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. Since XML data are often scattered throughout the web as text files in sometimes unknown locations, we address the problem of querying the entire internet to answer a query by presenting an indexing technique based on both structure and content information of the XML documents, and algorithms for locating relevant documents that match the query with high degree of precision.

We are first presenting 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. We are then presenting a framework for translating XML queries into OQL queries based on XML schema information. Our translation schemes are purely compositional. 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.

The second problem addressed by this paper is precise indexing of web-accessible XML data. We are presenting an inverse indexing technique that indexes text words and element tags in an XML document by taking into account the element structure in the document. Given an XML-OQL query, our system is capable of constructing an OQL query against the inverse index structures, which retrieves the locations of all web-accessible XML documents that satisfy the containment constraints of the XML-OQL query. This query construction is compositional, that is, each XML-OQL syntactic construct is translated independently and all translations are combined in steps.

1 Motivation

XML [31] 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 [29] 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 [30]. 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 [31]. 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, schema information allows more effective data storage and access paths, and more efficient query evaluation.

Historically, computer languages, ranging from general-purpose to domain-specific, have evolved towards more strict type disciplines, starting from untyped languages, such as Lisp, followed by languages with rigid type systems, such as Pascal and C, and currently ending in functional languages with polymorphic type inference engines, such as Haskell and SML. Even though strong typing reduces the expressiveness of a language and demands extensive type annotations, it also enforces a programming discipline crucial in software engineering, allows the detection of many programming errors and flaws at compile time, and promotes program optimizations. Strong typing does not necessarily imply inflexibility in capturing the data diversity found on the web today. Modern computer languages address data heterogeneity by supporting union data types and parametric polymorphism. Long-time advocates of strong typing in programming languages find encouraging that the XML community is starting to appreciate the importance of strong typing disciplines for their proposed data models and languages. While the majority of XML data currently available on the web does not come with a predefined schema, we hope that this will change in the near future, as it has changed for programming languages in the past. Nevertheless, any research centered around XML must handle both schema-less and schema-based XML data in the same framework.

Although type information is valuable to end-users in writing valid XML queries, it should not affect the way XML queries are expressed. Instead, type information may be used by XML systems in validating queries at compile-time, resolving ambiguities, and choosing effective storage indexing techniques and efficient retrieval algorithms for XML data.

Research on semi-structured databases [4] started at about the same time as, and independently from the development of XML. The focus of that research is query languages and algebras for schema-less, tree-like database and web page structures. Not very long after XML was introduced, researchers in that area realized that the semi-structured model can be adapted to handle XML data. Indeed, there is an increasing research activity recently in that direction. The implicit assumption of the semi-structured database research is that traditional database models are too rigidly structured to capture the heterogeneity and the irregularities intrinsic to web and XML data, thus a radical new data model and algebra are needed that go beyond traditional models [14]. In fact, there are already several algebras for semi-structured data, including an algebra based on structural recursion [4], YATL [12, 10], SAL [3], and x-algebra [19].

On the other hand, much research effort has already been devoted to query algebras for complex data types [16, 11, 24], which have been used successfully as a formal basis for various optimization techniques for object-oriented and object-relational databases, such as path indexing [23] and query decorellation [13]. However, very few of these algebras can handle tree data structures, such as those implicit to schema-less XML data. But if an XML schema is provided, then XML data can be easily mapped to complex data, while XML queries can be translated into algebraic forms in one of these complex data algebras.

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 [28]. 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 [28], 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 [7], 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 the OODB technology has a better potential than the 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 [9] and the Lore project [21], there are still many open problems related to query processing and optimization.

The most important criticism against the fixed-schema approach is that, in practice, the source of XML data is often not part of some database. Rather XML data are scattered throughout the web as text files in sometimes unknown locations. Requiring that all XML data be stored in databases is, of course, unrealistic. Users familiar with web search engines would like to write queries such as:

```
select b.*.title
  from b in download("*").*.bib.*.book
 where b.*.author.*.lastname = "Smith"
```

to query the entire web for all the XML files that contain book references to find all the books written by Smith. They do not really care about DTDs and Schemas. They are willing to put as many wildcards in the query as necessary to narrow-down/expand the search space of the query to get relevant answers. To them, an XML query is an effective means of adding both structure and content requirements to their web search, which will hopefully result to a more selective and precise search. The challenge here is to retrieve only those documents from the web that, not only contain all the paths appearing in the query, but also have a high probability of successfully satisfying the query condition. Current document indexing techniques, such as inverted files, are based on content only. But these techniques may not be very effective for indexing XML data, since structure is an essential component of XML data. More specifically, evaluating the above query is retrieving all the relevant XML files from the web. To be more precise, a relevant XML file should match all the paths in this query, that is, it should match the paths *.bib.*.book.*.title and *.bib.*.book.*.author.*.lastname, and it should contain the requested partial content information, that is, it should contain at least one book authored by Smith. In addition to relevance, if the query contains approximate matching, the query results should be prioritized, as is done by current

web search engines. Unlike web search engines though, all relevant documents must be downloaded and queried, rather than just be returned as links to documents, which raises some interesting issues. Since some links may be invalid, query evaluation must proceed in parallel for multiple links. Furthermore, a good caching policy of XML data in the database may improve system performance considerably.

