

XML Queries via SQL

Cindy Xinmin Chen¹ and Ashok Malhotra²

¹ Computer Science Department
University of California at Los Angeles
Los Angeles, CA 90095, U.S.A.
`cchen@cs.ucla.edu`*

² IBM Thomas J. Watson Research Center
30 Saw Mill River Road
Hawthorne, NY 10532, U.S.A.
`petsa@us.ibm.com`

Abstract. As XML [12] becomes more popular, we expect XML documents to be stored in databases with different representations. So an important issue is to translate XML queries into the language of the database. XPath [13], the XML Path Language, is a standard for expressing navigation and selection in XML documents. Hence it can be used to specify certain kinds of XML queries. If XML documents are stored in a relational database, XPath queries have to be translated into SQL [11]. This paper proposes an algorithm to translate queries expressed in XPath to SQL statements. The algorithm uses information that specifies the mapping of XML documents to relational tables.

1 Introduction

XML [12] is the next generation mark-up language, it has more functionality than HTML [7] and is easier to learn and use than SGML. Usually, a set of XML documents has a common DTD (Document Type Definition) [12] file to specify the structure of the XML documents. XML is a good candidate for web repositories, so a major issue is to be able to query the contents of XML documents efficiently.

There are several proposed XML query languages. The most powerful of them is XML-QL [4], which evolved from the Strudel [9] query language for ordered, semi-structured data. XML-QL uses a directed labeled graph data model. In this model, each XML tag becomes an edge labeled with the tag name and directed to an individual node. Non-leaf nodes correspond to attribute-value pairs. Leaf nodes correspond to element values. Each node has a unique ID and there is no order relation between nodes representing sibling elements. XML-QL includes a **construct** clause to specify the structure of the result and allows nested **where** and **construct** clauses.

* This work was done while the author was at the IBM Thomas J. Watson Research Center during the summer of 1999.

Lorel [6] uses a data model similar to that of XML-QL. Its design is based on OQL [1], thus it is syntactically more complex than XML-QL.

XMAS [8] is the query language used in MIX [2]. It is declarative and rule-based. However, XMAS is more or less a subset of XML-QL and lacks many of features one expects in a standard query language.

For SQL users, these query languages have the disadvantage of requiring one to learn another language. XML users would prefer an XML-style query language but none of the languages described above is a standard. [10] discusses a possible way to query XML documents stored on relational database by keeping the translation to SQL transparent to XML users. They propose translating XML-QL queries into SQL statements. Yet [10] cannot handle all the semantics of semi-structured queries over XML data.

XPath [13] is a standard proposed by the World Wide Web Consortium to address navigation and selection in XML documents and can be used to express a certain subset of XML queries. An XPath location path consists of a sequence of one or more location steps separated by “/”. The steps in a location path are interpreted from left to right. The leading “/” selects the root element of the XML document. Each succeeding step, in turn, selects a set of children of the parent element. Selection criteria can be included in location paths by encoding them within square brackets “[]”.

In this paper we describe an algorithm to query XML documents efficiently by translating XPath location paths to SQL statements. This is similar to [10] although we use a standard XML language to express the queries. In Section 2, we describe how an XML document is stored in a database. In Section 3, we demonstrate how to query a set of XML documents with some examples. In Section 4, we discuss the design and implementation of the algorithm to translate an XPath queries into an SQL statements. Section 5 concludes the paper. The algorithm is shown in Appendix A.

2 Storing XML Documents in a Database

The Document Object Model (DOM) [5] is an application programming interface for parsed documents. It defines the logical structure of documents and the way a document and its parts are accessed and manipulated.

