter '/' is used to separate nodes from each other. Attribute nodes are prefixed with '@' symbol (e.g. */publication/article*, /*publication/article/author/@id*). This path table can be created during the XML document parsing process. Because the number of paths in an XML document is considerably less than the size of the actual XML document, this path table can be made as an in-memory table to reduce the expensive and time-consuming I/O operations if otherwise placed on a secondary storage. Because of the fact that *PT* is generated from an XML tree, where all the possible paths are present, even if there is a recursive query, such as *//monograph[title= "test"]//editor* which returns all the editors that are reachable from the monograph titled "test", we can find all the matched paths in *PT*. Thus we can generate one SQL statement for each case and outer union all the cases together to form a single SQL.

In our work, we take into consideration each of the basic components, such as */E, //E, E_1//E_2, E/@att, E/..*, of a path expression, where *E*, E_1 and E_2 represent XML elements, and *att* represents attributes of elements. The following sections show a general translation scheme for the most basic path expression */E*. The translation of the other components is built on /*E*. Note that the scheme we present here is used for the reconstruction mode where the information of all the descendents of a node is also returned. The translation scheme for an evaluation mode in which only the information of the current nodes is requested is much simpler.

To translate */E* in the reconstruction mode:

1) Find the root *R* of the document, check if *R=E*, if no, return error, otherwise,
2) Search PT for all the paths p_i (i =1..n) which start with /R;
3) Keep an integer *attNum* whose initial value is 0 to track the number of *null* that will be put in the *select*-clause of the Sem-SQL statement to substitute for the inapplicable attributes;
4) For each matched path p_i, search the KnowledgeBase (KB) (a Sem-ODB database where the mapping of an XML schema to a Sem-ODM semantic schema is stored) to create a navigation path that corresponds to p_i. Suppose the navigation path created for the *ith* matched path is $C_R\_\_C_1\_\_\ldots\_\_C_i$, and check whether there are paths with the same prefix but ends with *@att* which indicates these are attributes. Suppose there are *k* attributes in p_i, and then we create a Sem-SQL query as the following:

$$\begin{array}{ll} \textit{select} & i \textit{ as label}, C_R\_\_C_1\_\_\ldots\_\_C_i \textit{ as id}, C_R\_\_C_1\_\_\ldots\_\_C_i\_\_\textit{order as order}, \\ & \overbrace{\textit{null, null, }\ldots\textit{null}}^{attNum}, C_R\_\_C_1\_\_\overbrace{\ldots\_\_C_i\_\_att_1, C_R\_\_C_1\_\_\ldots\_\_C_i\_\_att_2, \ldots, C_R\_\_C_1\_\_\ldots\_\_C_i}^{k}\_\_att_k \\ \textit{from} & C_R \end{array}$$

Note that the first column *label* in the above statement represents different paths. It can be used by the system to tag the result. It starts with 0, representing the top-most level element. C_R represents the category corresponding to the root *R*; $C_R\_\_C_1\_\_ \ldots\_\_C_i$ corresponds to the element appeared as p_i; The third column *order* denotes the document order of element p_i.

5) Increment *attNum* by *k* (i.e. *attNum = attNum + k*)
6) Before generating the final Sem-SQL statement, append *null* to the select-clause of each individual statement to ensure that there are *3+attNum* items projected;
7) Outer-union all the Sem-SQL statements created above and append Order by *3* so that the result is in document order. The final Sem-SQL query is shown in Figure 2 (suppose there are *m* attributes in total in the final statement, and the root element and last element do not contain attributes).

```
select     0 as label, C_R as id , C_R__order as order, null, ….., null
from       C_R
union all                                          m
……………
union all
select               i as label,C_R__C_1__ …__C_i as id ,  C_R__C_1__ …__C_i __order as order,
           null,…null, C_R__C_1__ …__C_i__att_1, C_R__C_1__ …__C_i__att_2,…, C_R__C_1__ …__C_i __att_k, null, ……, null

from       C_R
…………                                                                              m
union all
select               n as label,C_R__C_1__ …__C_n as id , C_R__C_1__…__C_n __order as order,
           null,…., null
from       C_R                        m
order by 3
```

Figure 2: Sem-SQL Translation for */E* in the reconstruction mode

Note that in the above translation, we fill each individual Sem-SQL statement with *nulls* at the position of unmatched columns. In this way, several SQL statements with different column types can be unioned together as one statement, which is the ultimate goal of the *outer union* approach.

4.2 FLWOR Expression Translation Scheme

Figure 3 shows an informal representation of the FLWOR expression. Note that $\$x_1$, $\$x_2$, $\$a_1$, and $\$a_2$ are variables, and Exp_1, Exp_2, l_1, l_2, o_1, o_2, *and* g_i *(i=1..n)* represent expressions. Except g_i, which appears to construct the result documents and denotes the *ith* expression appearing in the return clause, the rest of the notation is self-explanatory.

```
for    $x_1 in Exp_1,
       $x_2 in Exp_2
let    $a_1 := ….
       $a_2: = ….
where  l_1 and l_2
order by o_1, o_2
return
              g_1
            …………
              g_n
```

Figure 3: Info R Expression

The translation scheme of an FLWOR follows.

1) Expand g_i with its descendant in rt all the descendant information into appropriate positions in a template clause of the FLWOR, so that we know what expressions are needed to be evaluated in the *return* clause. For instance, *return /author* should be translated into *return <author id = $u> <name> <first>$f</first> <last> $l </last></name><addr>$addr</addr></author>*. In this way, we might get more expressions than those present in the original FLWOR expression. Suppose we will have *m* expressions in the form of h_i (*i* =1..m) after inserting the descendants of g_i (*i* = 1..n), where m ≥ n.
2) Search the KB for the starting categories S_1, S_2 based on Exp_1 and Exp_2.

