World Wide Web (2008) 11:485–510
DOI 10.1007/s11280-008-0050-4

Coding-based Join Algorithms for Structural Queries on Graph-Structured XML Document

Hongzhi Wang · Jianzhong Li · Wei Wang · Xuemin Lin

Received: 1 February 2007 / Revised: 10 January 2008 /
Accepted: 9 July 2008 / Published online: 1 August 2008

Abstract In many applications, XML documents need to be modelled as graphs. The query processing of graph-structured XML documents brings new challenges. In this paper, we design a method based on labelling scheme for structural queries processing on graph-structured XML documents. We give each node some labels, the reachability labelling scheme. By extending an interval-based reachability labelling scheme for DAG by Rakesh et al., we design labelling schemes to support the judgements of reachability relationships for general graphs. Based on the labelling schemes, we design graph structural join algorithms to answer the structural queries with only ancestor-descendant relationship efficiently. For the processing of subgraph query, we design a subgraph join algorithm. With efficient data structure,

Support by the Key Program of the National Natural Science Foundation of China under Grant No.60533110; the National Grand Fundamental Research 973 Program of China under Grant No. 2006CB303000; the National Natural Science Foundation of China under Grant No. 60773068 and No. 60773063.

H. Wang (✉) · J. Li
Department of Computer Science and Technology, Harbin Institute of Technology,
P.O. Box 750, Harbin 150001, China
e-mail: wangzh@hit.edu.cn

J. Li
e-mail: lijzh@hit.edu.cn

W. Wang · X. Lin
School of Computer Science and Engineering, University of New South Wales,
Kensington, Australia

W. Wang
e-mail: weiw@cse.unsw.edu.au

X. Lin
e-mail: lxue@cse.unsw.edu.au

the subgraph join algorithm can process subgraph queries with various structures efficiently. Experimental results show that our algorithms have good performance and scalability.

Keywords XML · query processing · subgraph query · coding · structural join

1 Introduction

XML has become the de facto standard for information representation and exchange over the Internet. XML data has a nesting structure. XML data is often modelled as a tree. However, XML data may also have IDs and IDREFs that add additional relationships. With such property, XML data can also be represented in graph structure. In many applications, data can be modelled as a graph more naturally than a tree. For example, the relationship of publications and authors adapts to be represented as graph structure. Because one paper may have more than one authors and one author may have more than one papers. A fragment of such information is shown in Figure 1.

Of course, a graph-structured XML document can be represented in tree structure by duplicating the elements with more than one incoming paths. But such strategy will result in redundancy. If the information in Figure 1 is represented with tree structure, the element author will be duplicated.

Some query languages are proposed for XML data. XQuery [4] and XPath [8] are query language standards for XML data. Structural queries on graph-structured XML data show more power and can request subgraphs matching the general graph-modelled schema described in the query. For example, the query to a graph-structured XML document in Figure 1 requests author elements with publication both in proceeding and journal.

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

- More complex queries are defined on graph-structured XML data. Such queries are also graph-structured to retrieve subgraphs of an XML document. The schema of a subgraph can be various, possibly including nodes with multiple parents or circle. Existing method cannot process such queries efficiently.
- One way to processing structural queries on XML data is to encode the nodes of graph with some labelling scheme. With the labelling scheme, the structure relationship such as parent-child or ancestor-descendant can be judged without accessing other information. In the query processing on tree structured XML, it

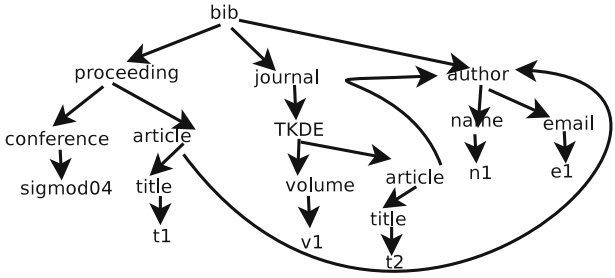


Figure 1 An example of graph-structured XML.

is a well-studied problem. But most of labelling schemes of XML representations and query processing methods are based on tree model. They cannot be applied on graph-structured XML data directly.

- Another kind of query processing methods for XML uses structural index such as 1-index [16], F&B index [14] to accelerate the query processing. However, the structural index of graph-structured XML document often contain 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 [14].

Using labels to represent the structural relationship between nodes is a practical method to process query on graph-structured XML data. With well-designed labelling scheme, the structural relationship between two nodes can be determined efficiently without accessing any other node. In this paper, we present reachability labelling scheme which can be used to process ancestor-descendant queries. We design interval-based reachability labelling scheme for general-graph-structured XML document.

Based on the labelling schemes presented in this paper, we present graph structural join to process reachability queries efficiently.

To process the complex queries with a graph schema on graph-structured XML document, we design a novel subgraph join algorithm based on the reachability labelling scheme. 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. The subgraph join algorithm can be used to process subgraph queries with both adjacent and reachability relationship.

The contributions of this paper can be summarized as follows:

- We generalize the coding scheme in [17] and present an effective labelling scheme to judge the reachability relationships between nodes in a general digraph.
- We use duplication to make the coding possible to be stored in relations or apply sorted based join algorithms on them.
- We present two efficient structural join algorithms on graph-structured XML data, GMJ and IGMJ, based on the labelling scheme for graph.
- We present subgraph query, a novel kind of structural query using general graph as matching schema. To process subgraph queries efficiently, a novel subgraph join algorithm is proposed.
- Our experiments show that our labelling scheme is efficient for XMark data. Our two structural join algorithms and subgraph join algorithm outperforms existing query processing methods significantly, respectively. All algorithms have good scalability.

The rest of the paper is organized as follows: Section 2 introduces some background knowledge. Section 3 presents the reachability labelling scheme. Structural join algorithms based on the labelling scheme are given in Section 4. Data preprocessing and subgraph join algorithm are presented in Section 5. We present our experimental results and analysis in Section 6. Related work is described in Section 7. We conclude the paper in Section 8.

