Path–Oriented Queries and Tree Inclusion Problem

Yangjun Chen
University of Winnipeg, Canada

INTRODUCTION

With the rapid advance of the Internet, management of structured documents such as XML documents has become more and more important (Marchiori, 1998). As a simplified version of SGML, XML is recommended by W3C (World Wide Web Consortium, 1998a; World Wide Web Consortium, 1998b) as a document description meta-language to exchange and manipulate data and documents on the WWW. It has been used to code various types of data in a wide range of application domains, including a Chemical Markup Language for exchanging data about molecules, the Open Financial Exchange for swapping financial data between banks and customers, as well as a Geographical Markup Language for searching geographical information (Bosak, 1997; Zhang & Gruenwald, 2001). Also, a growing number of legacy systems are adapted to output data in the form of XML documents.

In recent years, efforts have been made to find an effective way to generate XML structures that are able to describe XML semantics in underlying relational databases (Chen & Huck, 2001; Florescu & Kossmann, 1999; Shanmugasundaram et al., 1999, 2000; Yoshikawa, Amagasa, Shimura, & Uemura, 2001). However, due to the substantial difference between the nested element structures of XML and the flat relational data, much redundancy is introduced; i.e., the XML data is either flattened into tuples containing many redundant elements or has many disconnected elements. Therefore, it is significant to explore a way to accommodate XML documents which is different from the relational theory.

BACKGROUND

As a path-oriented language, XQL queries are represented by a line command which connects element types using path operators (‘/’ or ‘//’). ‘/’ is the child operator which selects from immediate child nodes. ‘//’ is the descendant operator which selects from arbitrary descendant nodes. In addition, symbol ‘@’ precedes attribute names. By using these notations, all paths of tree representation can be expressed by element types, attributes, ‘/’ and ‘@’. Exactly, a simple path can be described by the following Backus-Naur Form:

\[
\text{<simple path>} ::= \text{<PathOp>} \text{<SimplePathUnit>} | \text{<PathOp><SimplePathUnit>‘@’<AttName>}
\]

The following is a simple path-oriented query:

\[
/\text{letter}//\text{body} \text{[para $\text{contains}’\text{visit}’]} \quad \text{(1)}
\]

where /letter//body is a path and [para $\text{contains}’\text{visit}’] is a predicate, enquiring whether element “para” contains a word “visit.”

Several paths can be jointed together using $\wedge$ to form a complex query as follows:

\[
/\text{hotel-room-reservation/name}?x\wedge
/\text{hotel-room-reservation/location}\text{[city-or-district = Winnipeg]}\wedge
/\text{hotel-room-reservation/location/address}\text{[street = 510 Portage Ave]} \quad \text{(2)}
\]

EVALUATION OF PATH-ORIENTED QUERIES

In this section, we show different ways to evaluate a path-oriented query. First, we discuss the basic methods used in a database environment. Then a new strategy for tree-inclusion, which can be embedded into a document
database to provide an efficient way to evaluate path-oriented queries, is discussed in great detail.

**QUERY EVALUATION BASED ON INVERSION**

Inversion on Elements and Words

There is a lot of work that considers using relational database techniques to store and retrieve XML documents, such as Arnold-Moore, Fuller, Lowe, Thom, and Wilkinson (1995); Florescu and Kossman (1999); and Zhang, Naughton, DeWitt, Luo, and Lohman (2001). Among them, the most representative is the method discussed in Zhang et al. In this method, two kinds of inverted indexes are established for text words and elements, by means of which a text word (or an element) is mapped to a list, which enumerates documents containing the word (or the element) and its position within each document. To speed up the query evaluation, the position of a word (or an element) is recorded as follows:

- \((Dno, Wposition, level)\) for a text word
- \((Dno, Eposition, level)\) for an element

where \(Dno\) is its document number, \(Wposition\) is its position in the document, and \(level\) is its nesting depth within the document; \(Eposition\) is a pair: \(<s, e>\), representing the positions of the start and end tags of an element, respectively. For instance, the document shown in Figure 1(a) is indexed as shown in Figure 1(b). The index for elements is called \(E-index\) and the index for words is called \(T-index\).

Let \((d, x, l)\) be an index entry for an element \(a\). Let \((d', x', l')\) be an index entry for a word \(b\). Then, \(a\) contains \(b\) iff \(d = d'\) and \(x.s < x'.s\) and \(x.e > x'.e\). Let \((d'', x'', l'')\) be an index entry for another element \(c\). Then, \(a\) contains \(c\) iff \(d = d''\) and \(x.s < x''.s\) and \(x.e > x''.e\). Using these properties, some simple path-oriented queries can be evaluated. For example, to process the query: /hotel-room-reservation/location/[city-or-district = Winnipeg], the inverted lists of hotel-room-reservation, location, city-or-district, and Winnipeg will be retrieved and then their containment will be checked according to the above properties. In a relational database, \(E-index\) and \(T-index\) are mapped into the following two relations (note that primary keys are italicized):

\[
\begin{align*}
E-index & : (element, docno, begin, end, level) \\
T-index & : (word, docno, wordPosition, level)
\end{align*}
\]

These index structures are efficient for simple cases, such as whether a word is contained in an element. However, in the case that a query is a nontrivial tree, the evaluation based on these index structures is an exponential time process. To see this, consider the query: /hotel-room-reservation/location/address [street = Portage Ave.]. To evaluate this query, four joins have to be performed. They are the self-joins on \(E-index\) relation to connect hotel-room-reservation and location, location and address, and address and street, as well as the join between \(E-index\) and \(T-index\) relations to connect street and Portage Ave. In general, for a document tree with \(n\) nodes and a query tree with \(m\) nodes, the checking of containment needs \(O(n^m)\) time using this method.

**Figure 1. A sample XML file and its inverted lists**

```
<hotel-room-reservation filecode=''1302''>
  <name>Travel-lodge</name>
  <location>
    <city-or-district>Winnipeg</city-or-district>
    <state>Manitoba</state>
    <country>Canada</country>
    <address>
      <number>500</number>
      <street>Portage Ave.</street>
      <post-code>R3B 2E9</post-code>
    </address>
  </location>
  <type>
    <room>one-bed-room</room>
    <price>$119.00</price>
  </type>
  <reservation-time>
    <from>April 28, 2003</from>
    <to>May 01, 2003</to>
  </reservation-time>
</hotel-room-reservation>
```

```
E-index: (1, <1, 45>, 0)...
(1, <2, 4>, 1)...
(1, <5, 28>, 2)...

T-index: (1, 3, 2)...
(1, 7, 3)...
(1, 10, 3)...
```

```
(a) (b)
```

```
hotel-room-reservation

name

location

Travel-lodge

Winnipeg

Manitoba

(b)
```