XML Retrieval in the Classroom

Steven M. Beitzel, Eric C. Jensen, Angelo J. Pilotto, Ophir Frieder, David Grossman [1]
Information Retrieval Laboratory
Illinois Institute of Technology

There are currently a variety of systems for searching XML. However, they are typically research tools or specialized commercial products that have limited usability in the classroom. The SQLGenerator is a complete implementation of the XML-QL query language that translates queries to SQL for execution by an underlying relational database. It is unique in that it is a scalable, portable implementation of a semi-structured query language, that is also open source and extremely well-documented by a 140-page user's and devloper's manual. It allows students to see the logic of their semi-structured queries represented in the familiar relational language of SQL. In contrast to many of the relational mapping tools for XML, the SQLGenerator operates on a fixed relational schema that conceptually represents the hierarchical nature of XML data of any schema without requiring XSchemas or DTD's. A case study of using the SQLGenerator to teach the concepts of semi-structured search to a graduate level database class is presented. The SQLGenerator source code, along with all of its documentation and a demo interface, can be found at http://ir.iit.edu/sqlgen.

Categories and Subject Descriptors: D.2.7 [**Education**]: Education

General Terms: XML, SQL, Education

Additional Key Words and Phrases: XML, SQL, query languages

Introduction

The eXtensible Markup Language (XML) has become the standard for platform-independent data exchange. Its semi-structured nature provides the basis for many existing data integration solutions [6; 10]. Despite its extensive practical applications, the concepts of semi-structured data and search are often briefly noted in existing Computer Science curricula at best. Courses dealing with how to actually implement semi-structured search systems, much less scalable ones, are rare. One reason for this may be the lack of XML retrieval systems and suitable course material available for use in the classroom. Most systems that are publicly available are research systems that are insufficiently documented or specialized commercial products (such as relational database mapping tools) that do not address the general problem of XML search with a semistructured query language. Neither of these approaches are suited for presenting the fundamentals of semi-structured search in a classroom setting. The SQLGenerator is implemented as a platform-indepedent (tested on Solaris, Linux, and Windows with MySQL) system using only Java and ANSI SQL, with its full indexing and query-translating source code available under the Lesser GNU General Public License (LGPL). A regression test suite of 120 queries is provided to ensure functionality and provide a large number of example queries and their known results. Most importantly, it is extremely well documented

[1]Computer Science Department, 10 W 31 St., Chicago, IL 60616. {beitzel,jensen,pilotto}@ir.iit.edu

both in the sense of its query language (documented in its own manual and by the standards and tutorials publicly available) and its internal operation, making it suitable for use as both a system for students to execute XML-QL queries and as a system that can be modified in class projects.

The query languages that are implemented by many of the available systems are either lacking many of the features desirable for semistructured query languages (such as XPath) or nonstandard, often unintuitive, proprietary languages. There are a large number of proposed query languages for XML data [8; 26; 2; 3; 16]. The SQLGenerator system implements XML-QL, a query language developed by AT&T [7]. XML-QL was designed to meet the requirements of a full-featured XML query language set out by the World Wide Web Consortium [27]. In addition to its range of capabilities, it provides an intuitive means of writing semi-structured queries resembling SQL that uses XML data bindings in a format very similar to the XML documents being searched. XML-QL has been used to teach graduate students about searching XML at several universities including Concordia University, University of Pennsylvania, and University of Southern California[2], all using the AT&T reference implementation that parses XML data at query time and translates XML-QL queries to StruQL queries for execution by their Strudel semi-structured query engine. While this engine is the reference point for the XML-QL standard, it is not designed to be a teaching system. The error reporting it provides is most often too complex for students to understand and it is lacking in documentation, especially that which would be needed for use in class projects. By contrast, our system, with its open source code and extensive documentation, is well-suited for use in the classroom, especially by undergraduates.

There have been a variety of methods proposed for storing XML data and accessing it efficiently [5; 1]. One approach is a customized tree file structure, but this lacks portability and doesn't leverage existing database technology [12]. Other approaches include building a database system specifically tailored to storing semistructured data from the ground up [14; 15] or utilizing common information retrieval structures unsuited for full-featured semistructured search [20; 21]. The SQLGenerator fully implements XML-QL by translating its semistructured queries to SQL queries for execution on a static relational schema. Although work has been done on defining static schemas for XML storage, we are unaware of an algorithm for translating all queries in a full-featured semi-structured query language to SQL that retrieves the correct results from those schemas. By storing XML documents in a fixed-schema relational database, we are able to utilize the performance advantages of over 30 years of research in the database community, while fully supporting the rich semistructured features of the XML-QL query language [4; 22; 17; 19]. This makes it scalable not only in terms of collection size, based almost solely on database performance, but by using a parallel database we are implicitly able to scale the number of nodes for query processing, based on the database's scalability. The impact of this in the classroom is that it provides a realistic solution to searching large collections of XML that maintains an intuitive, familiar view of the logic behind semistructured queries. Its static schema design accommodates data of

[2]http://www.cs.concordia.ca/ grad/thomo, http://db.cis.upenn.edu/XML-QL, http://infolab.usc.edu/csci585/Spring2001/hw3.pdf

any XML schema without the need for document-type definitions [24] or XSchemas [28], allowing students to draft their own XML and experiment with searching it. By implementing XML-QL, it uses a standard, intuitive query language. In section 2 we present a case study of our experiences with SQLGenerator in the classroom. Section 3 presents a brief overview of the properties of XML and how to write valid XML documents. Section 4 describes in detail the syntax XML-QL and demonstrates how to use its features to query sample XML documents. Finally, section 5 is dedicated to SQLGenerator. The storage schema is explained as well as the query translation process.

Case Study

Recently, we extended the undergraduate curriculum at IIT to include information retrieval and data mining [11]. In Fall, 2002 we began teaching XML retrieval as a topic in the graduate advanced database course at IIT.