2 Preliminaries

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

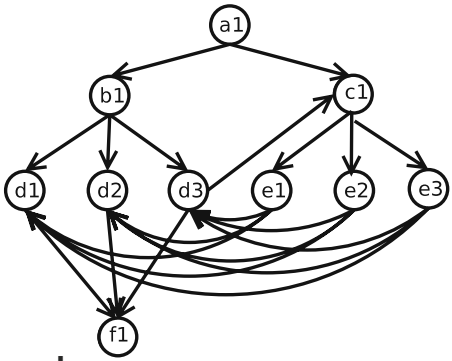
XML data is often modelled as a labelled tree: elements and attributes are mapped into nodes of the tree; directed nesting relationships are mapped into edges in the tree. A feature of an XML document is that from two elements in XML document, there may be an IDREF representing reference relationships [21]. With such feature, an XML document can be modelled as a labelled directed graph (digraph): elements and attributes are mapped into nodes of the graph; directed nesting *and* referencing relationships are mapped into directed edges in the graph. An example XML document is shown in Figure 2a. It can be modelled as the digraph shown in Figure 2b. Note that the graph in Figure 2b is not a DAG.

Queries in XML query languages such as XQuery[4] use tree pattern for matching relevant parts of data in a XML document. The query pattern node labels include element tags matching, attribute-value comparisons and string values and the query pattern edges are either parent-child edges or ancestor-descendant edges. In this paper, we focus on structural query with only element tags matching and edges representing relationship between tags but not value comparisons. In XPath, the parent-child relationship is represented in '/' and ancestor-descendant relationship is represented by '//'.

However, when an XML document is modelled as a graph, the ancestor-descendant relationship (as well as parent-child relationship) can be extended based on the notion of reachability. In [14], IDREF edges are represented as $\Rightarrow$ and $\Leftarrow$ for the forward and backward edges, respectively. In a graph-structured XML document,

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

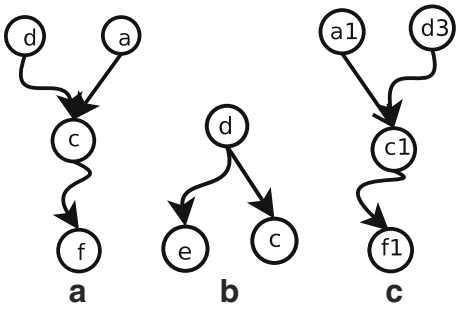
a An Example XML Document



b The Corresponding XML Graph

Figure 2 An example XML document and its graph structure.

Figure 3 Example queries.



structural queries are defined based on the structural relationship between nodes. In the graph structure G of an XML document, two nodes a and b satisfy *adjacent relationship* if and only if an edge from a to b exists in G; two nodes a and b satisfy *reachability relationship* if and only if a path from a to b exists in G. A *reachability query* $a \rightsquigarrow d$ is to retrieve all pairs (n_a, n_d) in G satisfies two conditions: 1)n_a has tag a and n_d has tag d; 2)n_a and n_d satisfy reachability relationship in G. For example, the result of reachability $a \rightsquigarrow e$ in the graph in Figure 2b includes e_1, e_2, e_3. *Adjacent queries* can be defined similarly.

The combination to reachability restraints forms *subgraph queries*. 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 the *query graph*. The nodes in a query graph represent the tag names of required elements. The edges in a query graph represent the relationship between required elements. If an edge in a query graph represents adjacent relationship, it is called an *adjacent edge*. If an edge in a query graph represents reachability relationship, it is called a *reachability edge*. For example, the query shown in Figure 3a on the XML document shown in Figure 2a represents the query to retrieve all the subgraphs with the structure that an a node connects to a c node, d node reaches to this c node and this c node reaches an f node. The result is shown in Figure 3c.

Since trees are a special case of graphs, twig queries can be represented as a special case of subgraph queries. For example, a twig query "d[//e AND c]" on an XML document shown in Figure 2a can be represented as the subgraph query shown in Figure 3b.

3 The coding of graph-model XML document

In this section, we describe a labelling scheme of general-graph-structured XML documents. We extend the labelling scheme for directed acyclic graphs (DAGs) in [17] to support general digraphs. Therefore, with such labelling scheme, the reachability relationship between two nodes in a general digraph can be judged efficiently without accessing other information.

3.1 The coding of DAGs

In this subsection, we describe DAG encoding method in [17] briefly. In such coding scheme, a list of intervals and an id is assigned to each node. We briefly summarize the encoding method for a DAG D in the following: *First, an optimal tree-cover T of the DAG D is found . T is traversed in a depth-first manner. During the traversal, an interval $[x, y]$ is assigned to each node n of T, where y is the postorder of n during the traversal. x is the smallest postorder number of all n's descendants in T. Next, all the nodes of D are examined in the reverse topological order. At each node n, if possible, all the intervals of its out-going nodes in D are copied and merged to its code.*

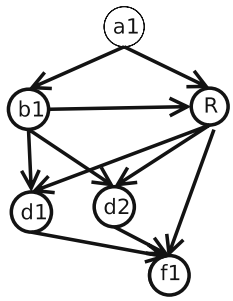
The judgement of the reachability relationship between two nodes a and b is to check whether the postorder of b is contained in one of the intervals of a.

An example of encoding a DAG D is shown as following. The coding of the DAG in Figure 4a is in Figure 4b. In Figure 4b, the subgraph formed with solid line is the tree cover of D. We use *postid* to denote the postorder number of each node, which is also the second value of the first interval of its code.

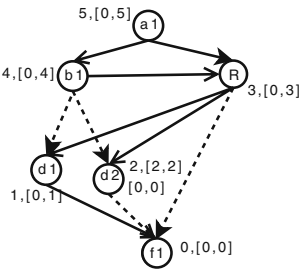
3.2 The coding of general graphs

We now generalize above labelling scheme for directed cyclic graphs. We assume that the graph consists 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.

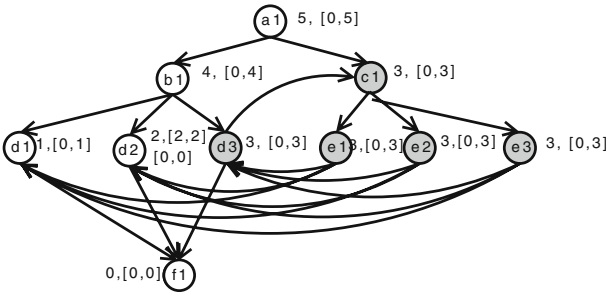
Figure 4 The coding scheme.



a The DAG contracted from XML Graph



b Interval Coding for the DAG



c Interval Coding for the XML Graph

The reachability coding of a general digraph G is generated with the following steps:

1. All strongly connected components (SCC) with number of nodes greater than one are found.
2. Each SCC C of G is contracted to a *representative node* n_C. Here, the contraction of a SCC C means that all the nodes of C are merged to n_C and each edge with one vertex in C and the other vertex v in $G - C$ is change to the edge between n_C and v. As a result, G is reduced to a DAG G' (the correctness of this step is proved in Theorem 1). Suppose SCC $S = \{node_0, node_1, \cdots, node_t\}$ 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 SCC 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 2b. In order to encode G, the first step is to contract all the SCCs. In G, there is only one such SCC, $S = \{d3, c1, e1, e2, e3\}$. By contracting this SCC, a DAG G' shown in Figure 4a is generated. In G', R is the representative node for SCC S. The interval codes of G' is shown in Figure 4b. 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 the graph G are shown in Figure 4c. 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.

