A Search Engine for Complex XML Queries

Leonidas Fegaras

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@cse.uta.edu*

Abstract

The implicit assumption behind most current XML query languages is that XML data are located in local or remote repositories at known URL addresses. But there are new proposals for XML languages now that allow queries over the whole Internet, as is done by current web search engines for web-accessible HTML documents. The ultimate goal is to see the Internet as a huge distributed database of XML data, stored in either native or database form, which can be accessed by a distributed query processing system that allows more powerful, more accurate results than traditional search engines. But XML query languages go beyond the simple keyword-based boolean formulas supported by current web search engines. One may wonder how feasible it is to evaluate XML queries that join web documents at unknown locations. Traditional document indexing techniques, such as inverted files, are based on keyword content only and may not be very effective for indexing XML data, since structure is an essential component of XML data.

Inspired by the "text-in-context" XML search engine of the Niagara project, we propose a general framework for 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 any complex XML query, our framework is capable of constructing an OQL query over the inverse index structures, which retrieves all the web-accessible XML document fragments that satisfy the containment and content constraints of the XML query. Our framework is based on the idea of state transformation, which is used often to give semantics to procedural languages. The state in our case is a binding from document variables to OQL queries that compute document fragments.

1 Introduction

XML has emerged as the leading textual language for representing and exchanging data on the web. Even though HTML is still the dominant format for publishing documents on the web, XML has become the prevalent exchange format for business-to-business transactions. It is expected that in the near future the internet will be populated with a large number of web-accessible XML files. One of the reasons of its popularity is that, by supporting simple nested structures of tagged elements, the XML format is able to represent both the structure and content of complex data very effectively. To take advantage of the structure of XML documents, new query languages had to be invented that go beyond the simple keyword-based boolean formulas supported by current web search engines. These queries allow a more precise searching of the web. One can now search for all XML documents containing bibliographies that match a particular structure and have at least one entry for a paper written by a given author, rather than searching for all documents that contain the author's name. Current document indexing techniques, such as inverted files, are based on keyword content only and may not be very effective for indexing XML data, since structure is an essential component of XML data.

Users familiar with web search engines would like to write XML queries, such as:

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

```
class Document ( extent documents ) { attribute string URL; attribute integer size; };
struct word_spec { Document doc; integer level; integer location; };
struct tag_spec { Document doc; integer level; integer ordinal; integer begin_loc; integer end_loc; };
class XML_word ( key word extent word_index )
{   attribute string word;
    attribute set⟨ word_spec ⟩ occurs; };
class XML_tag ( key tag extent tag_index )
{   attribute string tag;
    attribute set⟨ tag_spec ⟩ occurs; };
```

Figure 1: The XML Inverse Indexes

to query the entire web for all the XML files that contain book references to find all the books authored by Smith. Users tend to put as many wildcards in a 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 match all the paths appearing in the query, but also satisfy the query condition. For our example query, we would like to retrieve all documents from the web that match all the paths in this query, that is, they match the paths *.book.*.title and *.book.*.author.*.lastname, and contain the requested partial content information, that is, they contain at least one book authored by Smith.

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 [16, 15], we present a formally verifiable framework for indexing XML data that is more general 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. 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. Our approach is based on the simple idea, first introduced by the Niagara project [16], of letting the database query optimizer decide the best way to use the indexes based on statistical information. We present a set of rules that transform a complex XML query into an ODMG OQL [3] query against the two indexes. This query is called a *search query*. The result of the search query is a list of relevant documents that satisfy the original XML query in terms of both structure and content. In contrast to Niagara [16], which handles simple XPath queries, our framework can process any XML query of arbitrary complexity, with any number of document("*") clauses. Even though we have based our translations to our own XML query language, XML-OQL [9], we believe that our framework can be adapted to handle other complex XML query languages, such as XQuery [4] and Quilt [5].

In our framework, the search query generated from an XML query may retrieve database data but should not, in any way, access the web. All the web-accessible XML documents are summarized in two indexes, described by their ODMG ODL schema in Figure 1. These indexes can be populated off-line by web crawlers, as is done by current web search engines. The XML_word class associates words to documents. It is assumed that a typical 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 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 the first character of <t> in D while end_loc specifies the location of the last character of the matching </t>.

The ordinal component indicates the relative order of the element within its parent element.

Based on the two inverted lists in Figure 1, our framework translates the example XML query above into the following search query:

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

which returns all the web-accessible document fragments that satisfy the query. It is important to note that, when retrieved from the web, these document fragments will contain book titles only (namely, all titles of books written by Smith), and no further testing is necessary. The original XML query itself cannot be answered from the indexes alone, because book titles cannot be reconstructed from the word_index, since only the keywords in the title are indexed. But, if the web server of these documents is capable of sending XML fragments to clients, the title retrievals will take a very short time. One problem with document fragments is that they become invalid if the documents are changed after are indexed by the web crawler. One way to solve this problem is to include the timestamp of indexing as part of the Document class. This timestamp is sent along with a document fragment. The document server must be able to retrieve the proper document version at the timestamp point and to send the correct fragment.

A typical query optimizer will be able to use the primary indexes of the tag_index and word_index extents to evaluate the above search query efficiently, since there are key equalities in the query. Since there are five key equalities in the search query, there are many possible orders of indexing. A typical query optimizer will be able to select a good order based on statistics (the selectivities of the equalities), which may not necessarily be the indexed nested-loop join that proceeds from left to right in a path expression, as is implicit in the current XML search engines. The query graph of the above search query is a chain, but this may not be necessarily true for complex XML queries. A complex XML query may generate a large number of joins, one join for each XML projection, which may go beyond the capabilities of current relational database query optimizers (commercial relational DBMSs can handle up to 20 joins), since they use an exponential join ordering algorithm. But there are many heuristic cost-based query optimizers that can handle hundreds of joins. One such system is GOO [8], which is a bottom-up greedy algorithm that always performs the most profitable operations first. Our framework will be implemented on top of the λ-DB OODB management system [11], which uses GOO to optimize queries.