2 Our Approach

Inspired by the Niagara project [28], 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.

We are presenting 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 Quilt [8], 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 are then presenting a framework to handle both schema-less (semi-structured) and schema-based XML data uniformly. We first 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) [31]. We are then presenting 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 we 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 [9] 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 [16, 11], 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. In that way, XML data are stored in a more compact form and are manipulated more effectively than by using 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.

The second problem addressed by this is precise indexing of web-accessible XML data. We present an inverse indexing technique that indexes content words and element tags in XML documents by taking into account the element structure of the documents. Inspired by the "text-in-context" XML search engine of the Niagara project [27], we present a framework for indexing XML data that is more precise, easier to implement, and more efficient than that of Niagara. Like Niagara, our framework uses two inverted lists, one to index content words and another to index XML tags for all web-accessible XML documents. Unlike Niagara, though, in addition to word location and to tag ranges, the word/tag nesting level is stored, which may result to a more accurate indexing. The nesting level has been used successfully by some recent containment query processing engines [32]. The main contribution of our method is related to the way we search these indexes to retrieve relevant documents. Related approaches retrieve one inverted list for each tag name or word that appears in a path expression and then perform a kind of intersection between these lists that takes into account the containment restrictions of the path (derived from the tag order in the path). The order of intersections is important and poor choices may lead to performance degradation. For instance, there may be many XML documents with the same top-level tag. Hence, starting the search from that tag may result in the creation of very large intermediate inverted lists. Niagara uses a special indexing algebra, SEQL, but it is yet to be seen if there are effective optimization rules for this algebra. Our approach is novel and simple: we let the OODB query optimizer decide the best way to use the indexes based on statistical information. We present a set of rules that transform an XML-OQL query into an OQL query against the two indexes. The result of the index query is a list of relevant documents that satisfy the original XML-OQL query

in terms of both structure and content. In a way, our method is a non-standard interpretation of XML-OQL queries and is reminiscent to querying the system catalog during query translation. It requires minimal extensions to current OODB optimizers.

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 [17], as part of a previous work. 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 3 describes a small number of syntactic extensions to OQL to make it a full-fledged XML query language. Section 4 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 5, which describes our schema-driven mapping of data and queries. Section 6 discusses various practical problems that need to be addressed before our framework is used in a real web application. Finally, Section 7 reports on a prototype implementation of our framework and Section 8 compares our approach with related work.

3 The XML Object Query Language

Our XML query language is an extension of standard OQL. It captures all the important features found in most recent XML query languages. Unsurprisingly, 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 [9], Lorel [21], 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 query

translation than the 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 [16, 11]). It also supports a decidable type-checking system, well integrated with the type-checking system of OQL. Clear semantics is the foundation of query optimization frameworks. Correctness of program transformations can only be established with clear and well-formed semantics while efficiency can be demonstrated with well designed performance measurements.

Before we present the syntax of XML-OQL, we illustrate the basic features of the language using a simple query on XML data conforming 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)>
<!ELEMENT name (#PCDATA)>...
```

(The rest of the elements are #PCDATA.) Conforming to this schema, the following query 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:
EXAMPLE 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 document("bibliography").bib.vendor.book,
      b in bs
where b.year>1995 and count(b.author)>2 and b.title like "% computer %"
```

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

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

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 the database using the name of the document, document("bibliography"), or can be downloaded from the web using an URL address, such as download("http://www.acm.org/xml/index.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
$e.*$	any labeled element of e at any depth (wildcard)
$e.@A$	projection over the attribute A of e
$e[e']$	the subelement of e with ordinal number e' (indexing)
$e[\backslash v \Rightarrow e']$	the subelements of e that satisfy the predicate e' (filtering)

where e and e' are XML-OQL expressions, A is a tag or an attribute name, and v is a variable. All but the last form can be found in most XML query languages, including XPath [31]. The last form is an unambiguous version of XPath's element filtering: For each subelement v of e, it binds the variable v to this 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. Without the explicit use of a variable, the semantics of filtering would have been ambiguous if arbitrary nesting of expressions were allowed. For example, `book[title=@related_to[year>1999].title]` is ambiguous, while `book[\x⇒x.title=x.@related_to[\y⇒y.year>1999].title]` is not.

Note that this 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. Note also that 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 verse, and may mixed together in the same query.

4 Querying Schema-Less (Semi-Structured) 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 to 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 );
```

```
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

where the **select list** syntax is an extension to standard OQL that allows the construction of list collections from 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.