Theorem 1 *A directed cyclic graph G is converted to a DAG by contracting all the SCCs to representative nodes.*

Proof Two nodes n_{C_a}, n_{C_b} are representative nodes in contracted graph G' of G correspond node set C_a and C_b, respectively. n_{C_a} and n_{C_b} must not be strongly connected. Otherwise, the nodes in C_a and C_b satisfy strongly connection relationship. It is contradictory to that C_a is an SCC. So in the way stated above, G is converted into a DAG. □

Theorem 2 *For two nodes a and b of an XML graph encoded in steps stated, 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 SCCs, respectively. Then* $a \rightsquigarrow b$ *if and only if* $\exists i (0 \le i \le n)$ *such that* $a_i.x \le postid_b \le a_i.y$ *(condition P).*

Before proving Theorem 2, we define some symbols. $G = (E, V)$ is a directed graph. $V = V_o \cup V_c$. V_o is the set of nodes that are not contracted. V_c is the set of nodes to be contracted. $G' = (E', V')$ is the graph generated from G by contracting all SCCs. $V' = V'_o \cup V'_c$. V'_o is a subset of nodes in G. V'_c is the set of the nodes contracted from SCCs from G. $V_o = V'_o$. Suppose the SCC that a node $n \in V_c$ belongs to is C_n. The node in V'_c contracted from C_n is v_{C_n}. The code of each node is obtained in G'. If G is a DAG, then $V_c = \emptyset$.

Lemma 1 *Let a, b be two nodes in a graph G. Corresponding nodes of a and b in the contracted DAG G' are a' and b', respectively. If $a' \rightsquigarrow b'$, then $a \rightsquigarrow b$.*

Proof Suppose these is no path from a to b in G. a and b are not in the same SCC. Since the contraction does not bring new path in the graph but only make the path with edges in SCC short. So a' and b' are not connected in G'. It is contradictory with $a' \rightsquigarrow b'$. So $a' \rightsquigarrow b' \Rightarrow a \rightsquigarrow b$. □

According to Lemma 1, we give the proof of Theorem 2.

Proof (Theorem 2) Since G' is a DAG, based on [17], for $a, b \in G'$, $a \rightsquigarrow b$ iff the code of a and b satisfy P.

For two nodes $a, b \in G$ and $a \rightsquigarrow b$, there are four instances, $a \in V_o$ or $a \in V_c$ and $b \in V_o$ or $b \in V_c$. In the instance $a \in V_o$ and $b \in V_o$, [17] has proved the a and b satisfied P. If $a \in V_o$ and $b \in V_c$, in G', $a \rightsquigarrow v_{C_b}$. So that the codes of a and v_{C_b} satisfy P. b and v_{C_b} have same code. So the code of a and b satisfy P. For the same reason, when $a \in V_c$ and $b \in V_o$, the codes of a and b satisfy P. When $a \in V_c$ and $b \in V_c$, if $v_{C_a} = v_{C_b}$, a and b have same SCC codes, satisfying P. If $v_{C_a} \neq v_{C_b}$, $v_{C_a} \rightsquigarrow v_{C_b}$. The codes of v_{C_a} and v_{C_b} in G' satisfy P. Since a has same code as v_{C_a} and b has same code as v_{C_b}, the code of a and b satisfy P.

Suppose a and b are two nodes and their codes satisfy P. The corresponding nodes of a and b in G' are a' and b' respectively. $a' \rightsquigarrow b'$. According to Lemma 1, $a \rightsquigarrow b$. □

Time complexity analysis The finding of all SCCs can leverage the DFS-based algorithm in [20], 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 SCCs. The complex of encoding a DAG is $O(n')$ [17], 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 queries on graph-structured XML data using the labeling scheme presented in Section 3. The structural join algorithms compute the result of a reachability query $a \rightsquigarrow d$. The labelling schemes of nodes with tag a are in Alist and the labelling schemes of nodes with tag d are in Dlist. The format of Alist and Dlist will be presented in Section 4.1.

4.1 Preprocessing 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. The nodes to joined are preprocessed 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 the SCC of this element.

Then both inputs are sorted: 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 SCC have the same codes, hence intervals.
- Even if two nodes do not belong to the same SCC, 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 children will be appended to n. If some added interval of a child c cannot be merged in the existing interval of n, c and n with the same tag will have the same interval even if they do not belong to any SCC together.

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

Implementation-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 the query $d \rightsquigarrow e$ against the XML document shown in Figure 4c, Alist is $\{d1([0, 1]), d2([0, 0], [2, 2]), d3([0, 3])\}$, and Dlist is $\{e1(3), e2(3), e3(3)\}$. The intervals associated with a node is in the brackets following the node. After preprocessing, the Alist becomes { $[0, 4](d3)$, $[0, 1](d1)$, $[0, 0](d2)$, $[2, 2](d2)$ }, the Dlist becomes $\{3(e1, e2, e3)\}$. Preprocessed Alist and Dlist are sorted by the codes (intervals and *postid*s, respectively). The nodes corresponding to an interval i (or *postid*) is in the bracket following 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.

4.2 Structural join algorithms

After preprocessing, a naïve structural join algorithm can be obtained by generalizing the sort-merge based structural join algorithm in [1]. One subtlety is that the intervals in the preprocessed Alist might have the same starting or ending values (i.e., x or y). The codes shown in Figure 4c are such an example. We present the merge-based join algorithm on graph, named *Graph–Merge–Join* (GMJ), in Algorithm 1.

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

- Alist: $a1([1, 3]), a2([1, 1]), a3([4, 6]), a4([4, 5]), a2([8, 8])$
- Dlist: $d1(1), d2(4), d3(7)$

which correspond to a nodes and d nodes in Figure 5.

