Subgraph Join: Efficient Processing Subgraph Queries on Graph-Structured XML Document*

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

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

Abstract. The information in many applications can be naturally represented as graph-structured XML document. Structural query on graph structured XML document matches the subgraph of graph structured XML document on some given schema. The query processing of graph-structured XML document brings new challenges.

In this paper, for the processing of subgraph query, we design a subgraph join algorithm based on reachability coding. Using efficient data structure, subgraph join algorithm can process subgraph query with various structures efficiently.

1 Introduction

XML has become the *de facto* standard for information representation and exchange over the Internet. XML data has hierarchy nesting structure. XML data is often modeled as a tree. However, XML data may also have IDREFs that add additional relationship to XML data. With such property, XML data also can be represented in graph structure. In many applications, data can be modeled as a graph more naturally than a tree.

Of course, graph structured XML document can be represented in tree structure by duplicate the element with more than one incoming paths. But it will result in redundancy.

Query languages are proposed for XML data. XQuery [4] and XPath [6] are query language standards for XML data. Structure query on graph structured XML data has more power. Further than branching query on tree structured XML data, structure query on graph-structured XML data can request subgraphs matching the general graph modeled schema described query.

Query processing on graph structured XML data brings new challenges:

- More complex query can be defined on graph-structured XML data. The query can be also graph-structured to retrieve a subgraph of an XML document. The schema of the subgraph can be various, possibly including nodes

* This work was partially supported by UNSW FRG Grant (PS06863), UNSW Goldstar Grant (PS07248) and the Defence Pre- Research Project of the Tenth Five-Year-Planof China no.41315.2.3.

W. Fan, Z. Wu, and J. Yang (Eds.): WAIM 2005, LNCS 3739, pp. 68–80, 2005.

with multiple parents or circle. Existing method cannot process such query efficiently.
- One way to processing structural query on XML data is to encode the nodes of graph with some labelling scheme. With the code, the structure relationship such as parent-child or ancestor-descendant can be judgment quickly. In query processing on tree structured XML, it is a well-studied problem. But all existing labeling scheme of XML representations and query processing methods are based on tree model. They can not be applied on graph-structured XML data directly.
- Another kind query processing methods for XML is to use structural index such as 1-index[15], F&B index[13] to accelerate the query processing. But the structural index of graph structured XML document has many nodes. It is not practical to use structural index directly to process query on graph structured XML. For example, the number of nodes in F&B index of tree structured 100M XMark document has 436602 nodes while the number of nodes in F&B index of graph structured 100M XMark document has 1.29M nodes [13].

Using label to represent the relationship between nodes is a practical method to process query on graph-structured XML data. With well-designed labeling, the structural relationship between two nodes can be determined efficiently without accessing any other node. In this paper we use an extension of the code in [16] as reachability code.

To process the complex queries with a graph schema on graph-structured, we design a novel subgraph join algorithm based on the reachability code. In order to support the overlapping of intervals in the coding, we design a data structure *interval stack*. Subgraph join algorithm uses a chain of linked interval stacks to compactly represent partial results. Subgraph join algorithm can be used to process subgraph query with both adjacent and reachability relationship.

The contributions of this paper can be summarized as follows:

- We use duplication to make the coding possible to be storage in relation or apply sorted based join algorithms on.
- We present efficient graph structural join algorithms and efficient data structure, interval stack, to support join.
- We present subgraph query, a novel kind of structure query using general graph as matching schema. To process subgraph query, we design a novel subgraph join algorithm. It processes subgraph query efficiently.

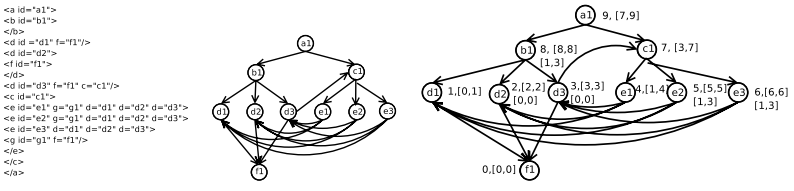
The reset of the paper is organized as follows: Section 2 introduces some background knowledge. Data preprocessing and subgraph join algorithm are presented in Section 3. We present our experimental results and analysis in section 4. Related work is described in Section 5. We conclude the paper in Section 6

2 Preliminaries

In this section, we briefly introduce Graph-structural XML model and some terms used in this paper.

