Paper: jc10-5-2585; 2006/5/11

Efficient Processing of XML Tree Pattern Queries

Yangjun Chen*, and Dunren Che**

*Department of Applied Computer Science, University of Winnipeg
515 Portage Avenue, Winnipeg, Manitoba R3B 2E9, Canada
E-mail: ychen2@uwinnipeg.ca
**Department of Computer Science, Southern Illinois University
Carbondale, IL 62901, USA
E-mail: dche@cs.siu.edu

[Received October 28, 2005; accepted March 17, 2006]

In this paper, we present a polynomial-time algorithm for TPQ (tree pattern queries) minimization without XML constraints involved. The main idea of the algorithm is a dynamic programming strategy to find all the matching subtrees within a TPQ. A matching subtree implies a redundancy and should be removed in such a way that the semantics of the original TPQ is not damaged. Our algorithm consists of two parts: one for subtree recognization and the other for subtree deletion. Both of them needs only O(n^2) time, where n is the number of nodes in a TPQ.

Keywords: XML documents, tree pattern queries, Xpath expressions, query evaluation

1. Introduction

XML (eXtensible Markup Language) is emerging as a standard for data and information exchange between applications on the Web and elsewhere. It offers a convenient syntax for representing data from heterogeneous data sources distributed over the Internet. As an important issue in building up Web services, XML queries have been extensively studied recently, and various optimization strategies to solve this problem are proposed. But most of them follow a conventional routine such as transforming a query into a logical-level plan first (such as in [8]), and then exploring the (exponential) space of possible plans in order to identify the optimal one that has the least estimated cost (such as in [1]). Yet XML queries are significantly different from the conventional *RDBMS* queries in that the former routinely involve a tree-shaped pattern that is to be matched against the database, and the queries are commonly referred to as *TPQ*s. Furthermore, TPQs often contain redundancies, especially when constraints such as those induced from the DTD/XSD (XMAL Schema Definition) are additionally considered. Redundancies are detrimental to the performance of XML query evaluation. Therefore, studying efficient mechanisms for TPQ minimization is of great importance for XML query processing [2, 3, 9–11].

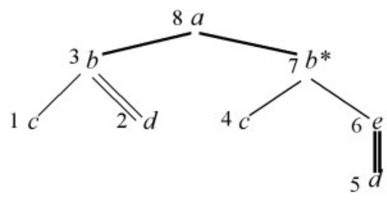

Fig. 1. A query tree.

2. Background and Data Model

2.1. Data Model and Queries

We consider a data model where information is represented as a forest of trees. Each node in the tree has an associated type. Types can be organized in a simple aggregation hierarchy for capturing the ancestorship.

To abstract from existing query languages for XML (e.g., [4–6]), we express queries as tree patterns where nodes are types and edges are *parent-child* or *ancestor-descendant* relationships. Among all the nodes of a query Q, one is designated as the output node, denoted by *output*(Q), corresponding to the output of the query. As an example, consider the query tree shown in **Fig.1**, which asks for any node of type b that is a child of some node of type a. In addition, the b-node is the parent of some c-node and some e-node, and an ancestor of some d-node. In the figure, a parent-child relationship is represented by a single edge while an ancestor-descendant relationship by a double edge. The output node is indicated by *. The query corresponds to the following XPath expression:

$$a[b[c \text{ and } //d]]/b[c \text{ and } e//d].$$

This tree patters is used to retrieve relevant portions of data from a database. Although the tree patterns do not capture all aspects of XML query languages such as ordering and restructuring, they form a key component of XML query languages by focusing on their structural aspect [7].

2.2. Semantics

Definition 2.1 (*Equivalent Queries*) Let $Q(D)$ denote the result of a query Q on a database D. We say that $Q_1 \subseteq Q_2$

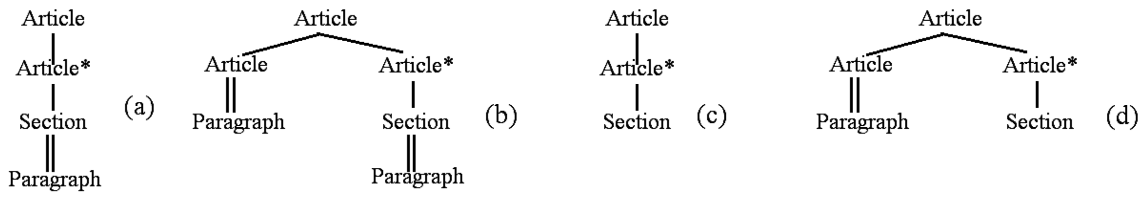

Fig. 2. Illustration of minimization scheme.

for queries Q_1 and Q_2, if $Q_1(D) \subseteq Q_2(D)$ for all databases D. We say Q_1 and Q_2 are equivalent if $Q_1 \subseteq Q_2$ as well as $Q_2 \subseteq Q_1$.

