A Theoretic Framework for Answering XPath Queries Using Views

Jian Tang[1] and Shuigeng Zhou[2]

[1] Department of Computer Science, Memorial University of Newfoundland
St. John's, NL, A1B 3X5, Canada
jian@cs.mun.ca
[2] Department of Computer Science, Fudan University
Shanghai, 200433, China
sgzhou@fudan.edu.cn

Abstract. Query rewriting has many applications, such as data caching, query optimization, schema integration, etc. This issue has been studied extensively for relational databases and, as a result, the technology is maturing. For XML data, however, it is still at the developing stage. Several works have studied this issue for XML documents recently. They are mostly application-specific, being that they address the issues of query rewriting in a specific domain, and develop methods to meet the specific requirements. In this paper, we study this issue in a general setting, and concentrate on the correctness requirement. Our approach is based on the concept of query containment for XPath queries, and address the question of how that concept can be adopted to develop solutions to query rewriting problem. We study various conditions under which the efficiencies and applicability can trade each other at different levels, and introduce algorithms accordingly.

1 Introduction

Query rewriting using views has many applications, such as data caching, query optimization, schema integration, etc. It can be simply described as follows. A user query wants to retrieve information from a given set of data. Instead of directly running the user query, we wish to rewrite it into another query by using a separate view query as a tool. The rewritten version then produces the output that can satisfy the users' needs. This issue has been studied extensively for relational databases [8, 9, 11, 13, 14]. As a result, the technology is maturing. For XML data, however, it is still at the developing stage. Several works have studied this issue for XML documents in specific contexts recently. In [3], the authors propose a system for answering XML queries using previously cached data. In [1, 5, 15], methods are suggested to rewrite queries using views to enhance the efficiencies. The work in [4, 17] study how the queries over the target schema can be answered using the views over the source data, and hence provide a way for schema integration. In [2, 7, 12], the authors study the query rewriting problem for semi-structured data. Since all these works are tailored to specific domains, they do not discuss how they can fit into a general setting. In this paper, we introduce a general framework for XPath query rewriting in a restricted context. Having such a framework has the following advantages. First, it characterizes

S. Bressan et al. (Eds.): XSym 2005, LNCS 3671, pp. 18–33, 2005.

the problem and the related solutions, and therefore provides us with insights into its theoretic nature. Second, it tells us how the two competing goals, completeness and efficiency, interplay and therefore suggests directions for further improvements over the existing solution.

Like the work in relational databases, most of the work on query rewriting using views for XML data are based on the concept of query containment of one kind or another. Recently, the research on query containment problem for XML queries has generated significant results. In [10], the authors study this problem in a limited class of XPath queries, which contains four kinds of symbols, /, //, [ ], *, and found that the problem of query containment is coNP-hard. In [6], the authors extend the results to the case where disjunctions, DTDs and some limited predicates are allowed.

There are two aspects for a general XPath query, *navigation script* and *tagging template* The navigation script guides the search for the required information while the tagging template provides the format for assembling the constructed document. Query rewritings for these two aspects are more or less orthogonal. In this paper, we study the framework for the rewriting for navigation script only, since this is the more vital and significant part of the two. We restrict our study also only to the subset of XPath queries that contains four kinds of symbols, /, //, [ ], *, as this can set up a foundation for further extension. We first provide a model for the problem, and then concentrate on the correctness problem. Our approach is based on the concept of query containment for XPath queries, and addresses the question of how that concept can be adopted to develop solutions to query rewriting problem. We study various conditions under which the efficiencies and applicability can trade each other at different levels, and introduce algorithms accordingly.

The rest of the paper is organized as follows. In Section 2, we first review some basic concepts, and then suggest a model for query rewriting problem. In Section 3, we present some solutions to query rewriting problem, and discuss the trade-offs between these solutions. We conclude the paper by summarizing the main results.

2 XPath Query and Rewriting

2.1 Pattern Trees and Input Trees

An XPath query can be denoted as a tree, called a *pattern tree*. Each node is attached with a label from an infinite alphabet, except for the root. The tree may contain branches, and can contain two kinds of edges, parent/child (denoted by single edges) and ancestor/descendent (denoted by double edges). If there is a child or descendant edge from n_1 to n_2, we say n_2 is a *C_child* or *D_child*, respectively, of n_1. Among all the nodes, there is a set of distinguished nodes, called *return nodes*. Although from the prescribed semantics, an XPath query should be considered to contain only a single return node, in this paper we do not restrict the number of return nodes to one. This will make the result applicable to more general query structures, such as those written in XQuery [18], where return nodes normally correspond to the last steps in path expressions, and accessing them (i.e., variable binding or referencing) triggers the creation of output nodes. (We use the convention that the root is always a return node.) Non-return nodes are called *transit nodes*.

XPath queries execute on XML document trees, which we will refer to as *input trees*. The execution proceeds by matching the nodes in the pattern tree to the nodes in the input tree. The following notations are from [10]. Let q be a pattern tree and t be an input tree. An *embedding* is a mapping e: nodes(q) $\rightarrow$ nodes(t) such that (1) for any node n $\in$ nodes(q), either label(n) = '*', or label(n) = label(e(n)) and (2) for any n_1, $n_2 \in$ nodes(q), if n_1 is a parent of n_2, then there is an edge from $e(n_1)$ to $e(n_2)$; if n_1 is an ancestor of n_2, then there is a path from $e(n_1)$ to $e(n_2)$. For each n $\in$ nodes(q), we say e matches n to e(n). Let return-nodes(q) = $\{n_1, \ldots, n_k\}$. The set anws(q, t) = $\{\{e(n_1), \ldots, e(n_k)\}$: e is an embedding from nodes(q) to nodes(t)$\}$ is called the answer to q on t. We say that pattern trees *q is contained in* pattern tree *p* if anws(q, t) $\subseteq$ anws(p, t) for all t.