2.1 Data Model

XML data is often modeled as a labelled 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 [23]. With this feature, XML data can be modeled as a labelled digraph: elements and attributes are mapped into nodes of graph; directed nesting and reference relationships are mapped into edges in the graph. An XML fragment is shown in Fig 1(b). It can be modeled as the graph shown in Fig 1(b). It is noted the graph in Fig 1(b) is not a DAG.



(a) An XML Fragment (b) XML Graph of Fig (c) The Reachibility code of Fig 1(b)

Fig. 1. An Example of Graph-structured XML

In a graph, a node without incoming edge is called *source*. A node without outcoming edge is called *sink*.

2.2 Subgraph Query

In graph-structured XML, the parent-child and ancestor-descendant relationship should be extended. In [13], the idref edges are represented as $\Rightarrow$ and $\Leftarrow$ for the forward and backward direction, respectively. We define the reachability relationship as two nodes a and b in the graph model G of XML data satisfy reachability relationship if and only there is a path from a to b in G. Each edge in this path can be either edges representing nested relationship or reference relationship. We represent reachability by $\rightsquigarrow$. For example, $a \rightsquigarrow e$ is to retrieve all the e elements with a path from a to it. In Fig 1(b), this query will retrieve $d1, d2$ and $d3$.

The combination to reachability restraints may forms *subgraph query*. Subgraph query will retrieve the subgraphs of graph-structured XML matching the structure given by the query. The graph corresponding to the query is called *query graph*. The nodes in query graph represent the tag name of required elements. The edges in query graph represent the relationship between required elements. If an edge in query graph represents adjacent relationship, it is called *adjacent edge*. If an edge in query graph represents reachability relationship, it is called *reachability edge*. For an example, the query shown in fig 2(a) on XML document shown in fig 1(b) represents the query to retrieve all the subgraphs of it with structure a node connects to a c node, d node reaches to this c node and this c node reaches a f node. the result is shown in fig 6.

2.3 Reachibility Coding

The goal of encoding XML is to represent the structural relationship so that the relationship between nodes in XML graph can be judged from the code quickly. With a good code, the query processing of structural query can be efficient. In this paper, we focus on *reachability coding*, which is used to judge the reachability relationship. We use an extension of reachability coding presented in [16]. In this coding, at first, all strongly connected components in the graph are contracted. Labeling is done by finding a spanning tree of the DAG generated in last step and assigning interval labels for nodes in the tree. The coding of the spanning tree is generated by post-order traversal. Each node is also assigned the number during traversal. The number is called *postid*. Next, to capture reachability relationships through non-spanning-forest edges, we add additional intervals to labels in reverse topological order of the DAG; specifically, if (u, v) is an edge not in the spanning forest, then all intervals of v are added to u (as well as labels of all nodes that can reach u). For an example, the reachability coding of graph in fig1(b) is shown in Fig 1(c). Using the spanning tree rooted at $a1$, we label $d2$, $f1$ with $[2, 2]$ and $[0, 0]$. In addition, d2 receives intervals from f1, resulting in that b2's code is $[2, 2], [0, 0]$. In this coding, $a \rightsquigarrow b$ if and only if $b.postid$ is contained some interval associated with a.

3 Subgraph Join

In this section, we discuss the processing of subgraph queries. We present subgraph join algorithm and the method of preprocessing query and data to support subgraph join algorithm.

3.1 Preprocess of the Input

The interval labelling scheme of a graph is different from that of tree. There may be more than one intervals assigned to one node. The processing unit of our method is interval. So that we should assign the *postid* of each node to all of its intervals. If several intervals associated to nodes with the same tag have the same x and y value but different *postid*, they are merged. The result of this step is a list of intervals, each of which is associated with one or more *postid*s. The list is called *candidate list*.

For the convenience of process, we will sort the intervals of all the nodes with the same tag by the value of x in ascending order and value of y in descending order. x is prior to y. It means only if two intervals have same x value, their y values are considered.

3.2 Preprocess for Subgraph Query

In order to apply subgraph join algorithm to process general subgraph query, some preprocess should be applied on the query when the query graph has circle.

If there are some circles in the query graph, a node n in each circle should be split to n_a and n_b break this circle. n_a includes all the incoming edges of n. n_b includes all the out edges of n. This node is the nodes related least edges in the circle.

When subgraph join is finished, the nodes in result corresponding to split query node are connected. Hash method is used.