Definition 2.2 (*Redundant Node*) A node $p \in Q$ is said to be redundant iff another query Q', obtained from Q by deleting p and all its descendants, is equivalent to Q, (i.e.$Q'(D) = Q(D)$).

Definition 2.3 (*Minimal Query*) A query Q is said to be minimal iff no query of smaller size is equivalent to Q. For a TPQ, the size is the number of the nodes in that query.

Definition 2.4 (*Coverage Relation*) Consider a TPQ $Q = (N, E)$ consisting of a set N of nodes and a set E of directed edges; each node $p \in Q$ has a type $\lambda(p)$ associated with it. We define the coverage relation in a TPQ Q as follows. We say that node p is covered by node v or v is the coverer of p (denote by $p \prec v$), whenever the following conditions hold:

- Preserve node types: $\lambda(p) = \lambda(v)$; also, if p is the output node then $v = p$.
- Preserve child edge relationships: if p has a direct child p', then v has to have a direct child v' such that $p' \prec v'$.
- Preserve descendant edge relationships: if p has a descendant child p'', then v has to have a descendant child v'' such that $p'' \prec v''$.

Definition 2.5 (*Cover-Set*) Let N be the set of the nodes of a certain TPQ Q and $cov(p)$ denote the set of all coverers of p for any $p \in Q$, we have:

1. If p is the output node, $cov(p) = \{p\}$.
2. If p is a leaf node, $cov(p) = \{v | v \in N, \lambda(p) = \lambda(v)\}$.
3. If p is a non-leaf node, $cov(p)$ is a set of nodes, in which each node v satisfies one of the following conditions:

   (i) For each child p' of p, v is the parent of some node in $cov(p')$, or

   (ii) For each descendant p'' of p, v is an ancestor of some node in $cov(p'')$.

2.3. Observations

As we stated before, our major task in this paper is to solve the problem of optimizing TPQS with/without the presence of constraints. The designing of our proposed algorithms is guided by the following two important observations.

1) *If not augmenting the queries at the first place, some TPQs may fail to be minimized.*

   It sounds to be unreasonable to "enlarge" a query first when our goal is to minimize it. However, in many cases, augmenting the input query is the only way to discover the potential opportunity to minimize it. As an example, consider the query of **Fig.2(d)**. In the absence of constraints, this query is minimal. If we are given the constraint: "*every section has a paragraph*", we still cannot perform any minimization directly on it. Yet, if we augment this query by adding an additional node "paragraph" and make it as a descendant of "section" as shown in **Fig.2(b)**, we then see that the entire left branch is now covered by the right branch, and thus can be completely dropped. This gives us a smaller but equivalent query as shown in **Fig.2(a)**. By removing the added (redundant) node "paragraph", we reach the ultimate minimal state as shown in **Fig.2(c)**.

2) *The order of applying constraints is important for obtaining a minimal equivalent query.*

   Constraints like the required child/descendant nodes inherently carry the information that the child/descendant nodes are redundant if the related parent/ancestor nodes are already in the query pattern. But the question is, should we apply the constraints first by dropping the redundant child/descendant nodes immediately or should we perform minimization first? Consider the query of **Fig.2(b)**. Suppose the constraint we are given is still the same – "*every section has a paragraph*". If we adopt the first strategy, then the query can be simplified to **Fig.2(d)**, by dropping the "paragraph" node immediately. Unfortunately, doing so will prevent any further simplification, either by applying constraints or constraint-independent minimization. However, the current state is obviously not minimal (the minimal state is the one shown in **Fig.2(c)**). In summary, the minimization of TPQs with/without the presence of constraints are orthogonal to each other. The kernel minimization scheme can be shared by both of the problems (for the minimization of TPQs in the presence of constraints, we just need to arrange additional processing steps). In general, our algorithm for tree pattern minimization can be described as a 3-phase process that consists of the following:

- *Augmentation Phase*: Augment the query with respect to the constraints that are present. If no constraints are given, move to next step directly.
- *Minimization Phase*: Identify and remove the redundant nodes in the augmented query.
- *Garbage Collection Phase*: Remove all temporary nodes added by the first step.

