Indexing graph-structured XML data for efficient structural join operation

Qun Chen a,*, Andrew Lim a, Kian Win Ong b, Ji Qing Tang a

a Department of Industrial Engineering and Engineering Management, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong
b Department of Computer Science and Engineering, University of California at San Diego, 9500 Gilman Drive, La Jolla, CA 92093-0114, USA

Received 3 February 2005; accepted 26 May 2005
Available online 23 June 2005

Abstract

Structural join has been established as a primitive technique for matching the binary containment pattern, specifically the parent–child and ancestor–descendant relationship, on the tree XML data. While current indexing approaches and evaluation algorithms proposed for the structural join operation assume the tree-structured data model, the presence of reference links in XML documents may render the underlying model a graph instead. In the more general category of semi-structured data, of which XML is an example, the data model is also usually supposed to be of graph structure. In this paper, we present an indexing approach and corresponding evaluation algorithms for efficiently performing the structural join operation on graph-structured data. Our approach encodes the structural containment relationship of a graph on multiple nested tree-structured layers, probably with the exception of the last one. With each tree-structured layer indexed with the inverted technique, the structural join operation on a graph can therefore be accomplished through recursively performing structural joins on nested layer trees. Our extensive experiments on both benchmark and synthetic XML data indicate that our proposed approach has good potential to perform significantly better than existing ones in term of both the I/O and CPU cost.

© 2005 Elsevier B.V. All rights reserved.

* Corresponding author.
E-mail addresses: qunchen@ust.hk (Q. Chen), iealim@ust.hk (A. Lim), okwin@ucsd.edu (K.W. Ong), jiqingtang@gmail.com (J.Q. Tang).
1. Introduction

All major XML query languages, such as XQuery [3] and Quilt [4], use path expressions to retrieve relevant parts of XML documents. A path expression usually consists of a sequence of location steps that specify the path from the document root to the target elements. For example, the path expression, //paper[author = "Jeffrey"]/title/text(), retrieves titles of papers authored by "Jeffrey".

A complex XML query pattern can be naturally decomposed into several basic parent–child and ancestor–descendant binary relationships. In the semi-structured data [7,8], of which XML is an example, such basic containment queries would also need to be supported by virtually all query engines. On the tree-structured data model of XML documents, the inverted index technique [9–11], which has its origin in the information retrieval systems, has been exploited to encode trees such that two nodes’ containment relationship can be decided from their codes alone. More specifically, positions of elements and string values on an XML tree are presented as (LeftPos:RightPos, Level). This representation succinctly captures the containment structural relationship between nodes on the tree: the element A contains the element B if and only if A’s position interval covers B’s position interval. Based on this representation, the stack-based structural join algorithm [12] achieves the I/O and CPU cost linear with the sum of sizes of inputs and output for the binary containment pattern matching on sorted inputs. The structural join technique enables the set-at-a-time strategy for matching XML query patterns. More recent work, such as PathStack and TwigStack [13] algorithms proposed for matching path and twig patterns respectively, further demonstrated the superiority of this strategy over the node-at-a-time tree traversal approach.

Our work is motivated by the fact that the existing structural join technique assumes the underlying tree structures of XML documents. This assumption is valid only if we disregard reference links between XML tree nodes. In several existing XML applications [14,15,17,18], reference edges were already treated as first class citizens, making the underlying data model of XML a graph instead of a tree. In addition, in the more general category of semi-structured data, the underlying data model is even more likely to be a graph. On the issue of binary containment, we note that the XPath standard [2] for path expressions has given a higher priority to tree edges. While there are ancestor and descendant axes for tree edges, there are no equivalent axes for reference edges. As a result, containment queries have been traditionally defined only on tree edges: each idref operator has to be explicitly stated. We take the view as in [4,5], however, that in many applications, for instance EDI (electronic data interchange), it makes sense to blur the distinction between the tree edge and the reference link since both logically “point to” other objects. This relaxation also allow users to write simple edge-traversal queries without regarding how a relationship between two objects is modeled within an XML database, be it a tree or reference edge. As a result, in this paper, we consider the general case of defining the containment query simply as the reachability between two nodes on the XML data graph. Elements can reach others transitively through both idref and tree edges alike.
We note that the family of bisimilarity-based indexing techniques [1,6,20,18,16] proposed for XML querying does consider the graph-structured model. Unfortunately, to fully support the binary containment query, the smallest singular index graph is the \((F + B)\)-index [16], which remains relatively large and may even approach the size of the data graph in certain cases. Therefore, it is expected to have limited usefulness in speeding up the containment queries on the graph model. This claim will be empirically validated by our experimental study. Note that the existing work on the bisimilarity-base indexes instead focuses on regular path expressions consisting of a sequence of parent–child but not ancestor–descendant axes.

The main contributions of this paper can be summarized as:

(1) We propose an indexing technique for facilitating the structural join operation on graph-structured model. It is based on the strategy of encoding binary containment relationships of a graph on multiple nested tree-structured layers.

(2) Based on our indexing framework, we present efficient algorithms for evaluating the binary containment pattern. Taking advantage of the previously proposed stack-based algorithm for the structural join operation on the tree structure, the algorithms consist of recursively performing structural joins on nested layers.

(3) We conduct an experimental study of our indexing approach on the benchmark and synthetic XML data sets. Our results demonstrate that, compared with the bisimilarity-based approach, our technique significantly cuts the I/O and CPU cost of the structural join operation on the XML graph data, where IDREFs are present only on a small portion of elements and the nested/reference direct containment relationships between elements more or less follow the 1-versus-N or N-versus-1 pattern.

