Query Engines for Web-Accessible XML Data

Leonidas Fegaras   Ramez Elmasri

Department of Computer Science and Engineering
The University of Texas at Arlington
416 Yates Street, P.O. Box 19015
Arlington, TX 76019-19015
email: {*fegaras,elmasri*}*@cse.uta.edu*

Abstract

Even though XML was first introduced as a schema-less, self-describing data representation language, there are now proposals for XML schema descriptions. The addition of schema information opens new opportunities in using the already established database management technology for storing and handling XML data, since databases are traditionally focused on data that conform to a fixed, predefined schema. Schema information allows better data integrity, more effective data storage, and more efficient query evaluation. This paper describes an effective framework for storing XML data in an object-oriented database and an optimization framework for translating XML queries into efficient algorithms. We first present a new type system for describing XML data, well integrated with the ODL type system of the ODMG standard, that captures both schema-less (semi-structured) and schema-based XML data. We then introduce a small set of syntactic extensions to ODMG OQL, powerful enough to make OQL a full-fledged XML query language. Next, we present a framework for translating XML queries into OQL queries based on XML schema information. Instead of inventing yet another semi-structured algebra for expressing our translations, the target of our transformation rules is OQL code, which not only has precise semantics, but has also been the focus of various optimization techniques. Schema information is an indispensable component of our transformations. It is used in disambiguating terms with multiple interpretations such as wildcard tag projections, in choosing the storage format for the XML data, and in generating OQL code guided by the choice of storage.

1 Introduction

XML [16] has emerged as the leading textual language for representing and exchanging data on the web. Even though XML has been developed without any active involvement by the mainstream database community, some database researchers have actively participated in the development of other standards centered around XML, such as query languages for XML, and have addressed storage and performance issues for XML data repositories. There is a wide range of opinions about the relationship between XML and databases. Some researchers believe that XML will dominate the database area and will eventually replace relational databases, while others dismiss it as nothing but a tree-structured representation, not much different from Lisp's S-expressions or hierarchical data models. Regardless of which side one takes, there is good news in the XML standardization arena, which may open new opportunities for database research in this area [15]. Even though XML was introduced as a schema-less, self-describing language, there are proposals now for XML schema languages, such as DTD (Document Type Declaration) and XML Schema [16]. The development of such schema descriptions was not driven by efficiency concerns per se, but rather by the need for enforcing data integrity. On the other hand, databases have been traditionally focused on data that conform to a fixed, predefined schema. This also remained the focus of the emerging database languages, such as the object-relational and object-oriented database (OODB) languages. There is a good reason for the tendency of databases towards static typing: In addition to safety and data integrity,

**Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001**

schema information allows more effective data storage and access paths, and more efficient query evaluation.

There have been many commercial products recently that take advantage of the already established database management technology for storing and handling XML data. In fact, nearly all relational database vendors provide now some functionality for storing and handling XML data in their systems, such as the Oracle XML Developer's Kit (XDK). We now see similar products showing up from OODB vendors, such as POET's Content Management Suite and Object Design's eXcelon system. Most of these systems support automatic insertion of canonically-structured XML data into tables, rather than utilizing the XML schemas for generating application-specific database schemas. They also provide methods for exporting database data into XML form as well as querying and transforming these forms using XML query languages. Consequently, these systems can be classified as semi-structured since they do not make use of XML schema information for better performance. Nevertheless, there are some systems that can be classified as schema-based, such as the Niagara project [14]. The basic assumption of the latter systems is that schema description is valuable information that can be used in optimizing storage and queries for XML data.

XML schema descriptions may contain nested elements in many levels, and thus they closely resemble nested collections of elements, rather than flat relational tables. Schema-based approaches based on the relational database technology, such as the Niagara project [14], have had moderate success so far, mostly due to the normalization of the nested XML structures into flat tables, which may require many joins for reconstructing the original XML data. Tree-structured data are more naturally mapped to nested objects than to flat relations, while navigations along XML paths are more easily expressible in term of OODB path expressions than in terms of joins. In addition, the query language of the ODMG standard for object-oriented databases [4], OQL, already provides the functionality for performing very complex operations on XML data, such as string pattern matching, sorting, grouping, aggregation, universal quantification, and random access of XML subelements, which are essential for any realistic XML query language. Thus, OQL can become a full-fledged XML query language with minimal effort. Relational languages, on the other hand, do not allow query nesting at arbitrary points in a query, such as in group-by values and in the query results, which makes the manipulation and construction of XML data very difficult. Consequently, we believe that OODB technology has a better potential than relational technology to become a good basis for storing and handling XML data. Even though there is already work on using OODBs for XML data, such as the work of Christophides, et al [7] and the Lore project [11], there are still many open problems related to query processing and optimization.

Inspired by the Niagara project [14], we believe that there are still many opportunities in using the already established database management technology for storing and handling XML data. Our claim is that schema description is valuable information that can be used in optimizing storage and queries for XML data. For example, if XML data were stored as trees, then each tree node must be allowed to have an undetermined number of children since XML elements may contain multiple subelements. This naturally leads to a nested list implementation of the XML data, which requires multiple deeply nested list scans to retrieve information from deep inside the tree structure. This may lead to bad performance, which can be avoided if we knew exactly how many children each tree node has, since we could have used records instead of lists. This information can be asserted from the schema.

In this paper, we present a new XML query language, called, XML-OQL, which is basically a small set of syntactic extensions to OQL. Our query language resembles current related proposals, such as XQuery [5] and Quilt [6], but is more uniformly integrated with OQL and has precise semantics. Nevertheless, we believe that our framework can be easily applied to other XML query languages as well.

