Bottom-Up Evaluation of Twig Join Pattern Queries in XML Document Databases

Yangjun Chen

Department of Applied Computer Science
University of Winnipeg
Winnipeg, Manitoba, Canada R3B 2E9
y.chen@uwinnipeg.ca

Abstract. Since the extensible markup language XML emerged as a new standard for information representation and exchange on the Internet, the problem of storing, indexing, and querying XML documents has been among the major issues of database research. In this paper, we study the twig pattern matching and discuss a new algorithm for processing *ordered twig pattern queries*. The time complexity of the algorithm is bounded by $O(|D|\cdot|Q| + |T|\cdot leaf_Q)$ and its space overhead is by $O(leaf_T\cdot leaf_Q)$, where T stands for a document tree, Q for a twig pattern and D is a largest data stream associated with a node q of Q, which contains the database nodes that match the node predicate at q. $leaf_T$ ($leaf_Q$) represents the number of the leaf nodes of T (resp. Q). In addition, the algorithm can be adapted to an indexing environment with *XB*-trees being used.

1 Introduction

The Extensible Markup Language (XML) is an emerging standard for data representation and exchange on the Internet. Tree pattern matching is one of the most important types of XML queries to extract information from XML sources. Normally, an XML document T is represented as a tree structure and typically a query Q specifies patterns of selection predicates on multiple elements that also have some specified tree structured relations. For instance, the XPath expression:

book[**title** = ‘Art of Programming’]//*author*[*fn* = ‘Donald’ and *ln* = ‘Knuth’]

asks for all those *author* elements that (i) have a child subelement *fn* with content ‘Donald’, (ii) have a child subelement *ln* with content ‘Knuth’, and are descendants of *book* elements that have a child *title* subelement with content ‘Art of Programming’. It can be represented as a tree structure. So the query evaluation in XML document databases is essentially a tree matching problem.

We distinguish between two kinds of tree matchings. One is the so-called unordered tree matching, by which the order of siblings is not significant. The other is the ordered tree matching, by which the order of siblings should be taken into account. In the following definitions, $u \rightarrow v$ in Q stands for a child edge (/-edge) for a parent-child relationship; and $u \Rightarrow v$ for a descendant edge (//-edge) for an ancestor-descendant relationship. We also use *label*(v) to represent the name tag (i.e., symbol $\in \sum \cup \{*\}$) or the string associated with v.

Definition 1 (*unordered tree matching*)**.** An embedding of a *twig* (small tree) pattern Q into an XML document T is a mapping f: $Q \rightarrow T$, from the nodes of Q to the nodes of T, which satisfies the following conditions:

(i) Preserve *node label*: For each $u \in Q$, $label(u) = label(f(u))$.

(ii) Preserve *parent-child/ancestor-descendant* relationships: If $u \rightarrow v$ in Q, then $f(v)$ is a child of $f(u)$ in T; if $u \Rightarrow v$ in Q, then $f(v)$ is a descendant of $f(u)$ in T.

If there exists a mapping from Q into T, we say, Q can be imbedded into T, or say, T contains Q. Notice that an embedding could map several nodes with the same tag name in a query to the same node in a database. It also allows a tree mapped to a path, by which the order of siblings is totally unconsidered. This definition is a little bit different from the ordered twig matching defined below.

Definition 2 (*ordered tree matching*)**.** An embedding of a twig pattern Q into an XML document T is a mapping f: $Q \rightarrow T$, from the nodes of Q to the nodes of T, which satisfies the following conditions:

(i) same as (i) in Definition 1.

(ii) same as (ii) in Definition 1.

(iii) Preserve *left-to-right order*: For any two nodes $v_1 \in Q$ and $v_2 \in Q$, if v_1 is to the left of v_2, then $f(v_1)$ is to the left of $f(v_2)$ in T.

v_1 is said to be to the left of v_2 if they are not related by the ancestor-descendant relationship and v_2 follows v_1. This kind of tree mappings is useful in practice. For example, an XML data model was proposed by Catherine and Bird [1] for representing interlinear text for linguistic applications, used to demonstrate various linguistic principles in different languages. For the purpose of linguistic analysis, it is essential to preserve the linear order between the words in a text [1]. In addition to interlinear text, the syntactic structure of textual data should be considered, which breaks a sentence into syntactic units such as noun clauses, verb phrases, adjectives, and so on. These are used by the language TreeBank [2] to provide a hierarchical representation of sentences. Therefore, by the evaluation of a twig pattern query against the TreeBank, the order between siblings should be considered [2, 3].