The rest of this paper will be organized as follows. Preliminaries will be presented in Section 2, which include descriptions of the XML data model, the inverted indexing technique and the structural join algorithm on the tree model. Our proposed indexing approach for the graph data model will be described in Section 3. Section 4 will be devoted to detailing the evaluation algorithms. We will present our experimental study in Section 5 and discuss more related work in Section 6. Finally, we will conclude this paper in Section 7 with some final thoughts about our work and suggestions for future research.

2. Preliminaries

Documents in an XML database are usually parsed into labeled trees. References between elements established through ID/IDREF attribute [1] or Xlink [6] are therefore mapped to reference edges between tree nodes on the data model. Each node has a unique id. Fig. 1(a) shows the data model of an example XML document storing the publication information of faculty members. The nested structural containment relationships in the document are represented by solid arrows, the reference links by dashed arrows. The reference link between elements person and paper stands for the “co-author” relationship. In this paper, we treat the reference links between elements as normal containment relationships. The underlying data model is thus a graph instead of a tree.
The inverted index has been used to encode XML tree data. Positions of nodes are represented by the 3-tuple code \((\text{DocNo}, \text{LeftPos:RightPos}, \text{LevelNo})\). \text{DocNo} is the identifier of the document. The pair of \text{LeftPos} and \text{RightPos} is generated by traversing the tree in the depth-first order and sequentially assigning a number at each visit of the nodes. \text{LevelNo} is the nesting depth of the nodes. The inverted index representation of an example XML tree is shown in Fig. 1(b). The traversal begins with the root node 1 and assign \text{LeftPos} = 1 to it. It continues to walk the nodes 2, 4 and 8, and assign \text{LeftPos} = 2, 3, 4 to them, respectively. Since the node 8 is a leaf node, its \text{RightPos} value can be set to be equal to its \text{LeftPos} value. According to the depth-first order, the process comes back to visit node 4 at the next step. Since it is the second visit to node 4, the number 5 is assigned to its \text{RightPos} value. The process continues to walk the nodes 5, 9, 5, 2 and 1 sequentially. Their \text{LeftPos} and \text{RightPos} values are assigned correspondingly.

With the inverted index representation, the structural relationship between two tree nodes can be determined easily: (1) containment or ancestor–descendant relationship: a tree node \(n_1, (LP_1:RP_1,L_1)\), contains a tree node \(n_2, (LP_2:RP_2,L_2)\), if and only if \(LP_1 < LP_2\) and \(RP_1 > RP_2\); (2) direct containment or parent–child relationship: \(n_1\) directly contains \(n_2\) if and only if \(LP_1 < LP_2, RP_1 > RP_2\) and \(L_1 = L_2 - 1\). One advantage of the inverted index is that checking the ancestor–descendant structural relationship is as easy as checking the parent–child structural relationship.

2.1. Structural join based on tree XML model

The structural join operation on the tree model takes two input inverted lists, both sorted on their \text{LeftPos} values, and conceptually interleaves them to determine the ancestor–descendant relationship. The Stack-Tree algorithm proposed in [12] uses the stack to store a sequence of ancestor nodes while proceeding the merge operation, with each node in the stack being a descendant of the node below it. It works by repeatedly considering the next node with the smaller value.
of $LeftPos$ in two input lists. The details of their algorithm are described in Algorithm 1. Analytically, the Stack-Tree algorithm avoids multiple scans on the input lists, thus achieves the I/O and CPU cost linear with the sum of sizes of the input lists and the output list.

**Algorithm 1. The Structural Join Algorithm on Tree Model: $A || B$**

Originally set the stack $S$ to be empty;

repeat

retrieve the next entry $e$ with the smaller $LeftPos$ value from $L_A$ and $L_B$;

while $top(S).RightPos < e.LeftPos$ do

    pop($S$);

end while

if $e$ is from $L_A$ then

    push($e, S$);

else if $e$ is from $L_B$ then

    for each entry $e_a$ in $S$ do

        return ($e_a, e$);

    end for

end if

until either (1) the end of $L_B$ is reached or (2) the end of $L_A$ is reached and $S$ is empty

---

3. Constructing indexes for graph data

In this section, we present the technique to index graph data for efficiently performing the structural join operation. Note that for the direct containment relationship, or the parent–child binary pattern, even the naive approach of directly indexing all edges of the graph scales linearly with the size of data. Therefore, we focus on handling the indirect containment relationship. Extending our indexing technique to accommodate the case of direct containment is relatively easy. It will be addressed along with the evaluation algorithms in the next section.

3.1. Motivation

Using XML data model as the example, we denote the nodes where reference links begin as outgoing ports and the nodes where reference links end as incoming ports. Our proposal is motivated by the observation that the indirect containment relationship between nodes can be determined through exploring the containment on the tree structure and the reachability from outgoing ports to incoming ports. In other words, a node $a$ contains the other node $b$ if and only if they satisfy either one of the following two conditions: (1) node $a$ contains node $b$ on the tree structure; (2) node $a$ contains some outgoing port $p_o$ and node $b$ is contained by some incoming port $p_i$ on the tree structure, and $p_o$ can reach $p_i$. The containment relationship between nodes on the tree structure can be determined efficiently by the inverted indexing technique. However, fully storing the reachability between ports may take space up to $O(t^2)$ in the worst case, where $t$ is the total number of ports. In the case that $t$ is large, the performance of the evaluation process may be
severely affected. Therefore, instead of directly recording the reachability between ports, conceptually our proposed indexing approach recursively represents the reachability between ports with a graph, which is itself encoded by a tree structure and its own port reachability graph, until the total number of ports is reduced to a manageable level in term of both the memory consumption and the evaluation cost.

