Labeling Scheme and Structural Joins for Graph-Structured XML Data*

Hongzhi Wang[1,2], Wei Wang[1,3], Xuemin Lin[1,3], and Jianzhong Li[2]

[1] University of New South Wales, Australia
{hongzhiw, weiw, lxue}@cse.unsw.edu.au
[2] Harbin Institute of Technology, Harbin, China
lijz@mail.banner.com.cn
[3] National ICT of Australia, Australia

Abstract. When XML documents are modeled as graphs, many challenging research issues arise. In particular, query processing for graph-structured XML data brings new challenges because traditional structural join methods cannot be directly applied. In this paper, we propose a labeling scheme for graph-structured XML data. With this labeling scheme, the reachability relationship of two nodes can be judged efficiently without accessing other nodes. Based on this labeling scheme, we design efficient structural join algorithms to evaluate reachability queries. Experiments show that our algorithms have high efficiency and good scalability.

1 Introduction

XML has become the *de facto* standard for information representation and exchange over the Internet. XML data has hierarchical nesting structures. Although XML data is often modeled as a tree, IDREFs within the XML document represent additional "referencing" relationships and are essential in some applications, e.g., to avoid redundancy and anomalies. Such XML data could be naturally modeled as a graph.

Query processing of graph-structured XML data brings new challenges:

- One traditional method of processing queries on tree-structured XML data is to encode the nodes of XML data tree with certain labeling scheme and process the query based on structural joins [2]. Under the coding scheme, the structure relationship between any two elements (such as parent-child or ancestor-descendant relationships) can be judged efficiently without accessing other elements. This property is the foundation of all structural join algorithms so far. However, none of the existing XML coding schemes can be applied to graph-structured XML data directly.
- Another query processing methods is based on structural index such as 1-index [14] and F&B index [12]. However, structural indexes of graph-structured XML documents are likely to have a large number of nodes. As

* This work was partially supported by ARC Discovery Grant – DP0346004.

Y. Zhang et al. (Eds.): APWeb 2005, LNCS 3399, pp. 277–289, 2005.

a result, they efficiency could be a problem when there is not enough memory. For example, the number of nodes in F&B index of the standard 100M XMark document, when modeled as a tree, has 0.44M nodes; the number of nodes in F&B index of the same document, when modeled as a graph, has 1.29M nodes [12].

Given the successes of query processing methods based on XML coding schemes and structural joins, in this paper, we adopt a similar approach dealing with query processing tasks for graph-structured XML data. We propose a reachability coding scheme for general digraph, which can be used to assist efficient reachability queries. In this coding scheme, all the strongly connected components of the digraph are contracted to single representative nodes such that the digraph is reduced to a DAG. Then a DAG labeling scheme is applied to the generated the code [15]. Based on the the features of the coding scheme, we design the efficient structural join algorithms, Graph-Merge-Join (GMJ) and its improved version Improved-Graph-Merge-Join (IGMJ). They can be viewed as the natural generalizations of the *tree-merge* and *stack-tree* algorithms [2] for the graph-structured XML data.

The contributions of the paper can be summarized as follows:

- We generalize the coding scheme in [15] and present an effective coding scheme to judge the reachability relationships between nodes in a general digraph.
- We present two efficient structural join algorithms, GMJ and IGMJ, based on the graph coding scheme.
- Our experiments show that our coding scheme is efficient for XMark data. Our two join algorithms outperform 1-index based query processing methods significantly and have good scalabilities.

The reset of the paper is organized as follows: Section 2 introduces some background knowledge and notations used in the paper. Section 3 presents the reachability coding scheme. Structural join algorithms based on the coding scheme are given in Section 4. We present our experiment evaluation results and analysis in Section 5. Related work is described in Section 6. We conclude the paper in Section 7.

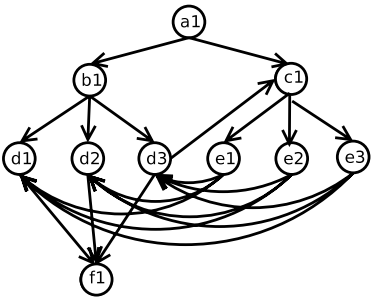
2 Preliminaries

In this section, we briefly introduce graph-structured XML data model as well as terms and notations used in this paper.