3. Identify and Remove Redundant Nodes

3.1. Algorithm

In the previous section, we have presented the general idea for TPQ minimization. Readers may already notice that the first and the third phase of the algorithm do not affect the core process of minimization (i.e., the second phase) at all. Hence, the major challenge falls on how to identify and remove redundant nodes in a TPQ. Due to limitation of space, we only address the core process in paper. However, the augmentation as well as the garbage collection can be reletively easily conducted based on the strategy to be presented.

The algorithm to be given is in fact a dynamic programming solution. During the process, three matrices are maintained and computed to facilitate the discovery of coverages. They are described as follows.

1. The nodes in a TPQ Q are numbered in postorder, and the nodes are then referred to by their postorder numbers.

2. The first matrix is established to keep the *proper* transitive closure of Q, in which each a_{ij} has value 0 or 1. Initially, we set $a_{ij} = 1$, if j is an ancestor of i; otherwise, $a_{ij} = 0$. We also notice that $a_{ii} (1 \leq i \leq n)$ is always set to 0 since any node i is not considered as a proper ancestor of itself. In addition, during the computation, if we find k covers i, each ancestor of k is considered as an ancestor of i. Therefore, the matrix will be dynamically changed. This matrix is denoted by $tran(Q)$.

3. The second matrix is used to calculate the cover-set, in which each c_{ij} has value 0 or 1. If $c_{ij} = 1$, it indicates that the subtree rooted at the node indexed by j covers the subtree rooted at the node indexed by i. Otherwise, $c_{ij} = 0$. This matrix is denoted by $cover(Q)$.

4. The third matrix is to store the parent j of any node k that covers i. Therefore, an entry $d_{ij} = 1$ indicates that there exists some child k of j, which covers i, i.e., $c_{ik} = 1$; otherwise, $d_{ij} = 0$. This matrix is denoted by $parent\text{-}cover(Q)$.

$cover(Q)$ is established by running the following algorithm, called *Coverage*. Initially, $c_{ij} = 0$ and $d_{ij} = 0$ for all i and j. During the execution of the algorithm, the values of c_{ij}'s will be changed according to Definition 2.5 while d_{ij}'s will be changed as specified above (see item (4)). The *Coverage* algorithm uses two convenient boolean functions, which are defined below:

- $f(i,i',j)$ – if (i,i') is a child edge and there exists an j' such that (j,j') is a child edge and $c_{i'j'} = 1$, return *true*; otherwise, *false*. This function needs a constant time by directly checking whether $d_{i'j} = 1$.

- $g(i,i',j)$ – if (i,i') is a descendant edge and there exists an j such that $c_{i'j'} = 1$ and $a_{j'j} = 1$, return *true*; otherwise, *false*. This function needs a constant time, too, by directly checking whether $a_{i'j} = 1$.

Our *Coverage* algorithm is given below:

```
Algorithm Coverage(Q)
input: Q
output: cover(Q)
begin
1. for $i = 1$ to $n$ do
2. {if $i$ = output($Q$) then $c_{ii} := 1$;
3.   else if $i$ is a leaf then {for $j = 1$ to $n$ do if $\lambda(i) = \lambda(j)$
                          then $c_{ij} := 1$;}
4.    else
5.    {let $i_1, i_2, \ldots, i_k$ be the children of $i$;
6.      for $j = 1$ to $n$ do
7.      if $\lambda(i) = \lambda(j)$ then
8.      {if for each child edge $(i, i_l)$ $(1 \leq l \leq k)$,
              $f(i, i_l, j)$ returns true and
9.          for each descendant edge $(i, i_l)$ $(1 \leq l \leq k)$,
              $g(i, i_l, j)$ returns true
10.        then $c_{ij} := 1$;}}
11. let $j_1, j_2, \ldots, j_h$ be the nodes that cover $i$;
12. set $d_{ip_l} = 1$ for each $p_l$, where $p_l$ is the parent of
        $j_l$ $(1 \leq l \leq h)$;
13. let $\{q_1, \ldots, q_m\}$ be a set such that each node in it is an
        ancestor of some $j_l$. Set $a_{iq_l} = 1$ for each
        $q_l$ $(1 \leq l \leq m)$;}
end
```

The above algorithm works in a bottom up way since we search the tree along the nodes' postorder numbers. During the process, c_{ij} is determined according to the coverage of i's and j's children as well as their descendants (see lines 7-10) while a_{ij} and d_{ij} are changed according to the newly changed c_{ij}'s (see lines 11-13).