3.2. On the issue of strongly connected component

Note that on a directed graph, the nodes in a strongly connected component have the same reachability with respect to other nodes. Therefore, for the general case considered by this paper, where the containment relationship between two nodes is defined to be their reachability, we treat the nodes in a strongly connected component as a whole unit while encoding containment relationships. Technically speaking, in the preprocessing phase, we shrink any strongly connected component present on the graph data into a single unit and record the mapping between them. As a result, the target graph to be encoded is acyclic.

3.3. Construction algorithm

The construction algorithm consists of the following two steps: (1) conducting a depth-first search (DFS) on the directed acyclic graph to extract a maximal tree structure; (2) with the edges not incorporated on the tree treated as reference links, building the reachability graph of outgoing and incoming ports. The algorithm repeats these two steps with the reachability graph as the input until the size of the resulting reachability graph is reduced to a manageable level. The algorithm results in multiple nested tree-structured layers and one final layer of graph structure. If the result of the first step includes multiple disjoint trees, a virtual root will be inserted to serve as the parent of all tree roots such that each layer has exactly one tree.

Performing DFS is a well-studied standard operation on directed graphs. We instead focus on elaborating the algorithm for constructing the reachability graph of ports. Intuitively, the purpose of the reachability graph \( G_r \) is that for any two nodes \( n_1 \) and \( n_2 \) satisfying that \( n_1 \) contains \( n_2 \) on the original data graph \( G \) but \( n_1 \) does not contain \( n_2 \) on the extracted tree \( T \), their containment relationship should be captured by the reachability graph. First of all, all edges of \( G \) left out by \( T \) (or reference links in XML term) should be included on \( G_r \). Secondly, some edges of \( G_r \) are to be inherited from the containment relationship between ports on \( T \).

Suppose that the node \( n_i \) contains \( n_j \) through the path \( P = n_i n_{i+1}, \ldots, n_j \) on \( G \). We consider the first and last reference link on \( P \), \( n_{i_1} \rightarrow n_{i_1+1} \) and \( n_{i_2} \rightarrow n_{i_2+1} \). Since the containment relationship between node \( n_i \) and \( n_j \) is established through the outgoing port \( n_{i_1} \) and incoming port \( n_{i_2+1} \), there should exist a path from \( n_{i_1+1} \) to \( n_{i_2} \) on \( G_r \). Therefore, all containment relationships on \( T \) between the ports on the path from \( n_{i_1+1} \) to \( n_{i_2} \) should be incorporated into \( G_r \). The containment relationship is from an incoming port to an outgoing port of the next reference link. Generally speaking, if there is a path on \( T \) from an incoming port \( n_i \) to an outgoing port \( n_o \), \( n_i \) should reach \( n_o \) on \( G_r \).

A naive solution to incorporate the containment relationships of \( T \) into \( G_r \) is to add an edge from \( n_i \) to \( n_o \) if there is a path from \( n_i \) to \( n_o \) on \( T \), in which \( n_i \) is an incoming port and \( n_o \) is an outgoing port. It takes \( \mathcal{O}(t^2) \) CPU time, where \( t \) is the total number of ports, provided that an inverted index is constructed for \( T \). Taking advantage of the tree structure of \( T \), we present
an alternative algorithm with the running time of $O(\text{size}(T))$. It has an additional benefit that the containment relationships on $T$ remain tree-structured on $G_r$. Thus in the worst case they can be encoded on one single layer.

The algorithm works by performing a depth-first traversal on $T$. Its purpose is to establish correct containment relationships between ports of $T$ on $G_r$. Besides the stack $S_T$ used to traverse $T$, we use an additional stack $S_p$ to store nested port nodes. We also keep a state variable $st(n)$ for each entry $n$ in $S_p$ to indicate whether the containment relationship between $n$ and the ports below $n$ in $S_p$ should be incorporated on $G_r$. Initially, $S_p$ is set empty. When the port nodes are pushed into or popped from stack $S_T$ while traversing $T$, they may also be pushed into or popped from stack $S_p$. If the current node $n$ being pushed into $S_T$ is an incoming port, the algorithm records it by pushing it into the stack $S_p$ because $n$ may have outgoing port nodes as its descendants on $T$. Its state variable $st(n)$ is set to 0. In the case that the node $n$ is an outgoing port, it is pushed into $S_p$ only if $S_p$ is not empty, and its state variable $st(n)$ is set to 1. Note that if $S_p$ is empty, it means that the outgoing port node $n$ has no ancestor incoming port node on $T$, thus $n$ does not need to be recorded in $S_p$. While popping a port node $n$ from the stack $S_T$, the algorithm also pops $n$ from stack $S_p$ if $n$ is at the top of $S_p$. After $n$ is popped from $S_p$, if $st(n) = 1$, the state variable of the current top port of $S_p$ is reset to 1 and an edge from it to $n$ is inserted on $G_r$. The algorithm is described in Algorithm 2. Its correctness is established by Lemma 1.

**Lemma 1.** If there is a path from an incoming port $n_i$ to an outgoing port $n_o$ on the extracted tree $T$, there is also one from $n_i$ to $n_o$ on the structure consisting of edges inserted as a result of Algorithm 2.

**Proof.** From the algorithm description, ports $n_i$ and $n_o$ should both be in the stack $S_p$ at some point of the procedure. The value of $n_o$’s state variable $st(n_o)$ should be 1. Therefore, when $n_o$ is popped out, an edge will be added between $n_o$ and the port node below it, $n_p$, in $S_p$ and $st(n_p)$ will be set to 1. As a result, the edges will similarly be added between any two adjacent entries in $S_p$ from $n_i$ to $n_o$. □

It is clear now that all edges inserted into $G_r$ as the result of Algorithm 2 amount to a tree structure since they have no cycle and each port has at most one parent. A construction example from $G$ to $G_r$ has also been shown in Fig. 2. Note that only containment relationships from incoming ports to outgoing ports on the extracted tree $T$ need to be incorporated on the reachability graph $G_r$. Therefore, in Fig. 2, there is an edge from node 2 to node 8 on $G_r$ since node 2 is an incoming port and node 8 is an outgoing port. Even though there is a path from node 4 to 10 on the extracted tree $T$, we do not need to insert an edge between them since node 4 is an outgoing port and 10 is an incoming port.