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 $a2([1, 1])$. When processing $d2$, the pointer of Alist moves to $a3$. $d2$ joins with $a3$ and $a4$. When processing $d3$, the pointer of Alist is on $a2([8, 8])$ with x value larger than the id of $d3$. The pointer of Dlist is then moves to the tail without outputting any results. So the algorithm terminates.

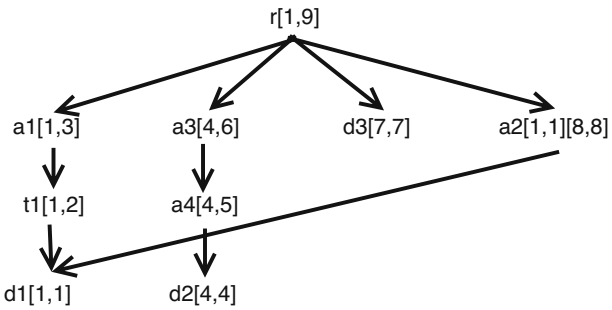
Algorithm 1 GMJ(*Alist*,*Dlist*)

```
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 do
11:             if a.y ≥ d.postid then
12:                 Append pair (a, d) to the output
13:             a = a.next()
14:         a = bookmark
15:     d = d.next()
```

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 [1] cannot be directly generalized to work with our coding scheme. This is because that two intervals may be partially overlapping. For example, a graph and its codes are shown in Figure 6. The intervals assigned to $a3$ and $a4$ are partially overlapping. As a result, stacks can no longer be used to represent the nesting relationship between intervals in our coding scheme.

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

Figure 5 An example of GMJ.



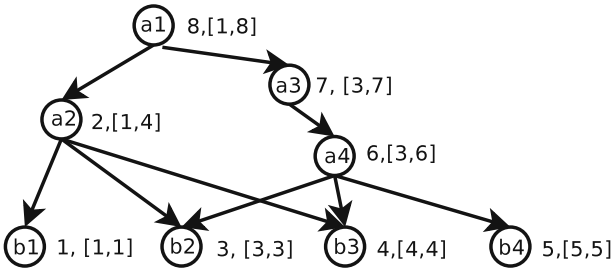


Figure 6 An example of overlapping code.

I to the RST; $trim(v)$ 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 $a2$([1, 1]) are smaller than $d1.postid$. At first, $a1$ and $a2$ are inserted into the RST. Then, $(a1, d1)$ and $(a2, d1)$ are appended to the result list. $a3$ and $a4$ are processed before $d2$. 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 $(a4, d2)$ are appended to the result list. $a3$ and $a4$ are trimmed from RST based on $d3$ then.

5 Subgraph join

In this section, we discuss the processing of subgraph queries. We present subgraph join algorithm and the method of query preprocessing. Before performing subgraph join, data is preprocessed in the same way as GMJ.

5.1 Preprocessing for subgraph query

In order to apply subgraph join algorithm to process general subgraph queries, some preprocessing strategies are applied on the query with circles. If there are some circles in the query graph, a node n in each circle should be split to n_a and n_b to break this circle. n_a includes all the incoming edges of n. n_b includes all the outgoing

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

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

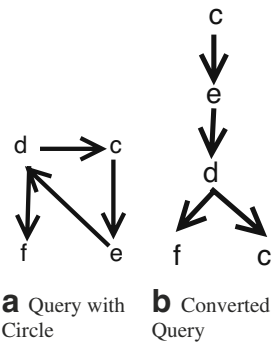


Figure 7 Examples of query with circles.

edges of n. For a circle, a node with the smallest corresponding nodes in the graph is chosen to split. It is because the splitting of such node will possibly results in smaller intermediate results.

When subgraph join is finished, the subgraph in the result with the same nodes corresponding to split query nodes is considered as the result and the such nodes are merged.

An example is used to illustrate the processing of subgraph queries with circles. The query is shown in Figure 7a. After splitting a node c, the query is converted to the query in Figure 7b. The results for such query on the graph shown in Figure 2b are $(c1, e1, d3, c1, f1)$, $(c1, e2, d3, c1, f1)$, $(c1, e2, d3, c1, f1)$. Since these three subgraphs all have the same c nodes corresponding the c nodes in the converted query. They are all included in the final result. The final result is $(c1, e1, d3, f1)$, $(c1, e2, d3, f1)$, $(c1, e2, d3, f1)$.

Theorem 3 *After connection processing in the last step, the splitting of query node will not affect the final result of the subgraph query.*

Proof In a subgraph query, the incoming edges and outgoing edges of a node n can be considered as the restrictions to this node. The nodes satisfying all these restrictions are retrieved. The splitting of the incoming edges and outgoing edges can be considered as a splitting of these restrictions. The connection of the final result is

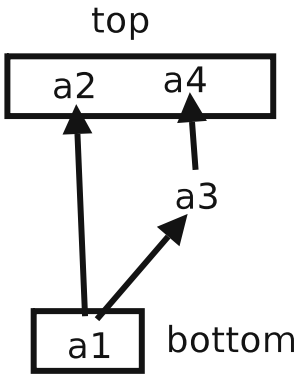


Figure 8 An instance of interval stack.

to obtain the intersection of the result set of the split query nodes n_1 and n_2. Since the results of n_1 and n_2 are selected from the same node set (nodes with the same tag), the intersection of these two result sets is to add all restrictions of the two nodes on this node set. Therefore, it equivalents to the result of the query node n with all incoming and outgoing edges as restrictions. □

For the efficiency of the query processing, before the process of the data stated in Section 4.1, the nodes in the same SCC in each candidate list are merged into one node. This node is called a *stub node*. Since the coding of nodes in the same SCC have same intervals, the coding of the stub node has these intervals; the id of the stub node is any id of the nodes belonging to the same SCC. The goal of such preprocessing is to prevent too large intermediate results during query processing without affecting the final result. For example, to process the query shown in Figure 9, 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 the intermediate result after processing these nodes.

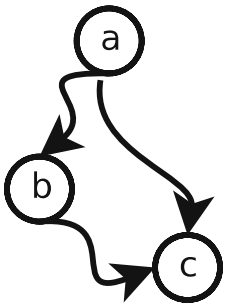
Corresponding to the merge, after the join is performed, the result should be extracted. The process of extraction is, for each result with the stub node, from the node set associated with each merged nodes, one node is selected once to put on the position of the merged node. With a different combination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

Theorem 4 *With extraction after all results are generated, the merging of nodes in the same SCC before query processing will not affect the final result.*

Proof The stub node is one of the nodes in the SCC. Since all nodes in the same SCC reaches same nodes and are reached by same nodes. So if the stub node is changed to other node in the same SCC with same tag. The result also satisfies the query.