Lesson Plan

The lessons consist of an introduction to semistructured data in the form of XML and to semistructured search using XML-QL. Initially we refresh the students on the nature of searching structured and unstructured data. Although they work all semester with structured data, we have found that many did not make the distinction between this and other data types. We exemplify these differences by a comparison to unstructured data in the form of plain text. We briefly show that it can be modeled as a bag of words for which a similarity score to a bag of query words can be found. Students easily understand the simplest example of an unstructured data query as something they might type into a popular search engine such as Google^{TM}. The key steps in the lesson plan are to introduce students to the basics of XML, have them author their own simple XML documents, and finally query their documents using XML-QL.

XML is presented as a type of semistructured data that posseses structured attributes, such as data within a document conforming to a specific DTD, but also exhibits unstructured properties because the contents of an element can be other unspecified (possibly recursive) elements or textual data. The example XML document presented to the students describes the members of a family where each member is described as a *person* element. This simple XML document is used to examine issues differentiating semistructured data from the structured (relational and object-oriented) databases they had previously learned about. Four important terms that we emphasize are: **element**, **attribute**, **tag**, and **markup**. These defined terms are used to explain the XML standard and why standards are important for data interchange. To give students a familiar example, we point to a number of real-world applications (e.g. e-commerce, file sharing programs, etc.).

Next, students given an assignment to create their own XML documents. A loose standard format for each of the student documents is discussed in class, defining the names of some example elements and what they might contain. The assignment is based on the autobiography examples supplied in class; each student creates their own XML autobiography including some additional elements such as education, family, and date of birth. Once a student has composed their XML file, they use the indexing feature of SQLGenerator so that it that can be queried using XML-QL

through a web interface.

In authoring their own XML the students achieve an understanding of the structure of an XML file and are familiar with XML terminology. They are then introduced to the basics of XML-QL. Students are taught that XML-QL consists of two basic clauses, WHERE and CONSTRUCT. A WHERE clause allows them to bind variables and pose constraints on those variables. A CONSTRUCT clause creates XML results from the variables obtained in a WHERE clause. An important concept is the idea of XML having paths within the document from the root element to some final value. Paths represent the structured properties of an XML document because they allow a user to take advantage of the known structure of a document to obtain values that occur at the end of a given path. A single path may occur multiple times within a document with different values found at the end of each path occurrence. We present the concept of a *regular path expression*, which presents a pattern for matching to a set of paths. For example, a single path expression represents a set of actual paths within the XML document. In XML-QL, variables can be bound to the values of paths. We explain that it is fundamental to use this feature of XML-QL when constructing queries. Consider the small XML fragment shown in Figure 1.

We see the path "[person, name]" has a value of "John Smith" at the end. A variable bound to a path can take on multiple values and that a query must be thought of as an iteration over matching paths of the XML document. The students can see that it is desirable to be able to bind a variable to the value at the end of the path so that the value can be used to construct XML later in the query.

We then present some example queries which operate on the autobiography XML presented. The results of the queries are explained so that the student clearly sees how XML-QL operates on the data it is querying and how the variables are used to create resultant XML. Finally, students are ready to construct their own queries and are assigned five query concepts to write in XML-QL and execute to obtain results. The student creates the query based on a question posed in English that can be answered from the set of XML documents written previously by the students themselves.

Interface

Having a web interface to show the execution of an XML-QL query also aids in the teaching process. The web interface provided is simple to use and uses a Java servlet to provide the results. It consists of an input box that accepts the query and a button that initiates the execution of the query. Once the student provides their XML file and it is indexed into the database, they can use the servlet interface to query their XML document. The servlet was especially useful as it allowed students to learn XML-QL by trial and error. The source code for the servlet is included with the rest of the SQLGenerator package. A screenshot of the servlet interface can be seen in Figure 2.

The query is executed when the button is clicked and user is redirected to a page (Figure 3) that displays the results if the XML-QL query was well-formed. In the case of syntax errors, a detailed error message is provided in order to aid the student in correcting the error and therefore further learning XML-QL syntax.

Fig. 1. XML Fragment

```
<Person>
    <Name>Joe Smith</Name>
    <Birthplace>
        <City>Honolulu</City>
        <State>HI</State>
        <Date Format="ISO 8601">1981-10-05</Date>
    </Birthplace>
    <Residence>
        <City>Honolulu</City>
        <State>HI</State>
    </Residence>
    <Residence>
        <City>Chicago</City>
        <State>IL</State>
    </Residence>
    <Education>
        <University>
            <Name>Illinois Institute of Technology</Name>
            <City>Chicago</City>
            <State>IL</State>
        </University>
        <High_School>
            <Name>University Laboratory School</Name>
            <City>Honolulu</City>
            <State>HI</State>
        </High_School>
    </Education>
    <Family>
        <Parents />
        <Siblings />
    </Family>
</Person>
```

Findings

The largest amount of teaching effort required was for introducting the students to the basic concepts and motivations behind semi-structured data and query languages. Once these basics were covered then a sample XML document and XML-QL query answered most of the questions students had.

After they completed the basics of the assignment, many of the students explored the XML their fellow students had written by exploiting the various features of XML-QL. They were curious to see the additions other students had made to the basic autobiography schema. For example, several students added elements which indicated where they were from, or included recursive *person* tags giving information about parents or siblings. In exploring, they also realized the challenges that dynamic schemas can pose, and the increasing complexity necessary in semistructured queries to accomodate them.

