Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Information Processing Letters 95 (2005) 480–486

Information Processing Letters

www.elsevier.com/locate/ipl

A path-based node filtering method for efficient structural joins [☆,☆☆]

Kyung-Sub Min [a,*], Hyoung-Joo Kim [b]

[a] *Interdisciplinary Program in Cognitive Science, Seoul National University, Seoul, Republic of Korea*
[b] *Department of Computer Science and Engineering, Seoul National University, Seoul, Republic of Korea*

Received 11 September 2004; received in revised form 14 May 2005; accepted 18 May 2005

Available online 1 July 2005

Communicated by F.Y.L. Chin

Keywords: Databases; XML; Structural join; Node filtering; Path structure summary

1. Introduction

Evaluating structure (path) predicates on elements is the core operation of XML path query processing. Structural join algorithms and path indexes are the methods to handle the predicates.

Basic structural join algorithm [1,9,11] accesses all the nodes corresponding to two input elements, not considering structural predicates of the elements. Thus, when processing a path query, a query processor using the join algorithm should perform join operations on all the elements in the query and access all the nodes in the elements. Some proprietary XML indexes [4,7] and a join algorithm [2] are proposed to reduce the access cost. However, all these algorithms still access all the nodes located in the paths that do not match to the structural predicates on the elements in a query.

For example, in order to retrieve the names of professors in a department we can express a query like "//department/professor/name". In this query, the professor element has a structural predicate, "//department/". Thus, it is desirable to scan only the child *name* nodes in *department*s. However, even when the professor element is located in the paths, "/department/course/professor" or "/department/publications/journal/professor", existing structural join algorithms scan all the nodes in these paths. Thus, large numbers of useless intermediate result pairs can be produced and in turn unnecessary high I/O and CPU cost can incur.

Meanwhile, path indexes [5,6,8,10] can efficiently process the structural predicates represented in the form of single paths or some pre-defined tree-shaped

☆ This work was supported in part by the Ministry of Information & Communications, Korea, under the Information Technology Research Center (ITRC) Support Program.

☆☆ This work was supported in part by the Brain Korea 21 Project.

* Corresponding author.

E-mail addresses: ksmin@oopsla.snu.ac.kr (K.-S. Min), hjk@oopsla.snu.ac.kr (H.-J. Kim).

0020-0190/$ – see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.ipl.2005.05.017

paths. However, it is very difficult to evaluate arbitrary path queries efficiently, because these indexes do not represent all the single and tree-shaped paths. Therefore, a query processor using path indexes first should find the elements needed to be joined (we call them join elements) and their structural predicates, and then perform a sequence of join operations on the node sets corresponding to the found join elements. Here, which algorithm is used to find join elements determines the performance of this type of query processing algorithms. BLAS [3] is the state-of-the-art among these algorithms. However, it still has room for reducing the number of join elements.

In this paper, we propose a filtering algorithm, P-Filter, which efficiently evaluates the structural predicates on join elements, and consequently filters out the nodes not matched to the predicates, before performing a structural join operation. To support this algorithm, we encode all the nodes in an XML document with their path information and also encode all the paths in the XML document (we call this UPET). Also, we propose a new query processing algorithm, P-QEval, using P-Filter and a query transformation algorithm to search for the minimal join elements.

Our experimental results show that P-QEval can efficiently process path queries compared to an existing structural join-based query processing algorithm or the state-of-the-art path index-based query processing algorithm, by using the P-Filter algorithm and the query transformation algorithm.

The rest of this paper is as follows. Section 2 shows some related work on evaluating path queries. We present a new encoding scheme for nodes and path structures in Section 3. In Section 4, we describe P-Filter algorithm and P-QEval algorithm. Experimental results are shown in Section 5. Finally, we conclude this paper in Section 6.

2. Related work

Several path indexes and structural join algorithms have been proposed to evaluate XML path queries. [5, 6,8,10] are the representative path indexes. These indexes can efficiently retrieve the nodes matched to a single path or predefined specific tree paths. Stack-Tree [1] is an optimal structural join algorithm that scans the nodes in two input sets only once. Meanwhile, Holistic Twig Join [2] eliminates large numbers of intermediate result pairs, which will not contained in the final result set, during processing a path query. [4,7] presented proprietary XML node indexes to skip the nodes that need not to participate in a structural join. As another query processing algorithm, BLAS [3] efficiently processes path queries, by encoding path information of nodes and transforming a user's query into a set of sub-queries by means of split, push-up, and unfold algorithms.