Projections of the form $e._$ are translated into any_projection(e), which is similar to tag_projection but with a true condition in the if-then-else expression. 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:

```
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 ⟩, attr_name: 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

$$e._ \;\rightarrow\; \text{any_projection}(e) \qquad \text{(U1)}$$

$$e.* \;\rightarrow\; \text{wildcard_projection}(e) \qquad \text{(U2)}$$

$$e.@A \;\rightarrow\; \text{attribute_projection}(e, \text{“}A\text{”}) \qquad \text{(U3)}$$

$$\frac{e : \text{list}\langle\ \text{XML_element}\ \rangle}{e.A \;\rightarrow\; \text{tag_projection}(e, \text{“}A\text{”})} \qquad \text{(U4)}$$

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

$$\frac{e : \text{list}\langle\ \text{XML_element}\ \rangle,\ e' : \text{integer}}{e[e'] \;\rightarrow\; \text{list}(e\mathbf{[}e'\mathbf{]})} \qquad \text{(U6)}$$

$$e[\backslash v \Rightarrow e'] \;\rightarrow\; \textbf{select list } v \textbf{ from } v \textbf{ in } e \textbf{ where } e' \qquad \text{(U7)}$$

Figure 2: Default Translation of XML-OQL Path Expressions

fraction is a conditional assertion. Rule (U4) requires that the type of e be 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 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:

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

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.

5 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 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 [21] 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 [8], 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.

5.1 The XML-ODL Type System

In this section, we extend the syntax of ODL 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 [22]. 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

t	::=	any	an arbitrary XML element
	|	()	identity
	|	$v[t]$	labeled (tagged) type
	|	$\{v_1 : s_1, \ldots, v_n : s_n\}\, t$	a type with attributes
	|	$t_1,\, t_2$	concatenation
	|	$t_1 \,\vert\, t_2$	alternation
	|	t^*	repetition
	|	$t?$	optionality
	|	primitive_type	integer, string, image, etc

s	::=	ID
	|	primitive_type
	|	any
	|	*path*
	|	*path**
path	::=	XML_type_name
	|	v
	|	*path.v*

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)

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 4 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;
```

It 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 following class for bibliography objects in our examples:

```
class biblio ( extent bibliographies ) { attribute bibliography entry; };
```

Note that, in most real-world applications, XML schemas are not predefined. Instead, XML data are downloaded along with their schemas (e.g., in DTD form) from one or more web sites and queries are executed over these data on the fly. We address this problem carefully in Section 6. However, it is assumed here here that XML schemas are predefined.

The XML-OQL query given is Section 4 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}{rcll}
\mathcal{T}([\![\text{any}]\!],\rho) & = & \text{list}\langle\ \text{XML_element}\ \rangle & \text{(T1)}\\
\mathcal{T}([\![()]\!],\rho) & = & \text{struct \{ \}} & \text{(T2)}\\
\mathcal{T}([\![v[t]]\!],\rho) & = & \mathcal{T}([\![t]\!],\rho.v) & \text{(T3)}\\
\begin{array}{l}\mathcal{T}([\![\{v_1:s_1,\ldots,v_n:s_n\}\,t]\!],\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}([\![t]\!],\rho) \text{ info;}\\ \qquad \text{attribute } \mathcal{S}([\![s_1]\!])\ v_1\text{; ...; attribute } \mathcal{S}([\![s_n]\!])\ 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}([\![\{v_1:s_1,\ldots,v_n:s_n\}\,t]\!],\rho) & = & \text{struct \{ } \mathcal{T}([\![t]\!],\rho) \text{ info; } \mathcal{S}([\![s_1]\!])\ v_1\text{; ...; } \mathcal{S}([\![s_n]\!])\ v_n\text{; \}} & \text{(T5)}\\
\mathcal{T}([\![t_1,\,t_2]\!],\rho) & = & \text{struct \{ } \mathcal{T}([\![t_1]\!],\rho) \text{ fst; } \mathcal{T}([\![t_2]\!],\rho) \text{ snd; \}} & \text{(T6)}\\
\mathcal{T}([\![t_1\,|\,t_2]\!],\rho) & = & \text{struct \{ union_kind tag; } \mathcal{T}([\![t_1]\!],\rho) \text{ left; } \mathcal{T}([\![t_2]\!],\rho) \text{ right; \}} & \text{(T7)}\\
\mathcal{T}([\![t^*]\!],\rho) & = & \text{list}\langle\ \mathcal{T}([\![t]\!],\rho)\ \rangle & \text{(T8)}\\
\mathcal{T}([\![t?]\!],\rho) & = & \mathcal{T}([\![t]\!],\rho) & \text{(T9)}\\
\mathcal{T}([\![\text{primitive_type}]\!],\rho) & = & \text{primitive_type} & \text{(T10)}\\
\hline
\mathcal{S}([\![\text{primitive_type}]\!]) & = & \text{primitive_type} & \text{(T11)}\\
\mathcal{S}([\![\mathit{path}]\!]) & = & \sigma[\mathit{path}] & \text{(T12)}\\
\mathcal{S}([\![\mathit{path}]\!]^*) & = & \text{list}\langle\ \sigma[\mathit{path}]\ \rangle & \text{(T13)}\\
\mathcal{S}([\![s]\!]) & = & \text{string} \qquad \textit{otherwise} & \text{(T14)}
\end{array}
$$