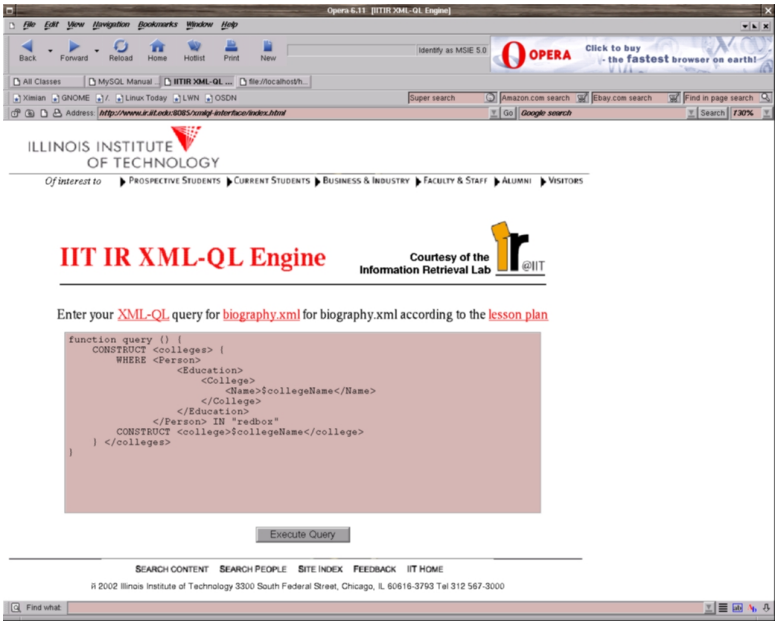

Fig. 2. The XML-QL Interface

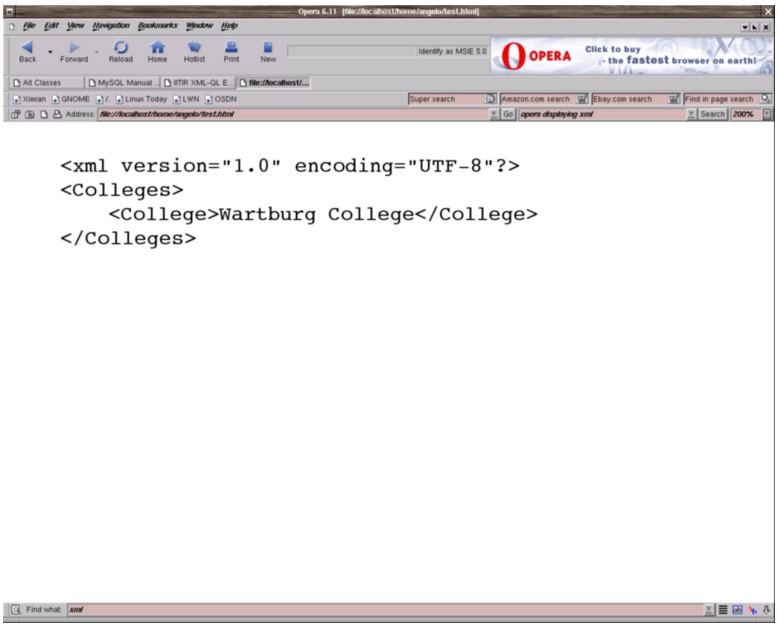

Fig. 3. The results

```
<Instruments>
    <Instrument>
        <name>Violin</name>
        <lowest_note>
            <octave>2</octave>
            <note>G</note>
        </lowest_note>
        <length unit="in">14</length>
    </Instrument>
    <Instrument>
        <name>Viola</name>
        <lowest_note>
            <octave>2</octave>
            <note>C</note>
        </lowest_note>
        <length unit="in">16</length>
    </Instrument>
    <Instrument>
        <name>Cello</name>
        <lowest_note>
            <octave>1</octave>
            <note>C</note>
        </lowest_note>
    </Instrument>
<Instruments>
```

Fig. 4. `string.xml`.

XML

A Brief Primer

Figure 4 is an example of an XML file. This file describes instruments: string instruments, in particular, although there is no reason why the document could not be extended to include wind, percussion, or any other type of instrument. Many XML search systems model XML as a tree. We can represent the XML in Figure 4 as a tree such as the one in Figure 5. Here the o_i values denote object identifiers (OIDs). These values are assigned using a depth first traversal of the XML document. We see that assigning the IDs in this manner does denote some type of order, however, there is no reason why the `<Instrument>` tag for violin should come before the tag for viola. The tags (starting with a `<` and ending with a `>`, like `<Researcher>`) wrap around human-readable text that the markup around it describes. Note that each tag is paired with a closing tag of the same name with a `/` before it. XML requires all opening tags to be matched with closing tags. An **element** is the entire fragment of the file from an opening to a closing tag. `string.xml` in its entirety can be referred to as an `<Instruments>` element. XML elements are ordered: unlike relations, it does matter which elements come before others. That means that we can say that the `<Instrument>` tag talking about violas is the second **child** element of the **root** element `<Instruments>`, where child elements are elements that are beneath so-called **parent** elements, and root

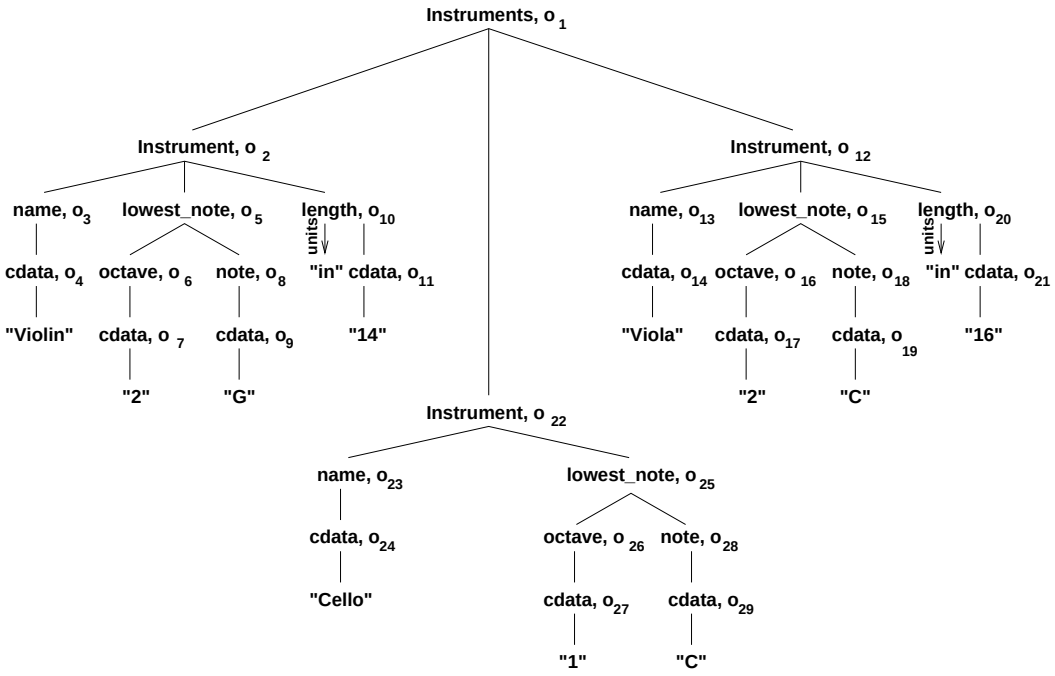

Fig. 5. The XML Data Model

elements are the top-most elements of the document. All XML documents have one and only one root element. Some tags also contain optional **attributes**, which are *unordered* key-value pairs, unlike elements, which are ordered. In other words, writing `<Tag a="1" b="2">` is the same thing as `<Tag b="2" a="1">`; additionally, attribute `a` of tag `Tag` has a value of 1 while attribute `b` has a value of 2. Because they are unordered, neither `a` nor `b` is the first or second attribute as far as computers are concerned.