3. Encoding scheme

In this section, we describe a new encoding scheme for nodes and paths in an XML document. We assume that an XML document is an ordered tree T. Each node in T holds a region, $\langle start, end\rangle$, which is assigned by visiting each node in depth-first order. Fig. 1 shows an XML data tree.

To encode the nodes and the paths of T, we make a tree, *LPTree*, which holds label paths in T and their element node extents in the tree nodes. Fig. 2 shows the *LPTree* of Fig. 1. For example, *lp16* node represents the label path '/deptList/dept/students/gradstudent' and the node extent $\{\ldots, (580, 599), (600, 619)\}$. Also, we allocate a path identifier, *pid*, for each node in the LPTree, which is represented by a region $\langle start, end\rangle$. For example, in Fig. 2, $(4, 15)$ is the *pid* of the label path of *lp4*.

Using the LPTree, we construct data node sets in the form of $\langle start, end, level, pid.start\rangle$ for data nodes in the extents of the tree. All encoded data nodes are

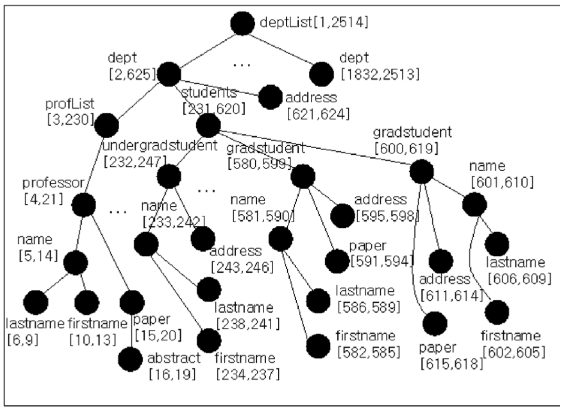
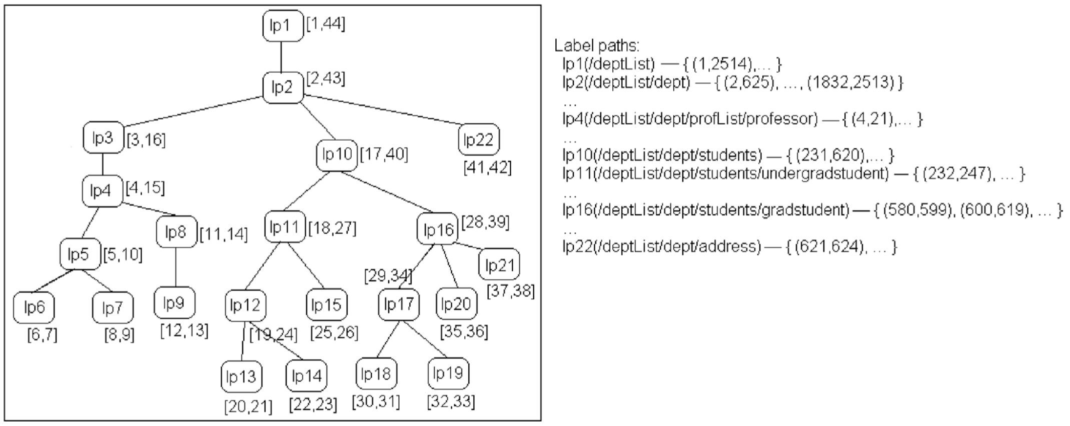


Fig. 1. A part of a data tree for deptList.dtd.



Label paths:
lp1(/deptList) — { (1,2514),... }
lp2(/deptList/dept) — { (2,625), ..., (1832,2513) }
...
lp4(/deptList/dept/profList/professor) — { (4,21),... }
...
lp10(/deptList/dept/students) — { (231,620),... }
lp11(/deptList/dept/students/undergradstudent) — { (232,247), ... }
...
lp16(/deptList/dept/students/gradstudent) — { (580,599), (600,619), ... }
...
lp22(/deptList/dept/address) — { (621,624), ... }

Fig. 2. LPTree for Fig. 1.