We also present a framework that handles both schema-less (semi-structured) and schema-based XML data uniformly. We describe a schema-less mapping of XML data to OODB objects using a fixed ODMG ODL schema, in the same spirit as the core interface proposed for the XML Document Object Model (DOM) [16]. We then present a method for translating XML queries into ODMG OQL queries over that fixed schema. We identify the problems caused by the lack of schema information and then present a schema-driven translation to address these problems. These schema-guided translations are the focus of this paper, since they have a potential for great performance improvement. We give extensions to the type system of ODL to incorporate XML types and a type system to type-check XML queries. We provide a number of compositional algorithms for mapping XML types to ODL schemas and for translating XML queries into OQL queries. Our type system allows a mixture of typed and untyped XML data, where parts of the XML data may have a fixed schema while others may be schema-less.

Our starting point is the work by Christophides, et al [7] on storing and handling SGML data with OQL, but we go beyond that by supporting a type system well integrated with that of ODMG ODL that supports both semi-structured and schema-based XML data. Furthermore, our query language supports XML data construction in the form of XML tags, a feature now present in most XML query languages. Unlike the related work, we provide precise compositional semantics to our query language. To our knowledge, this is the first attempt to give operational semantics to a non-trivial XML query language in the form of compositional transformation rules. Instead of inventing yet another algebra or calculus for expressing our semantic transformations, the target of our transformations is OQL, which, not only has precise semantics in the form of object algebras and calculi [9, 8], but has also been the focus of various optimization techniques, such as path indexing, path materialization, and query decorellation. These optimizations can now be used to speed up XML queries. Schema information is an indispensable component of our transformations. It is not only used to disambiguate terms with multiple interpretations, such as wildcard tag projections, but is also used in choosing the storage format for the XML data and in generating OQL code guided by the choice of storage. Because of this, XML data are stored in a more compact form and are manipulated more effectively compared to the use of a default schema-less mapping. The produced OQL code is as fast as one would have written by hand. For example, pointer dereferencing through an IDref attribute, which may link different XML documents, is simply mapped to an object reference. Thus, our system offers a high degree of data independence in which storage and access details are decided by the system but are hidden from the user.

Our translation schemes from XML-OQL into plain OQL are compositional, that is, the translation of an XML-OQL expression does not depend on the context in which it is embedded; instead, each subexpression is translated independently, and all translations are composed to form the final OQL query. This property makes the soundness of our translations easy to prove. Even though compositional translations are easy to express and verify on paper, the produced translations may contain many levels of nested queries, which can be overwhelmingly slow if they are interpreted as is. Hence, essential to the success of our framework is an OODB system that can support our translations effectively. We have already built a fully functional, high-performance OODB management system, called lambda-DB [10], as part of our previous research. This system has a sophisticated query optimizer that unnests all nested queries, materializes path expressions into pointer joins, uses a cost-based polynomial-time heuristic for join ordering, and uses a rule-based cost-driven optimizer to generate physical plans. Furthermore, its query evaluation engine is stream-based and supports many evaluation algorithms, including external sorting, block nested loop, indexed nested loop, pointer join, sort-merge join, and group-by. Lambda-DB is a good choice for implementing our framework, because, unlike commercial systems, lambda-DB performs complete query unnesting, which is essential for the performance requirements of the framework.

This paper is organized as follows: Section 2 describes a small number of syntactic extensions to OQL to make it a full-fledged XML query language. Section 3 describes our first attempt in storing and handling XML data using an ODMG database. In this attempt, we do not make use of any schema information; instead, XML data are stored in a tree-like form such that a tree node is an XML element that has a list of children nodes as subelements. We will see that this approach is problematic for many reasons, notably because it is inefficient. The main contribution of this paper is given in Section 4, which describes our schema-driven mapping of data and queries. Finally, Section 5 reports on a prototype implementation of our framework.

2 The XML Object Query Language

Our XML query language is an extension of standard OQL [4]. It captures all the important features found in most recent XML query languages, including XQuery [5] and Quilt [6]. It is not difficult to extend the OQL syntax with special constructs to handle XML data because these data have a tree-like structure that can be naturally mapped to linked objects. In fact there are several proposals for such extensions, such as POQL [7], Ozone [13], Lorel [11], WebOQL [2], and X-OQL [1]. Instead of using one of the proposed languages, we developed our own, called XML-OQL, because its syntax is better suited for our translations than other proposals. Unlike other proposals, XML-OQL was designed to have clear semantics in the form of a well-defined compositional translation into standard OQL (which in turn has clear formal semantics in the form of complex object algebras and calculi [9, 8]). It also supports a decidable type-checking system, well integrated with the type-checking system of OQL.

XML-OQL is essentially OQL extended with XML path expressions and XML data constructions. For example, the following XML-OQL query:

```
select list <bib><author>b.author.lastname </author>,
                <title>b.title</title>,
                <related>select list <title>r.title</title>
                                 from  r in b.@related_to
                </related>
             </bib>
from bs in retrieve("bibliography").bib.vendor.book,
      b in bs
where b.year>1995 and count(b.author)>2
  and  b.title like "% computer %"
```

retrieves information about books written by more than two authors, published after 1995, and containing the word "computer" in their title, along with the titles of their related documents. It conforms to the following partial DTD:

```
<!ELEMENT bib (vendor*)>
<!ELEMENT vendor (name, email, book*)>
<!ATTLIST vendor id ID #REQUIRED>
<!ELEMENT book (title, publisher?, year?, price, author+)>
<!ATTLIST book ISBN ID #REQUIRED>
<!ATTLIST book related_to IDrefs>
<!ELEMENT author (firstname?, lastname)>
```

where the rest of the elements are #PCDATA.

OQL is a very powerful functional query language that allows complex expressions to be composed from simpler ones. By following the OQL philosophy, the XML-OQL syntactic extensions to OQL are allowed to appear at any place an OQL expression is expected, including in complex aggregations, universal quantifications, sorting, and group-bys. In addition, ODL data can be converted to XML data, and vice versa, and both may be mixed together in the same query.