**Algorithm 2.** The Algorithm of Inserting Inheritance Edges on Reachability Graph

Set the stack $S_p = \emptyset$;

while traversing $T$ in depth-first order using the stack $S_T$ do
    if upon pushing a node $n$ into $S_T$ then
        if $n$ is an incoming port then
            push($n, S_p$);
            $st(n) = 0$;
else if \( n \) is an outgoing port and \( S_p \neq \emptyset \) then

\[
\text{push}(n, S_p);
\]

\[
\text{st}(n) = 1;
\]

end if

end if

if upon popping a node \( n \) from \( S_T \) then

if \( \text{top}(S_p) == n \) then

\[
\text{pop}(S_p);
\]

if \( \text{st}(n) == 1 \) and \( \text{top}(S_p) \neq \text{null} \) then

insert an edge from \( \text{top}(S_p) \) to \( n \) on \( G_r \);

\[
\text{st}(\text{top}(S_p)) = 1;
\]

end if

end if

end if

end while

The reachability graph \( G_r \) consists of all edges in \( G \) left out by the extracted tree \( T \) and all edges inserted by the Algorithm 2. We have the following theorem.

**Theorem 1.** For any two nodes \( n_i \) and \( n_j \) satisfying that \( n_i \) contains \( n_j \) on the original data graph \( G \), either one of the following two statements is true: (1) \( n_i \) contains \( n_j \) on \( T \); (2) \( n_i \) contains some outgoing port \( p_o \) and \( n_j \) is contained by some incoming port \( p_i \) on \( T \), and \( p_o \) contains \( p_i \) on \( G_r \).

**Proof.** Suppose that node \( n_i \) reaches \( n_j \) through the path \( P \) on \( G \). If there is no reference link on \( P \), obviously the first statement is true. Otherwise, consider the first and the last reference link on the path \( P \), \( p_{o1} \rightarrow p_{i1} \) and \( p_{ok} \rightarrow p_{ik} \). It is clear that \( n_i \) contains \( p_{o1} \) and \( p_{ik} \) contains \( n_j \) on \( T \). We proceed to prove that \( p_{o1} \) can reach \( p_{ik} \) on the reachability graph \( G_r \). Assume that the references links between \( p_{o1} \) and \( p_{ik} \) on \( P \) are \( p_{o1} \rightarrow p_{i1}, p_{o2} \rightarrow p_{i2}, \ldots, p_{ok} \rightarrow p_{ik} \). According to the Lemma 1, there are paths on \( G_r \) from \( p_{i1} \) to \( p_{i2}, \ldots \), and from \( p_{i(k-1)} \) to \( p_{ok} \) on \( G_r \). Therefore, \( p_{o1} \) contains \( p_{ik} \) on \( G_r \). \( \Box \)
The implication of Theorem 1 is that we can perform the structural join operation on graph-structured data through conducting the structural join on a tree structure and then consulting a reachability graph, which may be potentially much smaller in size if the reference links are present on only a small portion of graph nodes. Our construction algorithm recursively indexes the reachability graph by building its own tree structure and reachability graph.

3.4. A remedy for the N-versus-1 containment pattern

It is observed that the effectiveness of our indexing approach depends on how fast the size of the reachability graph is reduced as the nested tree-structured layers are being built. After studying the definition of ID/IDREF attribute in DTD or XML Schema [1], and Xlink [6], we expect that most direct containment relationships between nodes in XML data follow the 1-versus-N or N-versus-1 (N is large) pattern. The containment as a result of the nested document structure tends to be of the 1-versus-N pattern, while the containment as a result of the reference link structure tends to be of the N-versus-1 pattern. The edges of a 1-versus-N pattern can be incorporated on one single layer. To encode a N-versus-1 pattern, we need at least N layers since on each layer each node has at most one parent. As an example, the structure shown in Fig. 3 needs k layers to...
entirely encode its containment relationships. Therefore, in XML data where a lot of $N$-versus-1 containment patterns exist, the reduction speed of the reachability graph’s size may be intolerably slow. This phenomenon will be experimentally verified in Section 5.

Our remedy is based on the finding that the inverted index technique is also applicable to the structure satisfying the property that reversing the directions of its all edges results in a directed tree. We denote such structure as RTS (reverse tree structure) and the structure after the reversing operation as RTS$_r$. Since RTS$_r$ is a tree structure (TS), it can be indexed by the inverted technique. Keeping inverted indexes of the nodes on RTS$_r$ intact, we can say that node $n_i$ contains $n_j$ on the structure RTS if and only if $n_i$’s position interval is contained by $n_j$’s. Therefore, the stack-based structural join algorithm can still be applied on the structure RTS. The only difference is that the parent stack is associated with the descendant label instead of the ancestor label.

Since the structural join operation on the reverse tree structure can be equally efficiently performed as on the tree structure, the construction algorithm discussed in the last subsection can choose to extract either a tree structure or a reverse tree structure. To extract a reverse tree structure, we first reverse the direction of all edges on $G$ and perform a DFS search. Reversing the direction of the edges on the DFS tree structure results in the target reverse tree structure. To build the reachability graph for $G$, we first build the reachability graph for the reversed $G$ based on the DFS tree structure and reverse its edges’ direction to their original state. The purpose of introducing the reverse tree structure is very clear: to incorporate containments of the $N$-versus-1 pattern on one single layer. Its effectiveness can be illustrated by the example shown in Fig. 3. From Layer 1 to Layer 2, if a reverse tree structure is extracted, all edges into both $n_{k+1}$ and $n_{2k+2}$ nodes are included on Layer 2 and Layer 3 is left with a simple tree structure. In contrast, if only the tree structures are extracted from the reachability graph, totally $k$ layers are needed.