Figure 4: Mapping XML-ODL to ODL

5.2 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 $\mathcal{T}([\![t]\!], \text{type_name})$, where type_name is the name of the current XML type. The semantic brackets, $[\![\,]\!]$, are used in denotational semantics to separate syntactic structures from semantic operations. The extra parameter ρ in $\mathcal{T}([\![t]\!], \rho)$ 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 are global variables that 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 XML-ODL schema, bibliography, is list⟨ C2 ⟩, where the

$$
\begin{array}{rcll}
\mathcal{R}(\llbracket\{\ldots,A:s_k,\ldots\}\,t\rrbracket,x,e.@A) & = & x.A & \text{(R1)}\\
\mathcal{R}(\llbracket A[t]\rrbracket,x,e.A) & = & x & \text{(R2)}\\
\mathcal{R}(\llbracket A[t]\rrbracket,x,e._) & = & x & \text{(R3)}\\
\mathcal{R}(\llbracket A[t]\rrbracket,x,e.*) & = & \left\{\begin{array}{ll} 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{array}\right. & \text{(R4)}\\
\mathcal{R}(\llbracket t_1,\,t_2\rrbracket,x,e.a) & = & \left\{\begin{array}{ll} \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{array}\right. & \text{(R5)}\\
\mathcal{R}(\llbracket t_1\,|\,t_2\rrbracket,x,e.a) & = & \left\{\begin{array}{l} \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{array}\right. & \text{(R6)}\\
\mathcal{R}(\llbracket t^*\rrbracket,x,e.a) & = & \left\{\begin{array}{ll} \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{array}\right. & \text{(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) & \text{(R8)}\\
\mathcal{R}(\llbracket \text{any}\rrbracket,x,e) & = & \text{handled in Section 2 by Rules (U1) through (U7)} & \text{(R9)}\\
\mathcal{R}(\llbracket t^*\rrbracket,x,e[e']) & = & x[e'] & \text{(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' & \text{(R11)}\\
\mathcal{R}(\llbracket t\rrbracket,x,e) & = & \text{struct()} \qquad \textit{otherwise} & \text{(R12)}
\end{array}
$$

Figure 5: Translation of an XML-OQL Path Expression into OQL ($a \in \{A, @A, _, *\}$)

class C2 captures the attributes and content of a vendor (it is a class because the vendor type has an ID attribute):

```
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; };
```

and class C1 captures the bib attributes and content:

```
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; };
```

5.3 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 5. 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 <tag $a_1 = u_1 \ldots a_m = u_m$>$e_1, \ldots, e_n$</tag>. The type T_i of the expression e_i is allowed to be one of the following types:

$$
\begin{array}{rcl}
T & ::= & \mathrm{XML}[t] \\
 & | & \mathrm{struct}\{\ T_1\ \mathrm{fst};\ T_2\ \mathrm{snd};\ \} \\
 & | & \mathrm{list}\langle\ T\ \rangle \\
 & | & \mathrm{list}\langle\ \mathrm{XML_element}\ \rangle \\
 & | & \mathrm{primitive_type}
\end{array}
$$

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 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 without attributes, <tag>$e_1, \ldots, e_n$</tag>, is translated into:

struct(fst: ..., struct(fst: e_1, snd: e_2) ..., snd: e_n)

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

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

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. Pointer dereferencing is evaluated very fast by current OODBs.

5.4 Type-Checking XML Path Expressions

The type-checking of XML-OQL queries can be done after queries are translated into OQL code by using the standard OQL type-checking engine. Nevertheless, type-checking XML path expressions before code generation can provide us with the actual XML type of this expression, which may be useful information during query optimization. The type-checking can be accomplished with the help of typing-rules, one for each kind of path expression. Although, we have already developed a full type-checking framework for XML-OQL, we only give here a sample of these rules. The XML type of the XML projection $e.A$ is $\mathcal{P}(t, e.A)$, where t is the XML type of e:

$$\begin{array}{lcl}
\mathcal{P}([\![A[t]]\!], e.A) & = & t \\
\mathcal{P}([\![B[t]]\!], e.A) & = & () \\
\mathcal{P}([\![t_1,\, t_2]\!], e.a) & = & \left\{ \begin{array}{ll} \mathcal{P}([\![t_1]\!], e.A) & \text{if } \mathcal{P}([\![t_2]\!], e.A) = () \\ \mathcal{P}([\![t_2]\!], e.A) & \text{if } \mathcal{P}([\![t_1]\!], e.A) = () \\ \mathcal{P}([\![t_1]\!], e.A),\ \mathcal{P}([\![t_2]\!], e.A) & \textit{otherwise} \end{array} \right. \\
\mathcal{P}([\![t^*]\!], e.A) & = & \left\{ \begin{array}{ll} () & \text{if } \mathcal{P}([\![t]\!], e.A) = () \\ \mathcal{P}([\![t]\!], e.A)^* & \textit{otherwise} \end{array} \right.
\end{array}$$

For example, the type of biblio.entry in our example query is bibliography, which is equal to the XML type:

bib[vendor[{ id: ID } (name[string], email[string], book[BT]*)]*]

where BT is { ISBN: ID, related_to: bib.vendor.book.ISBN* } (title[string], ...)}, that is, it includes the attributes and the content of a book. Therefore, the type of biblio.entry.bib is:

vendor[{ id: ID } (name[string], email[string], book[BT]*)]*

which, in turn, implies that the type of biblio.entry.bib.vendor is:

({ id: ID } (name[string], email[string], book[BT]*)*

Now the type of the path expression biblio.entry.bib.vendor.book is ((), (), BT*)*, which, according to the third rule above, is reduced to BT**, since identity types, (), are eliminated from concatenations and repetitions. Therefore, the type-checker will infer that the range variable bs has type BT*, which implies that the range variable b has type BT.

6 Indexing Web-Accessible XML Data

The implicit assumption behind our translations is that the XML schema is fixed and predefined. But this is not a realistic assumption. XML data are often scattered throughout the web in sometimes unknown locations. Consider again the query discussed in Section 1:

```
select list b.*.title
       from b in download("*").*.bib.*.book
       where b.*.author.*.lastname = "Smith"
```

At a first glance, the above query may seem difficult to evaluate in a schema-based framework. But let us address the problems intrinsic to this type of query more carefully. Suppose that we were somehow able to retrieve all the relevant XML files from the web. Some XML files may not have any schema information at all, while others may have incompatible schemas; some may even cause a type-checking error for the query. Queries in a traditional database system are first type-checked against a fixed schema before are optimized. But it is not that difficult to make the type-checker able to download type information on the fly during type-checking. For example, when the type-checker encounters the XML-OQL expression, download("http://www.acm.org/xml/index.xml"), it can parse the first lines of the index.xml file and can retrieve the location of the DTD description, if one exists. Then, it can download the DTD file and convert it into an XML type, which will be in fact the result of type-checking the download expression. Before an XML file is imported into the database, the type-checker must send a request to the catalog manager to create a new schema and to the storage manager to create new transient class extents to store the imported objects.

Another problem that needs to be addressed is the multiplicity and heterogeneity of downloaded schemas, such as the schemas of the expression download("*") in the above query. Suppose that the type-checker finds, using a web search engine, that there are n relevant documents with URL locations, $u_1, \ldots, u_n$, and schemas, XML[t_1], ..., XML[t_n], respectively. (Recall that, if a document is schema-less, it is assigned the type XML[any].) Unfortunately, it is not always possible to assign the type XML[$t_1, \ldots, t_n$] to document("*") and proceed as it was just one query because each path expression in the query would be mapped to a concatenation of types (one type for each document). For example, the path b.*.author.*.lastname would be assigned the type XML[string,...,string] (one string for each document). A better alternative is to evaluate the query n times, one for each

document, and concatenate the results in pairs. That is, the ith query will be evaluated with document("u_i") in place of document("*"). If a document causes a type error in the query, the document is simply ignored. The situation becomes more complicated in the presence of multiple download("*") calls in the same query, such as when XML documents are joined via IDrefs. In that case, all permutations of downloaded files must be considered.

The most challenging problem that needs to be addressed when evaluating the above query is retrieving all the relevant XML files from the web. To be more precise, a relevant XML file should not cause a compile-time error for this query (structure requirement), that is, it should match the paths, *.bib.*.book.*.title and *.bib.*.book.*.author.*.lastname, and it should contain the requested partial content information (content requirement), that is, it should contain at least one book authored by Smith. In case of multiple download("*") calls in the same query, not all combinations of relevant files associated with these calls would be relevant, since each relevant document may have different paths from the others. Furthermore, deriving the actual paths may also be challenging since XML-OQL supports XML projections over any OQL expression. Our previous work on normalization and query unnesting [16] would be very valuable on extracting these paths.