The merging will not affect the results without nodes in any SCC. If there is a node n in the final result of a query q on the unmerged data belonging to some SCC, there must be a result in the result of q on the merged data. The position of n is the stub node of the SCC of n, then after extraction, there must be a result with the position n. Hence, each result of q on the unmerged data corresponds to one extracted result of q on the merged data. □

Figure 9 Example query.



5.2 Data structure for subgraph join

As mentioned before, there may be overlapping in the intervals. Therefore, the stack-based join of tree structured XML document cannot be applied to the coding directly.

We design a data structure, *interval stack*, to support the 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. Bottom is the list of sources. They are both sorted by x values of the intervals. An example of interval stack is shown in Figure 8.

There are mainly two operators of interval stack, append and trim. The former is to append an interval to the interval stack. The latter is to delete useless intervals from the interval stack. The pseudo code of the implementations of these two operators are in Algorithm 3 and Algorithm 5, respectively. The function *delSubgraph*(n) is to delete the subgraphs in the interval stack with root n.

5.3 Subgraph join algorithms

With the interval stack, we present a join algorithm to support subgraph queries.

We have following observations of the compacted interval list:

- The *postid* of a node is contained in 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 coding of a, only one contain the $b.id$.

Algorithm 3 append(S,i)

```
1:  if S.top is empty then
2:      S.top.addback(i)
3:      S.bottom.addback(i)
4:  else if i.x > S.bottom.end.y then
5:      S.top.clear
6:      S.bottom.clear
7:      S.top.addback(i)
8:      S.bottom.addback(i)
9:  else if node.y > S.bottom.end.y then
10:     S.top.addback(i)
11:     S.bottom.addback(i)
12: else
13:     for each interval n in S.bottom do
14:         if n.y ≤ i.x then
15:             insert(i, n)
16:     S.top.addback(i);
```

Algorithm 4 insert(i,n)

```
1: if i.x ≤ n.x and n.y ≤ i.y then
2:     if n is not accessed then
3:         b = false
4:         for each child c of n do
5:             if insert(i,c) = true then
6:                 b =true
7:         if b =false then
8:             n.child.addback(i)
9:             if n is in S.top then
10:                S.top.delete(n)
11:    return true
12: else
13:    return false
```

After preprocessing, the input query is visualized as a rooted DAG. Because the circle in the input query are broken in preprocess. If there is no root, a dummy root is added to the query.

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

For each node in the query graph, a structure is built which includes an interval stack (S) and its current cursor (C), the pointers to its parents and children in the query graph. The interval stack is described in Section 5.2. M is a hash map, mapping *postid* of node to its children. The algorithms of subgraph join are described in Algorithm 6.

The subgraph join algorithm has two phases. In the first phase, all pairs of nodes satisfying partial reachability relation described in the query are outputted. In the second phase, the nodes in the intermediate result unsatisfied the whole query are trimmed. That such nodes are possibly included in the 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 example, for the query shown in Figure 9, some of the intervals to process are shown in Figure 10, the ids in brackets are the *postid*s corresponding to the interval. Suppose the first number in the brackets is in corresponding interval and others is not in the interval. During

Algorithm 5 trim(S,i)

```
1: for each node n in S.bottom do
2:     if n.y < i.x then
3:         delSubgraph(n)
4: for each node n in S.top do
5:     if n.y < i.x then
6:         top.delete(n)
7:         if n has parents then
8:             for each parent p of n do
9:                 if p has no child other than n then
10:                    top.insert(p)
```

Algorithm 6 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
```

query processing, although a_{31} and c_{21} are not in the final result, the pair (a_{31}, c_{21}) is still outputted.

During the processing of the query in Figure 9, a_1 contains c_1. Based on Observation 1, only pairs (a_{11}, c_{11}), (a_{12}, c_{11}), (a_{13}, c_{11}) are appended to the intermediate result. This is because from the containment of these two intervals, only that c_{11} is contained in a_1 can be determined. Therefore, only the reachability of all nodes in the extent of a_1 and c_{11} is true.