XML data in our framework are either schema-less (semi-structured) data or schema-based data. This distinction is hidden from programmers: if schema-less XML data are assigned a schema and, thus, become schema-based, queries do not need to be changed. The optional schema information is used in catching errors at compile-time rather than run-time, in disambiguating terms with multiple interpretations such as wildcard tag projections, in choosing the storage format for the XML data, and in generating efficient OQL code guided by the choice of storage.

XML elements are constructed using the following syntax in XML-OQL:

<tag $a_1 = u_1 \ldots a_m = u_m$>$e_1, \ldots, e_n$</tag>

for $m, n \geq 0$. This expression constructs an XML element with name, "tag", attributes $a_1, \ldots, a_m$, and subelements $e_1, \ldots, e_n$ for content. Each attribute a_i is bound to the result of the expression u_i. XML data can be accessed directly from a database by name, retrieve("bibliography"), or can be downloaded from a local file or from the web using a URL address, such as document("http://www.acm.org/xml/journals.xml"). The tree structure of XML data can be traversed using path expressions of the form:

$e.A$	projection over the tag name A
$e.\_$	projection over any tag
$e.*$	all subelements of e at any depth
$e.@A$	projection over the attribute A of e
$e[\backslash v \rightarrow e']$	the subelements of e that satisfy e'
$e[e']$	the subelement of e at position e'

where e and e' are XML-OQL expressions, A is a tag or an attribute name, and v is a variable. Filtering is an unambiguous version of XPath's element filtering: For each subelement v of e, it binds the variable v to the subelement and evaluates the predicate e'. The result is all subelements of e that satisfy e'. The scope of variable v is within the expression e' only. Note that $e.A$ is slightly different from the path expression e/A found in languages based on XPath, since $e.A$ strips out the outer tag names $<A>\ldots</A>$ from e. Note also that our syntax allows IDref dereferencing, as is done for r.title in the inner *select*-statement of our example query, since variable r is an IDref that references a book element.

3 Querying Schema-Less XML Data

In our first attempt, we store XML data using a fixed ODL schema without taking into account any type information about the XML data. Then we present a framework for translating XML-OQL queries into standard OQL queries over this fixed schema. In the next section, we will do the same translation guided by schema information, which will result in more efficient queries.

Figure 1 shows a possible ODL schema for storing schema-less XML data. An element projection $e.A$, where e is an expression of type list⟨ XML_element ⟩, can be translated into the OQL expression tag_projection(e, "A") of type list⟨ XML_element ⟩, defined as follows in OQL:

```
define tag_projection
    ( e: list⟨ XML_element ⟩, tag_name: string )
    : list⟨ XML_element ⟩ as
select list y
from x in e,
      y in ( case x.element of
               PCDATA: list(),
               TAG: if x.element.tag.name = tag_name
                      then x.element.tag.content
                      else list()
               end );
```

where the **select list** syntax is an extension to standard OQL that allows the construction of list collections by a select-from-where query, much like the **select distinct** allows the construction of sets rather than bags. The **case** expression is another extension to OQL that allows the decomposition of union values. The above query scans the list of subelements of all elements in e to find those that have the tag name, tag_name. Since there may be several elements with the same tag name, or no elements at all, it returns a list of elements. For non-matching elements, the empty list, list(), is returned.

Projections of the form $e.\_$ are translated into any_projection(e), which is similar to tag_projection but with a true if-then-else condition. Wildcard projections, $e.*$, require a transitive closure, which is not supported directly in OQL but can be simulated with a recursive function. More specifically, $e.*$ is translated into wildcard_projection(e), defined as follows:

```
enum attribute_kind { CDATA, IDref, ID, IDrefs };
enum element_kind { TAG, PCDATA };

union attribute_type switch (attribute_kind)
{   case CDATA:   string            value;
    case IDref:   string            id_ref;
    case IDrefs:  list⟨ string ⟩    id_refs;
    case ID:      string            id;
};
struct attribute_binding
{   string          name;
    attribute_type  value; };
```

```
struct node_type {
    string                       name;
    list⟨ attribute_binding ⟩    attributes;
    list⟨ XML_element ⟩          content;
};
union element_type switch (element_kind)
{   case TAG:       node_type tag;
    case PCDATA:    string    data;
};
class XML_element ( extent Elements )
{   attribute   element_type   element; };
```

Figure 1: The ODL Schema for the Schema-Less Storage of XML Data

```
define wildcard_projection
    ( e: list⟨ XML_element ⟩ ) : list⟨ XML_element ⟩ as
  e + ( select list y
        from x in e,
             y in ( case x.element of
                        PCDATA: list(),
                        TAG: wildcard_projection
                                 (x.element.tag.content)
                    end ) );
```

where + is list concatenation. An attribute projection $e.@A$ is translated into attribute_projection(e, “A”), which has signature:

```
attribute_projection ( e: list⟨ XML_element ⟩,
                       aname: string ) : list⟨ attribute_type ⟩
```

and can be easily defined as an OQL query. Dereferencing an IDref attribute is expensive for schema-less XML data because it requires the scanning of the class extent, Elements, to find the XML element whose ID is equal to the IDref value. It has signature:

```
deref ( idrefs: list⟨ attribute_type ⟩ ) : list⟨ XML_element ⟩
```

Figure 2 presents the translation rules for XML-OQL path expressions. The notation $e \rightarrow e'$ means that e is translated into e', the assertion $e : t$ means that expression e has type t, and a fraction is a conditional assertion. Rule (U4) requires that the type of e is list⟨ XML_element ⟩, to distinguish a regular OQL projection from an XML projection. Rule (U5) enforces an IDref dereferencing if e is the result of an attribute projection (which is indicated by the type of e). Rule (U6) converts a path indexing into a list indexing (to differentiate them, bold-faced square brackets indicate list indexing while regular square brackets indicate path indexing). Finally, Rule (U7) translates a path filtering into a select-list query.