XML data is often modeled as a labeled tree: elements and attributes are mapped into nodes of graph; directed nesting relationships are mapped into edges in the tree. A feature of XML is that from two elements in XML document, there may be a IDREF representing reference relationships [18]. With this feature, XML data can be modeled as a labeled directed graph (digraph): elements and attributes are mapped into nodes of graph; directed nesting *and* referencing

```
<a id="a1">
<b id="b1">
</b>
<d id ="d1" f="f1"/>
<d id="d2">
<f id="f1">
</d>
<d id="d3" f="f1" c="c1"/>
<c id="c1">
<e id="e1" g="g1" d="d1" d="d2" d="d3">
<e id="e2" g="g1" d="d1" d="d2" d="d3">
<e id="e3" d="d1" d="d2" d="d3">
<g id="g1" f="f1"/>
</e>
</c>
</a>
```

(a) An Example XML Document



(b) The Corresponding XML Graph

Fig. 1. An Example XML Document and Its XML Graph

relationships are mapped into directed edges in the graph. An example XML document is shown in Fig 1(a). It can be modeled as the digraph shown in Figure 1(b). Note that the graph in Figure 1(b) is not a DAG.

XML query languages, such as XQuery [4], are based on the tree model and allow retrieving part of the XML document by the structural constraints. For example, the XPath query $b//e$ will retrieve all e elements nested within b elements (// represents the ancestor-descendant relationship). In the example XML document in Figure 1(a), the query result is empty. However, when we model XML document as a graph, the ancestor-descendant relationship (as well as parent-child relationship) can be extended based on the notion of reachability. In [12], IDREF edges are represented as $\Rightarrow$ and $\Leftarrow$[1] for the forward and backward edge[2], respectively. Two nodes, u and v belong to a graph G satisfy reachability relationship if and only there is a path from u to v in G (denoted as $u \rightsquigarrow v$). Each edge in this path can be either an edge representing nesting relationship or referencing relationship. For example, the query $b \rightsquigarrow e$ will return $e1, e2$, and $e3$ for the example XML data graph in Figure 1(b). Such queries are referred to as **reachability queries** in the rest of the paper.

3 The Coding of Graph-Model XML Document

In this section, we describe the coding scheme of graph-structured XML document. We extend the coding scheme for directed acyclic graph (DAG) in [15] to support directed cyclic digraph. Therefore, with this coding, the reachability

[1] **FIXME**
[2] **FIXME**

relationship between two nodes in a general digraph can be judged efficiently without accessing any other node.

3.1 The Coding of DAGs

In this subsection, we encode DAG using the method introduced in [15]. In this coding scheme, each node is assigned a list of intervals. We briefly summarize the encoding method for a DAG G in the following: *First, find an optimal tree-cover T of the DAG D. T is traversed in a depth-first manner. During the traversal, an interval $[x, y]$ is assigned each node n of T, where x is the postorder of n in the traversal. y is the smallest postorder number of all n's descendants in T. Next, examine all the nodes of D in the reverse topological order. At each node n, copy and merge, if possible, all the intervals of its out-going nodes in G to its code.*

The judgement of the reachibility relationship between two nodes a and b is to check whether the postorder of b is contained in one of the intervals of a.

An example of encoding a DAG G is shown as following. The coding of the DAG in Fig 2(a) is the in Fig 2(b). We use *postid* to denote the postorder number of each nodes, which is also the second value of the first interval of its code.

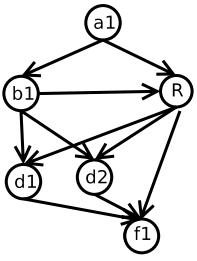
3.2 The Coding of General Graph

We now generalize the above scheme for the case of a directed cyclic graph. We assume that the graph consists of only one connected component with a single root. The root is a node without any incoming edge. Otherwise, we can pick up any root node or add a virtual root node.