4. Structural join algorithms

In this section, we present the structural join algorithm on graph-structured data. The case of querying the direct containment (parent–child) pattern will also be addressed. Note that the index structures are multiple nested tree layers and one reachability graph on the last layer. The structural join algorithm works recursively as follows. Given two nodes, it first checks whether they satisfy the containment relationship on the current layer. If not, it proceeds to the next layer to determine whether they do so through ports.

4.1. Index structures

Nodes on each tree layer are indexed with the inverted technique. We omit the document $id$ from our presentation, assuming that nodes are from the same document. The generalization of our framework to handle XML data consisting of multiple documents is trivial.

On the first layer, a list of entries is stored for each distinct node label. It has two additional lists, one of outgoing port nodes and the other of incoming port nodes. All entries in the lists have the same structure ($NodeID$, $LeftPos$, $RightPos$, $LevelNo$), where $NodeID$ is the unique $id$ assigned to each node in the XML data model and the values of $LeftPos$ and $RightPos$ indicate the position...
of NodeID on the tree of this layer. The lists are sorted by their entries’ LeftPos. The two lists of outgoing and incoming port nodes are required for detecting the containment relationship between given data nodes and ports on this layer.

Note that beginning with Layer 2, the nodes on the current layer are the previous layer’s port nodes. They are divided between two lists, one of incoming ports and the other of outgoing ports. (LeftPos, RightPos, LevelNo) of each entry records its position on the current tree layer. Note that the positions of the same node on two different layers are unrelated. These two lists serve to retrieve the positions of the previous layer’s ports on the current layer. Their entries are sorted by NodeID. There are also two additional lists for the current layer’s own outgoing and incoming ports. Their entries are sorted by the LeftPost value. Therefore, there are totally four lists, two corresponding to ports of the previous layer and two corresponding to ports of the current layer. If the current layer is a reverse tree structure, the LeftPos, RightPos and LevelNo values of entries are positions of nodes on its corresponding tree structure.

On the last layer of the reachability graph, the assumption is that the number of ports has been reduced to a manageable level, for instance there is enough memory to store the reachability matrix. The reachability list is used to record the reachability between outgoing and incoming ports. It consists of entries of the structure (NodeIDo, NodeIDi, Indicator), where the Indicator parameter is for differentiating between the indirect or direct containment relationship. In case of the short reachability list, entries are hashed according to the key (NodeIDo, NodeIDi). In case that the number of outgoing and incoming ports is large such that they cannot fit in memory, probing their reachability involves the I/O costly fetch-in and fetch-out of data on pages of one of the lists. If, for instance, all incoming ports can be put in memory with at least two extra buffer pages left over, the reachability between ports can be determined by scanning the list of the outgoing ports and the reachability list only once without repeatedly reading entries of the incoming port list into memory, provided that both incoming port and outgoing port list are sorted by NodeID and the reachability list is sorted by the key (NodeIDo, NodeIDi). Therefore, reducing the number of ports does help to improve the performance of evaluating reachability analytically in term of I/O cost.

The overall structure of indexes is presented in Fig. 4: (1) lb(j)s represent the distinct labels of graph data nodes; (2) L1k and L2k denote the kth layer’s two lists of outgoing and incoming ports from the previous layer; (3) Lok and Lik denote the two lists of the kth layer’s own outgoing and incoming ports; (4) Lr is the last layer’s reachability list.

![Figure 4](image-url)
4.2. Algorithms

Compared with the stack-based structural join algorithm on the tree model, the algorithm on the graph model is more involved on three aspects: (1) on each tree layer, besides the containment relationship between two input lists, it should also determine the containment relationship between input lists and port lists; (2) proceeding from the current layer to the next one, it should retrieves the ports’ position values on the next tree layer; (3) except on the first layer, the entries of input lists on each tree layer are NOT guaranteed to be sorted by the LeftPos value.

Given a containment pattern, for instance \( A \| B \), the evaluation algorithm returns all pairs of \((\text{NodeID}_A, \text{NodeID}_B)\) with \( A \)-labeled node \( \text{NodeID}_A \) containing \( B \)-labeled node \( \text{NodeID}_B \) on the graph data \( G \). Firstly, two input lists corresponding to label \( A \) and \( B \) respectively on the first layer are exploited to find all pairs matching \( A \| B \) through this layer’s tree structure. Secondly, we should also evaluate the containment relationship between input lists and port lists. If the three containment patterns, \( A \) containing \( B \), \( A \) containing outgoing ports and incoming ports containing \( B \) respectively, are evaluated independently using the structural join technique, the lists of \( A \) and \( B \) will both be scanned linearly twice. Actually, we can reduce the number of scanning on both lists \( A \) and \( B \) to only once by evaluating these three patterns at the same time. Our solution is that for each of three patterns, we keep two stacks in memory, one associated with its ancestor list and the other with its descendant list. The entry with the smallest \( \text{LeftPos} \) value in lists \( L_A \), \( L_{o1} \), \( L_{i1} \) and \( L_B \) is sequentially retrieved; if equality happens, the priorities of lists are \( L_A > L_{o1} > L_{i1} > L_B \). The processing of each pattern’s structural join is exactly the same as the case it is performed independently. Just remember that while processing an entry from list \( L_A \), it should be considered to be pushed into two stacks; similarly while processing an entry from list \( L_B \), it should be considered to be joined with entries in two parent stacks. Further optimization of using one stack for each of lists \( L_A \) and \( L_B \) can also be easily implemented.