An element construction of the form

<tag $a_1 = u_1 \ldots a_m = u_m$>$e_1, \ldots, e_n$</tag>

allows expressions u_i of type string and expressions e_i of one of the following types: XML_element, list⟨ XML_element ⟩, string, or list⟨ string ⟩. It is translated into the OQL expression:

```
XML_element(element: element_type(TAG: struct(
    name: tag, attributes: list($v_1, \ldots, v_m$),
    content: ($s_1 + s_2 + \cdots + s_n$) )))
```

where v_i is attribute_type(CDATA: u_i) and s_i is the translation of e_i that depends on the type of e_i. That is, if e_i is of type string, then s_i is

```
list(XML_element(element: element_type(PC_DATA: $e_i$)))
```

while, if e_i is of type list⟨ string ⟩, then s_i is

```
select list XML_element(element: element_type(PC_DATA: v))
from v in $e_i$
```

4 Schema-Guided Translation

Our translation scheme for XML-OQL queries against schema-less XML data is problematic for many reasons: First, projections are very expensive since they require nested list scans. If we knew the schema, we could have used records to store the content of elements instead of lists. In that case, XML-OQL projections would have been translated into the more efficient record projections. In addition, programmers do not like to learn the complete details of an XML structure, so they tend to write queries with many wildcard projections. If transitive closures were used to implement wildcard projections, programmers would tend to avoid them due to their high cost, which defies the declarativeness of the language. Second, pointer dereferencing for IDrefs is very expensive in this framework. We would like to have it in constant time, as it is done in OODBs. Finally, the predicate b.year>1995 in the example query is not type-correct because b.year is translated into an OQL expression of type list⟨ XML_element ⟩. This list may be empty if there are no subelements in b with tag name, year, or it may contain multiple elements with the same tag name. This leads to ambiguity. In addition, it is not clear what the result of comparing a

$$e.\_ \;\rightarrow\; \text{any\_projection}(e) \tag{U1}$$

$$e.* \;\rightarrow\; \text{wildcard\_projection}(e) \tag{U2}$$

$$e.@A \;\rightarrow\; \text{attribute\_projection}(e, \text{“}A\text{”}) \tag{U3}$$

$$\frac{e : \text{list}\langle\ \text{XML\_element}\ \rangle}{e.A \;\rightarrow\; \text{tag\_projection}(e, \text{“}A\text{”})} \tag{U4}$$

$$\frac{e : \text{list}\langle\ \text{attribute\_type}\ \rangle,\ a \in \{A, @A, \_, *\}}{e.a \;\rightarrow\; \text{deref}(e).a} \tag{U5}$$

$$\frac{e : \text{list}\langle\ \text{XML\_element}\ \rangle,\ e' : \text{integer}}{e[e'] \;\rightarrow\; \text{list}(e[e'])} \tag{U6}$$

$$e[\backslash v \rightarrow e'] \;\rightarrow\; \textbf{select list } v \textbf{ from } v \textbf{ in } e \textbf{ where } e' \tag{U7}$$

Figure 2: Default Translation of XML-OQL Path Expressions

string with an integer is. The correct predicate should have either been **exists** x **in** b.year: x.data>“1995” or element(b.year).data>“1995”. In fact, under this translation scheme, most predicates that refer to XML data must undergo a similar treatment. This is quite tedious and error-prone. The Lore project [11] addresses this problem by implicitly coercing datatypes and by concatenating the text content of the set-valued attributes, which may not be what the programmer has intended. Other XML query languages, such as Quilt [6], allow this syntax but do not provide semantics. We address these problems in this section by making use of the type information to disambiguate projections and to translate XML-OQL queries into more efficient programs.

4.1 The XML-ODL Type System

In this section, we extend the ODL syntax to handle XML data. Our goal is to integrate XML schema information with ODL types into a single type system, called XML-ODL, and use it to translate XML-OQL queries. XML types are defined in XML-ODL using a special type declaration of the form

typedef XML[t] type_name;

where the syntax of t is described in Figure 3. It defines a new XML type with name type_name. Following this declaration, type_name can appear at any point an ODL type is expected. Furthermore, anonymous XML types of the form XML[t] can also appear at any point an ODL type is expected, but cannot be referenced by other XML types through IDrefs.

Our XML types are regular expressions and are influenced by XDuce [12]. Even though at a first glance they seem different from the proposed XML document description standards, such as DTD and XML Schema, they can capture the most important features of these proposals. Concatenation represents the type of XML element pairs. An alternation is a union type and represents a choice of two types. The optionality operator can be defined in terms of alternation ($t? = t\,|\,()$) but is treated separately. The combination of a labeled type with a type with attributes defines the type of an XML element with attributes. We decided to separate attributes from labeled types to make the semantics easier to express. Concatenation and alternation are left-associative and concatenation has identity, (), that is, t, () = (), t = t. Similarly, repetition and optionality have identity, (), that is, $()^* = ()? = ()$. There are other normalization rules to simplify types, such as $t^{**} = t^*$, $t\,|\,t = t$, and $(t^*, t^*) = t^*$, which are not used. The type s_i of an attribute can be a *path* (if it is a single IDref) or *path** (if it contains multiple IDrefs). A *path* can precisely specify the location of the ID attribute that the IDref refers to. If it points within a different XML type, *path* starts with the XML type name. Otherwise, if the type name is missing, it is assumed to be the name of the current XML type. Since one of our goals is to integrate schema-less and schema-based XML data in the same framework, the attribute type s_i is also allowed to point to schema-less elements (when s_i is equal to *any*).