```
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
```

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 the least x value is chosen. If some intervals have the same x value, the interval with the largest y is chosen. If two intervals have the same x and the same y and their corresponding query nodes have reachability relationship, the interval corresponding to the query node as ancestor is chosen. Otherwise, some result will be lost. For example, the query in Figure 9 is considered. On the element sets visualized in Figure 10, the interval a_1 has the same x and y as interval b_1. The nodes in the interval of b_1 corresponding to a_1 should be outputted with the nodes corresponding to b_1. But if b_1 is chosen former than a_1, these pairs will not be outputted. Since a_1 contains b_2, the nodes corresponding to a_1 should be outputted with the nodes corresponding to b_2. But if b_2 is chosen at first, these pairs will be lost.

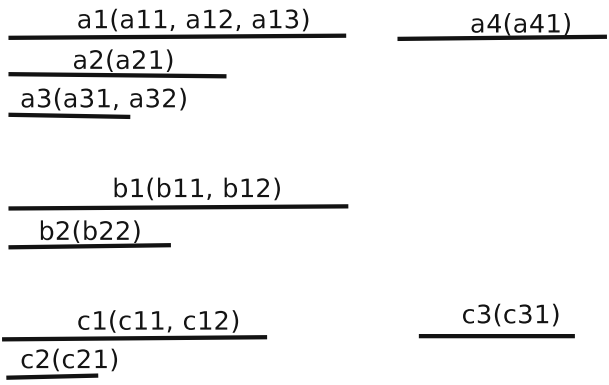


Figure 10 Element sets for Figure 9.

Algorithm 7 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 that the function *emptyParent*() is to check whether the nodes in current interval satisfies the restriction of all incoming paths in the query. In our example, when interval c_3 is met, since interval stack of b is empty, c_3 will not be considered.

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 pair (*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.

Theorem 5 *The logical I/O number of the subgraph join algorithm is linear to the number to the pair of nodes satisfying the reachability relationship described in the query.*

Proof In the first phase, the nodes and the pairs are appended to disk when the condition is satisfied. When each node is first met, it will be added to the intermediate result. When a pair of nodes (n_1, n_2) satisfies a reachability relationship described in query, this pair is added to the intermediate result. The adding method is to link n_2's position to n_1's child link. For adding each pair, twice logical I/O are required, one for appending pointer to the intermediate result, the other for filling back the address to the original tail of the child list of n_1. The logical I/O in this step is $N + 2M$ where N is the number of nodes in the disk and M the number pairs of nodes satisfying the reachability relationship described in the query.

After the first phase, the intermediate result in the disk can be considered as a graph. The second phase is to scan the graph once. The number of edges in this graph equals to the number of pairs of nodes satisfying the reachability relationship described in the query. This step requires $N + M$ times logical I/O.

The total number of logical I/O is $2N + 3M$. This is linear to M, the number of the pairs of nodes satisfying the reachability relationship described in the query. □

Discussion of processing subgraph query with adjacent edges Subgraph queries with adjacent edges can be processed with the method similar to that discussed in this section. Each node is given a adjacent code. The generation of adjacent code is that for each node with postid i, interval $[i,i]$ is assigned to each of its parents. The benefit of such coding is that the judgement of adjacent relationship is the same as that of reachability relationship so that the method in this session can be applied to the subgraph query with adjacent edges. Note that if one query node has both reachability and adjacent outgoing edges, this node should be split into two query nodes with only reachability and adjacent outgoing edges, respectively. It is because different intervals are to be used to judge reachability and adjacent relationship. The incoming edges are not split. It is because the judgements of reachability and adjacent relationships use the same set of *postid*s.

6 Experiments

In this section, we present results and analysis of part of our extensive experiments of the new labelling scheme and the algorithms presented in this paper.

6.1 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**), Improved-Graph–Merge–Join (**IGMJ**) and subgraph join algorithm using the file system as the storage engine. For comparison, we also implemented a traversal-based query processing algorithm based on the 1-index [16] (**1-index**) and F&B index [14] for graph-structured XML document. F&B index supports all the subgraph queries for XML. We also implement GRIPP [22], the latest reachability labelling scheme and reachability query processing method based on it, the best subgraph query processing algorithm , StackD [5], as well. We use LRU policy for buffer replacement.

We use the XMark benchmark dataset [19] in our experiments. It is a frequently used dataset with irregular schema. We measure the performance of different algorithms on the 20M XMark dataset (with scale factor 0.2). It has 351,241 nodes and its 1-index has 161,679 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 reachability 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	3,158	14,222	8,032
Q2	site⇝item	1	4,549	4,350
Q3	person⇝category	3,158	200	199
Q4	people⇝privacy	205	1,195	1,182

In order to better test and understand the characteristics of the subgraph join algorithm, we designed three queries with different characteristics. The query graph of them are shown in Figure 11a, b, c, respectively.

6.2 Space overhead of the coding

We measure the space overhead of our labelling 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 latter measurement represents the average number of intervals associated with one node to be processed during structural join.

The results of the size of codes are shown in Table 2, where IAP means the number of total intervals after preprocessing. IPNJ is smaller 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 the join is meaningful by exploiting the sharing of intervals and *postid*s, respectively.

We compare the coding presented in this paper with GRIPP [22]. The numbers of intervals and ids in GRIPP are shown in columns of GRINT and GRID, respectively. From the comparison result, even though the number of intervals in GRIPP is smaller than that of the labelling scheme in this paper, the number of intervals of GRIPP is larger than that of the intervals after preprocessing. It means that after preprocessing the space overhead of our labelling scheme is smaller than that of GRIPP.