To understand the difficulty of the problem, consider the following XML-OQL query:

```
select <info><name>e.name</name>,
                <title>b.*.title</title>
       </info>
 from e in Instructors, b in document("*").*.book
where b.*.author.name = e.name
  and ( exists a in document("http://www.nsf.gov/*").*.award:
              a.*.PI.name = e.name and a.*.amount > 100000 )
```

which retrieves the books of all instructors who have received an NSF grant larger than $100K. The instructor information is stored in the local database, the NSF grant information is searched at the NSF web site, while the book information is searched over the entire web. We may even have a join between two arbitrary XML documents, as is implicit in cross-document links supported by XLink, or aggregations over the entire web, such as count(document("*")), which counts all web-accessible XML files.

Our framework is based on the idea of state transformation, which is used often to give semantics to procedural languages. The state in our case is a binding list that associates document variables to OQL queries over the XML inverse indexes that compute sets of document fragments. A document variable is a range variable whose domain is all web-accessible XML documents. Each document("*") in a query is assigned a different document variable whose initial value is all documents, but is later refined to meet the structural and content constraints of the query. The state is threaded unchanged through the regular OQL

operations but is modified in the presence of XML path expressions. Our framework is general enough to handle any XML-OQL query since the only place that needs special attention is during XML paths. Hence, our framework can be easily adapted to handle any XML query language that supports similar XML path syntax.

2 Preliminaries: XML-OQL

Our XML query language is an extension of standard OQL [3]. It captures all the important features found in most recent XML query languages, including XQuery [4] and Quilt [5]. 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 [6], Ozone [14], Lorel [12], 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 [10, 7]). It also supports a decidable type-checking system, well integrated with the type-checking system of OQL. The semantics and type-checking rules for XML-OQL are given elsewhere [9].

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

```
select <bib><author>b.author.lastname </author>,
             <title>b.title</title>,
             <related>select <title>r.title</title>
                          from r in b.@related_to </related>
       </bib>
 from b in document("bibliography.xml").bib.*.book
where b.year>1995 and count(b.author)>2 and contains(b.title,"computer")
```

which 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 may both 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. The XML-OQL typechecking rules and the translation schemes from XML-OQL to OQL are described in another paper [9]. Here we only give a brief description of XML-OQL.

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

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

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	(tag projection)
$e.\_$	projection over any tag	(any projection)
$e.*$	all subelements of e at any depth	(wildcard)
$e.@A$	projection over the attribute A of e	(attribute projection)
$e[\backslash v \rightarrow e']$	the subelements of e that satisfy the predicate e'	(filtering)
$e[e']$	the subelement of e at position e'	(indexing)
$e[e_1 : e_2]$	all subelements of e starting at position e_1 and ending at position e_2	(range indexing)

where e, e', e_1, and e_2 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.

A few words about the XML-OQL typing system. Schema-less XML data are of the ODL type, XML, while schema-based data are of the ODL type, XML[t], where t is an XDuce-style XML type [13]. Schema-based XML-OQL path expressions can be easily and unambiguously typechecked and translated to OQL [9]. Schema-less XML-OQL path expressions are translated to OQL expressions against a fixed schema and have a high cost: A tag projection requires unnesting (i.e. scanning of the list of subelements) while a wildcard projection requires a transitive closure, which can be simulated with a call to a recursive OQL function. The typing of schema-less paths is easier though: if e is of type XML or list⟨XML⟩, then $e.A$ is of type list⟨XML⟩. The typechecker may wrap an XML-OQL path with the following coercion functions when necessary: element (a function from list⟨XML⟩ to XML), XML_to_string (from XML to string), and XML_to_int (from XML to integer), as is done for the predicate b.year>1995 in the above query. The wrapping is done during the type equivalence (or type unification) testing between two types. For example, since the left operand of b.year>1995 has type list⟨XML⟩ where an integer was expected, the typechecker will convert it into XML_to_int(element(b.year))>1995. The OQL element function may raise a run-time exception, which basically materializes the run-time typechecking. XML constructions are always schema-less: The elements e_i of the element construction $<A>e_1,\ldots,e_n</A>$ are expected to be of type list⟨XML⟩. Hence an integer/string e_i is lifted to an XML object, an XML object is lifted to a singleton list⟨XML⟩ value, and an XML[t] value is converted to an XML object (through a schema-driven function). Handling XML constructions as schema-based values is a better alternative that we have not investigated, but requires a complex, incomplete type inference [13].

3 The Construction of the Search Query

This section presents the main contribution of this paper, namely the generation of a web search query from a web-accessing XML-OQL query. We will first concentrate on XML-OQL queries that may contain multiple document("*") references, but will later extend our framework to handle patterns in document names (URL patterns). The main idea of our approach is to assign a unique variable name to each document("*") clause in a query, called a *document variable*, and bind it to a set of document fragments in a binding list σ. A document fragment is of type:

struct doc_fragment { Document doc; **integer** level; **integer** begin_loc; **integer** end_loc; };

and specifies the depth level and the location of an element <t>...</t> in a document doc. That is, the begin_loc attribute specifies the location of the first character of <t> in the document while the end_loc attribute specifies the location of the last character of the matching </t>. A document fraction contains a document element needed by the query. That is, it is an element that satisfies the structural and content requirements of the query. The binding list, σ, is of type

$$\mathsf{set}\langle\,(\,\mathsf{string},\,\mathsf{set}\langle\,\mathsf{doc\_fragment}\,\rangle\,)\,\rangle$$

and associates a name to a set of document fragments. It supports the following two operations:

- $\sigma[v/e]$ extends or updates σ with the binding from the document variable v to a set of fragments e.
- $\sigma[v]$ retrieves the document fragments associated with the document v.