For example, the DTD given in Section 3 can be captured by the following XML-ODL type:

```
typedef XML[bib[vendor[ { id: ID }
   ( name[string], email[string],
     book[ { ISBN: ID, related_to: bib.vendor.book.ISBN* }
          ( title[string], publisher[string]?,
            year[integer]?, price[integer],
            author[firstname[string]?,lastname[string]],
            author[firstname[string]?,lastname[string]]* )
          ]* )
   ]* ]] bibliography;
```

This defines a new XML type, called bibliography. The attribute related_to is a list of IDrefs, which are ISBNs of other books. Notice that author elements appear twice in a book; the first author element is the first book author while the second author element is the list of coauthors. This is a standard way of expressing a *type*$^+$ in terms of *type**. This will also serve as an example when we describe a method to disambiguate tag projections. Finally, we will use the class biblio with extent bibliographies for bibliography objects that contains an attribute entry of type bibliography.

The XML-OQL query given is Section 3 has the following type in our type system:

```
list⟨ XML[bib[author[string,string*],title[string],
            related[title[string]*]]] ⟩
```

that is, it returns a list of XML objects.

$$
\begin{array}{lcll}
t & ::= & \text{any} & \text{an arbitrary XML element} \\
 & | & () & \text{identity} \\
 & | & v[t] & \text{labeled (tagged) type} \\
 & | & \{v_1 : s_1, \ldots, v_n : s_n\}\, t & \text{a type with attributes} \\
 & | & t_1,\, t_2 & \text{concatenation} \\
 & | & t_1 \,|\, t_2 & \text{alternation} \\
 & | & t^* & \text{repetition} \\
 & | & t? & \text{optionality} \\
 & | & \text{primitive\_type} & \text{integer, string, image, etc}
\end{array}
\qquad
\begin{array}{lcl}
s & ::= & \text{ID} \\
 & | & \text{primitive\_type} \\
 & | & \text{any} \\
 & | & \mathit{path} \\
 & | & \mathit{path}^* \\
\mathit{path} & ::= & \text{XML\_type\_name} \\
 & | & v \\
 & | & \mathit{path}.v
\end{array}
$$

Figure 3: Syntax of the XML Types (where t, t_1, t_2 are XML types, $v, v_1, \ldots, v_n$ are names, and $s, s_1, \ldots, s_n$ are attribute types)

$$
\begin{array}{rcll}
\mathcal{T}(\llbracket \text{any} \rrbracket, \rho) & = & \text{list}\langle\ \text{XML\_element}\ \rangle & \text{(T1)} \\
\mathcal{T}(\llbracket () \rrbracket, \rho) & = & \text{struct } \{\ \} & \text{(T2)} \\
\mathcal{T}(\llbracket v[t] \rrbracket, \rho) & = & \mathcal{T}(\llbracket t \rrbracket, \rho.v) & \text{(T3)} \\
\begin{array}{l} \mathcal{T}(\llbracket \{v_1 : s_1, \ldots, v_n : s_n\}\, t \rrbracket, \rho) \\ \text{where } s_k\text{=ID} \end{array} & = & \left\{ \begin{array}{l} \text{a reference to a new class } C\text{:} \\ \quad \text{class } C \text{ (extent \_}C \text{ key } v_k\text{) } \{ \\ \qquad \text{attribute } \mathcal{T}(\llbracket t \rrbracket, \rho) \text{ info;} \\ \qquad \text{attribute } \mathcal{S}(\llbracket s_1 \rrbracket)\ v_1\text{; ...; attribute } \mathcal{S}(\llbracket s_n \rrbracket)\ v_n\text{; \};} \\ \text{bind path } \rho.v_k \text{ to } C \text{ in } \sigma \\ \text{and path } \rho.v_k \text{ to } \{v_1 : s_1, \ldots, v_n : s_n\}\, t \text{ in } \delta \end{array} \right. & \text{(T4)} \\
\mathcal{T}(\llbracket \{v_1 : s_1, \ldots, v_n : s_n\}\, t \rrbracket, \rho) & = & \text{struct } \{\ \mathcal{T}(\llbracket t \rrbracket, \rho) \text{ info; } \mathcal{S}(\llbracket s_1 \rrbracket)\ v_1\text{; ...; } \mathcal{S}(\llbracket s_n \rrbracket)\ v_n\text{; } \} & \text{(T5)} \\
\mathcal{T}(\llbracket t_1,\, t_2 \rrbracket, \rho) & = & \text{struct } \{\ \mathcal{T}(\llbracket t_1 \rrbracket, \rho) \text{ fst; } \mathcal{T}(\llbracket t_2 \rrbracket, \rho) \text{ snd; } \} & \text{(T6)} \\
\mathcal{T}(\llbracket t_1 \,|\, t_2 \rrbracket, \rho) & = & \text{struct } \{\ \text{union\_kind tag; } \mathcal{T}(\llbracket t_1 \rrbracket, \rho) \text{ left; } \mathcal{T}(\llbracket t_2 \rrbracket, \rho) \text{ right; } \} & \text{(T7)} \\
\mathcal{T}(\llbracket t^* \rrbracket, \rho) & = & \text{list}\langle\ \mathcal{T}(\llbracket t \rrbracket, \rho)\ \rangle & \text{(T8)} \\
\mathcal{T}(\llbracket t? \rrbracket, \rho) & = & \mathcal{T}(\llbracket t \rrbracket, \rho) & \text{(T9)} \\
\mathcal{T}(\llbracket \text{primitive\_type} \rrbracket, \rho) & = & \text{primitive\_type} & \text{(T10)} \\
\hline
\mathcal{S}(\llbracket \text{primitive\_type} \rrbracket) & = & \text{primitive\_type} & \text{(T11)} \\
\mathcal{S}(\llbracket \mathit{path} \rrbracket) & = & \sigma[\mathit{path}] & \text{(T12)} \\
\mathcal{S}(\llbracket \mathit{path} \rrbracket^*) & = & \text{list}\langle\ \sigma[\mathit{path}]\ \rangle & \text{(T13)} \\
\mathcal{S}(\llbracket s \rrbracket) & = & \text{string} \qquad \mathit{otherwise} & \text{(T14)}
\end{array}
$$