deptList	$\{\langle 2, 43, 0\rangle, \langle 3, 16, 0\rangle, \ldots\}$
deptList#	$\{\langle 1, 44, 0\rangle\}$
...	...
professor	$\{\langle 5, 10, 3\rangle, \langle 6, 7, 3\rangle, \langle 8, 9, 3\rangle, \langle 11, 14, 3\rangle, \langle 12, 13, 3\rangle\}$
professor#	$\{\langle 4, 15, 3\rangle\}$
paper	$\{\langle 12, 13, 4\rangle\}$
paper#	$\{\langle 11, 14, 4\rangle, \langle 35, 36, 4\rangle\}$
abstract#	$\{\langle 12, 13, 5\rangle\}$
name	$\{\langle 6, 7, 4\rangle, \ldots, \langle 30, 31, 4\rangle, \langle 32, 33, 4\rangle\}$
namef	$\{\langle 5, 10, 4\rangle, \langle 19, 24, 4\rangle, \langle 29, 34, 4\rangle\}$
firstname#	$\{\langle 8, 9, 5\rangle, \langle 22, 23, 5\rangle, \langle 32, 33, 5\rangle\}$
lastname#	$\{\langle 6, 7, 5\rangle, \langle 20, 21, 5\rangle, \langle 30, 31, 5\rangle\}$
...	...

Fig. 3. UPET for Fig. 2.

stored in [*pid*.*start*, *start*] order. Also, we make a label node set, UPET, in the form of [*label_name*(with # or not), $\{\langle pid, level\rangle\}$] by using label paths. Here, *label_name* means an element name that appears in a label path. Especially, a *label_name* with # means that the element name appears in the last level of a label path. And level is the position where a *label_name* appears in the label path. Fig. 3 shows the part of UPET for Fig. 2. For example, the *label_name* professor appears at five label paths. All the $\langle pid, level\rangle$ pairs in UPET are stored in the [*pid.start*, *level*] order. After encoding all the nodes and paths, the LPTree is eliminated.

4. P-Filter

In this section, we describe the filtering algorithm, P-Filter, which filters out the nodes not matched to the structural predicates of the elements to be joined. Also, we describe a new query processing algorithm, P-QEval, using P-Filter and a query transformation algorithm.

P-Filter algorithm is composed of two steps. First, it finds the *pid*s that match to the structure predicate, *s_pred*, on each join element E. (Algorithm FilterElmPIDs.) To find them, it first retrieves the *pid*s corresponding to each label in *s_pred*. Then, for the *pid* sets, $p_1, p_2, \ldots, p_k$, it sequentially performs a semi-join like operation that extracts descendant *pid*s which satisfy $p_i.start < p_j.start$ and $p_i.end > p_j.end$ $(1 \leqslant i \leqslant k-1, 2 \leqslant j \leqslant k, i < j-1)$.

For example, when E is the *name* element and *s_pred* of E is "//professor//name", FilterElmPIDs extracts $\{(5, 10, 3), (6, 7, 3), (8, 9, 3), (11, 14, 3), (12, 13, 3)\}$ for *profess* label and $\{(5, 10, 4), (19, 24, 4), (29, 34, 4)\}$ for *name#* label. Then, it performs one semijoin-like operation between the two sets. As a result, it produces $\{(5, 10, 4)\}$ as the *pid* set of *name* element. Thus, all the nodes in $\{(19, 24, 4), (29, 34, 4)\}$ are filtered out.

For all the join elements, P-Filter performs the above algorithm. As a result, it obtains the *pid* sets for the elements. Next, it additionally filters out some *pid*s that do not satisfy path containment relationship (we call this PCR) between the filtered *pid* sets. (Algorithm FilterPCRPIDs.) In this step, all the *pid*s that do not satisfy PCR with any *pid* sets are eliminated.

As a simple example, assume that two elements *paper* (E1) and *abstract* (E2) should be joined, and produce result pairs in the form of $\langle paper, abstract\rangle$. And the filtered *pid*s of *paper* label are $\{(11, 14, 4), (35, 36, 4)\}$, and those of *abstract#* label are $\{(12, 13, 5)\}$. As a result of checking PCR between the *pid* sets, it finds that the *pid* $(35, 36, 4)$ of E1 has no match with *pid*s of E2 and filters out the *pid* from E1.

Based on this P-Filter algorithm, we present a new query processing algorithm. (Algorithm P-QEval.) To process a query, P-QEval first transforms the query into ⟨a join element, its structural predicate⟩ pairs. Here, a join element means an element that is located in significant positions (branch, terminal, or retrieval) in a query. P-QEval finds join elements, while visiting query nodes in depth-first order. Next, it filters out the nodes that need not to participate in join operations by using P-Filter. Then, it performs a structural join algorithm like holistic twig join [2] on the join elements.