Our translation schemes 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 supports basic query unnesting, as is the case for the λ-DB OODB management system [11]. The most common form of query unnesting that we expect any OODB system to be able to support is when the domain of a range variable is the result of another OQL query:

$$\textbf{select } h_1 \textbf{ from } ..., v \textbf{ in } (\textbf{select } h_2 \textbf{ from } v_1 \textbf{ in } e_1, ..., v_n \textbf{ in } e_n \textbf{ where } p_2), ... \textbf{ where } p_1$$

In that case, the query can be simply rewritten into:

$$\textbf{select } h_1 \textbf{ from } ..., v_1 \textbf{ in } e_1, ..., v_n \textbf{ in } e_n, v \textbf{ in } \{h_2\}, ... \textbf{ where } p_1 \textbf{ and } p_2$$

which may require some variable renaming to avoid variable capture. The resulting query can be further reduced by removing v **in** $\{h_2\}$ and substituting v for h_2.

3.1 State Propagation

The binding list, σ, contains one binding for each document variable in a query. The initial binding of a document variable is the set of all documents accessible on the web but is eventually refined to meet the structural and content constraints of the query. The structural refinements are based on the tag_index while the content refinements are based on the word_index. The content refinements are very complex to capture as they may involve complex relationships between XML-OQL paths, which may be combined in complex boolean expressions or in existential and universal quantifications. One way to approach this problem in formal semantics is through state transformers. In the context of our translations, the state is the binding list, σ, which must be threaded through all XML-OQL operations in a query, and be refined in the presence of structural or content constraints.

State transformers are often used to give semantics to procedural languages. One way of handling side effects in denotational semantics is to map terms that compute first-order values of type T into functions of type $S \rightarrow T \times S$, called state transformers, where S is the type of the state in which all side effects take place. That is, a term of type T is mapped into a function that takes some initial state s_0 of type S as input and generates a value of type T and a new state s_1. If a term performs side effects, the state transformer maps s_0 into a different state s_1 to reflect these changes. Otherwise, the state remains unchanged. For example, the constant integer, 3, is mapped into the state transformer $\lambda s.(3, s)$ which propagates the state as is.

Our state transformer is a bit unconventional: A first-order value e of type T is lifted to a state transformer $\mathcal{T}[\![e]\!]$ of type $S \rightarrow \mathrm{set}\langle\, T \times S\,\rangle$, where S is the type of the binding list σ. We decided that the state transformer returns a set of pairs rather than one pair to allow multiple returned values for different states, as it will be apparent when we give semantics for content restrictions. The semantic brackets, $[\![\,]\!]$, give the meaning of the syntax enclosed by the brackets in terms of pure OQL. In general, if e is of type t, then the type of $\mathcal{T}[\![e]\!]$, denoted by t^*, is defined inductively as follows:

$$\begin{array}{rcll} (t_1 \rightarrow t_2)^* & = & S \rightarrow \mathrm{set}\langle\,(t_1 \rightarrow t_2^*) \times S\,\rangle & \textit{for a higher-order type} \\ t^* & = & S \rightarrow \mathrm{set}\langle\,t \times S\,\rangle & \textit{for a first-order type } t \end{array}$$

The following equations give the standard interpretation of the call-by-value λ-calculus with state propagation:

$$\mathcal{T}[\![v]\!]\,\sigma = \{(v, \sigma)\} \quad (1)$$

$$\mathcal{T}[\![(e_1, e_2)]\!]\,\sigma = \textbf{select } ((v_1, v_2), \sigma_2) \textbf{ from } (v_1, \sigma_1) \textbf{ in } \mathcal{T}[\![e_1]\!]\,\sigma,\ (v_2, \sigma_2) \textbf{ in } \mathcal{T}[\![e_2]\!]\,\sigma_1 \quad (2)$$

$$\mathcal{T}[\![\lambda v.e]\!]\,\sigma = \{(\lambda v.\lambda\sigma'.\mathcal{T}[\![e]\!]\,\sigma', \sigma)\} \quad (3)$$

$$\mathcal{T}[\![e_1\, e_2]\!]\,\sigma = \textbf{select } (v, \sigma_3) \textbf{ from } (f, \sigma_1) \textbf{ in } \mathcal{T}[\![e_1]\!]\,\sigma,\ (x, \sigma_2) \textbf{ in } \mathcal{T}[\![e_2]\!]\,\sigma_1,\ (v, \sigma_3) \textbf{ in } (f\, x\, \sigma_2) \quad (4)$$

Rule (3) handles higher-order terms. For example, $\mathcal{T}[\![\lambda v.v]\!]$ has type $S \to \text{set}\langle (T \to S \to \text{set}\langle T \times S \rangle) \times S \rangle$ and is translated into $\{(\lambda v.\lambda\sigma'.\{(v,\sigma')\},\sigma)\}$ when applied to a state σ. Rule (4) assumes a call-by-value interpretation: e_2 in the function application $e_1(e_2)$ is evaluated before e_1 is applied.