The operation to retrieve the ports’ positions on the next layer can be accomplished by either one of two standard join techniques in relational databases, sort-merge join and hash join. Although the sort-merge join algorithm has been shown to be generally inferior to the hash join [1], the setting here is slightly different since one of relations can be sorted or hashed beforehand. In the last subsection, we say that the lists of the previous layer’s outgoing and incoming ports are sorted by their \( \text{NodeIDs} \). This structure is to facilitate processing the sort-merge join algorithm. If the hash join algorithm is adopted, entries will be hashed into several partitions of roughly same size such that each partition can fit in memory. A different in-memory hash-table will be also built for each partition to reduce the CPU cost of the probing phase.

Algorithm 3. The Structural Join Algorithm on Graph Data with \( A \| B \)

Set the current layer to be the first layer;
Initially, \( L_A \) and \( L_B \) is set to be two input lists of label \( A \) and \( B \) on the first layer;
repeat
• perform structural joins of \( L_A \| L_B \), \( L_A \| L_{o1} \) and \( L_{i1} \| L_B \) on this layer;
• output results of matching \( L_A \| L_B \);
• with the result of matching \( L_A \| L_{o1} \) denoted as \( L_{Ak} \), join \( L_{Ak} \) with \( L_{1(k+1)} \) through \( \text{NodeID} \) to retrieve outgoing ports’ positions on the \((k+1)\)th layer;
• with the result of matching \( L_{ik} \| L_B \) denoted as \( L_{Bk} \), join \( L_{Bk} \) with \( L_{2(k+1)} \) through NodeID to retrieve incoming ports’ positions on the \((k + 1)\)th layer;
• sort lists \( L_{Ak} \) and \( L_{Bk} \) according to the LeftPos value;
• set \( L_A = L_{Ak} \), \( L_B = L_{Bk} \), and proceed to next layer;
until the current \((k)\)th layer is the final one
for each entry in \( L_A \) do
  for each entry in \( L_B \) do
    check their containment relationship through the reachability hashtable and output results correspondingly;
  end for
end for

Since the structural join algorithm assumes two sorted input lists, on each tree layer from the second layer on, lists of outgoing or incoming ports from the previous layer should be externally sorted by the LeftPos value before they are evaluated on the current layer. On the last layer, supposing that the reachability between ports are hashed in memory, the matching algorithm between two lists of ports is implemented in a nested loop way. The structural join algorithm on the graph model is described in Algorithm 3. Note that we use \( L_A = L_{Ak} \) and \( L_B = L_{Bk} \) only for the convenience of presentation. The structure of \( L_{Ak} \) and \( L_{Bk} \) is actually different from the initial \( L_A \) and \( L_B \). Each entry in the list \( L_{Ak} \) stores the corresponding outgoing port’s NodeID and all \( A \)-labeled nodes that can reach this port. Similarly, each entry in the list \( L_{Bk} \) stores the corresponding incoming port’s NodeID and all \( B \)-labeled nodes that can be reached through this port. Therefore, while outputting results, if an entry \( e_A \) from \( L_A \) contains an entry \( e_B \) from \( L_B \), it is obvious that any \( A \)-labeled node coming along with \( e_A \) contains any \( B \)-labeled node in \( e_B \).

We have the following theorem. Its proof is straightforward as the result of Theorem 1.

**Theorem 2.** The algorithm presented in Algorithm 3 correctly returns all matching results of a binary containment pattern on the graph data \( G \).

4.3. Handling direct containment

Exploiting the level information, our indexing approach can smoothly accommodate the structural join operation of the direct containment relationship. Note that determining the direct containment relationship only needs to check parent–child edges on the original graph data \( G \). We differentiate between two types of edges on our layered structures: edges of \( G \), original, and all other edges inserted because of inheritance, inherited. While building inverted indexes for tree layers, level is added up by one if encountering an original edge, two if encountering an inherited edge. On the last layer, \( (NodeID_o, NodeID_i, Indicator) \), Indicator is set to be 1 if node \( NodeID_o \) directly contains \( NodeID_i \) by an original edge, 0 otherwise. The evaluation procedure of a direct containment pattern, \( A/B \), thus involves taking two input lists \( L_A \) and \( L_B \) and on each layer structure, determining if entries in \( L_A \) directly contain entries in \( L_B \) in the usual way by checking their levels.
Up to now, we ignore the preprocessing phase in which all strongly connected components on the graph data $G$ are shrunk into single units. Note that in this case, all incoming and outgoing edges of strongly connected components, and edges inside them should be stored in a separate index structure for facilitating the direct containment join operation.

5. Experimental study

In this section, we present our experimental results on both benchmark and synthetic XML data. The two datasets we use are:

1. XMark. This is a popular XML data from the XML Benchmark project [24], which simulates information about activities of an auction site.
2. NASA Data. This data set is generated by the IBM data generator using a real DTD file, nasa.dtd [25], which is a markup language for the data and meta-data at the astronomical data center at NASA/GSFC. It possesses a broader, deeper and less regular structure than the XMark data. It also has more reference links.

We compare our technique, termed the layer index, with the existing bisimilarity-based approach. To fully support the containment join on a single structural summary, the $F + B$ index, or the forward and backward stable index, is needed since the binary patterns, $A[||B]$ corresponding to a location step or $A[||B]$ corresponding to a predicate, may require the join operation to return either the ancestor or descendant node. From the statistics in Table 1, the $F + B$ index of either of our test data remains large, with a size close to the size of original graph data. Therefore, to fully explore the potential of the bisimilarity-based approach, we also implement the option of building two separate $F$- and $B$-indexes to support the containment join. We evaluate the binary patterns on the graph index through the standard breadth-first search (BFS). It is supported by the $B+$-tree structure.