In 2003, Wang *et al.* [4] proposed a first index-based method, called *ViST*, for handling ordered twig pattern queries, by which the XML data are transformed into structure-encoded sequences and stored in a disk-based virtual *trie* using B^+-trees. One of the problems of this method is that the query processing strategy by straightforward sequence matching may result in false alarms. Another problem, as pointed out in [3], the size of indexes is higher than linear in the total number of elements in an XML document. Such problems are removed by a method, called *PRIX*, discussed in [3]. This method constructs two Prüfer sequences to represent an XML document: a numbered Prüfer sequence and a labeled Prüfer sequence. For all the labeled Prüfer sequences, a virtual trie is constructed, used as an index structure. In this way, the size of indexes is dramatically reduced to $O(|T|)$. But it suffers from very high *CPU* time overhead according to the following analysis. The method consists of a string matching phase and several so-called refinement phases, for which $O(k|Q|\log|Q|)$ time is needed (see page 328 in [3]), where k is the number of subsequences of a labeled Prüfer document sequence, which match Q's labeled Prüfer sequence. However, by the string matching defined in [3], a query pattern string can match non-consecutive segments within a document target string (see Definition 4.1 in [3], page 306). So in the worst case k is in the order of $O(|T|^{|Q|})$ since for each position i (in the target) matching the first element in the pattern string the second element of the pattern can match possibly at $|T| - i - 1$ positions; and for each position j matching the second element in the pattern, the third element in the pattern can possibly match at $|T| - j - 1$ positions, and so on. As an example, consider the following Prüfer string:

A … ab … bc … cd …d

in which each substring containing the same characters is of length $n/4$. Assume that the Prüfer string for a query is *abcd.* Then, there are $O(n^4)$ matching positions. For each of them, a tree embedding will be examined. (We note that if the string matching is restricted to consecutive segments, there is at most one matching for each position, at which the first element in the pattern matches. But it is not the case discussed in [3].)

In this paper, we propose a new method for processing ordered twig pattern queries. The main idea behind it is an algorithm for reconstructing tree structures from data streams as well as a new tree labeling technique for queries to represent *left-to-right relationships.* The new algorithm runs in $O(|D|\cdot|Q| + |T|\cdot leaf_Q)$ time and $O(leaf_T\cdot leaf_Q)$ space, where $leaf_T$ ($leaf_Q$) represents the number of the leaf nodes of T (resp. Q), and D is a largest data stream associated with a node q of Q, which contains the database nodes that match the node predicate at q.

The remainder of the paper is organized as follows. In Section 2, we restate the tree encoding [5], which can be used to facilitate the recognition of different relationships among the nodes of trees. In Section 3, we discuss our algorithm for evaluating ordered twig pattern queries. The paper concludes in Section 4.

2 Tree Labeling

In [5], an interesting tree encoding method was discussed, which can be used to identify different relationships among the nodes of a tree. Let T be a document tree. We associate each node v in T with a quadruple $\alpha(v) = (d, l, r, ln)$, where d is the document identifier (*DocId*), l = *LeftPos*, r = *RightPos*, and ln = *LevelNum*. Here, *LeftPos* and *RightPos* are generated by counting word numbers from the beginning of the document until the start and end of the element, respectively. By using such a data structure, the structural relationship between the nodes in an XML database can be simply determined [5]:

(i) *ancestor-descendant*: a node v_1 associated with (d_1, l_1, r_1, ln_1) is an ancestor of another node v_2 with (d_2, l_2, r_2, ln_2) iff $d_1 = d_2$, $l_1 < l_2$, and $r_1 > r_2$.
(ii) *parent-child*: a node v_1 associated with (d_1, l_1, r_1, ln_1) is the parent of another node v_2 with (d_2, l_2, r_2, ln_2) iff $d_1 = d_2$, $l_1 < l_2$, $r_1 > r_2$, and $ln_2 = ln_1 + 1$.
(iii) *from left to right*: a node v_1 associated with (d_1, l_1, r_1, ln_1) is to the left of another node v_2 with (d_2, l_2, r_2, ln_2) iff $d_1 = d_2$, $r_1 < l_2$.