Let q be a pattern tree, and m be the number of descendant edges in q. Let $\vec{u} = <u_1, \ldots, u_m>$, where for all $1 \le i \le m$, u_i is a non-negative integer. The $\vec{u}$ - *extension of q with z* is an input tree, t_u, formed by modifying q as follows: replacing the label * with symbol z, and replacing the *i*th descendant edge, say ab, by a path $a\lambda_i b$ where λ_i contains u_i nodes, all labeled z. Note that λ_i is not originally in tree q. We call it the *ith guest path* in t_u. (Refer to Figure 2 for an example.) Thus except for the nodes in any guest path, all the nodes in a $\vec{u}$ -extension belong to the original q. To avoid ambiguity, we say that they are copies of those in q, and use symbol π for the mapping from the nodes in q to their copies in any of its $\vec{u}$ -extensions. If for all $1 \le i \le m$, $u_i = c$, i.e., all the guest paths contain equal number of nodes, c, then that $\vec{u}$ -extension is referred as a *c-extension*. For example, the tree in Figure 2.b is a 3-extension of the pattern tree in Figure 2.d. Call a path a *star-path* in a pattern tree if all its edges are child-edges and nodes are labeled *. For any tree or path t, we use |t| to denote the number of nodes it contains.

2.2 Query Rewriting

Regardless of the context, any technique for query rewriting uses substitution one way or another. The differences lie on the levels at which the substitution is made. For XML queries, most techniques apply the substitution implicitly at the pattern tree node level. The idea is, given pattern trees p and q, and an input tree to both, if the answer to be produced by p can be reproduced by q, then we can 'delegate' the task of p to q by rewriting the nodes of p in terms of the nodes of q [1]. The following two definitions formalize this idea.

Definition 1: Let p and q be pattern trees. A *rewriting of p by q* is a triplet *(p, q, h)* where *h*: return_nodes(p) $\rightarrow$ return_nodes(q) is called a *return node mapping* (RNM), p and q are respectively called *user* query and *view* query.

---

1 Whether or not the answers reproduced should be complete is dictated by the correctness criteria for different applications. For example, if q is used to optimize the performance of p, then it must generate complete solutions. On the other hand, if q is used for the purpose of schema integration, it normally generates only partial answers.

In the above definition, the return node mapping is arbitrary. In particular, it does not have to be one-to-one or onto. This definition is applicable to any pair of pattern trees and RNMs. Our interest, however, is in a restricted class, as described below.

Definition 2: Rewriting (p, q, h) is *correct on an input tree t* if for all embedding e: nodes(q) → nodes(t), there is an embedding f: nodes(p) → nodes(t) such that f and $e \circ h$ are consistent on return-nodes(p) (i.e., for all $n \in$ return-nodes(p), f(n) = e(h(n))). If it is correct on all input trees, then we say it is *correct*.

Intuitively, a correct rewriting of p should produce the answers that are acceptable to p on any input. For example, consider the pattern trees in Figure 1.

Suppose return_nodes(p) = {1, 2, 4} and return_nodes(q) = {5, 6, 7}. Let the RNM h: return_nodes(p) → return_nodes(q) be defined as: h(1) = 5, h(2) = 6, h(4) = 7. We now show that the rewriting (p, q, h) is correct. Let t be any input tree, and e: nodes(q) → nodes(t) be an embedding.

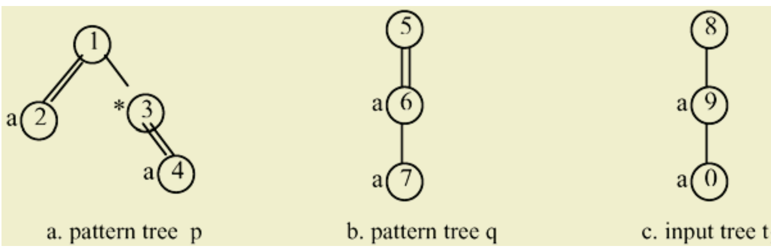

Fig. 1. An example for query rewriting

From the labels and the structure of q, we must have: e(5)) = root(t), label(e(6)) = a, label(e(7)) = a, there is a path λ from root(t) to e(6), and there is an edge ε from e(6) to e(7). Define g: nodes(p) → nodes(t) as: g(1) = root(t), g(2) = e(6), g(3) = child of root(t) in λ and g(4) = e(7). Clearly, g is an embedding, and g is consistent with $e \circ h$. This completes the proof.

Now we define another RNM as follows: $h_1(1) = 5$, $h_1(2) = 6$, $h_1(4) = 6$. It can be shown that the rewriting (p, q, h_1) is not correct. Indeed, consider the input tree in Figure c, and embedding e_1: nodes(q) → nodes(t_1) defined as $e_1(5) = 8$, $e_1(6) = 9$, $e_1(7) = 0$. Clearly, there does not exist an embedding that can match 2 to 9 and 4 to 9.

Note that for the same pair of queries, there may be more than one correct rewriting. For example, we can define h_2: return_nodes(p) → return_nodes(q) as: $h_2(1) = 5$, $h_2(2) = 7$, $h_2(4) = 7$. It can be easily shown that (p, q, h_2) is also a correct rewriting.

We now present some informal argument for the expressive power of our formulation. We argue that the condition in Definition 2 is the weakest one can assume conforming with the common notion used in practice for query rewriting, that is, substitution of view query for user query at the node level. First, note that, to be able to rewrite p using q, it is necessary that any result generated by q is acceptable by p. (This is a different way of saying that p contains q.) Although this assumption is weaker than ours, it nonetheless does not capture the idea of node substitution mentioned above. To capture that idea, additional component needs to be incorporated into the model to relate the view query nodes to the user query nodes in a manner that is independent of the input trees. This is way the RNM mapping is introduced in the above two definitions.

Now, there arise issues of how to determine efficiently if a given rewriting is correct and, given two pattern trees, how we find all the correct rewritings. We will study these issues in the subsequent sections.