Example 1. We apply $Coverage(Q)$ to the TPQ Q shown in **Fig.1**. First, the algorithm will generate $tran(Q)$ as shown in **Fig.3**. We recall that in Q each node is referred to by its postorder number. In addition, each c_{ij} as well as each d_{ij} is initially set to 0.

In each step of the outer **for**-loop, $cover(Q)$, $parent\text{-}cover(Q)$, and $tran(Q)$ will be changed in the way as illustrated in **Fig.4**.

From $cover(Q)$ created in step 8, we can see
$cov(1) = cov(4) = \{1,4\}$,

cover(*Q*): *parent-cover*(*Q*): *tran*(*Q*):

step 1:

$$\begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&1&0&0&1&0&0&0&0\\ 2&0&0&0&0&0&0&0&0\\ 3&0&0&0&0&0&0&0&0\\ 4&0&0&0&0&0&0&0&0\\ 5&0&0&0&0&0&0&0&0\\ 6&0&0&0&0&0&0&0&0\\ 7&0&0&0&0&0&0&0&0\\ 8&0&0&0&0&0&0&0&0 \end{array} \quad \begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&0&0&1&0&0&0&1&0\\ 2&0&0&0&0&0&0&0&0\\ 3&0&0&0&0&0&0&0&0\\ 4&0&0&0&0&0&0&0&0\\ 5&0&0&0&0&0&0&0&0\\ 6&0&0&0&0&0&0&0&0\\ 7&0&0&0&0&0&0&0&0\\ 8&0&0&0&0&0&0&0&0 \end{array} \quad \begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&0&0&1&0&0&0&1&1\\ 2&0&0&1&0&0&0&0&1\\ 3&0&0&0&0&0&0&0&1\\ 4&0&0&0&0&0&0&1&1\\ 5&0&0&0&0&0&1&1&1\\ 6&0&0&0&0&0&0&1&1\\ 7&0&0&0&0&0&0&0&1\\ 8&0&0&0&0&0&0&0&0 \end{array}$$

step 3:

$$\begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&1&0&0&1&0&0&0&0\\ 2&0&1&0&0&1&0&0&0\\ 3&0&0&0&0&0&0&1&0\\ 4&0&0&0&0&0&0&0&0\\ 5&0&0&0&0&0&0&0&0\\ 6&0&0&0&0&0&0&0&0\\ 7&0&0&0&0&0&0&0&0\\ 8&0&0&0&0&0&0&0&0 \end{array} \quad \begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&0&0&1&0&0&0&1&0\\ 2&0&0&0&0&0&0&0&0\\ 3&0&0&0&0&0&0&0&1\\ 4&0&0&0&0&0&0&0&0\\ 5&0&0&0&0&0&0&0&0\\ 6&0&0&0&0&0&0&0&0\\ 7&0&0&0&0&0&0&0&0\\ 8&0&0&0&0&0&0&0&0 \end{array} \quad \begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&0&0&1&0&0&0&1&1\\ 2&0&0&1&0&1&1&1&1\\ 3&0&0&0&0&0&0&0&1\\ 4&0&0&0&0&0&0&1&1\\ 5&0&0&0&0&0&1&1&1\\ 6&0&0&0&0&0&0&1&1\\ 7&0&0&0&0&0&0&0&1\\ 8&0&0&0&0&0&0&0&0 \end{array}$$

step 5:

$$\begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&1&0&0&1&0&0&0&0\\ 2&0&1&0&0&1&0&0&0\\ 3&0&0&0&0&0&0&1&0\\ 4&1&0&0&1&0&0&0&0\\ 5&0&1&0&0&1&0&0&0\\ 6&0&0&0&0&0&0&0&0\\ 7&0&0&0&0&0&0&0&0\\ 8&0&0&0&0&0&0&0&0 \end{array} \quad \begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&0&0&1&0&0&0&1&0\\ 2&0&0&0&0&0&0&0&0\\ 3&0&0&0&0&0&0&0&1\\ 4&0&0&1&0&0&0&1&0\\ 5&0&0&0&0&0&0&0&0\\ 6&0&0&0&0&0&0&0&0\\ 7&0&0&0&0&0&0&0&0\\ 8&0&0&0&0&0&0&0&0 \end{array} \quad \begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&0&0&1&0&0&0&1&1\\ 2&0&0&1&0&1&1&1&1\\ 3&0&0&0&0&0&0&0&1\\ 4&0&0&1&0&0&0&1&1\\ 5&0&0&3&0&0&1&1&1\\ 6&0&0&0&0&0&0&1&1\\ 7&0&0&0&0&0&0&0&1\\ 8&0&0&0&0&0&0&0&0 \end{array}$$