Inspired by the "text-in-context" XML search engine of the Niagara project [27] and its algebra, SEQL, we present a framework for indexing XML data that is more precise, easier to implement, and more efficient than that of Niagara. Like Niagara, our framework uses two inverted lists, word_index and tag_index, one for indexing content words in XML elements and another for indexing tag names. In OODBs, inverted lists can be specified as class extents:

```
class XML_word ( key word extent word_index )
{   attribute string word;
    attribute set⟨  struct{ string URL; integer level; integer location; }  ⟩ occurs; };
struct element_spec { string URL; integer level; integer begin_loc; integer end_loc; };
class XML_tag ( key tag extent tag_index )
{   attribute string tag;
    attribute set⟨  element_spec  ⟩ occurs; };
```

The first class associates words to documents. Here we assume that an OODB will create a primary index, such as a B^+-tree, for the primary key. In that case, the class extent would behave like an inverted list. The occurs attribute contains all occurrences of a word in all web-accessible XML documents. In addition to the location of the word in a document, we keep the nesting level (depth) of the word in the document. The word level makes our indexing technique more precise than that of Niagara. The second inverted list, tag_index, indexes XML element tags. For some tag t in a documents D, the begin_loc attribute specifies the location of <t> in D while end_loc specifies the location of the matching </t>.

Our indexing framework is basically a non-standard interpretation of XML path expressions. The standard interpretation of path expressions yields, of course, XML elements (all elements

reachable by the path). But our non-standard interpretation, $\mathcal{I}(\llbracket e \rrbracket)$, of a path expression e yields a set of document URLs (a set of element_spec, to be more precise). These are all the documents matching the path e in terms of both content and structure. The non-standard interpretation of indexing is achieved by the following compositional rules: For a tag projection $e.A$, $\mathcal{I}(\llbracket e.A \rrbracket)$ is equal to

```
select y from x in I([[e]]), a in tag_index, y in a.occurs
 where a.tag = "A" and x.URL = y.URL and x.level+1 = y.level
   and x.begin_loc < y.begin_loc and x.end_loc > y.end_loc
```

If e is a wildcard projection, then x.level < y.level is used in the predicate instead of x.level+1 = y.level. Under that correction, $\mathcal{I}(\llbracket e.* \rrbracket)$ is equal to $\mathcal{I}(\llbracket e \rrbracket)$. The interpretation of $\mathcal{I}(\llbracket e._ \rrbracket)$ is the same as $\mathcal{I}(\llbracket e \rrbracket)$ but with an increased depth level:

```
select URL: x.URL, level: x.level+1, begin_loc: x.begin_loc, end_loc: x.end_loc from x in I([[e]])
```

Content indexing is necessary when there is a comparison between a path e and a constant value c, such as b.*.author.*.lastname = "Smith", as is done for regular indexes in databases. In that case, for some comparison operator, *cmp*, $\mathcal{I}(\llbracket e \; cmp \; c \rrbracket)$ is equal to:

```
select y from x in I([[e]]), a in word_index, y in a.occurs
 where a.word cmp "c" and x.URL = y.URL and x.level+1 = y.level
   and x.begin_loc < y.location and x.end_loc > y.location
```

For these translations to be effective, the query optimizer must be able to perform basic query unnesting since each projection yields one nested query. For example, using the above rules and after query unnesting, the condition *.author.*.lastname = "Smith" is translated into:

```
select z
  from a in tag_index, x in a.occurs, b in tag_index, y in b.occurs, c in word_index, z in c.occurs
 where a.tag = "author" and b.tag = "lastname" and x.URL = y.URL and x.level < y.level
   and x.begin_loc < y.begin_loc and x.end_loc > y.end_loc
   and c.word = "Smith" and y.URL = z.URL and y.level+1 = z.level
   and y.begin_loc < z.location and y.end_loc > z.location
```

Note that a typical OODB optimizer will use the primary index of the tag_index and word_index extents, since there are key equalities in the query. In general, from *all* XML path expressions in a query, even in the presence of multiple document("*") calls, *only one* query is composed that calculates the relevant document URLs. There are many advantages of implementing web searching using one query against the inverted indexes. A good OODB query optimizer should be able to find the best way to evaluate the query based on statistical information, which may not necessarily be the nested-loop join that proceeds from left to right in a path expression, as is implicit in the Niagara web search engine. For example, there may be many XML documents that have the same top-level tag. Thus, starting searching the dictionary from this tag may not be wise. Hence, we expect to achieve a more efficient search than that of Niagara. Furthermore, our method is very