(See Fig. 3(a) for illustration.) In the rest of the paper, if for two quadruples $\alpha_1 = (d_1, l_1, r_1, ln_1)$ and $\alpha_2 = (d_2, l_2, r_2, ln_2)$, we have $d_1 = d_2$, $l_1 < l_2$, and $r_1 > r_2$, we say that α_2 is subsumed by α_1. For convenience, a quadruple is considered to be subsumed by itself. If no confusion is caused, we will use v and $\alpha(v)$ interchangeably. We can also assign LeftPos and RightPos values to the query nodes in Q for the same purpose as above. Finally we use $T[v]$ to represent a subtree rooted at v in T.

3 Main Algorithm

In this section, we describe our method. First, we discuss a kind of data stream transformation in 3.1, which provides the input to our main procedure. Then, in 3.2, the main algorithm is described in great detail.

3.1 Data Stream Transformation

As with *TwigStack* [4], each node q in a twig pattern (or say, a query tree) Q is associated with a data stream $B(q)$, which contains the positional representations (quadruples) of the database nodes v that match q (i.e., $label(v) = label(q)$). All the quadruples in a data stream are sorted by their (DocID, LeftPos) values. Therefore, iterating through the stream nodes in sorted order of their LeftPos values corresponds to access of document nodes in preorder. However, our algorithm needs to visit them in *postorder* (i.e., in sorted order of their RightPos values). For this reason, we maintain a global stack ST to make a transformation of data streams using the following algorithm. In ST, each entry is a pair (q, v) with $q \in Q$ and $v \in T$ (v is represented by its quadruple.)

```
Algorithm. stream-transformation($B(q_i)$'s)
input: all data streams $B(q_i)$'s, each sorted by LeftPos.
output: new data streams $L(q_i)$'s, each sorted by RightPos.
begin
1. repeat until each $B(q_i)$ becomes empty
2.    { identify $q_i$ such that the first element $v$ of $B(q_i)$ is of the minimal LeftPos
        value; remove $v$ from $B(q_i)$;
3.      while $ST$ is not empty and $ST.top$ is not $v$'s ancestor do
4.      { $x \leftarrow ST.pop()$; Let $x = (q_j, u)$;
5.        put $u$ at the end of $L(q_j)$; }
7.      $ST.push(q_i, v)$;
8.    }
end
```

In the above algorithm, ST is used to keep all the nodes on a path until we meet a node v that is not a descendant of $ST.top$. Then, we pop up all those nodes that are not v's ancestor; put them at the end of the corresponding $L(q_i)$'s (see lines 3 - 4); and push v into ST (see line 7.) The output of the algorithm is a set of data streams $L(q_i)$'s with each being sorted by RightPos values. However, we remark that the popped nodes are in postorder. So we can directly handle the nodes in this order without explicitly generating $L(q_i)$'s. But for ease of explanation, we assume that all $L(q_i)$'s are completely generated in the following discussion. We also note that the data streams associated with different nodes in Q may be the same. So we use $\boldsymbol{q}$ to represent the set of such query nodes and denote by $L(\boldsymbol{q})$ ($B(\boldsymbol{q})$) the data stream shared by them. Without loss of generality, assume that the query nodes in $\boldsymbol{q}$ are sorted by their RightPos values. We will also use $L(Q) = \{L(\boldsymbol{q}_1), ..., L(\boldsymbol{q}_l)\}$ to represent all the data streams with respect to Q, where each $\boldsymbol{q}_i$ (i = 1, ..., l) is a set of sorted query nodes that share a common data stream.

3.2 Main Procedure

First of all, we notice that iterating through $L(\boldsymbol{q}_1), ..., L(\boldsymbol{q}_l)$, i.e., the data streams sorted in increasing RightPos values, we navigate T in postorder. So, our algorithm works bottom-up. For the purpose of checking ordered tree embedding, we will first search Q in the breadth-first fashion, generating a number (called a *breadth-first number*) for each node q in Q, denoted as $bf(q)$, which can be used to represent the left-to-right order of siblings in a simple way (See Fig. 1(a) for illustration). Then, we use *interval*(q) to represent an interval covering all the breadth-first numbers of q's children. For example, for Q shown in Fig. 1(a), we have *interval*(q_1) = [2, 3] and *interval*(q_2) = [4, 5]. In the following, we will use q and $bf(q)$ interchangeably.