Threading the state through a select-from-where OQL statement is more complex. We translate queries of the form:

$$\textbf{select } e_h \textbf{ from } v_1 \textbf{ in } e_1, \ldots, v_n \textbf{ in } e_n \textbf{ where } e_p$$

in steps, one range variable at a time. First, we classify the range variables v_i into two categories: those ranging over XML values and those ranging over standard ODL collections. If v_1 ranges over an XML value e_1, then

$$\mathcal{T}[\![\textbf{select } e_h \textbf{ from } v_1 \textbf{ in } e_1, v_2 \textbf{ in } e_2, \ldots, v_n \textbf{ in } e_n \textbf{ where } e_p]\!]\sigma$$

is equal to:

$$\begin{array}{l}\textbf{select } (\ w, \sigma_2\ ) \\ \textbf{from } (v_1,\sigma_1) \textbf{ in } \mathcal{T}[\![e_1]\!]\sigma, \\ \qquad (w,\sigma_2) \textbf{ in } \mathcal{T}[\![\textbf{select } e_h \textbf{ from } v_2 \textbf{ in } e_2, \ldots, v_n \textbf{ in } e_n \textbf{ where } e_p]\!]\sigma_1\end{array}$$

Otherwise, if v_1 ranges over a collection, the above query becomes:

$$\begin{array}{l}\textbf{select } (\ (\textbf{select } h \textbf{ from } v_1 \textbf{ in } s, h \textbf{ in } w), \sigma_2\ ) \\ \textbf{from } (s,\sigma_1) \textbf{ in } \mathcal{T}[\![e_1]\!]\sigma, \\ \qquad (w,\sigma_2) \textbf{ in } \mathcal{T}[\![\textbf{select } e_h \textbf{ from } v_2 \textbf{ in } e_2, \ldots, v_n \textbf{ in } e_n \textbf{ where } e_p]\!]\sigma_1\end{array}$$

The bottom case is when there are no more range variables. In that case, **select** e_h **from where** e_p is equivalent to **if** e_p **then** { e_h } **else** { }. Hence, $\mathcal{T}[\![\textbf{select } e_h \textbf{ from where } e_p]\!]\sigma$, has the following interpretation:

$$\begin{array}{l}\textbf{select } (\ \textbf{if } p \textbf{ then } \{\ h\ \} \textbf{ else } \{\ \}, \sigma_2\ ) \\ \textbf{from } (p,\sigma_1) \textbf{ in } \mathcal{T}[\![e_p]\!]\sigma, \\ \qquad (h,\sigma_2) \textbf{ in } \mathcal{T}[\![e_h]\!]\sigma_1\end{array}$$

More complex query forms with a group-by, order-by, etc, can be handled in a similar way.

The above rules thread the state σ without changing it. The state changes in the presence of XML-OQL paths, as it is described in the next subsection. These rules will be our guidelines for translating the XML-OQL paths: the state is propagated through the component expressions in the same order as the evaluation order in the call-by-value λ-calculus. The above rules satisfy a very important property: if e is a standard first-order OQL expression (i.e., an OQL query that does not contain XML-OQL paths), then $\mathcal{T}[\![e]\!]\sigma$ is equal to $\{(e,\sigma)\}$, as it can be easily verified using structural induction.

Note that we have expressed our semantics using OQL pseudo-code (the distinct keyword has been omitted and pair patterns have been used as range variables). A more elegant way of expressing the semantics would have been in terms of monads and monad comprehensions, which are often used in lazy functional languages to hide the standard (and ugly) plumbing. But our goal is to generate OQL code, which will constitute the synthesized search query, because we would like to use a standard OQL optimizer to optimize this code. (After all, OQL queries are equivalent to comprehensions.)

3.2 Non-Standard Interpretation of XML-OQL Paths

In our interpretation, an XML-OQL path expression is mapped to a variable name, which is bound in σ to a set of document fragments that satisfy the path expression. A new variable name, nv, is created for each document("*") expression:

$$\begin{array}{rcl}\mathcal{T}[\![\text{document("*")}]\!]\sigma & = & \{(\text{nv},\sigma[\text{nv}/(\ \textbf{select } \text{doc: d, level: 0, begin\_loc: 1, end\_loc: d.size} \\ & & \qquad \textbf{from } \text{d } \textbf{in } \text{documents }\ )])\}\end{array}$$

The variable name associated with the tag projection $e.A$ is equal to that of e. The binding of this variable is refined to retrieve the sub-fragments tagged with A:

$$\begin{array}{rcl}\mathcal{T}[\![e.A]\!]\sigma & = & \textbf{select } (v,\sigma'[v/(\ \textbf{select } \text{doc: y.doc, level: y.level, begin\_loc: y.begin\_loc, end\_loc: y.end\_loc} \\ & & \qquad \textbf{from } \text{x } \textbf{in } \sigma'[v], \text{a } \textbf{in } \text{tag\_index, y } \textbf{in } \text{a.occurs} \\ & & \qquad \textbf{where } \text{a.tag} = \text{"A"} \textbf{ and } \text{x.doc} = \text{y.doc } \textbf{and } \text{x.level}+1 = \text{y.level} \\ & & \qquad\quad \textbf{and } \text{x.begin\_loc} < \text{y.begin\_loc } \textbf{and } \text{x.end\_loc} > \text{y.end\_loc }\ )]) \\ & & \textbf{from } (v,\sigma') \textbf{ in } \mathcal{T}[\![e]\!]\sigma\end{array}$$