Theorem 1. *After connection processing in the last step, the splitting of query node will not affect the final result of subgraph query.*

For the efficiency of query processing, before the process of data stated in Section 3.1, the nodes in the same SCC in each candidate list should be merged into one node. This node is called *stub node*. Since the coding of nodes in the same SCC have same intervals, the new node has these intervals, the number of the stub node is any of the number of the nodes belonging to the same SCC. Applying such preprocess is to prevent too large intermediate result during query processing without affecting the final result. For example, to process query shown in fig 3, there is a cycle in graph of the XML document with 100 a nodes, 100 b nodes and 100 c nodes respectively. Since they are reachable to each other, there will be 10^6 items in intermediate result after processing these nodes.

Corresponding to the merge, after the join is processed, the result should be extracted. The process of extraction is, for each result with stub node, from node set associated each merged nodes, one node is selected for one time to put on the position of the merged node. With a different combination of the selected nodes, one result is generated.

Theorem 2. *With extraction after all results are generated, the merging of nodes in the same SCC before query processing will not affect the final result.*

3.3 Data Structure for Subgraph Join

In our coding, there may be overlap in the intervals. Therefore, the stack based join of tree structured XML document can not be applied to our coding directly. We design a data structure, *interval stack*, to support efficient graph structural join. The interval stack is a DAG. Each node represents an interval. Each edge $e = n_1 \rightarrow n_2$ represents the interval of n_1 contains the interval of n_2. The child of each node is sorted by the x values of the intervals.

There are two additional structures of the digraph, top and bottom. Top is the list of the sinks which are intervals without any interval containing them. Bottom is the list of sources which are intervals without any interval contained in them. They are both sorted by x of the intervals.

There are mainly two operators of interval stack, append and trim. The former is to append an interval to interval stack. The latter is to delete useless intervals from interval stack. During the performing of these two operations, the property of interval stack should be kept and top and bottom are maintained.

3.4 Subgraph Join Algorithms

With interval stack, we improve stack-based twig join [3]algorithm to support subgraph queries.

Of compacted interval list, we have following observations:

- The *postid* of a node is contained one and only one of its intervals.
- If two nodes have reachability relationship, it can and only can be checked by one interval. That is, if $a \rightsquigarrow b$, among all the intervals of the reachability of a, only one contain the $b.number$.

Suppose the input query can be visualized as a rooted DAG. The circle in input query will be broken in preprocess. If there is no root. A dummy root is added to the query.

The join candidates are a series lists of intervals with a list of nodes it corresponds to.

For each node in query graph, a structure is build which includes an interval stack(S) and its current cursor(C), the parents and children of it in query graph. The interval stack has the same function as that in structural join. M is a hash map, mapping *postid* of node to its children. The algorithms of subgraph join are described in Alg 1.

The subgraph join algorithm has two phases. In the first phase, each pair of nodes satisfying partial reachability relation described in query is outputted. In the second phase, the nodes in intermediate result unsatisfied the whole query are trimmed. Such nodes being included in intermediate result is because in the first phase, when each pair of nodes is outputted, only partial reachability relation related to these two node is considered. For an example, for query shown in fig 3, some of the intervals to process are shown in fig 4, the ids in brackets are the *postid*s corresponding to the interval. Suppose the first number in bracket is in corresponding interval and others is not in the interval. During query processing, although a_{31} and c_{21} are not in final result, the pair (a_{31}, c_{21}) is still outputted.

During processing the query in fig 3 , interval a_1 contains interval c_1. Based on observation 1, only pairs (a_{11}, c_{11}), (a_{12}, c_{11}), (a_{13}, c_{11}) are appended to intermediate result. This is because from the containment of these two intervals, only that c_{11} is in interval a_1 can be determined. So only the reachability of all nodes in the extent of a_1 and c_{11} is true.

$getNext()$ is to find the next entry to process. It has similar function as $getNext$ of twigjoin in [3]. First of all, the interval with least x value is chosen. If some intervals have same x value, the interval with largest y is chosen. If two intervals have same x and same y and their corresponding query nodes have reachability relation, the interval corresponding query node as ancestor is chosen. Otherwise, some result will be lost. For an example, consider query in fig 3. on the element sets visualized in fig 4, the interval a_1 has the same x and y as interval b_1. The nodes corresponding to a_1 should be outputted with the nodes corresponding to b_1 and in the interval of b_1. But if b_1 is chosen former than a_1, these pairs will not be outputted. Since interval a_1 contains interval b_2, the nodes corresponding to a_1 should be outputted with the nodes corresponding to b_2 and in the interval of b_2. But if b_2 is chosen former, these pairs will lose.