3 Relating Query Containment to Query Rewriting

In the relational databases, query containment is the base for query rewriting. In this section, we study their relationship for XPath queries. To simplify the presentation, in this section when we mention rewriting (p, q, h) we assume implicitly that h is onto. This is because when the onto-condition is not met, we can always consider only the subset of the return nodes of q to which h is mapped. Then our results will follow without essential changes.

3.1 A Necessary and Sufficient Condition for Correct Rewriting

Query containment requires that any set of input nodes that are matched by the return nodes of the view query are also matched by the return nodes of the user query, while a correct query rewriting requires that this be true at the node level. This suggests that the latter is at least as strong a condition as the former. In the following theorem let L be the number of nodes in the longest star-path, and z be a label not used by any node in p.

Theorem 1[2]: A rewriting (p, q, h) is correct if and only if it is correct on the $\vec{u}$-extension of q for all $\vec{u} = \langle u_1, \ldots, u_m \rangle$, where for all $1 \le i \le m$, $u_i \le L + 1$.

Proof: The 'only if' part is straightforward. We explain the idea for the proof of 'if' part. Let t be any input tree, and e: nodes(q) $\rightarrow$ nodes(t) be an embedding. Let $\langle a_i, b_i \rangle$ be the *i*th descendant edge in q. It must be matched to a path $e(a_i)\lambda_i e(b_i)$ in t, where λ_i is a path in t with a length of possibly zero. Define $\vec{u} \equiv \langle v_1, \ldots, v_m \rangle$, where for all $1 \le i \le m$, $v_i = |\lambda_i|$ when $|\lambda_i| \le L$, and $v_i = L+1$ when $|\lambda_i| \ge L+1$. Let this $\vec{u}$-extension of q with z be t_u. Denote by μ_i the ith guest path in t_u. Thus $|\mu_i| = v_i$ and all nodes in μ_i are labeled z. (Refer to Sec. 2.1.) By the assumption, (p, q, h) is a correct rewriting in t_u. Thus there is an embedding g: nodes(p) $\rightarrow$ nodes(t_u) such that for all n $\in$ return_nodes(p), g(n) = e(h(n)). We can define a mapping f: nodes(p) $\rightarrow$ nodes(t) in such a way that it is an embedding and for all n $\in$ return_nodes(p), f(n) = e(h(n)), as follows. If $g(n) \notin \mu_i$ for all $1 \le i \le m$, let $f(n) = e \circ \pi^{-1} \circ g(n)$. (Recall π maps the nodes in q to their copies in the $\vec{u}$-extension.) If $g(n) \in \mu_i$ for some $1 \le i \le m$, consider following two cases. (1) $|\mu_i| \le L$. In this case we let f(n) be the *j*th node in λ_i if g(n) is

---

[2] For this and the following theorems, we present only the informal argument. The formal proof is found in [16].

the *j*th node in μ_i. This is possible since $|\mu_i| = |\lambda_i|$. (2) $|\mu_i| = L+1$. In this case $|\mu_i| \leq |\lambda_i|$. We have the following observations. First, since label(g(n)) = z, we have label(n) = *. Second, let α be the longest star-path in p such that $n \in \alpha$. By assumption, we have $|\alpha| \leq L$. Thus at least one end node of α is incident with a descendant edge and is also matched to a node in μ_i, otherwise we would have $|\alpha| \geq |\mu_i| = L+1$, which is impossible. Because of this, we can let f match, one by one, all the nodes preceding the descendant edge that were previously matched to μ_i by g *onto a prefix* of λ_i, and all the nodes following the descendant edge that were previously matched to μ_i by g *onto a suffix* of λ_i. We call such a way of mapping 'prefix-suffix mapping'. Shown in Figure 2 is an example of prefix-suffix mapping. It can then be easily shown that the function f so defined is an embedding from p to t, and for all $n \in$ return_nodes(p), f(n) = e(h(n)). □

We have mentioned that the correct-rewriting problem is a subset of containment problem. A natural question is, how big is this subset? At this time, we do not have a definite answer. Rather than exploring the difference of the two, however, our interest is how we can solve the rewriting problem with the help of the solutions to the containment problem, and with what a price.

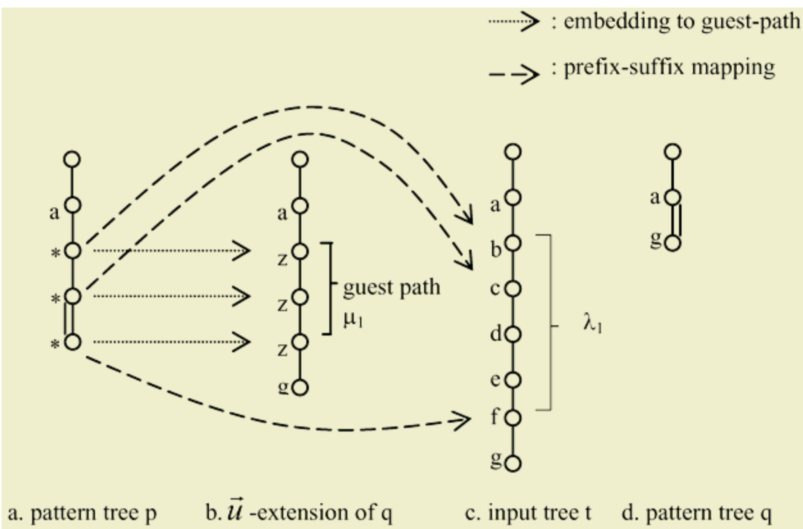

Fig. 2. Prefix-suffix mapping from p to λ_1, L = 2, $|\mu_1| = 3$, $|\lambda_1| = 5$

It is worth mentioning here that the RNM for a correct rewriting is not equivalent to homomorphism introduced in [10]. A homomorphism from p to q requires explicitly every node and edge in p to follow some structural pattern, depending on those in q, while a correct rewriting requires only some mapping from the return nodes in p to those in q that can meet the correctness criterion, without imposing structural constraints. It can be shown that homomorphism implies correct rewriting, but the reverse is not true [16].