Since the bisimilarity-based approach can only return either the ancestor or descendant node, to allow an “apple to apple” comparison, the evaluation algorithm of our new approach is also accordingly adjusted for this task. Notably, if ancestor nodes should be returned, the algorithm does not record descendant nodes that can be reached through incoming ports. Similarly, if

<table>
<thead>
<tr>
<th>Statistics of datasets and $FB$ indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doc size (M)</td>
</tr>
<tr>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>XMark data</td>
</tr>
<tr>
<td>XMark ($F + B$)</td>
</tr>
<tr>
<td>XMark $F$</td>
</tr>
<tr>
<td>XMark $B$</td>
</tr>
<tr>
<td>NASA data</td>
</tr>
<tr>
<td>NASA ($F + B$)</td>
</tr>
<tr>
<td>NASA $F$</td>
</tr>
<tr>
<td>NASA $B$</td>
</tr>
</tbody>
</table>
descendant nodes should be returned, it does not record ancestor nodes that can reach outgoing ports.

We build the layer index by alternately extracting the tree structure (TS) and the reverse tree structure (RTS) from the reachability graph. The effectiveness of this strategy in reducing the sizes of reachability graphs is shown in Fig. 5(a). If the extracted structure is limited to be tree, the sizes of reachability graphs can only be reduced by $1-5\%$ on each layer after layer 2. In contrast, the alternate combination of TS and RTS manages to reduce the size of reachability graph to 0 within 6 layers. It is run on a small 1 MB XMark data, but exemplifies the inherent structure of both XMark and NASA data sets.

The statistics of the $F+B$ index, the $F$- and $B$-index, and the layered index structures are summarized in Table 1 and Fig. 5(b). On both the XMark and NASA data, the $F+B$ index graph has roughly the same number of nodes and edges as the original graph data (the difference is within 10%); but the $F$- or $B$-index graph is significantly smaller. As for the layer index structures, the reachability graphs of two datasets are both reduced very fast to be of size 0, within 8 layers. It is interesting to point out that from layer 2 on, the back-to-back extractions of a TS and RTS repeatedly reduce the number of nodes in the reachability graph by at least about 90%.

We run experiments to compare the performance of the layer index with the ($F+B$) index and the $F_B$ index in which $F$- and $B$-index are maintained separately. The performance is measured by the I/O cost and the running time which includes both the I/O and the CPU cost. Our machine features a Pentium 2.0 GHz processor, a 512 MB RAM and runs Linux. We set the page size to 4 KB and vary the cache size from 8 MB up to 32 MB. On implementing the evaluation algorithm on the layered indexes, we partition the cache such that 2 MB is reserved for the buffer pool of external sorting. On both datasets, we choose 10 binary indirect containment query patterns with different selectivities. The query load actually consists of 20 patterns with each pattern having two versions, the one returning the ancestor nodes and the other one returning the descendant nodes. These patterns’ order in the query load is set randomly to more accurately simulate queries on real XML data. We find that if we repeatedly run the query load on indexes, the overall performance only fluctuates slightly. Therefore, the results we present here are independent of the cold or warm cache setting. The I/O cost is measured by the number of page accesses. Both the I/O cost and the running time are of the whole query load. Note that the vertical axes of all figures follow a logarithmic scale, since there are marked differences in performance.

![Fig. 5. Effectiveness of alternating direction while building tree layers. (a) Small data and (b) test data.](image-url)
5.1. Comparison results on XMark data

The I/O performance of three indexing approaches are presented in Fig. 6(a). On all cache sizes, the layer index takes significantly less I/O than either of two other indexes. The $F + B$ index performs considerably worse than the $F_B$ index. This observation can be analytically verified since the $F + B$ index graph is substantially larger than either the $F$- or $B$-index, thus its BFS search is more I/O expensive. From Fig. 6(a), we also have the observation that as the cache size is reduced, the I/O cost increase on the layer index is more modest than on the $F_B$ index. When the cache size is between 16 MB and 32 MB, the layer index takes about 10–20% I/O cost of the $F_B$ index. When the cache size is reduced to 12 MB, the percentage is reduced to about 5%. When the cache size is further reduced to 8 MB, the percentage is only about 3%. These results demonstrate that compared with the $F_B$ index, the I/O performance of the layer index is less sensitive to the constraints placed by the cache size. This characteristics is analytically expectable because most of the evaluation operations on the layer index involves linearly scanning lists.

Now let us take a look at a particular case when the cache size is 12 MB. We differentiate between two types of patterns, those returning ancestors and those returning descendants. In

Fig. 6. Results on XMark data set. (a) I/O cost, (b) running time, and (c) unbalance of $F$- and $B$-index.
Fig. 6(c), these two types of patterns have roughly the same I/O cost on the layer index. But on the $F_B$ index, the patterns returning descendants consume significantly more I/O cost than the patterns returning ancestors. Since the $F$-index has a much smaller size compared with the $B$-index, its BFS search is more I/O efficient. This performance discrepancy is observed across all cache settings.

The running-time comparison is presented in Fig. 6(b). As expected, the $F_B$ index performs much better than the $F + B$ index. Between the layer index and the $F_B$ index, the layer index performs much better across all cache settings. As the cache size varies, the running time of the layer index only fluctuates slightly, up no larger than 2%. As for the $F_B$ index, the running time goes up only slightly as the cache size is decreased from 32 MB to 16 MB since the cost is still dominated by the CPU consumption. But as the cache size becomes smaller, the performance deterioration becomes more visible, with the running time being up about 20% from 16 MB to 12 MB and 35% from 12 MB to 8 MB. This observation results from the dramatically increased I/O cost.