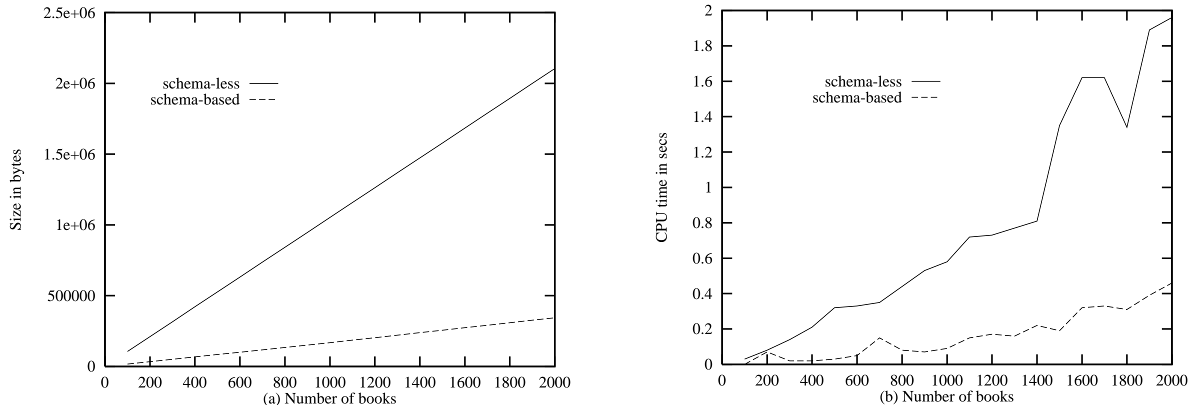

Figure 6: Comparing Schema-Less with Schema-Based Implementations

effective for multiple document retrievals in the same query, since only one query is generated that precisely relates the different documents involved, given that one document may be linked to others in the same query via idrefs or content constraints. In addition, our framework is extendible, since it is rule-based, and is easy to validate.

7 Implementation

We have already implemented the schema-less translation from XML-OQL to OQL using the lambda-DB object-oriented database management system [17], which is built on top of the SHORE object storage system [6]. This implementation is now part of the latest release (1.5) 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 6.(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.book.author,<title>x.book.title</title></book>
    from  x in document(”books”).vendor.books
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. We are also planning to implement and compare our web indexing method with two other related proposals, in terms of accuracy and efficiency, possibly the Niagara system and one commercial XML indexing system, such as GoXML (`http://www.goxml.com`).

8 Related Work

There are several recent proposals for XML query languages, such as UnQL [5], StruQL [18], XML-QL [15], YATL [12], WebOQL [2], Lorel [21], POQL [9], Quilt [8], and X-OQL [1]. A survey of some of these languages can be found elsewhere [20]. We do not claim that our query language, which is basically an extension of OQL, is more powerful or better than others. Instead, we use our own language because our goal is to capture the most important features [25] found in many of these languages in the simplest way possible and give them clean, compositional semantics. We believe that our semantics and transformations can also be applied to other languages, albeit with much more effort. However, there are features lacking from our language, such as primitive recursion for traversing tree-like structures, which are particularly important for semi-structured data. Primitive recursion is the most important feature of XSLT [31] (the XML language for stylesheet transformations) and is an important component of UnQL [5] (in the form of structural recursion). Primitive recursion is missing from OQL but can be simulated with recursive functions. However, recursion is difficult to optimize. Finally, some XML query languages support features for restructuring XML data, such as the FILTER construct of Quilt [8], which we suspect may have very complex formal semantics.

There are already commercial OODB systems, such as POET's Content Management Suite and Object Design's eXcelon system, that provide some functionality for storing and handling XML data. Furthermore, there are research projects that use OODB technology for XML query optimization, such as Lore [26], but are purely based on schema-less XML data. Our work is highly influenced by the work of Christophides, et al [9] on storing and handling SGML data using OQL, based on DTD schema information. We go beyond that work by supporting a type system well integrated with that of ODMG ODL, which 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. Even though their work describes an algebra based on path expressions and object identity, they only give a brief sketch of the translation from the language to the algebra. Unlike that work, we provide precise compositional semantics to the query language. Instead of inventing yet another semi-structured algebra, all our transformations

are targeting the core OQL itself, which in turn can be translated into one's favorite complex object algebra (there are already plenty of them).

Our web indexing engine is influenced by Niagara [27], which uses a special search engine, called text-in-context, to retrieve the relevant XML documents for an XML-QL query. Niagara uses a special indexing algebra, SEQL, as an intermediate form for XML-QL queries, but it is yet to be seen if there are effective optimization rules for this algebra. Our approach does not require any special algebra. Instead, we let the OODB query optimizer decide the best way to use the indexes based on statistical information. At the time of this writing, another proposal for XML indexing in relational databases came up [32], which also uses a tag and a word inverse indexes and is closer to our framework than Niagara. Unlike our work though, their indexing technique concentrates on simple XPath queries, which are restricted to work on a single document source and do not support XML data construction on the fly. Path expressions in XML-OQL as well as in many other recently proposed XML query languages, such as Quilt, may have a context, such as the path expression b.title in our example query, where variable b is bound into another path expression elsewhere in the query. The translation of such queries can be easily accomplished using compositional transformation rules where each subexpression is translated independently, and all translations are composed to form the final translation. The context is propagated throughout the translations in the form of binding lists, as we have shown elsewhere in the paper.