Using Theorem 1, we can develop a sound and complete algorithm to determine if (p, q, h) is a correct rewriting. However, checking whether or not the rewriting is correct on all the $\vec{u}$-extensions of q specified in the theorem is intractable: there are $(L+1)^r$ $\vec{u}$-extensions to consider in total, where r is the number of descendant edges in q. In the subsequence sections, we will present alternative methods that can have better performance, at the price of stronger assumptions.

3.2 An Efficient Method

We observe that the main cost of the algorithm mentioned above results from a large number of $\vec{u}$-extensions that must be considered. By adding a little strong condition, we can reduce this number to one, as described by the following theorem.

Theorem 2: Assume any transit node of p labeled '*' is not incident with a descendant edge, L is the number of nodes in the longest star-path in p, and z is a label not used in p. Then the following three statements are equivalent:

1. p contains q.
2. There exists an embedding e: nodes(p) $\rightarrow$ nodes(t_{L+1}) such that e(return_nodes(p)) = π(return_nodes(q)), where t_{L+1} is the (L+1)-extension of q with z, and π maps the nodes in q to their copies in t_{L+1}.
3. There exists an RNM h: return_nodes(p) $\rightarrow$ return_nodes(q) such that (p, q, h) is a correct rewriting.

Proof:

1 $\Rightarrow$ 2: Clearly, π: nodes(q) $\rightarrow$ nodes(t_{L+1}) is an embedding. Since p contains q, by definition, there is an embedding e: nodes(p) $\rightarrow$ nodes(t_{L+1}) such that e(return_nodes(p)) = π(return_nodes(q)).

2 $\Rightarrow$ 3: Let h = $\pi^{-1} \circ$ e: nodes(p) $\rightarrow$ nodes(q). We will show that (p, q, h) is a correct rewriting. (In the triplet, h should be understood as restricted on return_nodes(p).) First note that for any node x $\in$ nodes(q), π(x) does not belong to any guest-path t_{L+1}. In particular, π(return_nodes(q)) is disjoint with any guest-path in t_{L+1}. We now prove the claim that for all n $\in$ nodes(p), e(n) does not belong to any guest-path in t_{L+1}. If n $\in$ return_nodes(p), then by assumption, e(n) $\in$ π(return_nodes(q)). The claim follows. Now assume n is a transit node. Assume the contrary, i.e., e(n) belongs to some guest-path, r. Let s be the longest star-path containing n whose nodes are *all* matched to r by e. Thus $|s| \le L$. On the other hand, from the above arguments, all the nodes in s are transit nodes. By assumption they are incident only with child edges. Note that neither the node preceding s nor the node following s is matched to r. Thus we must have $|s| = |r| = L + 1$. This is impossible, implying e(n) cannot belong to any guest-path. Our claim follows. Now, let t be any input tree. Let g: nodes(q) $\rightarrow$ nodes(t) be an embedding. Consider mapping g $\circ$ h: nodes(p) $\rightarrow$ nodes(t). Let o $\in$ nodes(p) be an arbitrary node. If label(o) $\neq$ '*', then label(o) = label(e(o)) $\neq$ 'z', implying label(e(o)) = label(π^{-1}(e(o))) $\neq$ '*'. Since g is an embedding, label(π^{-1}(e(o))) = label(g(π^{-1}(e(o)))). Thus label(g $\circ$ h(o)) = label(g(π^{-1}(e(o)))) = label(o). Now let o_1, o_2 $\in$ nodes(p) be arbitrary

nodes. First assume there is a child edge from o_1 to o_2. Then there is an edge from $e(o_1)$ to $e(o_2)$. Since neither $e(o_1)$ nor $e(o_2)$ belongs to a guest-path, there is a child edge from $\pi^{-1}(e(o_1))$ to $\pi^{-1}(e(o_2))$, implying there is an edge from $g(\pi^{-1}(e(o_1)))$ to $g(\pi^{-1}(e(o_2)))$. Second, assume there is a descendant edge from o_1 to o_2. Then there is a path from $e(o_1)$ to $e(o_2)$. Again, since $e(o_1)$ and $e(o_2)$ are not in guest-paths, there must be a path from $\pi^{-1}(e(o_1))$ to $\pi^{-1}(e(o_2))$ (which may contain some guest-path as sub-path). This implies that there is a path from $g(\pi^{-1}(e(o_1)))$ to $g(\pi^{-1}(e(o_2)))$. We have proven that the mapping $g \circ h$ is an embedding. Note $\pi^{-1}(\pi(\text{return\_nodes}(q))) = \text{return\_nodes}(q)$. Since by assumption, $e(\text{return\_nodes}(p)) = \pi(\text{return\_nodes}(q))$, we have $\pi^{-1}(e(\text{return\_nodes}(p))) = \text{return\_nodes}(q)$, or $h(\text{return\_nodes}(p)) = \text{return\_nodes}(q)$. Let $n \in \text{return\_nodes}(p)$. We have $(g \circ h)(n) = g(h(n))$. Therefore $h = \pi^{-1} \circ e$ is the desired RNM.

3 $\Rightarrow$ 1: This follows from that given any input tree t, (p, q, h) being correct on t implies $p \supseteq q$ on t. □

The assumption in Theorem 2 somewhat constrains the cases where the theorem can be used. It nonetheless covers many common cases. From the proof of the theorem, if statement 2 is true, and we know π and e, then we can construct a correct rewriting, i.e., $(p, q, \pi^{-1} \circ e)$. To determine π, we first construct t_{L+1}. This can be done in a single scan of the nodes and edges in q. For each node scanned, we create its correspondence in t_{L+1} consistent with π. Each time when a child edge is scanned in q, we create an edge in t_{L+1}, and when a descendant edge is scanned, we create a guest-path of length L+1. When we finish scanning q, the construction of t_{L+1} is completed.