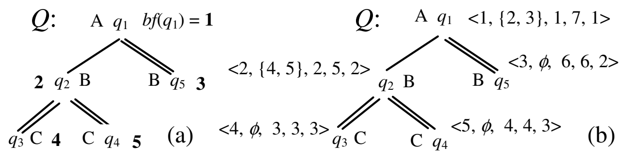

Fig. 1. Illustration for $L(q_i)$'s

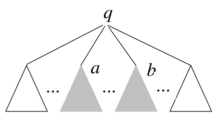

Fig. 2. Subtrees in Q

Next, we associate each q with a tuple g(q) = <$bf(q)$, *interval*(q), LeftPos(q), RightPos(q), LevelNum(q)>, as shown in Fig. 1(b). We say, a q is subsumed by a pair (L, R) if $L \leq$ LeftPos(q) and $R \geq$ RightPos(q). When checking the tree embedding of Q in T, we will associate each generated node v in T with a linked list A_v to record what subtrees in Q can be embedded in $T[v]$. Each entry in A_v is a quadruple e = (q, *interval*, L, R), where q is a node in Q, *interval* = [a, b] $\subseteq$ *interval*(q) (for some $a \leq b$), L = LeftPos(a) and R = RightPos(b). Here, we use a and b to refer to the nodes with the breadth-first numbers a and b, respectively. Therefore, such a quadruple represents a set of subtrees (in $Q[q]$) rooted respectively at a, a + 1, ..., b (i.e., a set of subtrees rooted at a set of consecutive breadth-first numbers.) See Fig. 2 for illustration. In addition, the following two conditions are satisfied:

i) For any two entries e_1 and e_2 in A_v, $e_1.q$ is not subsumed by ($e_2.L$, $e_2.R$), nor is $e_2.q$ subsumed by ($e_1.L$, $e_1.R$). In addition, if $e_1.q = e_2.q$, $e_1.interval \not\subset e_2.interval$ and $e_2.interval \not\subset e_2.interval$.
ii) For any two entries e_1 and e_2 in A_v with $e_1.interval$ = [a, b] and $e_2.interval$ = [a', b'], if e_1 appears before e_2, then RightPost($e_1.q$) < RightPost($e_2.q$) or RightPost($e_1.q$) = RightPost($e_2.q$) but $a < a'$. Condition (i) is used to avoid redundancy due to the following lemma.

Lemma 1. Let q be a node in Q. Let [a, b] be an interval. If q is subsumed by (LeftPos(a), RightPos(b)), then there exists an integer $0 \leq i \leq b - a$ such that $bf(q)$ is equal to $a + i$ or q is an descendant of $a + i$.

Then, by imposing condition (i), A_v keeps only quadruples which represent pairwise non-covered subtrees. Condition (ii) is met if the nodes in Q are checked along their increasing RightPos values. It is because in such an order the parents of the checked nodes must be non-decreasingly sorted by their RightPos values. Since we explore Q bottom-up, condition (ii) is always satisfied.

See Fig. 3 for a better understanding.