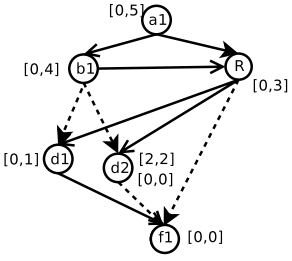
The process of coding a general digraph G is sketched as the following steps:

1. Find all Maximal Strongly Connected Components (MSCC) with number of nodes greater than one. A maximal strongly connected component C is a MSCC if and only if there is no strongly connected component (SCC) that contains C in G.
2. Each MSCC of G is contracted to a *representative node*. As a result G is reduced to a DAG G' (the correctness of this step is proved in Theorem 1). Suppose MSCC $S = \{node_0, node_1, \cdots, node_t\}$ in G is contracted in to $node_S$. $node_S$ is the representative node in G'. If a node in G is not contracted, then its corresponding node is itself in G'.
3. G' is encoded using the method introduced in Section 3.1.
4. For each $node_S$ in G', assuming its code is C_s, C_s is assigned to every $node_0, node_1, \cdots, node_t$ in G. It means that all nodes in the same MSCC have the same codes, i.e., the list of intervals and the *postid* number.

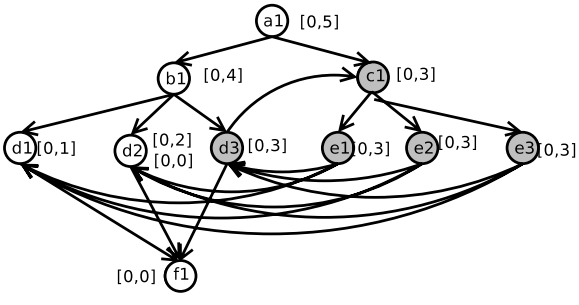
For example, a directed cyclic graph G is shown in Figure 1(b). In order to encode G, the first step is to contract all the maximal strongly connected components. In G, there is only one such MSCC, $S = \{d3, c1, e1, e2, e3\}$. By contracting this MSCC, a DAG G' shown in Figure 1(b) is generated. In G', R is the representative node for MSCC S. The interval codes of G' is shown in



(a) The DAG contracted from Graph XML



(b) Interval Coding for the DAG



(c) Interval Coding for the XML Graph

Fig. 2. The Coding Scheme

Figure 2(b). At last, the intervals associated with R, i.e., $[0, 3]$, is assigned to each node in S in G. As a result, $d3$, $c1$, $e1$, $e2$, $e3$ all have the same interval code $[0, 3]$. The interval codes of graph G is shown in Figure 2(c). The *postid* of each node is the second value of its first interval. For example, $d3.postid$ is 3.

The following theorems ensure the correctness of the encoding method. In the interest of space, we don't show the proofs of the theorems.

Theorem 1. *A directed cyclic graph G is converted to a DAG in the way stated above.*

Theorem 2. *For two nodes a and b of XML graph encoded in steps stated, suppose the $code_a = \{[a_0.x, a_0.y]$, ..., $[a_n.x, a_n.y]$, $postid_a\}$ and $code_b = \{[b_0.x, b_0.y]$, ..., $[b_m.x, b_m.y]$, $postid_b\}$. $postid_a$ and $postid_b$ are the postorders of a and b in the tree cover T of G' generated from G by contracting MSCCs, respectively. Then $a \rightsquigarrow b$ if and only if $\exists i (0 \leq i \leq n)$ such that $a_i.x \leq postid_b \leq a_i.y$.*

Time Complexity Analysis. The finding of all MSCCs can leverage the DFS-based algorithm in [1], and its time complexity is $O(n)$, where n is the number of the nodes of G. The efficiency of contracting step is $O(n_c)$, where n_c is the total number of nodes belonging to the MSCCs. The complex of encoding a DAG is $O(n')$ [15], where n' is the number of nodes in the DAG. The last step needs $O(n_c)$ time. Since both n_c and n' are smaller than n, the time complexity of encoding method is $O(n)$.

4 Join Algorithms Based on the Labeling Scheme

In this section, we design two join-based algorithms to process reachability query on graph-structured XML data using the labeling scheme presented in Section 3. The structural join algorithms compute the result of reachability query $a \rightsquigarrow d$, where $a \in Alist$ and $d \in Dlist$ are element sets.