Algorithm 1. GJoin(*root*)

```
1: while not end(root) do
2:     q = getNext(root)
3:     if not isSource(q) then
4:         if isSource(q) OR not emptyParent(q) then
5:             cleanNodes(q)
6:             push(q)
7:             advance(q)
8: obtainResult()

1: function END(q)
2:     return ∀q_i : isSink(q_i) ⇒ end(q_i.C)

1: procedure CLEARNODES(q)
2:     q.S.Trim(q.C)

1: function EMPTYPARENT(q)
2:     return ∃pi ∈ q_i.parents : p_i.C = p_i.end

1: procedure PUSH(q)
2:     for each node n ∈ q.C.context do
3:         if q = root then
4:             q.extent.add(n)
5:         if n.id > q.C.y then
6:             insertEntry(q.M, n)
7:             n.type = q
8:         else if n.id ≥ q.C.x then
9:             for each p ∈ q.parents do
10:                pointTo(p,q,n.id)

1: procedure POINTTO(p,q,id)
2:     for each entry i ∈ p.S do
3:         if id ≥ i.x AND id ≤ i.y then
4:             for each node n ∈ i.context do
5:                 M[n.id].child.add(id)

1: procedure OBTAINRESULT
2:     for each node n ∈ root.extent do
3:         b = generateResult(n)
4:         if b = FALSE then
5:             delete n from root.extent

1: function GENERATERESULT(node)
2:     if node is visited then
3:         return node.isresult
4:     b = TRUE
5:     for each child c of node do
6:         tb = generateResult(c)
7:         if tb = FALSE then
8:             delete c from node.child
9:             b = FALSE
10:        else if NOT c.type ∈ node.childtype then
11:            node.childtype.add(c.type)
12:        if node.childtype.size = node.type.child.size then
13:            node.isresult = TRUE
14:            return TRUE
15:        else
16:            node.isresult = FALSE
17:            return FALSE
```

Algorithm 2. getNext(q)

```
1: function GETNEXT(q)
2:     if isSink(q) then
3:         return q
4:     for q_i ∈ q.children do
5:         n_i = getNext(q_i)
6:         if n_i.left < n_min.left then
7:             n_min = n_i
8:         else if n_i.left = n_min.left then
9:             if n_i.right > n_min.right then
10:                n_min = n_i
11:            else if n_i.right = n_min.right AND n_i is a ancestor of n_min then
12:                n_min = n_i
13:    n_max = maxarg_{n_i}{n_i.C.x}
14:    while q_i.C.y < q_max.C.y do
15:        advance(q_i.C)
16:    if q_i.C.x ≤ q_min.C.x AND q_i.C.y ≥ q_min.c.y then
17:        return q
18:    else
19:        return n_min
```

Note the function $emptyParent()$ is to check whether the nodes in current interval satisfies the restriction of all incoming paths in the query. In out example, when interval c_3 is met, since interval stack of b is empty, it will not be considered.

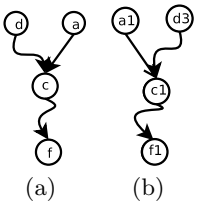


Fig. 2. Example Quries

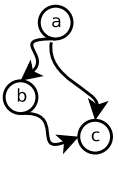


Fig. 3. Example Query

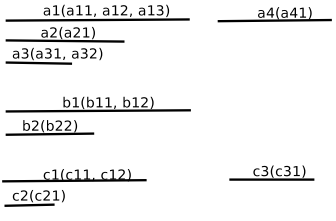


Fig. 4. Element sets for fig 3

Outputted pairs are organized by the ancestors. The main memory may be not enough to store intermediate results. External memory is used to store intermediate results. Since each node may have more than one descendant during query processing, children of one node are stored as a list in disk. The head of the list associated with a node record the number of the node, the query node corresponding to the node and the pointer to the first entry of the list. Each of entries in the list includes a 2-ary,($node$, $next$), where $node$ is the pointer to the node this entry corresponding to and $next$ is the pointer to next entry of the list. In the hash map, each entry e_n corresponds one node n. Each entry contain the head of the the position of the head and tail of list of n.