step 7:

$$\begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&1&0&0&1&0&0&0&0\\ 2&0&1&0&0&1&0&0&0\\ 3&0&0&0&0&0&0&1&0\\ 4&1&0&0&1&0&0&0&0\\ 5&0&1&0&0&1&0&0&0\\ 6&0&0&0&0&0&0&0&0\\ 7&0&0&0&0&0&0&1&0\\ 8&0&0&0&0&0&0&0&0 \end{array} \quad \begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&0&0&1&0&0&0&1&0\\ 2&0&0&0&0&0&0&0&0\\ 3&0&0&0&0&0&0&0&1\\ 4&0&0&1&0&0&0&1&0\\ 5&0&0&0&0&0&0&0&0\\ 6&0&0&0&0&0&0&0&0\\ 7&0&0&0&0&0&0&0&1\\ 8&0&0&0&0&0&0&0&0 \end{array} \quad \begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&0&0&1&0&0&0&1&1\\ 2&0&0&1&0&1&1&1&1\\ 3&0&0&0&0&0&0&0&1\\ 4&0&0&1&0&0&0&1&1\\ 5&0&0&0&0&0&1&1&1\\ 6&0&0&0&0&0&0&1&1\\ 7&0&0&0&0&0&0&0&1\\ 8&0&0&0&0&0&0&0&0 \end{array}$$

cover(*Q*): *parent-cover*(*Q*): *tran*(*Q*):

step 2:

$$\begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&1&0&0&1&0&0&0&0\\ 2&0&1&0&0&1&0&0&0\\ 3&0&0&0&0&0&0&0&0\\ 4&0&0&0&0&0&0&0&0\\ 5&0&0&0&0&0&0&0&0\\ 6&0&0&0&0&0&0&0&0\\ 7&0&0&0&0&0&0&0&0\\ 8&0&0&0&0&0&0&0&0 \end{array} \quad \begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&0&0&1&0&0&0&1&0\\ 2&0&0&0&0&0&0&0&0\\ 3&0&0&0&0&0&0&0&0\\ 4&0&0&0&0&0&0&0&0\\ 5&0&0&0&0&0&0&0&0\\ 6&0&0&0&0&0&0&0&0\\ 7&0&0&0&0&0&0&0&0\\ 8&0&0&0&0&0&0&0&0 \end{array} \quad \begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&0&0&1&0&0&0&1&1\\ 2&0&0&1&0&0&1&1&1\\ 3&0&0&0&0&0&0&0&1\\ 4&0&0&0&0&0&0&1&1\\ 5&0&0&0&0&0&1&1&1\\ 6&0&0&0&0&0&0&1&1\\ 7&0&0&0&0&0&0&0&1\\ 8&0&0&0&0&0&0&0&0 \end{array}$$

step 4:

$$\begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&1&0&0&1&0&0&0&0\\ 2&0&1&0&1&0&0&0&0\\ 3&0&0&0&0&0&0&1&0\\ 4&1&0&0&1&0&0&0&0\\ 5&0&0&0&0&0&0&0&0\\ 6&0&0&0&0&0&0&0&0\\ 7&0&0&0&0&0&0&0&0\\ 8&0&0&0&0&0&0&0&0 \end{array} \quad \begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&0&0&1&0&0&0&1&0\\ 2&0&0&0&0&0&0&0&0\\ 3&0&0&0&0&0&0&0&1\\ 4&0&0&1&0&0&0&1&0\\ 5&0&0&0&0&0&0&0&0\\ 6&0&0&0&0&0&0&0&0\\ 7&0&0&0&0&0&0&0&0\\ 8&0&0&0&0&0&0&0&0 \end{array} \quad \begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&0&0&1&0&0&0&1&1\\ 2&0&0&1&0&1&1&1&1\\ 3&0&0&0&0&0&0&0&1\\ 4&0&0&1&0&0&0&1&1\\ 5&0&0&0&0&0&1&1&1\\ 6&0&0&0&0&0&0&1&1\\ 7&0&0&0&0&0&0&0&1\\ 8&0&0&0&0&0&0&0&0 \end{array}$$