XML is a markup language standard, but the tag names themselves are not part of the standard. However, through the use of Document-Type Definitions (DTDs) or XML Schemas, these tags and their ordering can be restricted. DTDs define rules for how elements and attributes of an XML file are supposed to be enclosed in one another. For example, a DTD for `string.xml` might stipulate that the `<Instrument>` element may only appear as children of the `<Instruments>` element.

Overview of Existing Query Languages

There are several popular XML query languages in use today [13]. XML-QL, the semistructured language of our particular system, was developed by a team of AT&T researchers in 1998. In August 1997, initial work on XPath, likely the most basic path-based query language, was submitted to the W3C [23]. The specification describing XPath as it is known today was released on July 9, 1999. In December 2001, XPath 2.0 was released. One of the newest semistructured query languages,

XQuery, is also among the most powerful. It borrows many ideas from prior work on other semistructured query languages such as XML-QL and XPath, as well as from relational query languages like SQL. The first public draft of the XQuery 1.0 spec was released in June 2001.

Existing XML Retrieval Systems

The majority of systems freely available are unrealistic solutions to the problems of searching large volumes of XML with varying schemas, These are systems that use in-memory indexes. Many of them require XML that adheres to specified XSchemas [28] or document-type definitions (DTD's) [25; 24], which provides an obstacle to teaching students because they do not learn an XML query language. In contrast, the SQLGenerator accomodates XML of any XML schema with a single, fixed relational schema. Unlike in-memory systems, such as AT&T's XML-QL implementation, the SQLGenerator uses a relational database for persistent storage; parsing and indexing of XML data is performed before query time. This allows us to scale to collection sizes much larger than the physical memory of the system available. Unlike the relational mapping tools provided by many of the major database vendors, changes in the schema of XML data being stored do not require changes in the underlying relational schema, and a fully-functional semistructured query language is provided.

XML-QL

Data Model

Although elements in XML are ordered, the default data model for XML-QL does not retain order. As most relational databases also use unordered tuples that can be explicitly sorted or grouped as needed at query time, this property allows the SQLGenerator to make assumptions that are very useful for translating XML-QL queries to SQL.

Basics

All XML-QL queries return a block of XML as their results. Every query must contain a `CONSTRUCT` clause, which specifies a template for building XML query results. The trivial XML-QL query in Figure 6 doesn't actually query over XML; it merely specifies XML to build. Because well-formed XML documents have a single root element, XML-QL wraps the result tags inside of an artificial `<XML>` root element when no root element is defined.

Variables and the `WHERE` Clause

`CONSTRUCT` clauses only format retrieved data. The real power of XML-QL lies in the `WHERE` clause. A `WHERE` clause cannot exist without a `CONSTRUCT` clause, as `WHERE` only specifies what data to look for and not how to format query results. A `WHERE` clause contains a comma-separated set of **conditions** that can be imposed on the query to limit results. Conditions appear in two forms: tag patterns, where results that fit a specific XML form are retrieved, and predicate conditions, where mathematical comparisons or string `LIKE` comparisons can be performed on variables. Most useful queries such as the one in Figure 7 use `WHERE` clauses with

```
Query:
CONSTRUCT <result>Hello world.</result>
          <!-- And now for some math... -->
          <result>10 &gt; 4</result>

Results:
<?xml version="1.0" encoding="UTF-8"?>
<XML>
    <result>Hello world.</result>
    <!-- And now for some math... -->
    <result>10 > 4</result>
</XML>
```

Fig. 6. `simple.xmlql` and its output.

tag-patterns and variables. In this example, the variable `$b` matches on all character data that occurs in a `<name>` tag that is a child of the `<book>` element that is a child of the `<books>` root element. Similarly, the variable `$p` is bound to the character data for the `<price>` tag. Finally, a predicate condition is imposed that requires result books to have a price greater than 5 units (in this case, US Dollars). A file that this query might be applied to, `books.xml`, and its corresponding results are also shown in Figure 7. An XML tag can either contain more XML tags or character data. The contents of an XML tag are said to be complex when they contain more XML tags. Complex tags cannot be compared to one another like character data can. To explicitly request the character contents of an XML tag instead of its complex contents, one must append .PCDATA to the tag name in the XML-QL path expression. A shorthand for closing tags in path expressions is `</>`. Many strings that substitute for tag names in XML-QL (such as regular path expressions) are not true tag names, and actually require this type of closing tag. The tag pattern in the WHERE clause of `books.xmlql` (taken from Figure 7) specifies that for all XML character data that exist in `<name>` only under `<book>` only under `<books>`, `$b` will represent this value and the `CONSTRUCT` template will be repeated as many times as necessary to display all the possible bindings of `$b`.

The XML-QL processor needs to know where to look for these tags. This is accomplished by specifying a file name: `IN "books.xml"`. The keyword `IN` is used to specify the data source for the given tag pattern. Figure 8 shows examples of `WHERE` clause data sources: external source (files, which can be local files or absolute or relative URLs), a variable bound to XML, or a variable bound to a file location. In those examples, `books.xml` is the name of a file, `http://www.mylibrary.net/books.xml` is a URL where a file can be found, `$q` is bound to some XML fragment, and `$filename` is bound to the URL or local reference of a file. Obviously, as we're indexing the XML data ahead of time and storing it persistently in a relational database, one can also choose to query over all indexed XML.