Theorem 3. *The logical I/O number of subgraph join algorithms is linear to the number to the pair of nodes satisfying the reachability relationship described in query.*

4 Experiments

In this section, we present results and analysis of part of our extensive experiment of subgraph join algorithms based on reachibility coding.

4.1 Experimental Setup

The Testbed. All our experiments were performed on a PC with Pentium 1GMHZ CPU, 256M main memory and 30G IDE hard disk. The OS is Windows 2000 Professional. We implemented all the algorithms using Microsoft Visual C++ 6.0. We implemented the encoding of graph and subgraph join algorithms. We use LRU policy for buffer replacement.

For comparison, we also implemented F&B index [13]for graph structured XML document. F&B index supports all the subgraph queries for XML.

Dataset. The dataset we tested is the standard XMark benchmark dataset[21]. We used scale factor 0.1, 0.2, 0.3, 0.4 and 0.5, which generated XML document with size 10M, 20M, 30M, 40M and 50M respectively. It has complicated schema, including circle.

Some statistics information of test XML documents are shown in Table 1.

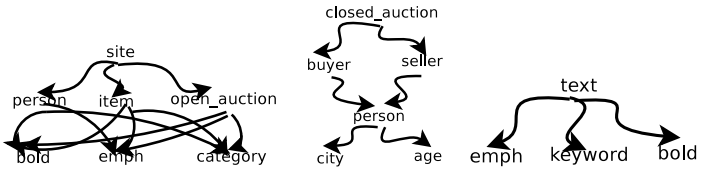
Table 1. Information of Test Document

Document size	11.3M	22.8M	34.0M	45.3M	56.2M
Node number	175382	351241	524067	697342	870628
Edge number	206129	413110	616228	820437	1024072

Query Set. In order to better test and understand the characteristics of the algorithms, we designed a set of queries that has different characteristics. We design three queries. They represent various structures. The query graph of them are shown in fig 5(a), fig 5(b) and fig 5(c), respectively.

4.2 Changing System Parameters

In this subsection, we investigate the performance of our system by varying various system parameters. We use physicalI/O and run time to reflect the impact of different parameter setting.



(a) GSQ1 (b) GSQ2 (c) GSQ3

Fig. 5. Test Queries

Scalability Experiment. We test the queries on XML documents with various sizes. In order to test the scalability of the subgraph join algorithm. We choose SGQ2 and SGQ3 as test query. We fix main memory 8M and block size 4096. The results are shown in fig 6(a) and fig 6(b), respectively. SGQ1 is a simple twig query. The nodes related to SGQ1 in XML document is not in any SCC and all have single parent. Therefore, the increase trend is nearly linear. SGQ2 is a complex subgraph query. One person node may be reached by more than one seller nodes and only parts of person nodes are reached by both seller node and buyer node. The trend of run time is faster than linear but still slower than square.

Varying Buffer Size. The physicalIO change with block number of SGQ1 is shown in fig 6(c). From the fig 6(c), we can find that without enough main memory, the second phase result more physical I/O than the first phase. This is because in the second phase the whole intermediate result is traversed while in the first phase, the operation is mainly append.

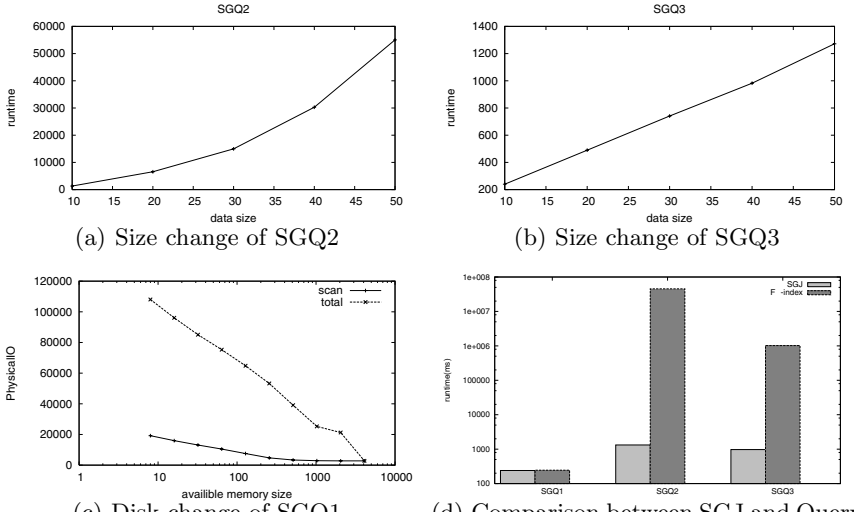
4.3 Comparison Experiment

We do comparison in 10M XML document. Its F&B-index has 167072 nodes. We naive implemented the depth first traversal-based query processing by F&B-index. The reason why we do not compare larger XML document is that when XML document gets larger, the query processing in F&B-index becomes too slow.