step 6:

$$\begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&1&0&0&1&0&0&0&0\\ 2&0&1&0&0&1&0&0&0\\ 3&0&0&0&0&0&0&1&0\\ 4&1&0&0&1&0&0&0&0\\ 5&0&1&0&0&1&0&0&0\\ 6&0&0&0&0&0&0&0&0\\ 7&0&0&0&0&0&0&0&0\\ 8&0&0&0&0&0&0&0&0 \end{array} \quad \begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&0&0&1&0&0&0&1&0\\ 2&0&0&0&0&0&0&0&0\\ 3&0&0&0&0&0&0&0&1\\ 4&0&0&1&0&0&0&1&0\\ 5&0&0&0&0&0&0&0&0\\ 6&0&0&0&0&0&0&0&0\\ 7&0&0&0&0&0&0&0&0\\ 8&0&0&0&0&0&0&0&0 \end{array} \quad \begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&0&0&1&0&0&0&1&1\\ 2&0&0&1&0&1&1&1&1\\ 3&0&0&0&0&0&0&0&1\\ 4&0&0&1&0&0&0&1&1\\ 5&0&0&0&0&0&1&1&1\\ 6&0&0&0&0&0&0&1&1\\ 7&0&0&0&0&0&0&0&1\\ 8&0&0&0&0&0&0&0&0 \end{array}$$

step 8:

$$\begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&1&0&0&1&0&0&0&0\\ 2&0&1&0&0&1&0&0&0\\ 3&0&0&0&0&0&0&1&0\\ 4&1&0&0&1&0&0&0&0\\ 5&0&1&0&0&1&0&0&0\\ 6&0&0&0&0&0&0&0&0\\ 7&0&0&0&0&0&0&1&0\\ 8&0&0&0&0&0&0&0&0 \end{array} \quad \begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&0&0&0&0&0&0&0&0\\ 2&0&0&0&0&0&0&0&0\\ 3&0&0&0&0&0&0&0&0\\ 4&0&0&0&0&0&0&0&0\\ 5&0&0&0&0&0&0&0&0\\ 6&0&0&0&0&0&0&0&0\\ 7&0&0&0&0&0&0&0&0\\ 8&0&0&0&0&0&0&0&0 \end{array} \quad \begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&0&0&1&0&0&0&1&1\\ 2&0&0&1&0&1&1&1&1\\ 3&0&0&0&0&0&0&0&1\\ 4&0&0&1&0&0&0&1&1\\ 5&0&0&0&0&0&1&1&1\\ 6&0&0&0&0&0&0&1&1\\ 7&0&0&0&0&0&0&0&1\\ 8&0&0&0&0&0&0&0&0 \end{array}$$

Fig. 4. Sample trace.

$$\begin{array}{c|cccccccc} & 1&2&3&4&5&6&7&8\\ \hline 1&0&0&1&0&0&0&0&1\\ 2&0&0&1&0&0&0&0&1\\ 3&0&0&0&0&0&0&0&1\\ 4&0&0&0&0&0&0&1&1\\ 5&0&0&0&0&0&1&1&1\\ 6&0&0&0&0&0&0&1&1\\ 7&0&0&0&0&0&0&0&1\\ 8&0&0&0&0&0&0&0&0 \end{array}$$

Fig. 3. Initial $tran(Q)$.

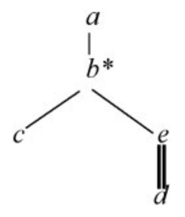

Fig. 5. A minimized tree.

$cov(2) = cov(5) = \{2, 5\}$,
$cov(3) = \{7\}$, and
$cov(7) = \{7\}$.

Based on $cover(Q)$ obtained by the *Coverage* algorithm, Q can be minimized by doing the following with each node $v \in Q$:

(i) Let $v_1, v_2, \ldots, v_k$ be the children of v;

(ii) For each v_i,

if (v, v_i) is a child edge and there exists $v_j (j \neq i)$ such that (v, v_j) is a child edge and $c_{v_i v_j} = 1$, then remove the subtree rooted at v_i if v_j has not been removed;

if (v, v_i) is a descendant edge and there exists $v_j (j \neq i)$ such that (v, v_i) is a child or descendant edge and $c_{v_i v_j} = 1$ or $a_{v_i v_j} = 1$, then remove the subtree rooted at v_i if v_j has not been removed.