Advanced `CONSTRUCT` clauses

The `CONSTRUCT` clause (or any element that appears in a `CONSTRUCT` clause) may contain expressions, elements, or query blocks. Expressions are simply mathemati-

```
Query:
  WHERE <books>
            <book>
                <name.PCDATA>$b</>
                <price.PCDATA>$p</>
            </book>
        </books> IN "books.xml",
        $p > 5
  CONSTRUCT <result>$b</result>

Document:
  <books>
      <book>
          <name>The Great Gatsby</name>
          <author>F. Scott Fitzgerald</author>
          <price currency="USD">9.99</price>
      </book>
      <book>
          <name>Cat in the Hat</name>
          <author alias="true">Dr. Seuss</author>
          <price currency="USD">14.99</price>
      </book>
      <pamphlet>
          <name>Common Sense</name>
          <author>Thomas Paine</author>
      </pamphlet>
  </books>

Results:
  <?xml verson="1.0" encoding="UTF-8"?>
  <XML>
      <result>The Great Gatsby</result>
      <result>Cat in the Hat</result>
  <XML>
```

Fig. 7. `books.xmlql`, `books.xml`, and the output of the query.

```
WHERE <...>
    IN "books.xml"
WHERE <...>
    IN "http://www.mylibrary.net/books.xml"
WHERE <...>
    IN $q
WHERE <...>
    IN SOURCE($filename)
```

Fig. 8. Examples of `WHERE` clause data sources.

```
WHERE <books>
          <book>
              <name.PCDATA>$n</>
              <price.PCDATA>$p</>
          </book>
      </books> IN ``books.xml''
CONSTRUCT <result>
              <name>$n</name>
              <tax>$p * 0.05</tax>
          </result>
```

Fig. 9. `books2.xmlql`

cal formulas where numbers, bound variables, or values of function calls are added, subtracted, multiplied, or divided. The query `books2.xmlql` of Figure 9 computes 5% tax on the prices of the books in *books.xml*. For every name-price pair encountered, the name is returned unchanged and the price is multiplied by 0.05. The `PCDATA` expression is absolutely essential here—without it, the processor would attempt to take 5% of a complex tag, when in fact, we very specifically want the character data within the tag.

Figure 11 shows the usage of a variable as a tag name, and an XML attribute. Recall that the closing tag of any element can be expressed as `</>`. In this `CONSTRUCT` clause, `</>` is actually necessary because `$t` can potentially be bound to anything. The XML result contains the closing tag appropriate to the matching tag name. The inner tag has a name specified by `$t`. The value of this bound variable can be stored and later used in a `CONSTRUCT` statement as contents of a tag or the name of a tag. The attribute eatsVeggies will be constructed with all values of `$f`.

Advanced WHERE clauses

In addition to basic tag patterns, regular tag path expressions allow for searching across multiple tag names. Say, for example, we wanted to extract only pamphlets and magazines from `books2.xml`. We could either write two separate queries since our sought-after results are in two different tag paths, or write one query that encapsulates both types of tags as seen in Figure 10. If one wants to retain the name of the tag that matched, the tag name variables and predicate conditions are used. As seen in Figure 11, Regular path expressions are composed of three operators, listed in order of precedence, lowest first:

(1) Alternation ("|"). Matches an occurrence of any of the expressions separated by the vertical bar. `<cats|dogs>` matches "`<cats>`", "`<dogs>`", but not "`<cats><dogs>`"

(2) Concatenation (“.”). Matches the concatenation of the expressions separated by periods. `<$t.$x>` matches "`<cat><dog>`" if `$t`="`<cat>`" and `$x`="`<dog>`"

(3) Kleene operators (“+”, “?”, “*”). Matches one or more, zero or one, or zero or more occurrences, respectively. `<cat*>` matches "", "`<cat>`", "`<cat><cat>`", `<dog>` matches "" or "`<dog>`" but not "`<dog><dog>`" `<bird+>` matches "`<bird>`", "`<bird><bird>`", not ""

Fig. 10. A query that captures both kinds of tags.

```
CONSTRUCT <results> {
WHERE <books>
          <magazine|pamphlet>
              <name.PCDATA>$n</>
              <price.PCDATA>$p</>
          </>
      </books>
CONSTRUCT <result>
              <name>$n</name>
          </result>
} </results>
```

Fig. 11. Use of tag name variables and predicate conditions

```
CONSTRUCT <results> {
    WHERE <books>
              <$t>
                  <name.PCDATA>$n</>
              </>
          </books> IN "books2.xml", $t IN {magazine, pamphlet}
    CONSTRUCT <result type=$t>
                  <name>$n</name>
                  <price>$p</price>
              </result>
} </results>
```

Fig. 12. Binding Tag Contents to a Variable

```
CONSTRUCT <results> {
    WHERE <books>
              <$t>
                  <name.PCDATA>$n</>
                  <price.PCDATA>$p</>
              </> ELEMENT_AS $e
          </books> CONTENT_AS $c IN "books2.xml", $p > 10, $n LIKE "Dr*"
    CONSTRUCT $t
} </results>
```

Tag patterns can also bind their contents or their entire elements to variables. The query in Figure 12, we see that `$c` and `$e` are both bound to the matching `<$t>` tag, illustrating `ELEMENT_AS` and `CONTENT_AS` in action.

Fig. 13. Using an Aggregate Predicate in the CONSTRUCT

```
Query:
CONSTRUCT <results>
        <cheapest_price>MIN($p)</cheapest_price> {
            WHERE <books>
                      <$t>
                          <name.PCDATA>$n</>
                          <price.PCDATA>$p</>
                      </> CONTENT_AS $e
                  </books> IN "books2.xml"
                  CONSTRUCT <result type=$t>$e</result>
} </results>

Results:
<?xml version="1.0" encoding="UTF-8"?>
<results>
    <cheapest_price>3.95</cheapest_price>
    <book>
        <name>The Great Gatsby</name>
        <author>F. Scott Fitzgerald</author>
        <price currency="USD">9.99</price>
    </book>
    <book>
        <name>Cat in the Hat</name>
        <author alias="true">Dr. Seuss</author>
        <price currency="USD">14.99</price>
    </book>
    <magazine>
        <name>Time</name>
        <issue>January 14, 2002</issue>
        <price currency="USD">3.95</price>
    </magazine>
<results>
```

Aggregate Predicates

Aggregate predicates are special functions that appear in the contents of an element in a `CONSTRUCT` clause. With the exception of `COUNT()`, the aggregate predicates aggregate over all the values that a particular bound variable can have. The query in Figure 13 produces the cheapest price in `books2.xml`, followed by the matches. The `PCDATA` expression is required here; aggregations of complex tag patterns are not supported. Notice that `<pamphlet>` is not shown. There is no `<price>` in `<pamphlet>`, and therefore `<pamphlet>` does not match the specified tag pattern. The numeric aggregate predicates supported by XML-QL are very similar to their SQL counterparts; MIN, MAX, AVG, and SUM. `COUNT(*)` is a special aggregate predicate that returns the number of tags that are at the same level as its enclosing tag, not counting its own enclosing tag.