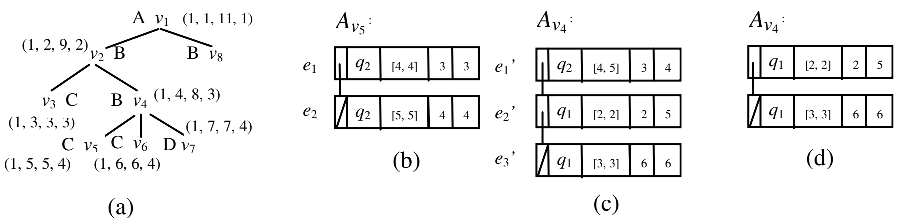

Fig. 3. Illustration for linked lists

In Fig. 3(a), we show a document tree T. Fig. 3(b) shows the linked list created for v_5 in T when it is generated and checked against q_3 and q_4 in Q shown in Fig. 1(a). Since both q_3 and q_4 are leaf nodes, $T[v_5]$ is able to embed either $Q[q_3]$ or $Q[q_4]$ and so we have two entries e_1 and e_2 in A_{v_5}. Note that $bf(q_3) = 4$ and $bf(q_3) = 5$. So we set their *intervals* to [4, 4] and [5, 5], respectively. In addition, each of them is a child of q_2. Thus, we have $e_1.q = e_2.q = q_2$. In Fig. 3(c), we show the linked list for v_4. It contains three entries e_1', e_2' and e_3'. Special attention should be paid to e_1'. Its *interval* is [4, 5], showing that $T[v_4]$ is able to embed both $Q[q_3]$ and $Q[q_4]$. In this case, $e_1'.L$ is set to 3 and $e_1'.R$ to 4. However, since $e_1'.q = q_2$ is subsumed by $(e_2'.L, e_2'.R) = (2, 5)$, the entry will be removed, and the linked list is reduced to a data structure shown in Fig. 3(d). With the linked lists associated with the nodes in T, the embedding of a subtree $Q[q]$ in $T[v]$ can be checked very efficiently. First, we define a simple operation over two intervals $[a, b]$ and $[a', b']$, which share the same parent:

$$[a, b]\ \Delta\ [a', b'] = \begin{cases} [a, b'] & \text{if } a \le a' \le b+1, b < b' \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

For example, in A_{v_5}, we have an entry (q_2, [4, 4], 3, 3). In A_{v_6} (which is exactly the same as A_{v_5}), we have an entry (q_2, [5, 5], 4, 4). We can merge these two entries to form another entry (q_2, [4, 5], 3, 4), which can be used to facilitate checking whether $T[v_4]$ embeds $Q[q_2]$.

The general process to merge two linked list is described below.

1. Let A_1 and A_2 be two linked list associated with the first two child nodes of a node v in T, which is being checked against q with $label(v) = label(q)$.
2. Scan both A_1 and A_2 from the beginning to the end. Let e_1 (from A_1) and e_2 (from A_2) be the entries encountered. We will perform the following checkings.
   – If RightPos($e_2.q$) > RightPos($e_1.q$), $e_1 \leftarrow next(e_1)$.
   – If RightPos($e_2.q$) < RightPos($e_1.q$), then $e_2' \leftarrow e_2$; insert e_2' into A_1 just before e_1; $e_2 \leftarrow next(e_2)$.

– If RightPos($e_2.q$) = RightPos($e_1.q$), then we will compare the intervals in e_1 and e_2. Let $e_1.interval = [a, b]$. Let $e_2.interval = [a', b']$.
If $a' > b + 1$, then $e_1 \leftarrow next(e_1)$.
If $a \leq a' \leq b + 1$ and $b < b'$, then replace $e_1.interval$ with $[a, b] \Delta [a', b']$ in A_1; e_1.RightPost ← RightPos(b'); $e_1 \leftarrow next(e_1)$; $e_2 \leftarrow next(e_2)$.
If $[a', b'] \subseteq [a, b]$, then $e_2 \leftarrow next(e_2)$.
If $a' < a$, then $e_2' \leftarrow e_2$; insert e_2' into A_1 just before e_1; $e_2 \leftarrow next(e_2)$.

3. If A_1 is exhausted, all the remaining entries in A_2 will be appended to the end of A_1. The result of this process is stored in A_1, denoted as $merge(A_1, A_2)$. We also define

$$merge(A_1, ..., A_k) = merge(merge(A_1, ..., A_{k-1}), A_k),$$

where $A_1, ..., A_k$ are the linked lists associated with v's child nodes: $v_1, ..., v_k$, respectively. If in $merge(A_1, ..., A_k)$ there exists an e such that $e.interval = interval(q)$, $T[v]$ embeds $Q[q]$.

For the merging operation described above, we require that the entries in a linked list are sorted. That is, all the entries e are in the order of increasing RightPos($e.q$) values; and for those entries with the same RightPos($e.q$) value their intervals are 'from-left-to-right' ordered. Such an order is obtained by searching Q bottom-up (or say, in the order of increasing RightPos values) when checking a node v in T against the nodes in Q. Thus, no extra effort is needed to get a sorted linked list. Moreover, if the input linked lists are sorted, the output linked lists must also be sorted.

The above merging operation can be used only for the case that Q contains no /-edges. In the presence of both /-edges and //-edges, the linked lists should be slightly modified as follows.

i) Let q_j be a /-child of q with $bf(q_j) = a$. Let A_i be a linked list associated with v_i (a child of v) which contains an entry e with $e.interval = [c, d]$ such that $c \leq a$ and $a \leq d$.
ii) If $label(q_j) = label(v_i)$ and v_i is a /-child of v, e needn't be changed. Otherwise, e will be replaced with two entries:

– ($e.q$, $[c, a - 1]$, LeftPos(c), LeftPos(a - 1)), and
– ($e.q$, $[a + 1, d]$, LeftPos(a + 1), LeftPos(d)).