Figure 4: Mapping XML-ODL to ODL

```
class C2 (extent _C2 key id) {
   attribute struct{ struct{ string fst; string snd; } fst;        // vendor's name and email
                     list⟨ C1 ⟩ snd; } info;                        // books by this vendor
   attribute string id; };
class C1 (key ISBN) {
   attribute struct{ struct{ struct{ struct{ struct{ string fst; string snd; } fst;     // title and publisher
                                             integer snd; } fst;                        // year
                                     integer snd; } fst;                                // price
                             struct{ string fst; string snd; } snd; } fst;              // first author
                     list⟨ struct{ string fst; string snd; } ⟩ snd; } info;             // coauthors
   attribute string ISBN;
   attribute list⟨ C1 ⟩ related_to; };
```

Figure 5: The ODL type of the Bibliography Schema

$$\mathcal{R}(\llbracket \{\ldots, A : s_k, \ldots\}\, t \rrbracket, x, e.@A) = x.A \tag{R1}$$

$$\mathcal{R}(\llbracket A[t] \rrbracket, x, e.A) = x \tag{R2}$$

$$\mathcal{R}(\llbracket A[t] \rrbracket, x, e.\_) = x \tag{R3}$$

$$\mathcal{R}(\llbracket A[t] \rrbracket, x, e.*) = \begin{cases} x & \text{if } \mathcal{R}(\llbracket t \rrbracket, x, e.*) = \text{struct()} \\ \text{struct(fst: } x\text{, snd: } \mathcal{R}(\llbracket t \rrbracket, x, e.*)) & \textit{otherwise} \end{cases} \tag{R4}$$

$$\mathcal{R}(\llbracket t_1,\, t_2 \rrbracket, x, e.a) = \begin{cases} \mathcal{R}(\llbracket t_1 \rrbracket, x, e.a) & \text{if } \mathcal{R}(\llbracket t_2 \rrbracket, x, e.a) = \text{struct()} \\ \mathcal{R}(\llbracket t_2 \rrbracket, x, e.a) & \text{if } \mathcal{R}(\llbracket t_1 \rrbracket, x, e.a) = \text{struct()} \\ \text{struct(fst: } \mathcal{R}(\llbracket t_1 \rrbracket, x.\text{fst}, e.a), & \\ \qquad \text{snd: } \mathcal{R}(\llbracket t_2 \rrbracket, x.\text{snd}, e.a)) & \textit{otherwise} \end{cases} \tag{R5}$$

$$\mathcal{R}(\llbracket t_1 \,|\, t_2 \rrbracket, x, e.a) = \begin{cases} \text{if } x.\text{tag=INL} \\ \quad \text{then struct(tag: INL, left: } \mathcal{R}(\llbracket t_1 \rrbracket, x.\text{left}, e.a)\text{, right: NULL)} \\ \quad \text{else struct(tag: INR, left: NULL, right: } \mathcal{R}(\llbracket t_2 \rrbracket, x.\text{right}, e.a)) \end{cases} \tag{R6}$$

$$\mathcal{R}(\llbracket t^* \rrbracket, x, e.a) = \begin{cases} \text{struct()} & \text{if } \mathcal{R}(\llbracket t \rrbracket, x, e.a) = \text{struct()} \\ \textbf{select list } \mathcal{R}(\llbracket t \rrbracket, v, e.a) \textbf{ from } v \textbf{ in } x & \textit{otherwise} \end{cases} \tag{R7}$$

$$\mathcal{R}(\llbracket t? \rrbracket, x, e.a) = \text{if } x = \text{NULL then NULL else } \mathcal{R}(\llbracket t \rrbracket, x, e.a) \tag{R8}$$

$$\mathcal{R}(\llbracket \text{any} \rrbracket, x, e) = \text{handled in Section 2 by Rules (U1) through (U7)} \tag{R9}$$

$$\mathcal{R}(\llbracket t^* \rrbracket, x, e[e']) = x[e'] \tag{R10}$$

$$\mathcal{R}(\llbracket t^* \rrbracket, x, e[\backslash v \rightarrow e']) = \textbf{select list } \mathcal{R}(\llbracket t \rrbracket, v, e) \textbf{ from } v \textbf{ in } x \textbf{ where } e' \tag{R11}$$

$$\mathcal{R}(\llbracket t \rrbracket, x, e) = \text{struct()} \qquad \textit{otherwise} \tag{R12}$$

Figure 6: Translation of an XML-OQL Path Expression into OQL ($a \in \{A, @A, \_, *\}$)

4.2 Mapping XML-ODL to ODL

The rules for mapping XML types to ODL are given in Figure 4. More specifically, XML[t] is mapped to $\mathcal{T}(\llbracket t \rrbracket, \text{type\_name})$, where type_name is the name of the current XML type. The semantic brackets, $\llbracket\,\rrbracket$, are used in denotational semantics to separate syntactic structures from semantic operations. The extra parameter ρ in $\mathcal{T}(\llbracket t \rrbracket, \rho)$ is a path expression similar to the IDref path in the attribute types of Figure 3. Rules (T4) and (T5) map types with attributes. If a type has an attribute of type, ID, then a new class, C, is declared and the type is mapped to the class name (Rule (T4)); otherwise, it is mapped to a simple ODL struct (Rule (T5)). The environment lists σ and δ contain information about the generated classes. The environment σ binds the path expression of the ID attribute to the class name, while the environment δ binds the same path to the XML type itself. The environment σ is used in translating IDrefs into class references in Rules (T13) and (T14). To avoid ambiguity, no two classes are allowed to be assigned to the same path. Alternation is implemented as a struct in Rule (T7) because ODL does not support polymorphic union types and, therefore, union values cannot be constructed on the fly. The union tag is of type enum {INL,INR} and is used in choosing between the left and the right components of the struct. Optionality is simply mapped to a type that allows null values, which applies to every ODL type.