The SQLGenerator

The SQLGenerator is a scalable XML retrieval engine that fully implements the XML-QL query language by translating it to SQL. It incorporates XML documents of any schema without requiring modifications to the fixed underlying relational schema they are stored in. XML need not specify a schema or DTD, but rather incoming XML with dynamically changing schemas is immediately searchable via a full-featured semi-structured query language.

The Relational Database Schema

Our storage schema stores each unique XML path and its value from each document as a seperate row in a relational table. This is similar to the edge table described by Florescu and Kossman, named for the fact that each row corresponds to an edge in the XML graph representation [9]. Our edge table has the values inlined (in the same table). This is a static schema that is capable of storing any XML document without modification. The hierarchy of XML documents is kept in tact such that any document indexed into the database can be reconstructed using only the information in the tables. We also support the partitioning of this edge table by selected paths, producing a schema similar to a partially decomposed variant of Florescu's binary scheme. This allows for the seperation of key paths into their own tables, often enabling the database to build more effective indexes due to the increased homogeneity of the data in those paths and the obvious data partitioning advantages. Unlike the pure binary scheme, however, it does not require a table for every unique path in every XML document indexed. We shall use `books.xml` (see Figure 7) as an example file to show our storage scheme and the SQLGenerator's translation over it. All of the database tables shown in examples will reflect the data of this file.

`tagnametbl`, `tagpathtbl`, and `atrnametbl`. These tables store the metadata (data about the data) of the XML files. `tagnametbl` (Figure 14) and `tagpathtbl` (Figure 15) together store the information about tags and paths within the XML file. `tagnametbl` stores the name of each unique tag in the XML collection. `tagpathtbl` stores the unique paths found in the XML documents. `atrnametbl` (Figure 16) stores the names of all the attributes. In each of these tables, `vkey` is an integer assigned by the system and is the primary key of the table. These tables are populated each time a new XML file is indexed. This process consists of parsing the XML file and extracting all of this information and storing it into these tables.

vkey	value
1	books
2	book
3	name
4	author
5	price
6	pamphlet

Fig. 14. The `tagnametbl` Table

vkey	value
1	[books]
2	[books, book]
3	[books, book, name]
4	[books, book, author]
5	[books, book, price]
6	[books, pamphlet]
7	[books, pamphlet, name]
8	[books, pamphlet, author]

Fig. 15. The `tagpathtbl` Table

vkey	value
1	-
2	currency

Fig. 16. The `atrnametbl` Table

pinfiles. Since XML-QL allows users to specify what file(s) they wish to query, frequently we do not want to look at each record in the database but only a subset of records that correspond to that file. Each time a new file is indexed, it receives a unique identifier that is known as the `pin` value. This value corresponds to a single XML file. The `pinfiles` table creates a link from the `pin` value to the name of the XML document. When a user specifies an XML document whose `pin` value happens to be 5, the SQL generated needs to have a condition that only entries with this `pinnum` should be examined. An example entry in this table is shown in Figure 17.

vkey	value
1	books.xml
2	iit-ir.xml

Fig. 17. The `pinfiles` Table

`pinndx`

The `pinndx` table (Figure 18) stores the actual contents of all of the XML files that have been indexed. The `pinndxnum` column is a unique integer assigned to each element and attribute in a document. The `parent` column indicates the `pinndxnum` value of the tag that is the parent of the current tag. The `tagpath` entry is the value corresponding to the primary key value in the `tagpathtbl`. The `tagtype` column indicates whether the path terminates with an element or attribute. The `tagname` and `atrname` values correspond to the primary key values in the `tagnametbl` and `atrnametbl` tables respectively. The `pinnum` column indicates the XML document this row corresponds to. The `indexpos` column is used for queries that use the index expression feature of XML-QL and indicates the position of this element relative to others under the same parent (starting at zero). This column stores the original ordering of the input XML for explicit usage in users' queries.

pinndxnum	parent	tag-path	tag-type	tag-name	atr-name	pin-num	index-pos	value
1	0	1	E	1	1	1	0	
2	1	2	E	2	1	1	0	
3	2	3	E	3	1	1	0	The Great Gatsby
4	2	4	E	4	1	1	0	F. Scott Fitzgerald
5	2	5	E	5	1	1	0	9.99
6	5	5	A	5	2	1	0	USD
7	1	2	E	2	1	1	0	
8	7	3	E	3	1	1	0	Cat in the Hat
9	7	4	E	4	1	1	0	Dr. Seuss
10	7	5	E	5	1	1	0	14.99
11	10	5	A	5	2	1	0	USD
12	1	6	E	6	1	1	0	
13	12	7	E	3	1	1	0	Common Sense
14	12	8	E	4	1	1	0	Thomas Paine

Fig. 18. The **pinndx** Table

The Translation Process

While prior work has defined methods for storing XML in relational databases, it has not shown that all features of a rich semistructured query language such as XML-QL can be translated into equivalent SQL for the given schemas. Using the relations described above, we can translate any XML-QL query into the appropriate SQL that will return all of the information requested by the user. Data returned by the translated SQL query is used to create a DOM document object for representing the XML results. The general algorithm for processing a query block is shown in Figure 21. This represents a high level view of the inner workings of SQLGenerator and does not go into detail on the translation of many of the advanced features of XML-QL. The algorithm processes a query block which is a part of the XML-QL query that contains a CONSTRUCT clause and optional WHERE clause. Initially the CONSTRUCT clause is processed by examining each element in the clause. Elements or the contents of the clause are usually an XML element or variable that is used to create a new XML document. These elements are put into a template that is used later in the XML construction process. The optional WHERE clause is processed next. If the element of the WHERE clause is a tag pattern, this is stored in a special data structure. If the element is a predicate condition, this is stored as a string that can be appended to the generated SQL. Once each tag pattern has been stored, path creation and resolution is the next step. A tag pattern can represent multiple paths, finding these paths is the path creation process. A path can contain an expression to represent a real path, such as a Kleene star, finding all possible paths is path resolution. Once the paths and predicates are known, the SQL can be generated. The results obtained by the SQL are used in conjunction with the template created from the CONSTRUCT clause to create the new XML. Consider a query shown in Figure 19.