We now look into the question of how to search for embeddings from p to t_{L+1} efficiently. We shall now present an algorithm that searches for embeddings in the general case. We first need a data structure to store the embeddings with the matching between return nodes annotated. We use the term 'sub-graph tree' to refer to a tree that is a sub-graph. (A sub-graph tree is therefore not necessarily a subtree.) We use the term *embedding-tree* to refer to a tree that stores embeddings. An Embedding tree consists of two kinds of nodes, P_nodes (for pattern tree) and I_nodes (for input tree). All the nodes are labeled the ids of the corresponding tree nodes. Parents and children must be of different kinds. The root is a P_node. If a P_node is labeled x and it has an I_node child labeled y, then there is an embedding that matches node x to node y. Each embedding is represented by a subgraph-tree that contains *all* the P_nodes and, for each P_node, *exactly one* I_node child. Shown on the left side of the double arrow in Figure 3 is the embedding tree that contains all the embeddings for the pattern tree and the input tree respectively in figures 1.a and 1.c.

The circles denote P_nodes and the rectangles denote I_nodes. On the left side of the double arrow is the embedding tree, which contains two embeddings, represented by the sub-graph trees on the right side of the double arrow.

Algorithm 1 below constructs an embedding-tree for all the embeddings from a given pattern tree to an input tree[3]. For simplicity, we will use the phrase 'return

[3] This is an extension of the one in [10], which generates only binary answers.

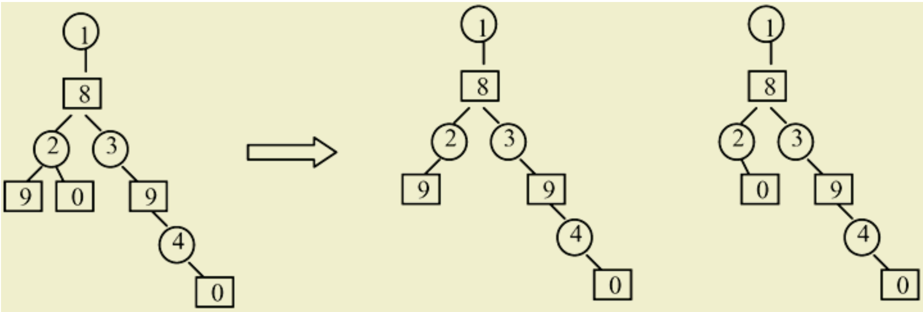

Fig. 3. An Embedding-tree

nodes' also to refer to those nodes in any $\vec{u}$ -extension of q that are copies of the return nodes in q.

Algorithm 1

Input: pattern tree p with root r_1, and a set R_1 of return nodes

Input tree t with root r_2, and a set R_2 of return nodes

Two dimensional arrays C and D, where each array entry takes as value a set of I_nodes, or U (i.e., undefined).

Output: the embedding tree containing the set of all embeddings: nodes(p) $\rightarrow$ nodes(t) that match r_1 to r_2

set every entry of C and D to U
insert as the root of the embedding tree a P_node n_1 for r_1
C_Build(r_1, r_2, C[r_1, r_2])
if C[r_1, r_2] = Φ then
    remove n_1
    return // no embedding found
create as the single child of n_1 the I_node C[r_1, r_2]

C_Build(PatternTreeNode x, InputTreeNode y, SetofInodes C[x, y])

if (label(x) ≠ * and label(x) ≠ label(y)) then C[x, y] ← Φ; return
if (x ∈ return_nodes(p) and y ∉ return_nodes(t)) then C[x, y] ← Φ; return
for each C_child x' of x //test if the sub-pattern rooted at x' can match a sub-tree //rooted at a child of y
    for each child y' of y
        if C[x', y'] = U then C_Build(x', y', C[x', y']) //make the call only if no earlier //call made on the same pair
    if C[x', y'] = Φ for every child y' of y, then C[x, y] ← Φ; return //no match
for each D_child x' of x //test if the sub-pattern rooted at x' can match a sub-tree //rooted at a descendant of y
    for each child y' of y
        if D[x', y'] = U then D_Build(x', y', D[x', y'])
        if D[x', y'] = Φ for every child y' of y, then C[x, y] ← Φ; return //no match
create an I_node n_1 labeled y //can match, so store it in embedding tree

```
12  for each C_child x' of x                    //also store all the matches for each child
13    create a P_node n_2 as a new child of n_1
14    for each child y' of y
15      if C[x', y'] ≠ Φ then let C[x', y'] be a new child of n_2 //C[x',y'] has a single member
16  for each D_child x' of x
17    create a P_node n_3 as a new child of n_1
18    for each child y' of y
19      if D[x', y'] ≠ Φ then let D[x', y'] be a new set of children of n_3  //D[x',y'] may have
                                                                               //multiple members
20  C[x, y] = {n_1}; return
```

C_Build(x, y, C[x, y]) stores into entry C[x, y] an I_node for y, or Φ, depending on whether or not the sub-pattern rooted at x can match the subtree rooted at y in such a way that all the return nodes in the sub-pattern are matched to the return nodes in the subtree. If the match is successful, it also creates the P_node children for the I_node, one for each child of x (lines 13 and 17). These P_nodes in turn have their I_node children (lines 15 and 19), storing the matching for the children of x. Line 5 checks and see if C[x', y'] is set by some earlier calls. This may happen due to the presence of the loop in line 22 in D_Build( ), whose pseudo-code is shown below.