Figure 11 Test queries.

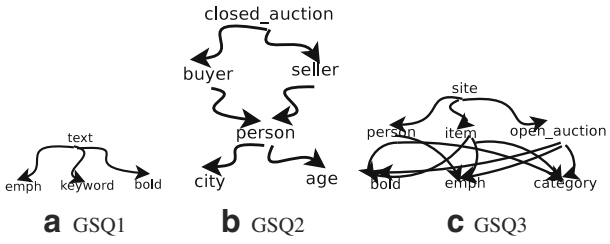


Table 2 The size of code.

Doc Size	Intervals	IAP	IPN	IPNJ	GRINT	GRID
11.3M	252,284	173,702	1.44	0.990	200,105	169,357
22.8M	503,112	346,014	1.43	0.985	401,061	339,191
34.0M	746,234	516,546	1.40	0.985	598,154	505,992
45.3M	999,784	687,596	1.43	0.986	796,338	673,421
56.2M	1,255,877	859,823	1.44	0.988	993,948	840,503

6.3 Efficiency of structural join

6.3.1 Execution time

We show in Figure 12 the execution time of IGMJ, GRIPP-based method and 1-index for Q1 to Q4 on the 20M XMark dataset. Note that Y-axis is in logarithm scale. IGMJ outperforms other two methods. 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. For the same reason, GRIPP-based method accesses the codes for many nodes while IGMJ only accesses the codes of the nodes with the labels in the query.

6.3.2 Scalability experiment

To evaluate the scalability of the algorithms, We ran Q1 on XMark documents with size ranging from 10M to 50M. The result is shown in Figure 13a. It can be seen that both algorithms scale linearly with the increment of the data size.

From the result, 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.

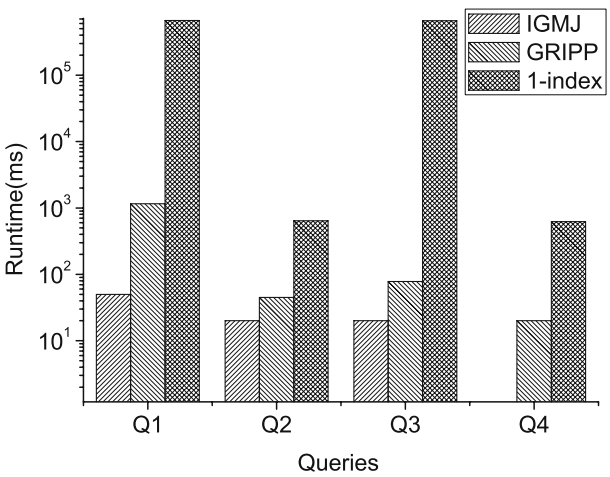
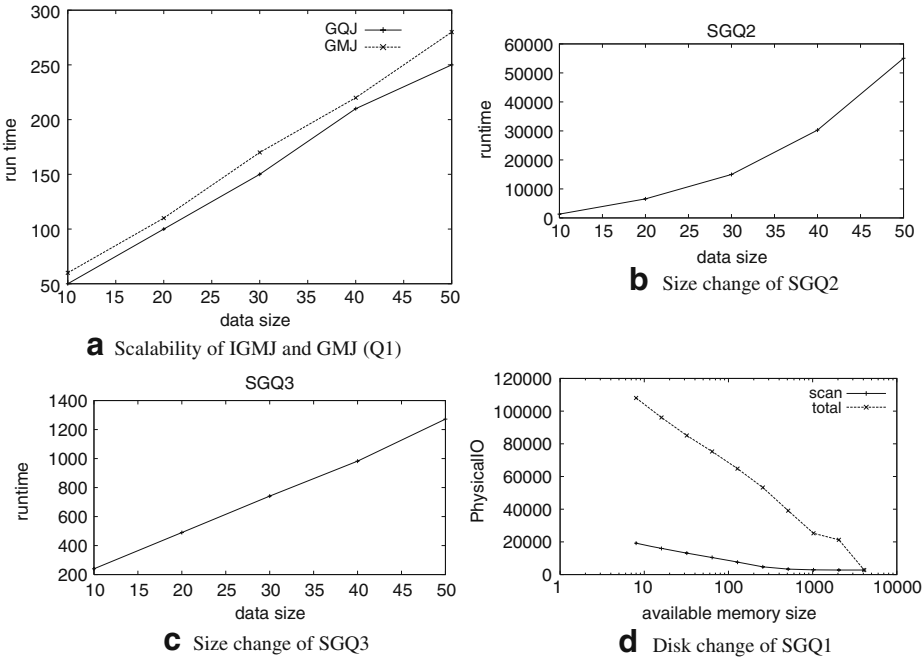


Figure 12 Comparisons of IGMJ.



a Scalability of IGMJ and GMJ (Q1)

b Size change of SGQ2

c Size change of SGQ3

d Disk change of SGQ1

Figure 13 Experiment results of comparisons.

6.4 Efficiency of subgraph join

6.4.1 Changing system parameters

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

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 queries. We fix main memory 8M and block size 4096. The results are shown in Figure 13b and c, respectively. SGQ1 is a simple twig query. The nodes related to SGQ1 in XML document are not in any SCC and all have a single parent. Therefore, the increasing 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 physical I/O change with the block number of SGQ1 is shown in Figure 13d. From the Figure 13d, we can find that without enough main memory, the second phase result in 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 appending.

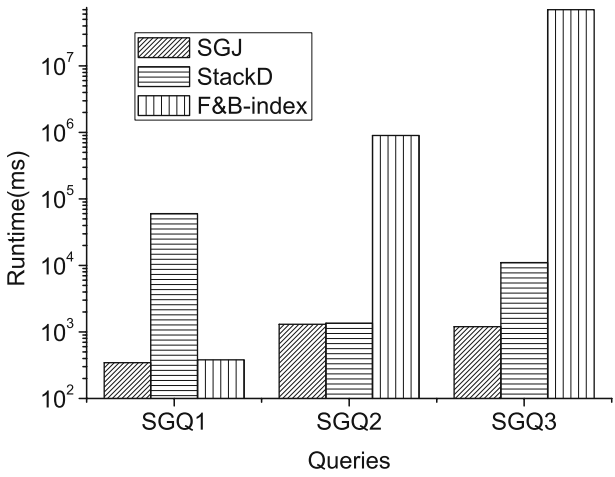


Figure 14 Comparisons for SGJ.

6.4.2 Comparison experiment

We do comparison on 10M XML document with F&B-index 167,072 nodes. We 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. We also compare our algorithms with StackD.

The result of comparisons is shown in Figure 14. *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. From the result, our algorithm outperforms StackD. It is because that for the subgraph queries, StackD requires large intermediate results and when the form of query is a DAG, additional join operations are required.

7 Related work

There are two kinds of work related to the work in this paper. One is reachability labelling scheme. The other is labelling-scheme-based structural query processing techniques.

Many reachability labelling schemes of DAG have been presented, including [9, 11, 17] and [6]. Vassilis Christophides and Plexousakis [23] presents a survey of labelling schemes on DAG and compare the labelling schemes on the coding of semantic web.

A two-hop reachability label are presented in [9]. Ralf Schenkel and Theobald [18] uses two-hop label to process the reachability query in complex XML document collections. An approximate algorithm for the computation of two-hop labelling by finding densest subgraphs is presented in [6]. He et al. [11] presents HLSS labelling. This labelling strategy obtains (*preorder*, *postorder*) for each node and then computes two-hop labelling on remaining edges. Similar as [11], the labelling scheme presented in [24] obtains (*preorder*, *postorder*) for each node at first and

then compute transmit closure matrix for remaining edges. With preorder and postorder, such matrix can be reduced. GRIPP [22] is the latest labelling scheme for reachability relationship. Such labelling scheme converts a graph into a tree with duplication nodes with multiple incoming edges as leaves and using reachability labelling scheme of tree to encode such tree. The labelling scheme and algorithms in this paper are based on the labelling scheme in [17]. It is because such scheme avoids costly set comparison and matrix looking up and is suitable for the computation of (ancestors, descendent) pairs from two node sets. Additionally, such labelling scheme is compatible with adjacent labelling scheme so that it is also suitable to process subgraph queries with both adjacent and reachability relationships. Some XML compression storage strategies are surveyed in [25], the tree structure is also a required feature for these methods. So these strategies cannot be applied on graph-structured the storage of 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 [1, 7, 10, 12, 15, 25, 26, 28]. They are all based on coding schemes that enable efficiently checking of structural relationship of any two nodes in a tree, and thus cannot be applied to the graph-structural XML data directly. There are some path-based indices structures for XML data [27]. These indices are based on tree structure and not suitable for graph-structured XML data.

A few methods have been proposed to process some kinds of subgraph queries on XML data in form some of special kinds of graphs. Chen et al. [5] presents a method, called StackD, to process twig queries on DAG-structured data. It is a modification of a holistic TwigJoin based method. However, StackD focuses on tree-structured twig queries and is not suitable for queries in form of complex graphs. Additionally, when there are many edges in the graph, StackD should maintain a very large data structure. In the case, very huge main memory space is required. It is not practical and becomes inefficient. We choose StackD for comparison. Zografoula Vagena and Moura Moro [29] presents an other method, which is also a modification of the holistic TwigJoin based method, to process twig query on graph-structured data. However, it only works on a kind of special graph, i.e. st-planar graphs [13], but not suitable for other graphs. XQBE [2] is a graphical environment to query XML Data with XQuery. It uses graph mode to represent the query. However, the data of the queries for XQBE is not graph-structured XML data. This method cannot be applied to graph-structured XML data directly. In summary, current methods cannot process general subgraph queries effectively or efficiently.