In the DOM, parsed documents are modeled as a tree of objects — specifically, a hierarchy of nodes. Some nodes may have child nodes while others are leaf nodes that contain values but no children. Figure 1 is a representation of a DOM tree of the following sample XML document.

```
<State>
  <City>
    <Locality>
      <Street>Shady Grove</Street>
      <Street>Aeolian</Street>
    </Locality>
    <Locality>
      <Street>Over the River, Charlie</Street>
```

```

    <Street>Dorian</Street>
  </Locality>
</City>
</State>

```

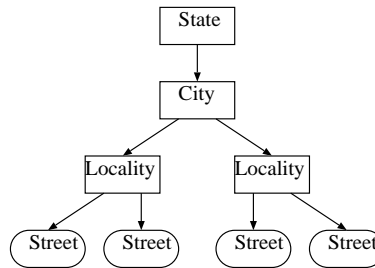


Fig. 1. DOM tree representation of the sample XML document

Each rectangle node in the DOM tree in Figure 1 corresponds to an element in XML that has children and each rounded rectangle node corresponds to an attribute of the element or a sub-element that has a value. Arrows between nodes indicate parent-child relationships.

To store a collection of XML documents that obey a particular DTD as a set of relational tables, we need to specify how the data in the documents is mapped into the relational tables. The IBM DB2 XML Extender [3] uses an XML file called a Document Access Definition (DAD) to specify this mapping. Typically, each type of element is stored in a table of its own. The attributes of the element appear as columns of that table.

The name of the column is typically the name of the attribute but can be different. In addition to this, the DAD file also stores the following information about each attribute:

- type — data type of the attribute of an XML element.
- path — the path from the root of the document to that attribute, i.e., to which XML element the attribute belongs and all of its ancestor XML elements. The DAD file also specifies the key for each table here, denoted by *[@Key]* after the name of the attribute.
- multi_occurrence — whether the attribute may appear more than once in the XML document.

3 Example of Querying XML Documents

In this section, we discuss how to query XML documents shredded into relational tables using SQL through the following examples.

Consider a set of XML documents containing information about customer orders. Each XML document contains a root `Order` element and each `Order`

element has several Customer and Part sub-elements. Each Part element has Shipment sub-element and Price attribute, etc.

```
<Order Key="1">
  <Customer>
    <Name>American Motors</Name> <Email>parts@am.com</Email>
  </Customer>
  <Part Key="68">
    <Color>red</Color> <Quantity>36</Quantity> <Price>34850.16</Price>
    <Tax>0.06</Tax>
    <Shipment>
      <ShipDate>1998-08-20</ShipDate> <ShipMode>AIR</ShipMode>
    </Shipment>
  </Part>
  <!-- information about other parts -->
</Order>
```

The DOM tree representation for such a document is shown in Figure 2. A possible DAD file is shown below.

```
<DAD>
  <dtdid>E:\dtd\order.dtd</dtdid>
  <validation>YES</validation>
  <Xcolumn>
    <table name="order_tab">
      <column name="order_key">
        type="integer" path="/Order [@Key]" multi_occurrence="NO"/>
    </table>
    <table name="customer_tab">
      <column name="Name">
        type="varchar(50)" path="/Order/Customer/Name [@Key]"
        multi_occurrence="NO"/>
      <column name="Email">
        type="varchar(50)" path="/Order/Customer/Email"
        multi_occurrence="NO"/>
    </table>
    <table name="part_tab">
      <column name="part_key">
        type="integer" path="/Order/Part [@Key]" multi_occurrence="NO"/>
      <column name="color">
        type="varchar(50)" path="/Order/Part/Color"
        multi_occurrence="YES"/>
      <column name="quantity">
        type="integer" path="/Order/Part/Quantity"
        multi_occurrence="YES"/>
      <column name="price">
        type="double"
        path="/Order/Part/Price"
        multi_occurrence="YES"/>
      <column name="tax">
        type="double" path="/Order/Part/Tax" multi_occurrence="YES"/>
    </table>
```

```

<table name="shipment_tab">
  <column name="shipdate"
    type="date" path="/Order/Part/Shipment/ShipDate"
    multi_occurrence="YES"/>
  <column name="shipmode"
    type="varchar(50)" path="/Order/Part/Shipment/ShipMode"
    multi_occurrence="YES"/>
</table>
<Xcolumn>
</DAD>