When applying these operations to the tree shown in **Fig.1**, the subtree rooted at 3 will be eliminated as shown in **Fig.5**.

Obviously, this process can be easily integrated into the Algorithm $Coverage(Q)$ by performing the above step immediately after $tran(Q)$ and $parent\text{-}cover(Q)$ are modified according to the newly changed c_{ij}'s. This is because after each inner **for**-loop, all the covering nodes of i is recorded into $cover(Q)$ and if there is any node k that is covered by i, it must be covered by a node that covers i. Thus, this will be certainly discovered in the subsequent computation. So i and all its descendants can be removed if possible without damaging the correctness. In terms of the above analysis, we give the following general query minimization algorithm.

```
Algorithm query-minimization(Q)
input: Q
output: minimized query – Q'
begin
1. for $i = 1$ to $n$ do
2. {if $i$ = output($Q$) then $c_{ii} := 1$;
3.   else if $i$ is a leaf then {for $j = 1$ to $n$ do if $\lambda(i) = \lambda(j)$
                                    then $c_{ij} := 1$;}
4.   else
5.   {let $i_1, i_2, \ldots, i_k$ be the children of $i$;
6.     for $j = 1$ to $n$ do
7.     if $j$ exists do
8.     {if $\lambda(i) = \lambda(j)$ then
9.       {if for each child edge $(i, i_l)$ $(1 \le l \le k)$,
              $f(i, i_l, j)$ returns true and
10.           for each descendant edge $(i, i_l)$ $(1 \le l \le k)$,
              $g(i, i_l, j)$ returns true
11.       then $c_{ij} := 1$;}}}
12. let $j_1, j_2, \ldots, j_h$ be the nodes that cover $i$;
13. set $d_{ip_l} = 1$ for each $p_l$, where $p_l$ is the parent of
        $j_l$ $(1 \le l \le h)$;
14. let $\{q_1, \ldots, q_m\}$ be a set such that each node in it is an
        ancestor of some $j_l$. Set $a_{iq_l} = 1$ for each
        $q_l$ $(1 \le l \le m)$;
15. if there is a sibling of $i$ satisfying the condition speci-
        fied above in (ii), remove $i$ and its descendants;}
end
```

The above algorithm is a slight change to Algorithm *Coverage*(Q). In line 7, we check whether a node is already deleted. If it is the case, the corresponding computation will not be performed, leading to some time reduction. In addition, some work in line 13 and 14 can also be saved. In line 15, we remove i if it can be removed according to the condition (ii) given above.

3.2. Correctness and Time Complexity

The correctness of Algorithm *Coverage*(Q) can be easily established. In terms of Definition 2.5, a node i is covered by another node j if it is an output node, a leaf node have the same label as j or the following conditions are satisfied:

(i) $\lambda(i) = \lambda(j)$ and j is the parent of some node in $cov(i')$ for each child i' of i.

(ii) $\lambda(i) = \lambda(j)$ and j is an ancestor of some node in $cov(i'')$ for each descendant i'' of i.

These two conditions are exactly checked in lines 8-9 in Algorithm *Coverage*(Q). The work done in line 12 and 13 of the algorithm is just to make the checking of the above conditions more efficient.

Now we analyze the time complexity of Algorithm *Coverage*(Q). First, we notice that each step in the inner **for**-loop needs $O(d_i)$ time, where d_i represents the outdegree of i. Then, the whole cost of the inner **for**-loop is bounded by $O\left(\sum_i d_i\right) = O(n)$. The time for the execution of line 12 is obviously bounded by $O(n)$. In addition, the time for line 13 is also $O(n)$ if we change *tran*(Q) as follows. Each time when we search the tree bottom-up from a covering node q_k $(1 \le k \le m)$ to find all its ancestors, we mark each node encountered and stop whenever we meet such a mark (made by a previous searching). So at most $O(n)$ nodes will be checked. From this analysis, we see that the total cost of the algorithm is $O(n^2)$. The time of Algorithm *query-minimization*(Q) is still $O(n^2)$ since line 15 in it takes at most $O(d_i)$ time.