4.1 Preprocess of the Input

One difference of the interval labeling scheme of a graph and that of a tree is that there may be more than one interval assigned to a node. The reachability relationship of two nodes a and b can be judged based on Theorem 2. We choose to preprocess the joining nodes by inverting the nodes and their corresponding interval codes. That is, if a node has k intervals, it is treated as k nodes: for Alist, each element has one interval; for Dlist, each element has a *postid* and the *id* of this element. Then both inputs are inverted: for the Alist, the list is sorted on the intervals $[x, y]$ by the ascending order of x and then the descending order of y; for the Dlist, the list is sorted by the ascending order of *postid*. The intuition is to leverage the order in the intervals and *postid*s to accelerate join processing.

We note that the same interval will occur more than once in the preprocessed Alist, for one of the following two reasons:

- All the nodes with the same tag in an MSCC have the same codes, hence intervals.
- Even if two nodes does not belong to the same MSCC, there could be some interval associated with both of them. This is because in the third step of the DAG encoding, when considering a node n with multiple children, some intervals of its child will be appended to n. If some added interval of a child c cannot be merged in the existing code of n, c and n with the same tage will have the same interval even if they do not belongs to any MSCC together.

Similar case exists in Dlist as well because all nodes in the same MSCC have the same *postid*.

Inmplementation-wise, in order to decrease the interval set of processing, repeated intervals with different node IDs are merged into one interval with multiple node IDs. Repeated *postid*s in Dlist are also merged in a similar way.

For example, before the preprocessing for answering query $d \rightsquigarrow e$ against the XML document shown in Figure 2(c), Alist is $\{d1([0,1]), d2([0,0],[0,2]), d3([0.4])\}$, and Dlist is $\{e1(4), e2(4), e3(4)\}$. The intervals associated with a node is in the brackets following the node. After preproocessing, the Alist becomes $\{[0,4](d3),[0,2](d2),[0,1](d1),[0,0](d2)\}$, the Dlist becomes $\{4(e1,e2,e3)\}$. Preprocessed Alist and Dlist are sorted by the codes (intervals and *postid*s, respectively). The nodes corresponds to an interval i (or *postid*) is in the bracket followed the interval (or the *postid*). In Alist, the intervals associated to $d2$ are separated. In Dlist, since $e1, e2, e3$ have the same postid, they are merged into the same *postid*.

4.2 Two Join Algorithms

After preprocessing, a naïve structural algorithm can be obtained by generalizing the sort-merge based structural join algorithm in [2]. One subtlety is that the intervals in the preprocessing Alist might have the same starting or ending values (i.e., x or y). The codes shown in Figure 2(c) is such an example. We present the merge based join algorithm on graph, named *Graph-Merge Join* (GMJ), in Algorithm 1.

Algorithm 1 GMJ(*Alist*,*Dlist*)

```
1: a = Alist.head()
2: d = Dlist.head()
3: while a ≠ NULL ∧ b ≠ NULL do
4:    while a.x > d.postid ∧ d ≠ NULL do
5:       d = d.next()
6:    while a.y < d.postid ∧ a ≠ NULL do
7:       a = a.next()
8:    if d ≠ NULL ∧ a ≠ NULL then
9:       bookmark = a
10:      while a ≠ NULL ∧ a.x ≤ d.postid ∧ a.y ≥ d.postid do
11:         Append (a, d) pair to the output
12:         a = a.next()
13:      a = bookmark
14:   d = d.next()
```

For example, assume the two lists (preprocessed) to be joined are:

- Alist: $a1([1,3]), a2([1,1]), a3([3,6]), a4([4,5])$,
- Dlist: $d1(1), d2(4), d3(7)$

In GMJ, the basic idea is to join intervals and *postid*s in a sort-merge fashion. Since intervals might be nested, a bookmark is needed to keep track of the current position of intervals while outputting results. In the example, the pointer of Alist, i.e., a, points to $a1$. $d1$ joins $a1$ and $a2$. When processing $d2$, the pointer of Alist moves to $a3$. $d2$ join with $a3$ and $a4$. When processing $d3$, the pointer moves to the tail of Alist, so the algorithm terminates.