```

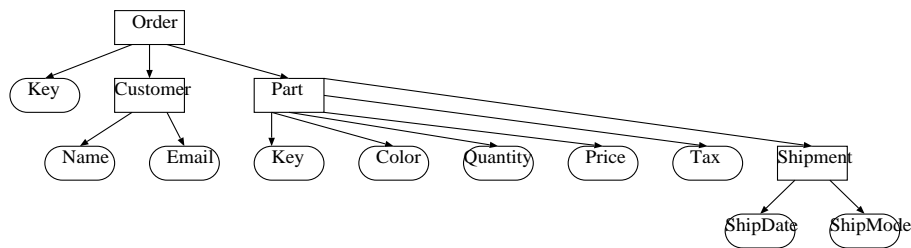


Fig. 2. DOM representation of order.xml

The above DAD file specifies that the order documents are stored in four tables; one table for each XML element. It also specifies that attributes of the element are stored as columns of each table.

To create a database according to this DAD file, some additional information is required. For those tables, which do not have a “[@Key]” in their column path, the keys of tables corresponding to their parent elements must be borrowed. The key columns of the parent tables are added into the child table and specified as the key of the child table. Keys of the parent tables are always included in the child table as foreign keys referring to the parent tables.

Thus, the database schema derived from the above DAD file is:

```

order_tab(order_key)
customer_tab(name, email, order_key)
part_tab(part_key, color, quantity, price, tax, order_key)
shipment_tab(part_key, shipdate, shipmode)

```

An XML query for this database may be: “find the shipping date for parts whose prices are greater than 20000”. This can be expressed in XPath syntax as:

```

/order/part[price>20000]/shipment/@shipdate

```

The initial “/” indicates the document root. Following this we navigate to the set of all `order` children of the root. From there we select the `part` children whose `price` attribute has a value greater than 20000. Further navigation gets us the set of `shipment` sub-elements of the selected parts and then their `shipdate` attributes which are returned.

According to our algorithm in Appendix A, and the DAD file shown above, this query can be expressed in SQL as:

```
SELECT shipment_tab.shipdate
FROM   order_tab, part_tab, shipment_tab
WHERE  part_tab.price > 20000 AND
       part_tab.order_key = order_tab.order_key AND
       shipment_tab.part_key = part_tab.part_key
```

Now consider the following query: “find the shipment element of the parts whose price are higher than 20000”.

The XPath syntax for this query is:

```
/order/part[price>20000]/shipment
```

The last step in the location path is now an XML element rather than an attribute. Thus every column in the table that stores the XML element should be returned.

The corresponding SQL statement is:

```
SELECT shipment_tab.part_key, shipment_tab.shipdate,
       shipment_tab.shipmode
FROM   order_tab, part_tab, shipment_tab
WHERE  part_tab.price > 20000 AND
       part_tab.order_key = order_tab.order_key AND
       shipment_tab.part_key = part_tab.part_key
```

Lastly, let us look at the following query: “find the part element and all of its sub-elements for the parts whose price are higher than 20000”.

The XPath syntax is:

```
/order/part[price>20000]
```

The last step in the location path is an XML element with sub-elements. So all the attributes of this element and all the attributes of its sub-elements, i.e., the nodes appear in the DOM tree below it are returned.

The corresponding SQL statement is:

```
SELECT part_tab.part_key, part_tab.color, part_tab.quantity,
       part_tab.price, part_tab.tax, part_tab.order_key,
       shipment_tab.part_key, shipment_tab.shipdate,
       shipment_tab.shipmode
FROM   order_tab, part_tab, shipment_tab
WHERE  part_tab.price > 20000 AND
       part_tab.order_key = order_tab.order_key AND
       shipment_tab.part_key = part_tab.part_key
```

4 Translating XPath Queries into SQL Statements

Since the DAD file contains all the information related to database schema design, the first step of the translation is to read the DAD file and store the information it contains in a data structure in main memory. We used a hash table for this. Each entry in the hash table represents an XML element or attribute. Each entry also contains the path information of the element or attribute, the

names of the table and column where the element or attribute is stored, and the key and the foreign key of the table. In addition to the hash table, we used an array to store the path information of all the columns defined in the DAD file.