In the query, we see a CONSTRUCT clause enclosing a nested block. When the SQLGenerator encounters a CONSTRUCT clause it begins to process its contents. In this example, the contents consist of another query block, which will be stored

```
CONSTRUCT <authors> {
  WHERE <books>
          <book>
            <author.PCDATA>$author</author>
          </book>
        </books> IN "books.xml"
  CONSTRUCT <author>$author</author>
}
```

Fig. 19. A Basic XML-QL Query

for execution until after the outer query block has been processed. Since the outer block is a single CONSTRUCT clause, SQLGenerator executes its algorithm again on the nested query block. The example query specifies a single `<authors>` tag as the root of the XML results, and under it will be a variable number of `<author>` tags depending on the number of authors found in books.xml. When the the resulting authors are obtained from the translated SQL query, the template for each author specified by the CONSTRUCT is filled in. As stated above, a WHERE clause consists of a number of conditions. In this example query, the sole condition is a tag pattern that binds the authors in the document to the variable `$author`. This tag pattern resolves to a single path (some tag patterns can represent multiple paths). The goal of this query is to find each instance of this path in the data source being searched and bind the value associated with it to the specified variable. Although SQLGenerator translates each nested XML-QL query into exactly one SQL query, a small number of simple SQL queries are executed beforehand in order to obtain metadata necessary for the main translation process. For queries containing regular path expressions (this example does not), the expression is resolved to a set of paths. This is accomplished by doing string comparisons of the values in the `tagpathtbl` with the path expression in the query. Once all paths in the query have been resolved to known paths, the integer keys corresponding to these paths are stored for later use. In this example, the integer key associated with the path is 4. The data source for each tag pattern must also be resolved into `pin` numbers by querying the `pinfiles` table. In our example query the data source is books.xml and the corresponding `pin` value is 1.

After obtaining the integer keys of the data source and all of the paths, SQL-Generator has enough information to begin construction of the SQL query that will retrieve the desired results, in this case all authors restricted to the path given in the query, located in the file books.xml. An alias to a new self-join on the `pinndx` table is generated for each path. This allows the SQLGenerator to perform the conjunctions of paths necessary to translate an XML-QL query into a single SQL query. The SQL generated for this query is shown in Figure 20.

Aliases created for the `pinndx` are labeled as `queryTable`n where n is an integer that depends on the number of necessary aliases. We see that the conditions we put on our results as expressed in the generated SQL query are that the path must have an ID of 4 and it must be of type 'E' (an XML Element). Also, valid results

```
SELECT DISTINCT queryTable0.pinnum,
queryTable0.value
FROM            pinndx queryTable0
WHERE           queryTable0.tagtype="E" AND
                queryTable0.tagpath=4  AND
queryTable0.pinnum=1
```

Fig. 20. Generated SQL From Example Query

must have a `pinnum` of 1, ensuring that that we only get results from the books.xml document. It is easy to see that this simple query will return all desired author names. Once the results are returned from the database, SQLGenerator iterates through the result set and replaces the known author names into the template obtained from the CONSTRUCT clause. The template accepts the current result and returns a DOM document fragment. A list of these fragments is compiled and appended to our root document. The document is then formatted and the resulting XML text is output to the user.

Performance

The SQLGenerator is unique in its capacity for scalability as it is bound soley by the scalability of the underlying database. Performance of indexing the XML documents is solely a function of the XML parser used and time required to insert a row into the database for each path in each document. This typically takes relatively little time, being bounded by the I/O capability of the relational database engine. Query translation is inconsequential compared to query execution time. Aside from some metadata lookup queries, the SQLGenerator makes every effort to combine the key processing into a single SQL statement, allowing the database to perform any possible optimizations. The only XML-QL feature which require more than one SQL query is nested queries, requiring one SQL query per nested query.

We indexed a collection of roughly 6.5 GB of XML documents which corresponds to a `pinndx` table of over 12 million rows. Figure 22 contains a subset of the queries from our regression test suite having various features discussed in this paper, and their execution time (in seconds). These queries were executed using a MySQL 4.0 database running on a Sun E450 with 4GB of RAM and 4 400MHz Ultrasparc II CPUs using a single SCSI drive. The first query binds one variable to a path and uses this variable in the CONSTRUCT to create XML results with one element each. The second query binds the value of an attribute as well as binds the value of two paths. The attribute and two element values are used in the CONSTRUCT to produce results with three elements all at the same depth. The first predicate query binds one variable to a path and places one relational condition on that variable. The variable is used in the CONSTRUCT to create results with one element. The second predicate condition is similar to the first but two conditions are placed on the variable to that it must fall within a certain range. The fifth query binds variables to paths using PCDATA binding, meaning that the contents of the path are not reconstructed manually but simply looked up using a column in the `pinndx` table.