For example, the ODL type of the bibliography schema is list⟨ C2 ⟩, given in Figure 5, where the class C2 captures the attributes and content of a vendor (it is a class because the vendor type has an ID attribute), and class C1 captures the element bib.

4.3 Translating XML-OQL into OQL

The OQL translation of an XML-OQL projection $e.a$, where $a \in \{A, @A, \_, *\}$, is $\mathcal{R}(\llbracket t \rrbracket, x, e.a)$, is given in Figure 6. The OQL expression x in $\mathcal{R}(\llbracket t \rrbracket, x, e.a)$ is the implementation of e, while t is the type of e.

As an example of an XML-OQL translation, biblio.entry.bib is translated into the OQL expression biblio.entry. Similarly, biblio.entry.bib.vendor is translated into **select list** v **from** v **in** biblio.entry and biblio.entry.bib.vendor.book is translated into:

```
select list (select list u from u in w.info.snd)
from w in (select list v from v in biblio.entry)
```

which, after basic query unnesting, is equivalent to:

```
select list (select list u from u in v.info.snd)
from v in biblio.entry
```

Thus, the line "bs **in** biblio.entry.bib.vendor.book, b **in** bs" in the example query is translated into:

```
bs in ( select list (select list u from u in v.info.snd)
        from v in biblio.entry ), b in bs
```

which, after basic query unnesting, is equivalent to v **in** biblio.entry, b **in** v.info.snd. The translation of b.author.lastname in the same query is interesting because the tag name author appears twice in the book b. Rule (R10) applies to all the subelements of the book b except the two author subelements. The last name of the first author is retrieved from b.info.fst.snd.snd while the last names of the coauthors are retrieved by **select list** c.snd **from** c **in** b.info.snd. Consequently, b.author.lastname is translated into:

```
struct( fst: b.info.fst.snd.snd,
        snd: ( select list c.snd from c in b.info.snd ) )
```

XML-OQL allows XML element constructions of the form:

$$<\text{tag } a_1 = u_1 \ldots a_m = u_m>e_1, \ldots, e_n</\text{tag}>$$

The type T_i of the expression e_i is allowed to be one of the following types:

$$\begin{array}{lll} T & ::= & \text{XML}[t] \\ & | & \text{struct}\{\ T_1\ \text{fst};\ T_2\ \text{snd};\ \} \\ & | & \text{list}\langle\ T\ \rangle \\ & | & \text{list}\langle\ \text{XML\_element}\ \rangle \\ & | & \text{primitive\_type} \end{array}$$

which can be unambiguously converted to an XML$[t_i]$ type for e_i. In addition, each u_i must be of a primitive type, a class reference, or a list of class references, where the class name in the two latter cases must be a member of the environment, σ. This also unambiguously generates a type t'_i for u_i. Thus the type of the above element construction is:

$$\text{XML}[\{a_1 : t'_1, \ldots, a_m : t'_m\}\,(t_1, \ldots, t_n)]$$

The translation of an XML-OQL construction to OQL follows the type translation and is straightforward. Therefore, an XML construction without attributes, $<\text{tag}>e_1, \ldots, e_n</\text{tag}>$, is translated into:

$$\text{struct(fst: ..., struct(fst: } e_1\text{, snd: } e_2\text{) ..., snd: } e_n\text{)}$$

while a construction with attributes of the form $<\text{tag } a_1 = u_1 \ldots a_m = u_m>e_1, \ldots, e_n</\text{tag}>$, is first translated into a construction without attributes:

$$\begin{array}{l} \text{struct(info: } <\text{tag}>e_1, \ldots, e_n</\text{tag}>, \\ \qquad a_1\text{: } u_1\text{, ..., } a_m\text{: } u_m\text{)} \end{array}$$

Using the translation rules for XML-OQL projections and constructions, the example XML-OQL query is translated into the following OQL query:

```
select list struct( fst: struct( fst: struct( fst: b.info.fst.snd.snd,
              snd: ( select list c.snd from c in b.info.snd ) ),
              snd: b.info.fst.fst.fst.fst.fst ),
    snd: ( select list r.info.fst.fst.fst.fst.fst
                from r in b.related_to ) )
from biblio in bibliographies,
      v in biblio.entry, u in v.info.snd
where b.info.fst.fst.fst.snd>1995
   and count(b.info.fst.snd)≥2
   and b.info.fst.fst.fst.fst.fst like "% computer %"
```

It does not really matter that our translations contain long chains of projections since record projections are converted into address offsets at compile time. Note also that there is an implicit pointer dereferencing in the path r.info.fst.fst.fst.fst.fst since r is a reference to the class C1.