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, $\mathcal{T}[\![e.*]\!]$ is equal to $\mathcal{T}[\![e]\!]$. The interpretation of $\mathcal{T}[\![e._]\!]$ is the same as $\mathcal{T}[\![e]\!]$, but with an increased depth level:

$$\begin{aligned}\mathcal{T}[\![e._]\!]\,\sigma \;=\; &\textbf{select } (v,\sigma'[v/(\textbf{ select } \text{doc: x.doc, level: x.level+1, begin\_loc: x.begin\_loc, end\_loc: x.end\_loc}\\ &\qquad\qquad \textbf{from } \text{x } \textbf{in } \sigma'[v]\ )])\\ &\textbf{from } (v,\sigma') \textbf{ in } \mathcal{T}[\![e]\!]\,\sigma\end{aligned}$$

Indexing uses the value of the index as the ordinal for the sub-element:

$$\begin{aligned}\mathcal{T}[\![e_1[e_2]]\!]\,\sigma \;=\; &\textbf{select } (v_1,\sigma_2[v_1/(\textbf{ select } \text{doc: y.doc, level: y.level, begin\_loc: y.begin\_loc, end\_loc: y.end\_loc}\\ &\qquad\qquad \textbf{from } \text{x } \textbf{in } \sigma_2[v_1], \text{a } \textbf{in } \text{tag\_index, y } \textbf{in } \text{a.occurs}\\ &\qquad\qquad \textbf{where } \text{a.ordinal}{=}v_2 \textbf{ and } \text{x.doc} = \text{y.doc } \textbf{and } \text{x.level}{+}1 = \text{y.level}\\ &\qquad\qquad\quad \textbf{and } \text{x.begin\_loc} < \text{y.begin\_loc } \textbf{and } \text{x.end\_loc} > \text{y.end\_loc }])\\ &\textbf{from } (v_1,\sigma_1) \textbf{ in } \mathcal{T}[\![e_1]\!]\,\sigma,\ (v_2,\sigma_2) \textbf{ in } \mathcal{T}[\![e_2]\!]\,\sigma_1\end{aligned}$$

while index ranging retrieves all sub-elements with ordinal number between the range values:

$$\begin{aligned}\mathcal{T}[\![e_1[e_2 : e_3]]\!]\,\sigma \;=\; &\textbf{select } (v_1,\sigma_3[v_1/(\textbf{ select } \text{doc: y.doc, level: y.level, begin\_loc: y.begin\_loc, end\_loc: y.end\_loc}\\ &\qquad\qquad \textbf{from } \text{x } \textbf{in } \sigma_3[v_1], \text{a } \textbf{in } \text{tag\_index, y } \textbf{in } \text{a.occurs}\\ &\qquad\qquad \textbf{where } v_2 \le \text{a.ordinal} \le v_3 \textbf{ and } \text{x.doc} = \text{y.doc } \textbf{and } \text{x.level}{+}1 = \text{y.level}\\ &\qquad\qquad\quad \textbf{and } \text{x.begin\_loc} < \text{y.begin\_loc } \textbf{and } \text{x.end\_loc} > \text{y.end\_loc }])\\ &\textbf{from } (v_1,\sigma_1) \textbf{ in } \mathcal{T}[\![e_1]\!]\,\sigma,\ (v_2,\sigma_2) \textbf{ in } \mathcal{T}[\![e_2]\!]\,\sigma_1,\ (v_3,\sigma_3) \textbf{ in } \mathcal{T}[\![e_3]\!]\,\sigma_2\end{aligned}$$

Finally, the predicate of filtering is handled as a lambda abstraction:

$$\begin{aligned}\mathcal{T}[\![e_1[\backslash v \to e_2]]\!]\,\sigma \;=\; &\textbf{select } (v_1,\sigma_2[v_1/(\textbf{ select } w \textbf{ from } w \textbf{ in } \sigma_2[v_1] \textbf{ where } (f\,w\,\sigma_2)\ )])\\ &\textbf{from } (v_1,\sigma_1) \textbf{ in } \mathcal{T}[\![e_1]\!]\,\sigma,\ (f,\sigma_2) \textbf{ in } \mathcal{T}[\![\lambda v.e_2]\!]\,\sigma_1\end{aligned}$$

There are two places where XML paths can appear in XML-OQL queries: The first place is when a range variable ranges over the elements of an XML-OQL path, as is b in the statement b **in** document("*").*.book in our first example query. In that case, b is bound to the document variable, and when is referenced, the value σ[b] is returned, where σ is the current state. The second place is when the element content returned by an XML-OQL path, e, is coerced to a string or an integer, as is the path b.*.author in the predicate b.*.author = "Smith" in our example query. In that case, the XML-OQL typechecker will wrap e with the proper coercion function, such as, XML_to_string(e) to coerce e to a string. The coerced XML value is translated as follows:

$$\begin{aligned}&\mathcal{T}[\![\text{XML\_to\_string}(e)]\!]\,\sigma \;=\\ &\quad\textbf{select } (\text{ a.word, } \sigma'[v/(\textbf{ select } \text{doc: y.doc, level: y.level, begin\_loc: y.location, end\_loc: y.location}\\ &\qquad\qquad \textbf{from } \text{x } \textbf{in } \sigma'[v], \text{y } \textbf{in } \text{a.occurs}\\ &\qquad\qquad \textbf{where } \text{x.doc} = \text{y.doc } \textbf{and } \text{x.level}{+}1 = \text{y.level}\\ &\qquad\qquad\quad \textbf{and } \text{x.begin\_loc} < \text{y.location } \textbf{and } \text{x.end\_loc} > \text{y.location })]\ )\\ &\quad\textbf{from } (v,\sigma') \textbf{ in } \mathcal{T}[\![e]\!]\,\sigma, \text{ a } \textbf{in } \text{word\_index}\end{aligned}$$