9 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 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. The second contribution of this paper is related to a novel framework that maps an XML-OQL query over all web-accessible XML data into an OQL query over two inverse indexes, which effectively summarize the content and structure of all these documents. Even though similar indexes have been used by other XML indexing systems, our framework utilizes them in the best way possible by using the OQL query optimizer itself to construct a usage plan based on statistical information.

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

References

[1] S. Abiteboul, V. Aguilera, S. Ailleret, B. Amann, S. Cluet, B. Hills, F. Hubert, J.-C. Mamou, A. Marian, L. Mignet, T. Milo, C. S. dos Santos, B. Tessier, and A.-M. Vercoustre. XML Repository and Active Views Demonstration. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99), Edinburgh, Scotland*, pages 742–745, 1999.

[2] G. Arocena and A. Mendelzon. WebOQL: Restructuring Documents, Databases, and Webs. In *Proceedings of the Fourteenth International Conference on Data Engineering, Orlando, Florida*, pages 24–33, Feb. 1998.

[3] C. Beeri and Y. Tzaban. SAL: An Algebra for Semistructured Data and XML. In *ACM SIGMOD Workshop on The Web and Databases (WebDB'99), Philadelphia, Pennsylvania*, pages 37–42, June 1999.

[4] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Canada*, pages 505–516, May 1996.

[5] P. Buneman, M. Fernandez, and D. Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDB Journal*, 9(1):76–110, 2000.

[6] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring Up Persistent Applications. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota*, pages 383–394, May 1994.

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

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

[9] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From Structured Documents to Novel Query Facilities. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota*, pages 313–324, May 1994.

[10] V. Christophides, S. Cluet, and J. Siméon. On Wrapping Query Languages and Efficient XML Integration. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas*, pages 141–152, May 2000.

[11] S. Cluet and C. Delobel. A General Framework for the Optimization of Object-Oriented Queries. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Diego, California*, pages 383–392, June 1992.

[12] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your Mediators Need Data Conversion! In *ACM SIGMOD International Conference on Management of Data, Seattle, Washington*, pages 177–188, June 1998.

[13] S. Cluet and G. Moerkotte. Nested Queries in Object Bases. In *Fifth International Workshop on Database Programming Languages, Gubbio, Italy*, page 8, Sept. 1995.

[14] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Maier, and D. Suciu. Querying XML Data. *IEEE Data Engineering Bulletin*, 22(3):10–18, 1999.

[15] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. *WWW8 / Computer Networks*, 31(11–16):1155–1169, 1999.

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

[17] L. Fegaras, C. Srinivasan, A. Rajendran, and D. Maier. λ-DB: An ODMG-Based Object-Oriented DBMS. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas*, page 583, May 2000.

[18] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for a Web-Site Management System. *SIGMOD Record*, 26(3):4–11, 1997.

[19] M. Fernandez, J. Simeon, and P. Wadler. An Algebra for XML Query. In *FST TCS, Delhi*, Dec. 2000.

[20] D. Florescu, A. Levy, and A. Mendelzon. Database Techniques for the World-Wide Web: A Survey. *SIGMOD Record*, 27(3):59–74, 1998.

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

[22] 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.

[23] A. Kemper and G. Moerkotte. Advanced Query Processing in Object Bases Using Access Support Relations. In *Proceedings of the Sixteenth International Conference on Very Large Databases, Brisbane, Australia*, pages 290–301. Morgan Kaufmann Publishers, Inc., Aug. 1990.

[24] T. Leung, G. Mitchell, B. Subramanian, B. Vance, S. Vandenberg, and S. Zdonik. The AQUA Data Model and Algebra. In *Fourth International Workshop on Database Programming Languages, Manhattan, New York City*, pages 157–175. Springer-Verlag, Workshops on Computing, Aug. 1993.

[25] D. Maier. Database Desiderata for an XML Query Language. In *Query Languages 98 (QL'98)*, 1998. Available at `http://www.w3.org/TandS/QL/QL98/pp/maier.html`.

[26] J. McHugh and J. Widom. Query Optimization for XML. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99), Edinburgh, Scotland*, pages 315–326, 1999.

[27] J. Naughton, D. DeWitt, and D. Maier. The Niagara Internet Query System . Submitted for publication. Available at `http://www.cs.wisc.edu/niagara/papers/NIAGRAVLDB00.v4.pdf`, 2000.

[28] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99), Edinburgh, Scotland*, pages 302–314, 1999.

[29] P. Wadler. The Next 700 Markup Languages. In *Second Conference on Domain Specific Languages (DSL'99), Invited Talk, Austin, Texas*, Oct. 1999.

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

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

[32] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, Santa Barbara*, May 2001.