D_Build(PatternTreeNode x, InputTreeNode y, SetofINodes D[x, y])

```
1   L ← Φ
2  if (label(x) ≠ * and label(x) ≠ label(y)) then go to 22     //cannot match y, let's turn
                                                               // to its descendants
3  if (x ∈ return_nodes(p) and y ∉ return_nodes(t)) then go to 22
4    for each C_child x' of x
5      for each child y' of y
6        if C[x', y'] = U then C_Build(x', y', C[x', y']))
7        if C[x', y'] = Φ for every child y' of y,  then go to 22
8     for each D_child x' of x
9        for each child y' of y
10          if D[x', y'] = U then D_Build(x', y', D[x', y'] )
11          if D[x', y'] = Φ for every child y' of y, then go to 22
12   create an I_node n_1 labeled y
13  for each C_child x' of x
14     create a P_node n_2 as a new child of n_1
15     for each child y' of y
16        if C[x', y'] ≠ Φ then let C[x', y'] be a new child of n_2
17  for each D_child x' of x
18     create a P_node n_3 as a new child of n_1
19     for each child y' of y
20         if D[x', y'] ≠ Φ then let D[x', y'] be a new set of children of n_3
21  L ← {n_1}
22  for each z ∈ children(y)  //recursively determine if x can match the descendants of y
23     if D[x, z] = U then D_Build(x, z, D[x, z])
24     L ← L ∪ D[x, z]
25  D[x, y] ← L; return
```

D_Build(x, y, D[x, y]) stores into entry D[x, y] a set of I_nodes if the sub-pattern rooted at x can match the subtree rooted at y and/or y's descendants, such that the return nodes are matched to the return nodes, and Φ otherwise. The way it stores the matching information for the children of x is identical to that in C_Build( ).

For the time complexity of the algorithm, notice that for any pair of a node in p and a node in t, once the corresponding entry in C or D array is set by a call, then no later calls will be made on the same pair. This means all the calls are made on different pairs. Thus the total number of calls is at most $|p|\bullet|t|$, implying a time complexity of $O(|p|\bullet|t|)$. (Note that once an embedding is returned, we need to check further if it maps the return nodes of p *onto* the return nodes of q. This can be done by simply comparing |e(return_nodes(p)| and |return_nodes(q)|, and incurs only a linear time complexity.)

3.3 A Weaker Condition

Correct rewriting imposes a fixed relationship between the return nodes of p and their RNM correspondences in q. This suggests that the paths delimited by the return nodes in p and those by their RNM correspondences in q may need to follow some patterns. In this section, we will look at this in detail. In the following, for any pattern tree or input tree, we use the notation $<a_1, \ldots, a_m>$ to refer to a path that starts with node a_1 and ends at node a_m. Note that this notation does not contain information about the kind of edges connecting the adjacent nodes when the path is in a pattern tree. Also, for any two paths λ and μ, and mapping g: nodes(λ) → nodes(μ), we use the notation g(λ) = μ to indicate that g maps respectively the beginning and the end nodes of λ to the beginning and the end nodes of μ, and preserves the order of any two nodes in λ when it maps them to different nodes in μ.

Definition 3: Let $\lambda = <x_1, \ldots, x_m>$ and $\mu = <y_1, \ldots, y_n>$. We say e: nodes(λ) → nodes(μ) is a *relay* from λ to μ, denoted as $e_{relay}(\lambda) = \mu$ (or simply $e_{relay}(\lambda)$), if e(λ) = μ, and the following conditions hold true:

1. m = 1, n = 1 and (label(x_1) = * or label(x_1) = label(y_1)), or
2. m = 2, n = 2, $e_{relay}(x_1) = y_1$, $e_{relay}(x_2) = y_2$, there is a child edge from x_1 to x_2, and there is a child edge from y_1 to y_2, or
3. $2 \le m \le n$, $e_{relay}(x_1) = y_1$, $e_{relay}(x_m) = y_n$, each node in sub-path $<x_2, \ldots, x_{m-1}>$ is labeled *, does not branch, and path $<x_1, \ldots, x_m>$ contains at least one descendant edge, or
4. there is $1 \le i \le m$, $1 \le j \le n$, such that $e_{relay}(<x_1,\ldots,x_i>)=<y_1,\ldots,y_j>$, and $e_{relay}(<x_i,\ldots,x_m>)=<y_j,\ldots,y_n>$

The idea is that for any embedding g from μ to an input path, we can compose e_{relay} and g to form an embedding from λ to the same input path. This is clearly the case when conditions 1 and 2 are met. When condition 3 is met, we retain the mapping for the two end nodes of λ, but apply the prefix-suffix mapping introduced in Section 3.1 for the inner nodes if necessary. Condition 4 simply makes the definition recursive. Consider Figure 4.

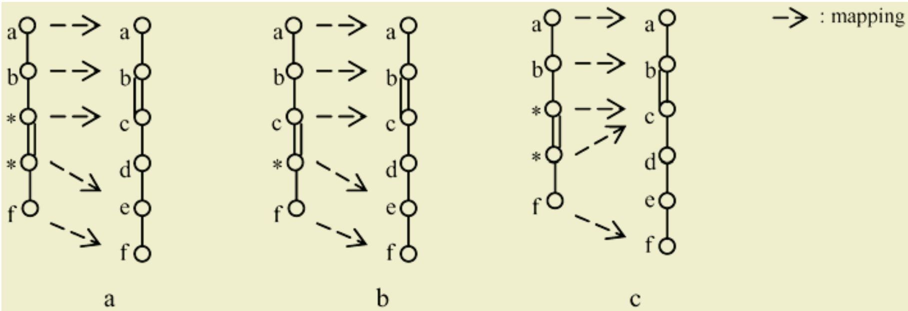

Fig. 4. a and c: relay, b: not relay

In Figure 4.a, the mapping is a relay, since the path on the left can be decomposed into two sub-paths, <a, b> and <b, f>. The former meets condition 2, and the latter meets condition 3. Then by applying condition 4 the claim follows. To see how condition 3 meets our intuition in this instance, suppose an embedding g matches path <a, f> on the right to an input path γ, and matches descendant edge <b, c> to the sub-path <b, c> of γ. Then the mapping shown in the figure as is can be composed with g to form an embedding from the path on the left to γ. On the other hand, if g matches <b, c> to sub-path <b, x, c> of γ, we directly match the upper node labeled * on the left to node x, and keep the rest of the composition unchanged. This still results in an embedding from the path on the left to γ. In general, no matter what the input path is, we can form an embedding from the path on the left to it, as long as there is an embedding from the path on the right to it. This is the idea behind the notion of relay.