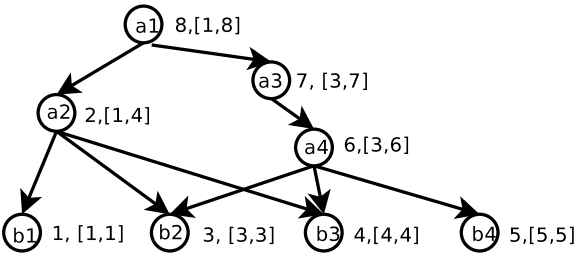


Fig. 3. An Example of Overlapping Code

GMJ suffers the same problem of tree-merge join algorithms in that part of the input might be scanned repeatedly. We note that stack-based structural join algorithm in [2] cannot be directly generalized and work with our coding scheme. This is because that two intervals may be partially overlapped. For an example, a graph and its codes is shown in Figure 3. The intervals assigned to a_2 and a_4 are partially overlapping. As a result, stacks can no longer be used to represent the nesting relationship between intervals in our coding scheme.

Algorithm 2 IGMJ(*Alist*,*Dlist*)

```
1: a = Alist.head()
2: d = Dlist.head()
3: rstree.insert(a)
4: a = a.next()
5: while a ≠ NULL ∧ d ≠ NULL do
6:    if a.x ≤ d.postid then
7:       rstree.trim(a.x)
8:       rstree.insert(a)
9:       a = a.next()
10:   else
11:      rstree.trim(d.postid)
12:      for all element a in rstree do
13:         Append (a, d) pair to the output
14:      d = d.next()
```

We design a new algorithm, named *Improved Graph Merge Join* (IGMJ) instead. The basic idea of GRJ is to store the intervals that can be joined in a range search tree (RST for brief). In the tree, the intervals indexed and organized according to their y values. When a new interval a of Alist arrives, it is compared with the current node d of Dlist. If a contains the postorder of d, a is inserted to the tree and all elements in the tree with y value smaller than $a.x$ are deleted (via the *trim*() method). Otherwise, we process current node d in Dlist. All elements in the tree with y value smaller than *d.postid* are deleted. Then output d with all the a nodes in the tree. The algorithm of IGMJ is shown in Algorithm 2. In this algorithm, *brtree* is a RST that supports the following methods: *insert*(I)

and $trim(v)$. $insert(I)$ will insert an interval I to the BRST; $trim(v)$ will batch delete the intervals in *brtree* whose y values smaller than v.

For example, let's consider running IGMJ on the same example above. x values of $a1$ and $a2$ are smaller than $d1.postid$. At first, $a1$ and $a2$ are inserted into the RST. Then, $(d1, a1)$ and $(d1, a_2)$ are appended to the result list. $a3$ and $a4$ are processed before $d2$[3]. They are inserted to RST. When $a3$ is processed, $a2$ is trimmed from the RST because $a2.y < a3.x$. $a1$ is trimmed from RST when processing $d2$, since $a1.y < d2.postid$. Then, $(a3, d2)$ and $(a4, d2)$ will be appended to the result list. $a3$ and $a4$ are trimmed from RST based on $d3$.

5 Experiments

In this section, we present results and analyses of part of our extensive experiments of the new coding scheme and the structural join algorithms.

5.1 Experimental Setup

All our experiments were performed on a PC with Pentium 1GHz CPU, 256M main memory and 30G IDE hard disk. The OS is Windows 2000 Professional. We implemented the encoding of graph, the Graph-Merge-Join (**GMJ**) and Improved-Graph-Merge-Join (**IGMJ**) using the file system as the storage engine. For comparison, we also implemented a naïve traversal-based query processing algorithm based on the 1-index [14] (**1-index**).

We use the XMark benchmark dataset [16] in our experiments. It is a frequently used dataset and features irregular schema. We measure the performance of different algorithms on the 20M XMark dataset (with scale factor 0.2). It has 351241 nodes and its 1-index has 161679 nodes. We generated other XMark datasets with sizes 10M, 20M, 30M, 40M, and 50M respectively. They are used in the scalability experiment.

We show the set of queries used in the experiments in Table 1. They represent different characteristics in terms of the sizes of Alist, Dlist, and result (based on the 20M XMark dataset).