3) Look up the Path Table *PT* to find all the paths that are necessary for evaluating o_1 , o_2 , l_1 , l_2 and h_i (i =1..m).
4) Check the KB to generate the navigation paths for all the above XQuery paths. Suppose they are *o_1_path, o_2_path, l_1_path, l_2_path, h_i_path (i=1..m).*
5) Create the following Sem-SQL statement shown in Figure 4 for the FLWOR expression:

```
            select 0, o1_path , o2_path, h1_order, h1_path, null, …,null
            from S1, S2
            where Predicate( l1_path) AND Predicate(l2_path)          m+1
union all
            select 1,  o1_path , o2_path, h2_order, null,h2_path, …,null
            from S1, S2
            where Predicate( l1_path) AND Predicate(l2_path)        m+1
union all
…………………
union all
             select m-1, o1_path , o2_path, hm_order,null, null, …,hm_path
             from S1, S2
             where Predicate( l1_path) AND Predicate(l2_path)          m+1
order by  2,3,4
```

Figure 4: Sem-SQL Translation for a Simple FLWOR Expression: the result is ordered by the ordering specifications $o_1$and o_2 in the *order by* clause and document order of h_i

Basically in this case one Sem-SQL statement is created for each *h_i_path* identified by a different *label* value and the final statement is produced by unioning all the Sem-SQL statements together. In case that any of o_1, o_2, and h_i corresponds to more than one navigation path, we create a statement for each different case with the same label if the difference comes from h_i and then use the union all operator. *h_i__order* represents the document order of each expression in the return clause. By projecting on and selecting *o_1_path, o_2_path,* and *h_i_path,* we achieve both user-defined and document ordering.

5.0 Conclusion and Future Work

The XML query translation is an inevitable step involved in using non-XML databases storing XML data. In this paper, we present the translation schemes for converting major XQuery expressions into Sem-SQL queries. The advanced features of Sem-SQL enable a seamless and natural translation from XQuery to Sem-SQL. For instance, the navigational query paradigm of the Sem-SQL makes it possible to traverse the categories in a Sem-SQL query the same way as it traverses nodes in an XQuery path expression, thus it reduces the number of joins operators. In addition, we propose a solution to dealing with recursive queries by utilizing a path table to accommodate the limitation of Sem-SQL with recursive queries. We extend the *sorted outer union* approach with consideration on user-defined orderings.

However, it is worth pointing out that the expressive power of XQuery as a query language for XML data from both databases and documents prevents a complete and one to one conversion to Sem-SQL as well as to SQL. Some features of XQuery such as some XML specific built-in functions cannot be implemented directly via SQL/Sem-SQL due to unmatched semantics, while other features such as mixed queries on meta-data and data cannot be translated due to incomparable expressive power. To fully and efficiently support those features, extensions to Sem-SQL/SQL both syntactically and semantically have to be considered.

6.0 References

[1] J. Bosak, et. al. W3C XML Specification DTD. http://www.w3.org/XML/1998/06/xmlspec-report.html .

[2] S. Boag, D. Chamberlin, et al (Eds.). XQuery 1.0: An XML Query Language. W3C Working Draft, 15 November 2002, http://www.w3.org/TR/xquery/ .

[3] D. Florescu, and D. Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Technical Report 3684, INRIA, 1999.

[4] G. Kappel, E. Kapsammer, S. Rausch-Schott, and W. Retschitzegger. X-Ray -Towards Integrating XML and Relational Database Systems. In Proceedings of ER'00, Salt Lake City, Utah, 2000.

[5] C. C. Kanne, and G. Moerkotte. Efficient Storage of XML Data, In ICDE'00, California, 2000.

[6] I. Manolescu, D. Florescu, and D. Kossmann. Pushing XML queries inside relational databases. Tech. Report No. 4112, INRIA, January 2001.

[7] N. Rishe. TERRAFLY: A High-Performance Web-based Digital Library System for Spatial Data Access. In Proceedings of ICDE'01, Heidelberg, Germany, 2001.

[8] N. Rishe. Database Design: The Semantic Modeling Approach. McGraw-Hill, 1992.

[9] N. Rishe. Semantic SQL. Internal Document, High-Performance Database Research Center, School of Computer Science, Florida International University, 1998.

[10] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, et al. Efficiently Publishing Relational Data as XML Documents. In Proceedings of VLDB'00, Cairo, Egypt, 2000.

[11] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, et al. Relational Databases for Querying XML Documents: Limitations and Opportunities. In Proceedings of VLDB'99, Edinburgh, 1999.

[12] F. Tian, D. J. DeWitt, J. Chen, and C. Zhang. The Design and Performance Evaluation of Alternative XML Storage Strategies. In ACM SIGMOD Record, 31(1):5-10, 2002.

[13] L. Yang, XML Storage and Retrieval Using the Semantic Binary Object-Oriented Database System (Sem-ODB), PhD Dissertation, Florida International University, 2003.

[14] I. Tatarinov, S. Viglas, K. S. Beyer, et al. Storing and Querying Ordered XML Using a Relational Database System. In Proceedings of SIGMOD'02, pp 204-215, Madison, Wisconsin, USA, 2002.

[15] M. Yoshikawa, T. Amagasa, et al. XRel: A Path-Based Approach to Storage and Retreval of XML Documents Using Relational Databases. In ACM Transactions on Internet Technology, 1(1): 110-141, 2001.