Similarly, the mapping in Figure 4.c is also a relay. The mapping in Figure 4.b, however, is not a relay. This is because the path on the left cannot be decomposed in accordance with condition 4. Thus, for example, if the right path is matched to input path γ but the descendant edge <b, c> in it is matched to sub-path <b, x, c> of γ, it is not possible to form an embedding from the left path to γ.

In the general case, we have the following lemma.

Lemma 1. Let λ and μ be two pattern tree paths, and g(λ) = μ be an arbitrary relay. Let t be an input path and e: nodes(μ)→nodes(t) be an embedding such that e(start-node(μ)) = start-node(t) and e(end-node(μ)) = end-node(t). Then there is an embedding f: nodes(λ)→nodes(t) such that f(start-node(λ)) = start-node(t) and f(end-node(λ)) = end-node(t).

Idea of Proof: If g meets conditions 1 or 2, we simply let f = e ∘ g. If e meets condition 3, then we can perform prefix-suffix mapping from the inner nodes of λ to the inner nodes of t, resulting in an embedding. If we have to decompose λ according to condition 4, then we can apply the above procedure to the resultant sub-paths. □

Based on the above lemma, we have the following

Theorem 3: Let p and q be pattern trees. If there is a mapping g: nodes(p) → nodes(q) satisfying the following conditions: (1) g(return_nodes(p)) = return_nodes(q), (2) for all path λ in p in which the beginning and ending nodes are return or leaf nodes, and the remaining are transit nodes, g is a relay, then (p, q, g) is a correct rewriting.

Proof: By condition 1, g can match return nodes in p only to return nodes in q. Let t be an input tree, and e: nodes(q) $\rightarrow$ nodes(t) be an embedding . Let λ be an arbitrary return-or-leaf-node-delimited path in p and $g(\lambda) = \mu$ where μ is a path in q. Let n be whichever delimiting node of λ that is a return node. By condition 1 in the theorem we have g(n) is also a return and delimiting node of μ. Applying lemma 1 to λ, we obtain an embedding that matches λ to $e(\mu)$, and n to e(g(n)). Note that since any inner node in λ does not branch, which is required by condition 3 in Definition 3, the possible prefix-suffix mapping performed for the inner nodes of λ will not affect the embeddings for other paths in p. Thus the embedding obtained collectively for all such paths is indeed an embedding from p to t. □

Is Theorem 3 weaker than Theorem 2? The answer is yes. The following is why. Suppose Condition 2 in Theorem 2 is true. Let $h = \pi^{-1} \circ e$: nodes(p) $\rightarrow$ nodes(q). The arguments in the proof of Theorem 2 have shown that for all $o \in$ nodes(p), label(o) $\neq$ * $\Rightarrow$ label(o) = label(h(o)), and for all $n_1, n_2 \in$ nodes(p), if there is a child edge from n_1 to n_2, then there is a child edge from $h(n_1)$ to $h(n_2)$, Thus the first two conditions in Definition 3 are true. This means h is a relay for any path in p. The proof for Theorem 2 also shows h(return_nodes(p)) = return_nodes(q). Together, these imply the two conditions in Theorem 3 (with h substituting for g). On the other hand, the conditions in Theorem 3 do not imply those in Theorem 2: they do not require that transit nodes labeled * not be incident with descendant edges. This means the assumption in Theorem 3 is strictly weaker than that in Theorem 2.

Theorem 3 gives an approach to searching for correct rewriting, i.e., searching for relays. In reality, we can narrow the search space by discarding “replicas”. Note that it is possible that multiple relays are identical when they are restricted to return nodes. When this happens, we need to consider only one of them.

Theorem 4: Let λ and μ be two pattern tree paths, and $g(\lambda) = \mu$ be a relay. Then there is a relay $f(\lambda) = \mu$ such that $\pi \circ f$: nodes(λ)$\rightarrow$ nodes(t_0) is an embedding, where t_0 is the 0-extension of μ and π maps each node in μ to its copy in t_0. (Refer to Sec. 2.1.)

Idea of Proof: If g meets condition 1 or 2, let f = g. In this case t_0 is identical to μ, since μ does not contain descendant edges. Thus $\pi \circ f$ is an embedding. If g meets condition 3, we obtain f by performing prefix-suffix mapping from the inner nodes of λ to the inner nodes of μ. In this case t_0 is identical to μ., except that in the place of each descendant edge in μ is an edge of t_0. Since prefix-suffix mapping always maps a child edge in λ to an edge in μ, which is also an edge in t_0, $\pi \circ f$ is surely an embedding. Note that f is still a relay from λ to μ, since prefix-suffix mapping does not alter condition 3. In case we need to decompose λ according to condition 4, we apply the above procedure to the resultant sub-paths of λ. □

Note that in Theorem 4, if n is an end node of λ, then f(n) = g(n). Thus if we want to find a relay that satisfies the conditions in Theorem 3, we can consider f only. For this purpose, according to Theorem 4 also, we can first find all the embeddings from λ to t_0, then determine those that can be transformed to relays from λ to μ. Searching for embeddings can be done by applying Algorithm 1. The transformation is done simply by applying π^{-1} to the embeddings. (In the following algorithm this is done implicitly.)

In the following algorithm, we use the term ‘D_edge’ to refer to an edge in t_0 that corresponds to the descendant edge in q. We term a path *segment* that is delimited by return or leaf nodes in p.