5 Implementation

We have already implemented the schema-less translation from XML-OQL to OQL using the lambda-DB object-oriented database management system [10], which is built on top of the SHORE object storage system [3]. This implementation is now part of the latest release (1.6) of lambda-DB and is available at `http://lambda.uta.edu/lambda-DB.html`. The schema-guided translations will require more substantial changes to the underlying OODB engine, especially to the type-checker and the query translator. We believe that we will gain a significant performance improvement when we implement our schema-driven framework. To support this claim, we compiled the bibliography XML-ODL schema into a regular ODL schema by hand. Figure 7.(a) indicates that the schema-less data take about six times more space than the schema-based data. Then we compared the performance of the following queries for various database sizes:

```
Q1: select <book>x.author,<title>x.title</title></book>
      from x in retrieve("books").vendor.books.book
Q2: select a.firstname, a.lastname, b.title
      from v in Vendors, b in v.books, a in b.authors
```

Query Q1 is in XML-OQL and is over the schema-less database while query Q2 is in OQL and is over the schema-based database. The two queries have been tested for various random databases, containing between 100 and 2000 books. The platform used for these experiments was a Pentium III/800MHz/256MBs, running RedHat Linux 7.0. The SHORE client buffer was set to 1MB. We can see that a schema-less query takes about four times more time than that of the schema-based query. We expect more substantial performance gain for complex XML-OQL queries that contain many path expressions, multiple documents, and IDref dereferencing.

The above preliminary results are encouraging but are far from complete. In the near future, we are planning to compare the performance of our schema-based implementations with that of our schema-less implementations for a wider spectrum of queries.

6 Conclusion

We have presented a framework for storing XML data into an OODB management system and for translating XML queries into OQL queries based on XML schema information. While schema-less XML data may benefit from a semi-structure model and algebra, if a schema is given, XML data can be naturally mapped to the nested storage structures of an OODB system. Using schema information, the type-checker can resolve ambiguities and fill-out missing details in a query, even in the presence of complex patterns in the path expressions. This is accomplished at compile-time, rather than evaluation-time, and resembles the

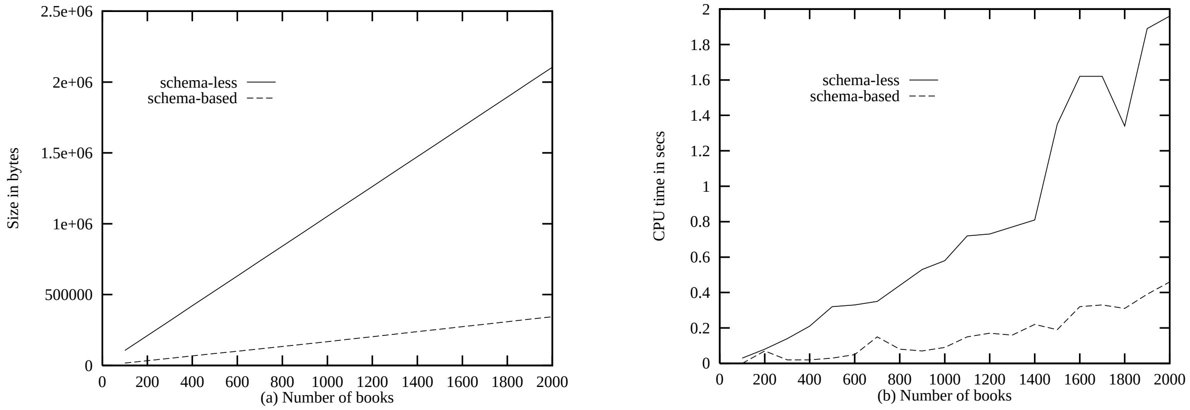

Figure 7: Comparing Schema-Less with Schema-Based Implementations

pattern decomposition techniques in functional programming languages. Our framework does not require any fundamental change to the query optimizer and evaluator of an OODB system, since XML queries are translated into OQL code after type-checking but before optimization.

Acknowledgments: This work is supported in part by the National Science Foundation under the grant IIS-9811525 and by the Texas Higher Education Advanced Research Program grant 003656-0043-1999.

References

[1] S. Abiteboul, *et al.* XML Repository and Active Views Demonstration. In *VLDB'99*, pages 742–745, 1999.

[2] G. Arocena and A. Mendelzon. WebOQL: Restructuring Documents, Databases, and Webs. In *ICDE'98*, pages 24–33, Feb. 1998.

[3] M. Carey, *et al.* Shoring Up Persistent Applications. In *SIGMOD'94*, pages 383–394, May 1994.

[4] R. Cattell, editor. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann, 2000.

[5] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery: A Query Language for XML . W3C Working Draft. Available at `http://www.w3.org/TR/xquery/`, 2000.

[6] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. *Workshop on the Web and Databases (WebDB'00), Dallas, Texas*, pages 53–62, May 2000.

[7] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From Structured Documents to Novel Query Facilities. In *SIGMOD'94*, pages 313–324, May 1994.

[8] S. Cluet and C. Delobel. A General Framework for the Optimization of Object-Oriented Queries. In *SIGMOD'92*, pages 383–392, June 1992.

[9] L. Fegaras and D. Maier. Optimizing Object Queries Using an Effective Calculus. *ACM Transactions on Database Systems*, Dec. 2000.

[10] L. Fegaras, C. Srinivasan, A. Rajendran, and D. Maier. λ-DB: An ODMG-Based Object-Oriented DBMS. In *SIGMOD'2000*, page 583, May 2000.

[11] R. Goldman, J. McHugh, and J. Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. *Workshop on the Web and Databases (WebDB'99), Philadelphia, Pennsylvania*, pages 25–30, June 1999.

[12] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular Expression Types for XML. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'00), Montreal, Canada.* ACM Press, Sept. 2000.

[13] T. Lahiri, S. Abiteboul, and J. Widom. Ozone: Integrating Structured and Semistructured Data . In *7th International Workshop on Database Programming Languages, DBPL'99, Kinloch Rannoch, Scotland, UK*, LNCS volume 1949, pages 297–323. Springer, Sept. 1999.

[14] J. Shanmugasundaram, *et al.* Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB'99*, pages 302–314, 1999.

[15] J. Widom. Data Management for XML: Research Directions. *IEEE Data Engineering Bulletin, Special Issue on XML*, 22(3):44–52, Sept. 1999.

[16] World Wide Web Consortium (W3C). *Extensible Markup Language (XML).* http://www.w3.org/XML/.