Table 1. The Query Set

ID	Query	Alist Size	Dlist Size	Result Size
Q1	person⇝emph	3158	14222	8032
Q2	site⇝item	1	4549	4350
Q3	person⇝category	3158	200	199
Q4	people⇝privacy	205	1195	1182

[3] **FIXME**

Table 2. The Size of Code

Doc Size	Intervals	Intervals after preprocessing	IPN	IPNJ
11.3M	252284	173702	1.44	0.990
22.8M	503112	346014	1.43	0.985
34.0M	746234	516546	1.40	0.985
45.3M	999784	687596	1.43	0.986
56.2M	1255877	859823	1.44	0.988

5.2 Space Overhead of the Coding

We measure the space overhead of our coding scheme with the following two parameters:

$$IPN = \frac{\text{number of total intervals}}{\text{number of nodes in the XML document}}$$

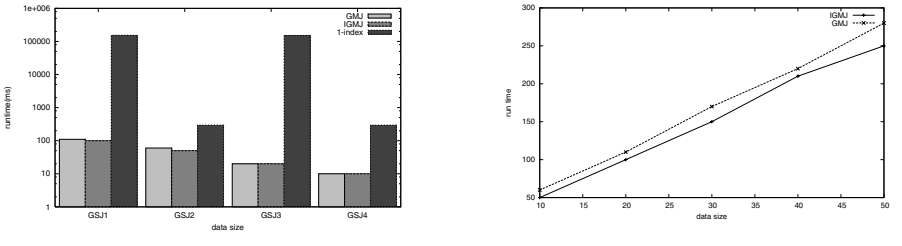
$$IPNJ = \frac{\text{number of total intervals after preprocessing}}{\text{number of nodes in the XML document}}$$

The former measurement represents the average number of intervals associated to one node. The later measurement represents the average number of intervals associated with one node that will be processed during structural join.

The results of the size of codes are shown in Table 2. IPNJ is small than 1.0. This is because some nodes in the interval sets may share the same interval. It can be observed from the result that even though the average number of intervals of a node is larger than one, the average number of intervals after preprocessing are smaller. This shows that the preprocessing of the Alist and Dlist in join is meaningful by exploiting the sharing of intervals and *postid*s, respectively.

5.3 Execution Time

We show in Figure 4(a) the execution time of GMJ, IGMJ, and 1-index for Q1 to Q4 on the 20M XMark dataset. Note that Y-axis is in logarithm scale. Both



(a) Execution Time

(b) Scalability of IGMJ and GMJ (Q1)

Fig. 4. Experiment Results

GMJ and IGMJ outperform 1-index. This is because there are many nodes in 1-index. During processing the query with form 'a//b', when a node a_1 with tag a is found, all the nodes in the subgraph formed with nodes that are reachable from a_1 need to be traversed. We also found that IGMJ is always faster than GMJ. It is because with the usage of RST, whenever an interval will not join with any d in the Dlist, it will be trimmed from the RST.

5.4 Scalability Experiment

To evaluate the scalability of the new algorithms, We ran Q1 on XMark documents with size ranging from 10M to 50M. The result is shown in Fig 4(b). It can be seen that both algorithms scale linearly with the increase of the data size.

6 Related Work

There are many reachability labeling schemes for trees. Recent work includes [3, 8, 11]. Reachability labeling schema schemes for directed acyclic graphs (DAGs) includes [15, 22]. [6] is a survey of labeling schemes for DAGs and compares several labeling schemes in the context of semantic web applications. [10] presents a reachability coding for a special kind of graph, which is defined as *planar-st* in [17]. Based on this coding, [19] presents a twig query processing method. [17] extends this coding scheme to spherical st-graph. Note that both "planar st" and "spherical st" are strong conditions. To the best of our knowledge, there is no direct generalization of the above two labeling schemes to support general digraph.

[7] presents a 2-hop reachability coding scheme. But the length of the label for a node could be $O(n)$. This might add to much overhead for the query processing for graph-structured XML data.

With efficient coding, XML queries can also be evaluated using the join-based approaches. Structural join is such an operator and its efficient evaluation algorithms have been extensively studied in [2, 21, 13, 8, 5, 9, 20]. They are all based on coding schemes that enable efficient checking of structural relationship of any two nodes in a tree, and thus cannot be applied to the graph-structural XML data directly.