In terms of the above discussion, we give our algorithm for evaluating ordered twig pattern queries. In the process, we will generate *left-sibling* links from the current node v to the node u generated just before v if u is not a child (descendant) of v. However, in the following description, we focus on the checking of tree embedding and that part of technical details is omitted.

Algorithm. *tree-embedding*($L(Q)$)
Input: all data streams $L(Q)$.
Output: S_v's, with each containing those query nodes q such that $T[v]$ contains $Q[q]$.

```
begin
1. repeat until each $L(\mathbf{q})$ in $L(Q)$ become empty
2. {identify $\mathbf{q}$ such that the first element $v$ of $L(\mathbf{q})$ is of the minimal RightPos value; remove $v$
     from $L(\mathbf{q})$;
3.   generate node $v$; $A_v \leftarrow$ f;
4.   let $v_1, ..., v_k$ be the children of $v$.
5. $B \leftarrow merge(A_{v_1}, ..., A_{v_k})$;
6. for each $q \in \boldsymbol{q}$ do { (*nodes in $\boldsymbol{q}$ are sorted.*)
7.    if $q$ is a leaf then {$S_v \leftarrow S_v \cup \{q\}$;}
8.    else (*$q$ is an internal node.*)
9.    {if there exists $e$ in $B$ such that $e.interval = interval(q)$
10.      then $S_v \leftarrow S_v \cup \{q\}$;}
11. }
12. for each $q \in S_v$ do {
13.   append ($q$'s parent, [$bf(q)$, $bf(q)$], $q$.LeftPos, $q$.RightPos to the end of $A_v$;}
14.   $A_v \leftarrow merge(A_v, B)$; Scan $A_v$ to remove subsumed entries;
15.   remove all $A_{v_j}$ 's;}
16. }
end
```

In Algorithm *tree-embedding*( ), the nodes in T is created one by one. For each node v generated for an element from a $L(\boldsymbol{q})$, we will first merge all the linked lists of their children and store the output in a temporary variable B (see line 5). Then, for each $q \in \boldsymbol{q}$, we will check whether there exists an entry e such that $e.interval = interval(q)$ (see lines 8 - 9). If it is the case, we will construct an entry for q and append it to the end of the linked list A_v (see lines 12 - 13). The final linked list for v is established by executing line 14. Afterwards, all the A_{v_j} 's (for v's children) will be removed since they will not be used any more (see line 15).

Proposition 2. Algorithm *tree-embedding*( ) computes the entries in A_v's correctly.

4 Conclusion

In this paper, we have discussed a new method to handle the ordered tree matching in XML document databases. The main idea is the concept of intervals, which enables us to efficiently check from-left-to right ordering. The time complexity of the algorithm is bounded by $O(|D|\cdot|Q| + |T|\cdot leaf_Q)$ and its space overhead is by $O(leaf_T\cdot leaf_Q)$, where T stands for a document tree, Q for a twig pattern and D is a largest data stream associated with a node q of Q, which contains the database nodes that match the node predicate at q. $leaf_T$ ($leaf_Q$) represents the number of the leaf nodes of T (resp. Q).

References

[1] Catherine, B., Bird, S.: Towards a general model of Interlinear text. In: Proc. of EMELD Workshop, Lansing, MI (2003)

[2] Müller, K.: Semi-automatic construction of a question tree bank. In: Proc. of the 4th Intl. Conf. on Language Resources and Evaluation, Lisbon, Portual (2004)

[3] Rao, P., Moon, B.: Sequencing XML Data and Query Twigs for Fast Pat-tern Matching. ACM Transaction on Data base Systems 31(1), 299–345 (2006)

[4] Wang, H., Meng, X.: On the Sequencing of Tree Structures for XML Indexing. In: Proc. Conf., Data Engineering, Tokyo, Japan, April 2005, pp. 372–385 (2005)

[5] Zhang, C., Naughton, J., Dewitt, D., Luo, Q., Lohman, G.: on Supporting containment queries in relational database management systems. In: Proc. of ACM SIGMOD (2001)