That is, the returned value is a keyword in word_index that satisfies the structural constraints. Since there may be many such keywords, a set of word/binding-list pairs is returned. The idea behind this translation is to generate, for example, a comparison a.word=b.word from a predicate $e_1 = e_2$ that compares two XML paths e_1 and e_2, which will hopefully suggest the use of an efficient equijoin algorithm to the query optimizer. Predicates of the form e cmp c, where e returns an XML value, c is a string constant, and cmp is a string comparison operator, can be handled by the above coercion rule but can be utilized better by the following translation:

$$\begin{aligned}&\mathcal{T}[\![e \text{ cmp } c]\!]\,\sigma \;=\\ &\quad\textbf{select } (\text{ true, } \sigma'[v/(\textbf{ select } \text{doc: y.doc, level: y.level, begin\_loc: y.location, end\_loc: y.location}\\ &\qquad\qquad \textbf{from } \text{x } \textbf{in } \sigma'[v], \text{a } \textbf{in } \text{word\_index, y } \textbf{in } \text{a.occurs}\\ &\qquad\qquad \textbf{where } \text{a.word cmp } c \textbf{ and } \text{x.doc} = \text{y.doc } \textbf{and } \text{x.level}{+}1 = \text{y.level}\\ &\qquad\qquad\quad \textbf{and } \text{x.begin\_loc} < \text{y.location } \textbf{and } \text{x.end\_loc} > \text{y.location })]\ )\\ &\quad\textbf{from } (v,\sigma') \textbf{ in } \mathcal{T}[\![e]\!]\,\sigma\end{aligned}$$

4 A Complete Example

In this section, we synthesize a search query for the following XML-OQL query:

$$\mathcal{T}[\![\ \textbf{select}\ \mathsf{b}\ \textbf{from}\ \mathsf{b}\ \textbf{in}\ \mathsf{document(\text{“}*\text{”}).*.book}\ \textbf{where}\ \mathsf{b.*.author} = \text{“Smith”}\ ]\!]\ \{\ \}$$

According to our standard interpretation rules for a select-from-where OQL statement, the search query is:

$$\begin{array}{l}\textbf{select}\ (\ \textbf{if}\ \mathsf{p}\ \textbf{then}\ \{\ \mathsf{h}\ \}\ \textbf{else}\ \{\ \},\ \sigma_3\ )\\ \textbf{from}\ (\mathsf{b},\sigma_1)\ \textbf{in}\ \mathcal{T}[\![\ \mathsf{document(\text{“}*\text{”}).*.book}\ ]\!]\ \{\ \},\\ \quad\ (\mathsf{p},\sigma_2)\ \textbf{in}\ \mathcal{T}[\![\ \mathsf{b.*.author} = \text{“Smith”}\ ]\!]\ \sigma_1,\\ \quad\ (\mathsf{h},\sigma_3)\ \textbf{in}\ \mathcal{T}[\![\ \mathsf{b}\ ]\!]\ \sigma_2\end{array}$$

We first expand $\mathcal{T}[\![\ \mathsf{document(\text{“}*\text{”})}\ ]\!]\ \{\ \}$ using the first non-standard interpretation rule:

$$\{(\ \text{“d”},\ \{(\ \text{“d”},\ (\textbf{select}\ \mathsf{doc:\ d,\ level:\ 0,\ begin\_loc:\ 1,\ end\_loc:\ d.size}\ \textbf{from}\ \mathsf{d}\ \textbf{in}\ \mathsf{documents})\ )\}\ )\}$$

where “d” is a new document variable. The expression $\mathcal{T}[\![\ \mathsf{document(\text{“}*\text{”}).*.book}\ ]\!]\ \{\ \}$ is expanded to:

$$\begin{array}{l}\textbf{select}\ (\ \mathsf{v},\ \sigma'[\mathsf{v}/(\textbf{select}\ \mathsf{doc:\ y.doc,\ level:\ y.level,\ begin\_loc:\ y.begin\_loc,\ end\_loc:\ y.end\_loc}\\ \qquad\qquad \textbf{from}\ \mathsf{x}\ \textbf{in}\ \sigma'[\mathsf{v}],\ \mathsf{a}\ \textbf{in}\ \mathsf{tag\_index},\ \mathsf{y}\ \textbf{in}\ \mathsf{a.occurs}\\ \qquad\qquad \textbf{where}\ \mathsf{a.tag} = \text{“book”}\ \textbf{and}\ \mathsf{x.doc} = \mathsf{y.doc}\ \textbf{and}\ \mathsf{x.level} < \mathsf{y.level}\\ \qquad\qquad\quad \textbf{and}\ \mathsf{x.begin\_loc} < \mathsf{y.begin\_loc}\ \textbf{and}\ \mathsf{x.end\_loc} > \mathsf{y.end\_loc})]\ )\\ \textbf{from}\ (\mathsf{v},\sigma')\ \textbf{in}\ \mathcal{T}[\![\ \mathsf{document(\text{“}*\text{”})}\ ]\!]\ \{\ \}\end{array}$$

which, using the previous form for $\mathsf{document(\text{“}*\text{”})}$, becomes:

$$\begin{array}{l}\{(\ \text{“d”},\ \{(\ \text{“d”},\ \textbf{select}\ \mathsf{doc:\ y.doc,\ level:\ y.level,\ begin\_loc:\ y.begin\_loc,\ end\_loc:\ y.end\_loc}\\ \qquad\qquad \textbf{from}\ \mathsf{x}\ \textbf{in}\ (\textbf{select}\ \mathsf{doc:\ d,\ level:\ 0,\ begin\_loc:\ 1,\ end\_loc:\ d.size}\ \textbf{from}\ \mathsf{d}\ \textbf{in}\ \mathsf{documents}),\\ \qquad\qquad\quad \mathsf{a}\ \textbf{in}\ \mathsf{tag\_index},\ \mathsf{y}\ \textbf{in}\ \mathsf{a.occurs}\\ \qquad\qquad \textbf{where}\ \mathsf{a.tag} = \text{“book”}\ \textbf{and}\ \mathsf{x.doc} = \mathsf{y.doc}\ \textbf{and}\ \mathsf{x.level} < \mathsf{y.level}\\ \qquad\qquad\quad \textbf{and}\ \mathsf{x.begin\_loc} < \mathsf{y.begin\_loc}\ \textbf{and}\ \mathsf{x.end\_loc} > \mathsf{y.end\_loc}\ )\}\ )\}\end{array}$$

and, after normalization, is simplified into the form:

$$\begin{array}{l}\{(\ \text{“d”},\ \{(\ \text{“d”},\ \textbf{select}\ \mathsf{doc:\ y.doc,\ level:\ y.level,\ begin\_loc:\ y.begin\_loc,\ end\_loc:\ y.end\_loc}\\ \qquad\qquad \textbf{from}\ \mathsf{d}\ \textbf{in}\ \mathsf{documents},\ \mathsf{a}\ \textbf{in}\ \mathsf{tag\_index},\ \mathsf{y}\ \textbf{in}\ \mathsf{a.occurs}\\ \qquad\qquad \textbf{where}\ \mathsf{a.tag} = \text{“book”}\ \textbf{and}\ \mathsf{d} = \mathsf{y.doc}\ \textbf{and}\ 1 < \mathsf{y.level}\\ \qquad\qquad\quad \textbf{and}\ 1 < \mathsf{y.begin\_loc}\ \textbf{and}\ \mathsf{d.size} > \mathsf{y.end\_loc}\ )\}\ )\}\end{array}$$

We then translate the XML-OQL path $\mathcal{T}[\![\ \mathsf{b.*.author}\ ]\!]\ \sigma_1$:

$$\begin{array}{l}\textbf{select}\ (\mathsf{v},\sigma'[\mathsf{v}/(\textbf{select}\ \mathsf{doc:\ y.doc,\ level:\ y.level,\ begin\_loc:\ y.begin\_loc,\ end\_loc:\ y.end\_loc}\\ \qquad\qquad \textbf{from}\ \mathsf{x}\ \textbf{in}\ \sigma'[\mathsf{v}],\ \mathsf{a}\ \textbf{in}\ \mathsf{tag\_index},\ \mathsf{y}\ \textbf{in}\ \mathsf{a.occurs}\\ \qquad\qquad \textbf{where}\ \mathsf{a.tag} = \text{“author”}\ \textbf{and}\ \mathsf{x.doc} = \mathsf{y.doc}\ \textbf{and}\ \mathsf{x.level} < \mathsf{y.level}\\ \qquad\qquad\quad \textbf{and}\ \mathsf{x.begin\_loc} < \mathsf{y.begin\_loc}\ \textbf{and}\ \mathsf{x.end\_loc} > \mathsf{y.end\_loc})])\\ \textbf{from}\ (\mathsf{v},\sigma')\ \textbf{in}\ \sigma_1[\mathsf{b}]\end{array}$$

which, after substitution of $\sigma_1[\mathsf{b}]$ and normalization, becomes:

$$\begin{array}{l}\{(\ \text{“d”},\ \{(\ \text{“d”},\ \textbf{select}\ \mathsf{doc:\ y.doc,\ level:\ y.level,\ begin\_loc:\ y.begin\_loc,\ end\_loc:\ y.end\_loc}\\ \qquad\qquad \textbf{from}\ \mathsf{x}\ \textbf{in}\ \sigma_1[\text{“d”}],\ \mathsf{a}\ \textbf{in}\ \mathsf{tag\_index},\ \mathsf{y}\ \textbf{in}\ \mathsf{a.occurs}\\ \qquad\qquad \textbf{where}\ \mathsf{a.tag} = \text{“author”}\ \textbf{and}\ \mathsf{x.doc} = \mathsf{y.doc}\ \textbf{and}\ \mathsf{x.level} < \mathsf{y.level}\\ \qquad\qquad\quad \textbf{and}\ \mathsf{x.begin\_loc} < \mathsf{y.begin\_loc}\ \textbf{and}\ \mathsf{x.end\_loc} > \mathsf{y.end\_loc})\}\ )\}\end{array}$$

which in turn is simplified to:

$$\begin{array}{l}\{(\ \text{“d”},\ \{(\ \text{“d”},\ \textbf{select}\ \mathsf{doc:\ y.doc,\ level:\ y.level,\ begin\_loc:\ y.begin\_loc,\ end\_loc:\ y.end\_loc}\\ \qquad\qquad \textbf{from}\ \mathsf{d}\ \textbf{in}\ \mathsf{documents},\ \mathsf{b}\ \textbf{in}\ \mathsf{tag\_index},\ \mathsf{z}\ \textbf{in}\ \mathsf{b.occurs},\ \mathsf{a}\ \textbf{in}\ \mathsf{tag\_index},\ \mathsf{y}\ \textbf{in}\ \mathsf{a.occurs}\\ \qquad\qquad \textbf{where}\ \mathsf{b.tag} = \text{“book”}\ \textbf{and}\ \mathsf{d} = \mathsf{z.doc}\ \textbf{and}\ 1 < \mathsf{z.level}\\ \qquad\qquad\quad \textbf{and}\ 1 < \mathsf{z.begin\_loc}\ \textbf{and}\ \mathsf{d.size} > \mathsf{z.end\_loc}\\ \qquad\qquad\quad \textbf{and}\ \mathsf{a.tag} = \text{“author”}\ \textbf{and}\ \mathsf{z.doc} = \mathsf{y.doc}\ \textbf{and}\ \mathsf{z.level} < \mathsf{y.level}\\ \qquad\qquad\quad \textbf{and}\ \mathsf{z.begin\_loc} < \mathsf{y.begin\_loc}\ \textbf{and}\ \mathsf{z.end\_loc} > \mathsf{y.end\_loc})\}\ )\}\end{array}$$

According to the last rule for string comparisons, $\mathcal{T}[\![$b.*.author = "Smith"$]\!]\,\sigma_1$ is equal to:

```
select ( true, σ'[v/( select doc: y.doc, level: y.level, begin_loc: y.location, end_loc: y.location
                      from x in σ'[v], a in word_index, y in a.occurs
                      where a.word = "Smith" and x.doc = y.doc and x.level+1 = y.level
                        and x.begin_loc < y.location and x.end_loc > y.location )] )
from (v, σ') in T[[b.*.author]] σ1
```

and becomes after simplification:

```
{( true, {( "d", select doc: y.doc, level: y.level, begin_loc: y.location, end_loc: y.location
                 from d in documents, b in tag_index, z in b.occurs,
                      c in tag_index, w in c.occurs, a in word_index, y in a.occurs
                 where b.tag = "book" and d = z.doc and 1 < z.level
                   and 1 < z.begin_loc and d.size > z.end_loc
                   and c.tag = "author" and z.doc = w.doc and z.level < w.level
                   and z.begin_loc < w.begin_loc and z.end_loc > w.end_loc
                   and a.word = "Smith" and w.doc = y.doc and w.level+1 = y.level
                   and w.begin_loc < y.location and w.end_loc > y.location )} )}
```

Finally, we put together all the components to form the final translation:

```
{( {"d"}, {( "d", select doc: y.doc, level: y.level, begin_loc: y.location, end_loc: y.location
                  from d in documents, b in tag_index, z in b.occurs,
                       c in tag_index, w in c.occurs, a in word_index, y in a.occurs
                  where b.tag = "book" and d = z.doc and 1 < z.level
                    and 1 < z.begin_loc and d.size > z.end_loc
                    and c.tag = "author" and z.doc = w.doc and z.level < w.level
                    and z.begin_loc < w.begin_loc and z.end_loc > w.end_loc
                    and a.word = "Smith" and w.doc = y.doc and w.level+1 = y.level
                    and w.begin_loc < y.location and w.end_loc > y.location )} )}
```

5 Extensions

Attribute projections, $e.@A$, in our framework are treated in the same way as tag projections. An extra boolean attribute can be added to the class XML_tag to distinguish a tag from an XML attribute and to make the search more accurate. The XML-OQL syntax, though, allows IDref dereferencing, as is done for r.title in the inner *select*-statement of the example query in Section 2, since variable r is an IDref that references a book element. IDref dereferencing requires a special index on all IDs of all web-accessible XML documents:

```
class XML_IDref ( key ID extent IDref_index )
{   attribute string ID;
    attribute set⟨ tag_spec ⟩ occurs; };
```

To handle patterns in URLs, as in document("http://*.edu/*/technical-reports/*"), which retrieves XML documents from the .edu domain in some technical-reports directory, can be handled using the following index:

```
struct url_spec { Document doc; integer level; };

class XML_url ( key name extent url_index )
{   attribute string name;
    attribute set⟨ url_spec ⟩ occurs; };
```

For example, the above URL pattern will require an OQL query that intersects all documents in url_index associated with the name edu with all those associated with the name technical-reports satisfying the appropriate containment restriction.

One assumption made earlier was that equalities between XML paths are between keywords, which are indexed by the inverse index, word_index. But suppose that we want to search using exact content, such as, b.title="A Search Engine for Complex XML Queries", or perform pattern matching, such as, b.title like "* Search Engine * XML Queries". These queries too can be evaluated with self-joins over word_index, by ignoring non-keywords and specifying containment constraints for the keywords.

6 Conclusion

We have presented a general framework for a precise indexing of web-accessible XML data. Unlike related proposals, our framework can be applied to complex XML queries. For any XML-OQL query, our framework is capable of generating one OQL query over the XML inverse indexes to return all the relevant web-accessible XML document fragments that satisfy the XML query. We are planning to implement our framework on top of the λ-DB OODB management system, which already handles semi-structured XML-OQL queries over database-resident XML data.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] L. Fegaras. A New Heuristic for Optimizing Large Queries. In *9th International Conference, DEXA'98, Vienna, Austria*, pages 726–735. Springer-Verlag, Aug. 1998. LNCS 1460.

[9] L. Fegaras and R. Elmasri. Query Engines for Web-Accessible XML Data . To appear in VLDB 2001. Available at `http://lambda.uta.edu/vldb01.ps.gz`, 2001.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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*, volume 1949 of *Lecture Notes in Computer Science*, pages 297–323. Springer, Sept. 1999.

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

[16] 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.