7 Conclusions

In this paper, we present a labeling scheme for graph-structured XML data. With such labelling scheme, the reachability relationship between two nodes in a graph can be judged efficiently. Based on the labeling scheme, we design efficient structural join algorithms for graph-structured XML, GMJ and IGMJ. Our experiments show that the labeling scheme has acceptable size while the proposed structural join algorithms outperform previous algorithms significantly. As one of our future work, we will design efficient index structure based on the labeling scheme to accelerate query processing.

References

1. *Introduction to Algorithms*. MIT Press, Cambridge MA, 1990.
2. Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002)*, pages 141–152, 2002.
3. Stephen Alstrup and Theis Rauhe. Small induced-universal graphs and compact implicit graph representations. In *Proceedings of 2002 IEEE Symposium on Foundations of Computer Science (FOCS 2002)*, pages 53–62, Vancouver, BC, Canada, November 2002.
4. Donald D. Chamberlin, Daniela Florescu, and Jonathan Robie. XQuery: A query language for XML. In *W3C Working Draft*, http://www.w3.org/TR/xquery, 2001.
5. Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient structural joins on indexed XML documents. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB 2002)*, pages 263–274, 2002.
6. Vassilis Christophides, Dimitris Plexousakis, Michel Scholl, and Sotirios Tourtounis. On labeling schemes for the semantic web. In *Proceedings of the Twelfth International World Wide Web Conference(WWW2003)*, pages 544–555, Budapest, Hungary, May 2003.
7. Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA 2002)*, pages 937–946, San Francisco, CA, USA, January 2002.
8. Torsten Grust. Accelerating XPath location steps. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002)*, pages 109–120, Hong Kong, China, August 2002.
9. Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-Tree: Indexing XML data for efficient structural join. In *Proceedings of the 19th International Conference on Data Engineering (ICDE 2003)*, pages 253–263, 2003.
10. Tiko Kameda. On the vector representation of the reachability in planar directed graphs. *Information Process Letters*, 3(3):78–80, 1975.
11. Haim Kaplan, Tova Milo, and Ronen Shabo. A comparison of labeling schemes for ancestor queries. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA 2002)*, pages 954 – 963, San Francisco, CA, USA, January 2002.
12. Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering indexes for branching path queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002)*, pages 133–144, 2002.
13. Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of 27th International Conference on Very Large Data Base (VLDB 2001)*, pages 361–370, 2001.
14. Tova Milo and Dan Suciu. Index structures for path expressions. In *Proceedings of the 7th International Conference on Database Theory (ICDE 1999)*, pages 277–295, 1999.
15. H. V. Jagadish Rakesh Agrawal, Alexander Borgida. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD 1989)*, pages 253–262, Portland, Oregon, May 1989.

16. Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A benchmark for XML data management. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB 2002)*, pages 974–985, 2002.
17. Roberto Tamassia and Ioannis G. Tollis. Dynamic reachability in planar digraphs with one source and one sink. *Theoretical Computer Science*, 119(2):331–343, 1993.
18. C. M. Sperberg-McQueen Franois Yergeau Tim Bray, Jean Paoli. Extensible markup language (xml) 1.0 (third edition). In *W3C Recommendation 04 February 2004*, http://www.w3.org/TR/REC-xml/, 2004.
19. Zografoula Vagena, Mirella Moura Moro, and Vassilis J. Tsotras. Twig query processing over graph-structured xml data. In *Proceedings of the Seventh International Workshop on the Web and Databases(WebDB 2004)*, pages 43–48, 2004.
20. Wei Wang, Haifeng Jiang, Hongjun Lu, and Jeffrey Xu Yu. PBiTree coding and efficient processing of containment joins. In *Proceedings of the 19th International Conference on Data Engineering (ICDE 2003)*, pages 391–402, 2003.
21. Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD 2001)*, pages 425–436, 2001.
22. Yoav Zibin and Joseph Gil. Efficient subtyping tests with pq-encoding. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001)*, pages 96–107, San Francisco, CA, USA, October 2001.