5.2. Comparison result on NASA data

The I/O cost performance comparison on the NASA data is presented in Fig. 7(a). It follows the trends observed on the XMark data. The slight difference is that when the cache size is no less than 20 MB, the $F_B$ index has roughly the same I/O cost as the layer index. Compared with the results on XMark data, the $F$- and $B$-index of the NASA data has a smaller size. Thus its BFS evaluation incurs less I/Os. But with the cache size below 16 MB, the I/O cost of the $F_B$ index shots up dramatically and the layer index clearly claims the better performance.

On the running time, the results of Fig. 7(b) show that the layer index also significantly outperforms both the $F + B$ index and $F_B$ index in all cache settings. As on the Xmark data, the running time of the layer index remains roughly the same as the cache size varies. In contrast, the performance of the $F_B$ index worsens considerably as the cache size is decreased to below 16 MB.

Fig. 7. Results on NASA data set. (a) I/O cost and (b) running time.
5.3. Discussions on experimental results

Summarizing the experimental results, we have the following observations:

(1) The combination of the tree structure (TS) and the reverse tree structure (RTS) is experimentally shown to be extremely effective in encoding XML graph data, where containment relationships between nodes follow the 1-versus-N and N-versus-1 patterns.

(2) In contrast to the bisimilarity-based approach, whose I/O performance deteriorates rapidly as the cache size decreases, the layer index’s I/O performance is much less sensitive to the cache size. Therefore, if the cache size is small in comparison to the sizes of the F- and B-indexes, the layer index performs significantly better.

(3) Concerning the combined I/O and CPU cost, the layer index also has a clear-cut advantage over the bisimilarity-based approach. Its superiority becomes more evident as the cache size becomes smaller.

6. More related work

The inverted index technique was pioneered by Consens and Milo [9,10], who used the representation, \((DocID, LeftPos:RightPos)\), to compute the containment relations between “text regions” in the text database. Works in [11–13] adopted this representation to index XML elements of a tree-structured model. Our work is closely related to theirs in that our solution for the graph model is based on the proposed structural join on the tree structure. It is also worth mentioning that complementary to these works, [23] uses B+-tree to index elements in the same linked list and thus achieves the experimentally sub-linear performance of the binary join by skipping unmatched elements; [22] addresses the twig pattern matching problem on partially indexed XML documents with the underlying model of a tree.

The approach of building structural summaries for graph-structured data is based on the concept of bi-similarity. The works of [16–19] contribute to the research in this direction. The structural summary has been shown to be effective in pruning the search space, thus speeding up the query processing. Even though the underlying graph model is assumed, these techniques only consider the path expression with parent–child edges and do not take I/O efficiency into their cost model. The F&B index was put forward by [21,20], where authors addressed the matching of branching path expressions. Their experiments show that, compared with the original document, the resulting F&B index tends to still have a substantially large size. This fact is also verified by our experiments.

7. Conclusion

In this paper, we propose an indexing framework, the layer index, and evaluation algorithms for performing the structural join operation on graph-structured XML data. Our approach constructs multiple nested layers of tree-structured indexes by recursively decomposing a graph into constituent trees. The evaluation algorithm consists of performing nested structural join opera-
tions on tree-structured layers. The experimental study on both benchmark and synthetic XML data validates the effectiveness of our approach. Compared with existing bisimilarity-based techniques, the layer index performs significantly better in terms of both the I/O and CPU cost.

As for the future research, even though the trick of alternating the edge direction while building layer trees in consecutive steps is empirically shown to be quite effective, it is interesting and important to investigate analytically how well our technique fares on the general graph of any structure. On the other hand, we have not taken advantage of sophisticated index structures, such as B+-tree, in our proposal. Therefore, it is worthy to explore the potential of such index structures in speeding up the containment join processing on the graph data.

References


Qun Chen obtained his bachelor degree in Management Information Systems from Tsinghua University of Beijing, China, in 1998. After a brief period working as a software engineer, he began his postgraduate study in Computer Science at National University of Singapore in 1999 and received his Ph.D. degree in 2004. He is currently a research associate of Department of Industrial Engineering and Engineering Management at the Hong Kong University of Science and Technology. His research interests are in querying and indexing semi-structured databases.

Andrew Lim obtained his Ph.D. degree in 1992 from the University of Minnesota. From 1992 to 1997, he was a system developer, project leader and consultant to many large private and government organizations in Singapore. From 1997 to 2002, he was an associate professor of Computer Science at the National University of Singapore. At the present moment, he is an associate professor at the Department of Industrial Engineering and Engineering Management in Hong Kong University of Science and Technology. Andrew’s interests include algorithms, software components, framework and architecture for Global Supply Chain management. He has published more than 185 papers in premier international conferences and journals. Professor Lim is also the Director of the HKUST Logistics and Supply Chain forum, and the founding director of the Logistics and Supply Chain Institute (China). He has consulted to a large number of companies in Hong Kong, Singapore and United States.

Kian Win Ong received his bachelors in Computer Engineering from the National University of Singapore in 2003. After working on various XML indexing structures in the Hong Kong University of Science and Technology, he is currently a Computer Science postgraduate student at University of California at San Diego. His research interests are in databases, including optimizations for semi-structured data, novel application frameworks and schema evolution.
Ji Qing Tang received his Bachelor Degree in Electronic Engineering in 2003 from Zhejiang University. From September 2003, he studied in Department of Industrial Engineering and Engineering Management at the Hong Kong University of Science and Technology and received his master degree in Industrial Engineering in January 2005. He is currently working at the high-tech company NVIDIA in Hong Kong. His research interests include operations research, integer programming, artificial intelligence and databases.