The input XPath query is then parsed into navigation tokens separated by the “/” symbol. The name of the XML element or attribute in each token is then used as the key to search the hash table. Information about the element or attribute is retrieved, and corresponding additions to the SELECT, FROM and WHERE clauses of an SQL statement are generated.

Column names are added into the SELECT clause; the table names are added into the FROM clause and the foreign key constraints are added into the WHERE clause. As we mentioned, strings representing selection criteria can appear in square brackets “[]”. These selection criteria are added into the WHERE clause as well while necessary additions to the SELECT and FROM clause associated with the column mentioned in the selection criteria are made at the same time.

The key point of translating an XPath query into an SQL statement is to keep the database query process transparent to XML users. Users are able to query XML documents using the cognitive model of the XML documents to write the query. This method efficiently handles all the semi-structured queries over XML data that can be expressed by XPath standard.

5 Conclusion

This paper discusses an algorithm to query XML documents by translating an XPath queries into SQL statements. The XML documents are stored in tables in a database with the schema of the database derived from the DAD file that specifies the mapping of the XML documents onto database tables. Foreign key constraints of the tables are used to maintain the parent-child relationship between the elements of the XML document.

References

1. A.M. Alashqur, S.Y.W. Su and H. Lam. OQL: A Query Language for Manipulating Object-oriented Databases. In *Proc. Fifteenth VLDB*, pp.433-442, 1989.
2. C. Baru, A. Gupta, B. Ludaescher, R. Marciano, Y. Papakonstantinou and P. Velikhov. XML-Based Information Mediation with MIX. *Exhibitions Program of ACM SIGMOD*, 1999.
3. The DB2 XML Extender. <http://www-4.ibm.com/software/data/db2/extenders/xmlxt/>, 1999.
4. A. Deutsch, M.F. Fernandez, D. Florescu, A. Levy and D. Suciu. XML-QL:A Query Language for XML. <http://www.w3.org/TR/NOTE-xml-ql>, 1999.
5. Document Object Model (DOM) Level 1 Specification Version 1.0. <http://www.w3.org/TR/REC-DOM-Level-1>, 1998.
6. 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)*, pp. 25-30, 1999.
7. Hypertext Markup Language, "HTML 3.2 Reference Specification" <http://www.w3.org/TR/REC-html32> , 1997.

8. B. Ludascher, Y. Papakonstantinou and P. Velikhov. A Brief Introduction to XMAS. <http://www.db.ucsd.edu/Projects/MIX/docs/XMAS-intro.pdf>, 1999.
9. M. Fernandez, D. Florescu, A. Levy and D. Suci. A Query Language for a Web-Site Management System, *SIGMOD Record*, vol.26, no.3, pp.4-11, 1997.
10. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. *Proc. 25th VLDB*, 1999.
11. The SQL Standard. http://www.jcc.com/SQLPages/jccs_sql.htm, 1999.
12. Extensible Markup Language (XML) Version 1.0. <http://www.w3.org/TR/REC-xml>, 1998.
13. XML Path Language (XPath) Version 1.0. <http://www.w3.org/TR/xpath>, 1999

A Algorithm

```

begin
  parse dad_file
  record all paths in array
  create hash_table for database schema

  parse xpath_query at '/' boundaries

  for each xpath_query step except the last one do
    begin
      parse the xpath_query step by '['
      if string following '['
        put string into WHERE clause
      search hash_table
      put table name into FROM clause
      put foreign key constraint into WHERE clause
    end

  for last xpath_query step
    if string following '['
      put string into WHERE clause
    if '@'
      search hash_table
      put column name into SELECT clause
      put table name into FROM clause
      put foreign key constraint into WHERE clause
    else
      compare xpath_query with the array of paths
      for each element in the difference
        begin
          search hash_table
          put column name into SELECT clause
          put table name into FROM clause
          put foreign key constraint into WHERE clause
        end
    end
end

```