4. Conclusion

In this paper, the minimization of TPQs in XML database systems has been discussed. For this, two algorithms were proposed: one for the recognition of matching subtrees within a TPQ, and the other for the TPQ minimization. The main idea behind the algorithm for subtree recognition is a dynamic programming strategy to find all coverages of nodes. It needs $O(n^2)$ time for this task, where n is the number of nodes in a TPQ Q. Based on this algorithm, the algorithm for the query minimization is devised. It removes a subtree immediately once the subtree is found to be covered by another. Thus, the query reduction does not require any extra costs. So the entire time complexity of our algorithms is bounded by $O(n^2)$. In the absence of XML constraints, it can always find the minimal equivalent query to the original one.

References:

[1] J. McHugh, and J. Widom, "Query Optimization for XML," Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

[2] S. Amer-Yahia etc. "Minimization of TPQs," Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 497-508, 2001.

[3] P. T. Woo, "Minimizing Simple XPath Expressions," WebDB 2001.

[4] G. Miklau, and D. Suciu, "Containment and Equivalence for an XPath Fragment," Proceedings of 21st ACM Symp., Principles of Database Systems, 2002.

[5] D. Chamberlin, J. Clark, D. Florescu, and M. Stefanescu, "XQuery1.0: An XML Query Language," http://www.w3.org/TR/ query-datamodel/.

[6] A. Deutch, M. Fernandex, D. Florescu, A. Levy, and D. Suciu, "A Query Language for XML," WWW'99.

[7] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," Proceedings of the 18th International Conference on Data Engineering, 2002.

[8] D. Che, and K. Aberer, "Query Processing and Optimization in XML Structured-Document Databases," The VLDB Journal (in press).

[9] S. Flesca, F. Furfaro, and E. Masciari, "On the Minimization of Xpath Queries," Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003.

[10] D. Lee, and W. W. Chu, "Constraints-preserving Transformation from XML Document Type Definition to Relational Schema," Proceedings of the 19th International Conference on Conceptual Modeling, pp. 323-338, 2000.

[11] C. Yu, and L. Popa, "Constraint-Based XML Query Rewriting for Data Integration," Proceedings of the ACM SIGMOD International Conference, Paris, France, 2004.

Name:
Yangjun Chen

Affiliation:
Associate Professor, Dept. Applied Computer Science, University of Winnipeg

Address:
515 Portage Ave. Winnipeg, Manitoba R3B 2E9, Canada

Brief Biographical History:
1982 Received B.S. degree in information system engineering from the Technical Institute of Changsha, China
1990 Received Diploma degree in computer science from the University of Kaiserslautern, Germany
1995 Received Ph.D. degree in computer science from the University of Kaiserslautern, Germany
1995-1997 Worked as a post-doctor at the Technical University of Chemnitz-Zwickau, Germany
1997-2000 Worked as a senior engineer at the German National Research Center of Information Technology (GMD), Germany
2000 Worked as a post-doctor at the University of Alberta, Canada
2000-present Professor in the Department of Applied Computer Science, the University of Winnipeg, Canada

Main Works:
- "On the Signature Tree Construction and Analysis," to appear in IEEE Transaction on Knowledge and Data Engineering.
- "Graph Traversal and Linear Binary-chain Programs," IEEE Transaction on Knowledge and Data Engineering, Vol.15, No.3, pp. 573-596, May/June, 2003.
- "Magic Sets and Stratified Databases," Int. Journal of Intelligent Systems, John Wiley & Sons, Ltd., Vol.12, No.3, pp. 203-231, March, 1997.

Name:
Dunren Che

Affiliation:
Assistant Professor, Department of Computer Science, Southern Illinois University Carbondale

Address:
Faner Hall 2125. SIU-Campus Carbondale, IL 62901, USA

Brief Biographical History:
1994 Received Ph.D. in Computer Science from the Beijing University of Aeronautics and Astronautics, Beijing, China
1994-2001 Gained postdoctoral research experience from various research Institutes, including, the Tsinghua University in China, the German National Research Center for Information Technology in Germany, and the Johns Hopkins University in the USA
2002-present Assistant Professor of Computer Science in the Southern Illinois University at Carbondale, USA

Main Works:
- "Query Optimization in XML Structured-Document Databases," The VLDB Journal, Vol.15, 2006.
- "Efficiently Processing XML Queries with Support for Negated Containments," International Journal of Computer & Information Science, Vol.6, No.2, pp. 109-120, June, 2005.