Algorithm 2

Input: pattern tree p; input tree t_0, with a set of D_edges; an embedding e: nodes(p) → nodes(t_0)

Output: a decision of whether or not e is a relay for all the segments in p

```
1   s ← next segment
2   if s = NULL return ‘yes’
3   if ¬Relay(s, e) return ‘no’
4   go to 1
```

Boolean Relay(Segment s, Embedding e)

```
1   <a, b> ← first child edge ε in s such that e(ε) is a D_edge;
2   if <a, b> = NULL then return ‘yes’
3   if label(b) ≠ * then return ‘no’
4   c ← first successor of b in s that is incident with a descendant edge and delimits
        a star-path with no branching
5   if c = NULL, then return ‘no’
6   d ← child of c
7   if Relay(<d, …, end-node(s)>, e) then return ‘yes’
8   return ‘no’
```

The idea should be clear. Each child edge in the prefix <start-node(s), a> is not matched to a D_edge, thus we have e_{relay}(<start(s), a>) is true by Condition 2 and 4 in Definition 3. If the test in line 5 evaluates to true, then there does not exist a sub-path starting from *a* that meets any condition in Definition 3, else path <a, …, d> meets Condition 3, e_{relay}(<a,…,d>) is true. Thus e_{relay}(start(s), d) is true by Condition 4. This means e_{relay}(s) is true iff e_{relay}(<d, …, end(s)>) is true. For example, when we apply the algorithm to the left path in Figure 4.a, two recursive calls will be made. The top call is on the entire path, and the nested call is on the path containing the bottom two nodes.

For time complexity, it is easy to see that on any path, Relay( ) returns in O(m) time where m is the number of nodes in the path, implying that the algorithm runs in O(n) where n is the number of nodes in p. Thus, to find all the correct rewriting based on Theorem 3, in addition to the cost of running Algorithm 1 on p and t_0, we need to pay an extra cost of O(w•n), where w is the number of embeddings from nodes(p) to nodes(t_0). In the general case, w is much smaller than n. Therefore, this extra cost is close to linear. Note that, if Algorithm 2 returns ‘no’ for all the embeddings, this does not necessarily mean that there does not exist a correct rewriting for p and q. This is because the conditions in Theorem 3 are not necessary conditions. When that happens, we may need to resort to the method based on Theorem 1 for the final judgment. (Refer to the discussion in the next section.)

4 Conclusion

We study the issue of query rewriting using views for XPath queries in a general setting. Several issues are studied, including conditions for correct query rewriting, search methods, and trade-offs between efficiency and applicability. Our solution can be used as a basis for developing solutions suited to special requirement. For example, for small queries, the method based on Theorem 1 should be used, as it is both sound and complete. If a query is large, but has no transit node with a wildcard label that is associated with a descendant edge, then the method based on Theorem 2 is the best, since it is both sound and complete (under that condition), as well as efficient. These methods can also be used in a hybrid manner. For example, if the condition in Theorem 2 is not met, we use the method based on Theorem 3. If it returns 'no', we then use Theorem 1.

There are several related issues. Suppose there does not exist a correct rewriting for p and q. (This is the case, for example, when p does not contain q.) How do we search efficiently for another query q' $\subset$ q such that there exists a correct rewriting for p and q'? Another issue is the extension of the model to incorporate more features of XPath queries. These issues can be viewed as immediate follow ups of the work in this paper, and deserve further study.

References

1. F. Ozcan, K. Beyer and R. Cochrane, "A Framework for Using Materialized XPath Views in XML Query Processing", In 30th VLDB Conf., 2004, pp 60 – 71.
2. D. Calvanese, G. Giacomo, M. Lenzerini and M. Vardi, "Answering Regular Path Queries Using Views", In 16th Intl. Conf. On Data Engg., 2000, pp 389 - 398.
3. L. Chen and E. Rundensteiner, "ACE-XQ: A Cache-Aware XQuery Answering System", In WebDB, pp 31 – 36.
4. V. Cristophides, S. Cluet and J. Simeon, "On Wrapping Query Languages and Efficient XML Integration", In SIGMOD Conf., 2000, pp141 – 152.
5. A. Deutsch and V. Tannen, "Reformulation of XML Queries and Constraints", In 9th Intl. Conf. on Database Theory, 2003, pp 225 – 241.
6. F. Neven and T. Schwentick, "XPath Containment in the Presence of Disjunction, DTDs and Variables", In 9th Intl. Conf. on Database Theory, 2003, pp 315 – 329.
7. G. Grahne and A. Thomo, "Query Containment and Rewriting Using Views for Regular Path Queries Under Constraints, In 22nd PODS, 2003, pp 111 – 121.
8. T. Kirk, A. Levy, Y. Sagiv and D. Srivastava, "The Information Manifold", AAAI Spring Sym. on Information Gathering from Heterogeneous and Distributed Environments, 1995.
9. A. Levy, A. Mendelzon, Y. Sagiv and D. Srivastava, "Answering Queries Using Views", In 14th PODS, 1995, pp 95 – 104.
10. G. Miklau and D. Suciu, "Containment and Equivalence for an Xpath Fragment", In 21st PODS, 2002, pp 65 - 76
11. P. Mitra, "An Algorithm for Answering Queries Efficiently Using Views", In 12th Australian Database Conf., 2001, pp 99 – 106.
12. Y. Papakonstantinou and V. Vassalos, "Query Rewriting Using Semistructured Views", In SIGMOD Conf., 1999, pp 455 – 466.

13. R. Pottinger and A. Levy, “A Scalable Algorithm for Answering Queries Using Views”, VLDB Journal, 10(2-3), 2001, pp 182 - 198.
14. X. Qian, “Query Folding”, In 12th Intl. Conf. On Data Engg., 1996, 48 – 55.
15. J. Shanmungasundaram, J. Kiernan, E. Shekita, C. Fan and J. Funderburk, “Querying XML Views of Relational Data”, In 27th VLDB Conf., 2001, pp 261 – 270.
16. J. Tang and S. Zhou, “Rewriting Queries Using Views for XML Documents”, TR-04, MUN, 2004.
17. Yu and L. Popa, “Constraint-Based XML Query Rewriting for Data Integration”, In SIGMOD Conf., 2004, pp 371 – 382.
18. XQuery: A Query Language for XML, http://www.W3.org/TR/xquery, 2003.