The result of comparison subgraph query process efficiencies of subgraph join algorithm and F&B index is shown in Fig 6(d). Y axis is in log scale. subgraph join algorithm outperforms the efficiency of F&B index. For SGQ1, the efficiency are similar. It is because the nodes in XML document related to SGQ1 is in tree structured in Xmark document and the search depth in F&B index is limited.

5 Related Work

With efficient coding, XML queries can also be evaluated on-the-fly using the join-based approaches. Structural join and twig join are such operators and their efficient evaluation algorithms have been extensively studied [27,14,8,10,5,25] [3,11]. Their basic tool is the coding schemes that enable efficient checking of



(a) Size change of SGQ2

(b) Size change of SGQ3

(c) Disk change of SGQ1

(d) Comparison between SGJ and Query Process with F&B-index

Fig. 6. Experiment Results

structural relationship of any two nodes. TwigStack [3] is the best twig join algorithm to answer all twig queries without using additional index. The idea of these work can be referenced to process query on graph. But these algorithms can not be applied on the coding of graph directly.

6 Conclusions

Information in some applications can be naturally stored as graph modeled data. The processing of graph structured XML data brings new challenges. To process structural query on graph structured XML data, in this paper, we present reachability labelling scheme for graph structured XML. With such labelling scheme, the reachability relationship between two nodes in graph structured XML can be judged efficiently. Based on the labelling scheme, we design graph structural join and subgraph join algorithms of graph structured XML to perform subgraph queries. From experiment, our labelling scheme has acceptable size. The subgraph join algorithm outperforms the query processing with F&B-index.

Our further work includes designing efficient index structure so support efficient query processing on graph structured XML document.

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. Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002)*, pages 310–321, 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. James Clark and Steve DeRose. XML path language (XPath). In *W3C Recommendation, 16 November 1999*, http://www.w3.org/TR/xpath, 1999.
7. Haim Kaplan Uri Zwick Edith Cohen, Eran Halperin. Reachability and distance queries via 2-hop labels. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA '02)*, 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. Ronen Shabo Haim Kaplan, Tova Milo. A comparison of labeling schemes for ancestor queries. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA '02)*, pages 954 – 963, San Francisco, CA, USA, January 2002.
10. 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.
11. Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig joins on indexed xml documents. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB 2003)*, pages 273–284, 2003.
12. Tiko Kameda. On the vector representation of the reachability in planar directed graphs. *Information Process Letters*, 3(3):78–80, 1975.
13. 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.
14. 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.
15. 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.
16. 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.
17. Gerhard Weikum Ralf Schenkel, Anja Theobald. Hopi: An efficient connection index for complex xml document collections. In *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology(EDBT04)*, pages 237–255, Heraklion, Crete, Greece, March 14-18 2004.
18. Ioannis G. Tollis Roberto Tamassia. Dynamic reachability in planar digraphs with one source and one sink. *Theoretical Computer Science*, 119(2):331–343, 1993.

19. A. Sayed and R. Unland. Indexing and querying heterogeneous xml collections. In *Proceedings of In 14th International Conference on Computer Theory and Applications*, Alex, Egypt, Septempber 2004.
20. Ralf Schenkel. Flix: A flexible framework for indexing complex xml document collections. In *Proceedings of International Workshop on Database Technologies for Handling XML Information on the Web(DATAX04)*, Heraklion, Crete, Greece, March 2004.
21. 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.
22. Theis Rauhe Stephen Alstrup. Small induced-universal graphs and compact implicit graph representations. In *Proceedings of 2002 IEEE Symposium on Foundations of Computer Science (FOCS '02)*, pages 53–62, Vancouver, BC, Canada, November 2002.
23. 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.
24. Michel Scholl Sotirios Tourtounis Vassilis Christophides, Dimitris Plexousakis. 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.
25. 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.
26. Joseph Gil Yoav Zibin. 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.
27. 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.
28. Vassilis J. Tsotras Zografoula Vagena, Mirella Moura Moro. 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.