8 Conclusions

In this paper, we presented 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 designed efficient structural join algorithms for graph-structured XML, GMJ, IGMJ and subgraph join. Our experiments showed that the labeling scheme has acceptable size while the proposed algorithms outperform previous algorithms significantly. As one of our future work, we will design efficient index structure for the labeling scheme to accelerate query processing.

Acknowledgements Supported by the Key Program of the National Natural Science Foundation of China under Grant No. 60533110; the National Grand Fundamental Research 973 Program of China under Grant No. 2006CB303000; the Key Program of the Natural Science Foundation of Heilongjiang Province of China under Grant No. ZJG03-05 ; the Program for New Century Excellent Talents in University of China under Grant No. NCET-05-0333; the Heilongjiang Province Scientific and Technological Special Fund for Young Scholars under Grant No.QC06C033; the National Natural Science Foundation of China under Grant No. 60703012, 60773068, 60773063.

References

1. Al-Khalifa, S., Jagadish, H.V., Patel, J.M., Wu, Y., Koudas, N., Srivastava, D.: Structural joins: a primitive for efficient XML query pattern matching. In: Proceedings of the 18th International Conference on Data Engineering (ICDE 2002), pp. 141–152 (2002)
2. Braga, D., Campi, A.: XQBE: a graphical environment to query xml data. World Wide Web **8**(3), 287–316 (2005)
3. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002), pp. 310–321 (2002)
4. Chamberlin, D.D., Florescu, D., Robie, J.: XQuery: a query language for XML. In: W3C Working Draft. http://www.w3.org/TR/xquery (2001)
5. Chen, L., Gupta, A., Kurul, M.E.: Stack-based algorithms for pattern matching on dags. In: VLDB, pp. 493–504 (2005)
6. Cheng, J., Yu, J.X., Lin, X., Wang, H., Yu, P.S.: Fast computation of reachability labeling for large graphs. In: Ioannidis, Y.E., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) EDBT. Lecture Notes in Computer Science, vol. 3896, pp. 961–979. Springer (2006)
7. Chien, S.-Y., Vagena, Z., Zhang, D., Tsotras, V.J., Zaniolo, C.: Efficient structural joins on indexed XML documents. In: Proceedings of 28th International Conference on Very Large Data Bases (VLDB 2002), pp. 263–274 (2002)
8. Clark, J., DeRose, S.: XML path language (XPath). In W3C recommendation, 16 November 1999. http://www.w3.org/TR/xpath (1999)
9. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. In: SODA, pp. 937–946 (2002)
10. Grust, T.: Accelerating XPath location steps. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002), pp. 109–120. Hong Kong, China August (2002)
11. He, H., Wang, H., Yang, J., Yu, P.S.: Compact reachability labeling for graph-structured data. In: Herzog, O., Schek, H.-J., Fuhr, N., Chowdhury, A., Teiken, W. (eds.) Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management (CIKM2005), Bremen, Germany, October 31–November 5, 2005, pp. 594–601. ACM (2005)
12. Jiang, H., Lu, H., Wang, W., Ooi, B.C.: XR-Tree: indexing XML data for efficient structural join. In: Proceedings of the 19th International Conference on Data Engineering (ICDE 2003), pp. 253–263 (2003)
13. Kameda, T.: On the vector representation of the reachability in planar directed graphs. Information Process Letters **3**(3), 78–80 (1975)
14. Kaushik, R., Bohannon, P., Naughton, J.F., Korth, H.F.: Covering indexes for branching path queries. In: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002), pp. 133–144 (2002)
15. Li, Q., Moon, B.: Indexing and querying XML data for regular path expressions. In: Proceedings of 27th International Conference on Very Large Data Base (VLDB 2001), pp. 361–370 (2001)
16. Milo, T., Suciu, D.: Index structures for path expressions. In: Proceedings of the 7th International Conference on Database Theory (ICDT 1999), pp. 277–295 (1999)
17. Rakesh Agrawal, H.V.J., Borgida A.: 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), pp. 253–262. Portland, Oregon, May (1989)
18. Ralf Schenkel, G.W., Theobald, A.: 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), pp. 237–255, Heraklion, Crete, Greece, March 14–18 (2004)
19. Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: XMark: a benchmark for XML data management. In: Proceedings of 28th International Conference on Very Large Data Bases (VLDB 2002), pp. 974–985 (2002)
20. Cormen, C.L.T., Rivest, R.: Introduction to Algorithms. MIT Press, Cambridge MA (1990)
21. Bray, J.P.T., Sperberg-McQueen, C.M., Yergeau, F.: Extensible markup language (xml) 1.0 (third edition). In: W3C Recommendation 04 February 2004. http://www.w3.org/TR/REC-xml/ (2004)
22. Trißl, S., Leser, U.: Fast and practical indexing and querying of very large graphs. In: SIGMOD Conference, pp. 845–856 (2007)
23. Vassilis Christophides, M.S.S.T., Plexousakis, D.: On labeling schemes for the semantic web. In: Proceedings of the Twelfth International World Wide Web Conference(WWW2003), pp. 544–555. Budapest, Hungary, May (2003)
24. Wang, H., He, H., Yang, J., Yu, P.S., Yu, J.X.: Dual labeling: answering graph reachability queries in constant time. In: Liu, L., Reuter, A., Whang, K.-Y., Zhang, J. (eds.) Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3–8 April 2006, Atlanta, GA, USA, p 75. IEEE Computer Society (2006)
25. Wang, H., Li, J., Wang, H.: Clustered chain path index for xml document: efficiently processing branch queries. World Wide Web **11**(1), 153–168 (2008)
26. Wang, W., Jiang, H., Lu, H., Yu, J.X.: PBiTree coding and efficient processing of containment joins. In: Proceedings of the 19th International Conference on Data Engineering (ICDE 2003), pp. 391–402 (2003)
27. Wong, K.-F., Yu, J.X., Tang, N.: Answering xml queries using path-based indexes: a survey. World Wide Web **9**(3), 277–299 (2006)
28. Zhang, C., Naughton, J.F., DeWitt, D.J., Luo, Q., Lohman, G.M.: On supporting containment queries in relational database management systems. In: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD 2001), pp. 425–436 (2001)
29. Zografoula Vagena, V.J.T., Moura Moro, M.: Twig query processing over graph-structured xml data. In: Proceedings of the Seventh International Workshop on the Web and Databases(WebDB 2004), pp. 43–48 (2004)