Fig. 21. Query block processing algorithm

```
    procedure GenerateDOMResults(QueryBlock)
    predicates = {∅}
    tagPatterns = {∅}
    nestedQueries = {∅}
    paths = {∅}
    sqlPredicates = {∅}
    sqlSelects = {∅}
    template = {emptyDOMtemplate}
    for all elements in the CONSTRUCT do
10:   if Contents is a nested QueryBlock then
        nestedQueries ← QueryBlock
      else
        template ← element
      end if
    end for
    if This QueryBlock contains a WHERE clause then
      for all Conditions in the WHERE clause do
        if The condition is a TagPattern then
          tagPatterns ← this TagPattern
20:     else
          predicates ← this predicate
        end if
      end for
      for all Elements in tagPatterns do
        Build a path from this TagPattern
        paths ← all resolved paths from path
        Assign aliases to the pinndx table for each element in paths
        for all path in paths do
          if path has a variable or literal bound within it then
30:         sqlSelects ← a value column for the alias
          end if
          sqlPredicates ← predicate to enforce the data source
          sqlPredicates ← predicate to require presence of this path
        end for
      end for
      for all Items in predicates do
        Find table alias for variable in predicate
        sqlPredicates ← translated predicate
      end for
40:   construct SQL from sqlPredicates and sqlSelects
      resultSet ← execution of SQL
    end if
    for all result in resultSet do
      Use result to replace all non-static items in template
      domResults ← result.toDOM()
    end for
    for all nested QueryBlocks do
      nestedResults ← GenerateDOMResults(QueryBlock)
      Replace nestedResults into domResults
50: end for
    return domResults
```

Fig. 22. Timings of queries on 6.5 gig collection

Feature	Execution Time (s)
1: Basic functionality 1 path, 667542 results	330
2: Attribute binding 2 paths, 1 result	1170
3: Predicate conditions 1 path, 317900 results	143
4: Predicate conditions 1 path, 114736 results	79
5: PCDATA binding 2 paths, 105489 results	872
6: Kleene star expression 1 path, 15667 results	360
7: CONSTRUCT formatting 2 paths, 6 results	2

The Kleene star query uses a Kleene star as the contents of an element to represent any element or number of elements whose parent element and child element match the other two elements in the query. A variable is bound to the resulting paths and used in the CONSTRUCT. Finally, the CONSTRUCT formatting query binds two variables to paths and uses them in the CONSTRUCT. However, there is a formatting string located within the CONSTRUCT that separates the two variables. This formatting string causes a new line and tab character to be placed in between the values of the variables.

Acknowldegements

This work is supported by BIT Systems, Inc. Special thanks to Fred Ingham and Tim Lewis for their assistance with the design and testing of various parts of the SQLGenerator.

REFERENCES

Serge Abiteboul, *Querying semistructured data*, Proceedings of the International Conference on Database Theory, 1997.

S. Abiteboul and D. Quass and J. McHugh and J. Widom and J. Wiener, *The Lorel Query Language for Semistructured Data*, International Journal on Digital Libraries, vol. 1, no. 1, pp. 68-88, 4/1997.

XML Query Languages: Experiences and Exemplars.
http://www-db.research.bell-labs.com/user/simeon/xquery.html

P. Boncz, A. Wilschut, M. Kersten. Flattening an Object Algebra to Provide Performance. *IEEE 14th International Conference on Data Engineering, 568-577, 1998.*

Peter Buneman, Susan Davidson, Gerd Hillebrand, Dan Suciu A query language and optimization techniques for unstructured data, *Proceedings of ACM-SIGMOD International Conference on Management of Data, 1996.*

Peter Buneman, Tutorial: Semistructured data *Proceedings of the ACM SIGMOD Symposium on Principles of Database Systems, 1997.*

A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu A query language for XML. *International World Wide Web Conference, 1999.*

Alin Deutsch, Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, David Maier, and Dan Suciu Querying XML Data *IEEE Data Engineering Bulletin, 22(3):10-18, 1999.*

D. Florescu, D. Kossman. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin, 22(3):27-34, 1999.*

R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. *In Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99), Philadelphia, Pennsylvania, June 1999.*

D. Grossman, N. Goharian, O. Frieder, N. Raju. Extending the Undergraduate Curriculum to Include Information Retrieval and Data Mining. *IASTED Fifth International Multi-Conference on Computers and Advanced Technology in Education.* Cancun, Mexico, May 2002.

C. Kanne, G. Moerkotte. Efficient storage of XML data. *Proc. Of the 16th Int. Conf. On Data Engineering, 2000.*

Luk, Robert W.P., Leong, H.V., Dillon, Thraam S., Chan, Alvin T.S., Croft, W. Bruce, Allan, James. A Survey in Indexing and Searching XML Documents. *Journal of the American Society of Information Science and Technology*, 53(6):415-437, 2002.

J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. *SIGMOD Record, 26(3):54-66, September 1997.*

D. Quass, J. Widom. R. Goldman, K. Haas, Q. Luo, J. McHugh, S. Nestorov, A. Rajaraman, H. Rivero, S. Abiteboul, J. Ullman, and J. Wiener. LORE: A Lightweight Object REpository for Semistructured Data. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Montreal, Canada, June 1996. Demonstration description.*

J. Robie. The design of XQL, 1999.
http://www.texcel.no/whitepapers/xql-design.html

A. Schmidt, M. Kersten, M. Windhouwer, F. Waas. Efficient Relational Storage and Retrieval of XML Documents *Proceedings of the Third International Workshop on the Web and Databases, 47-52, 2000.*

J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. *Proceedings of the 25th VLDB Conference, 1999.*

T. Shimura, M. Yoshikawa, S. Uemura. Storage and retrieval of xml documents using object-relational databases. *Proc. of DEXA, Florence, Italy. Lecture Notes in Computer Science, 1677:206–217, 1999.*

Dongwook Shin. BUS: An Effective Indexing and Retrieval Scheme in Structured Documents. *Proceedings of Digital Libraries '98.*

Dongwook Shin. Structured querying, indexing, and retrieval for SGML/XML documents. *Proceedings of SGML/XML Japan '98, pp. 199-214.*

R. Zwol, P. Apers, A. Wilschut. Modelling and querying semistructured data with Moa *proceedings of Workshop on Query Processing for Semistructured Data and Non-standard Data Formats, 1999.*

A Proposal for XSL http://www.w3.org/TR/NOTE-XSL.html

Extensible Markup Language *(XML)* Second Edition
http://www.w3.org/TR/2000/REC-xml-20001006

The Extensible Markup Language *(XML)*
http://www.w3c.org/XML

The Query Languages Workshop.
http://www.w3.org/TandS/QL/QL98/

XML Query Requirements
http://www.w3.org/TR/xmlquery-req

XML Schema: Formal Description
http://www.w3.org/TR/2001/WD-xmlschema-formal-20010320