By using this algorithm, we can reduce the number of joins because the algorithm does not perform joins on the elements not in significant position in a query. Also, we can significantly reduce CPU and I/O cost because the algorithm filters out the nodes in the *pid*s that need not participate in join operation.

5. Experimental results

In this section, we present preliminary results of the experiments that were performed to verify the effectiveness of the proposed filtering algorithm, P-Filter, and our query processing algorithm, P-QEval.

To show the effectiveness of P-Filter and P-QEval, we compared its performance with the state-of-the-art XML query processing algorithms, Holistic Twig Join [2] and BLAS [3]. The query processing algorithms were implemented on Windows 2000 with 256 MB memory and 40 GB hard disk using C++. All experimental data sets were syntactically generated using the DTD shown in Fig. 4. For these data sets, two types of path queries, linear and branch (twig), were chosen. For each type, we prepared 3 queries, each of which holds only parent-child (we call it P/C) relationships, one ancestor-descendant (we call it A/D) relationship, or two or more A/D relationships. Table 1 shows the prepared queries.

Fig. 5 shows the performance results on the query processing algorithms, HTJ, BLAS-PU, and P-QEval. In the figure, HTJ represents Holistic Twig Join algorithm, BLAS-PU represents BLAS using push-up

```
Algorithm: P-Filter(E)

E: the set of join elements.
[steps]
1.  for each element e in E do
2.     e.pidset := FilterElmPIDs(e. predicate);
3.  FilterPCRPIDs(E);
```

```
Algorithm: FilterElrnPIDs (p)

p: a structure predicate in the form of a linear path expression,
   e.g., //dept//professor//name for a join element name.
[steps]
1.  L := all label names in p.
2.  R := { }
3.  F := the entries in UPET that match to the first label name in L.
4.  for ( each label e in L ) do
5.     S := the entries in UPET that match to e.
6.     CurF := the cursor for entries of F.
7.     CurS := the cursor for entries of S.
8.     while ( CurF != NULL 88 CurS != NULL ) do
9.       if ( CurF. start < CurS. start ) then CurF++;
10.      else if ( CurF. start > CurS. start) then CurS++;
11.      else
12.           if ( CurF. level < CurS. level ) then append CurS to R.
13.           CurS++;
14.      end if.
15.    end while.
16  F := R;
17. end for.
```

```
Algorithm: FilterPCRPIDs(E)

E: the set of join elements.
[steps]
1.  while ( a join element in E is not joined ) do
2.     select two join elements $e_a$ (ancestor) and $e_d$ (descendant)
          in bottom-up fashion.
          [ $e_d$, can be a join element that was filtered by joining with
            its descendant join element or a leaf element that is not
            joined with any element. ]
3.     for ( each entry, x in e, and y in $e_d$) do
4.       if ( x contains y ) then link x with y and y.linkcount++;
5.       if ( x does not contain any entry in $e_d$ ) then
6.          if ( x joined with other descendant element, $e_0$, and
               has a link with an entry, z, of $e_0$ ) then
7.            z.linkcount- -;
8.            if ( z.linkcount= = 0 ) then
9.              recursively perform 6, 7 on descendant entries of z.
10.           end if.
11.         remove x from e.
12.      end if.
13.    end for.
14. end while.
```

Algorithm 1. P-Filter.

```
Algorithm: P-QEval(Q)

Q: input query
E: the set of join elements
e: a join element
e.name: element name,
e.predicate: structural predicate on e,
e.type: branch(B), terminal(T), or retrieval(R),
e.joinelm: the ancestor (parent) element that is to be joined with e,
e.joincond: the join condition (parent–child, ancestor-descendant),
e.pidset (filtered) pid list of e.

[steps]
1.  while (not visited all the query nodes in Q) do
      // query transformation
2.  e := the next query node (in depth-first order)
3.  if ( e is branch, terminal, or retrieval element) then
4.    set e.name, e.predicate, e.type.
5.    if ( an ancestor join element exists ) then
6.      e.joinelm := a;
7.      e.joincond = containment relationship between a and e,
8.    end if,
9.    append e into E,
10. end if,
11. end while,
12. P-Filter(E);        // filter out pids for join elements
13. Perform a structural join algorithm for the elements in E.
```

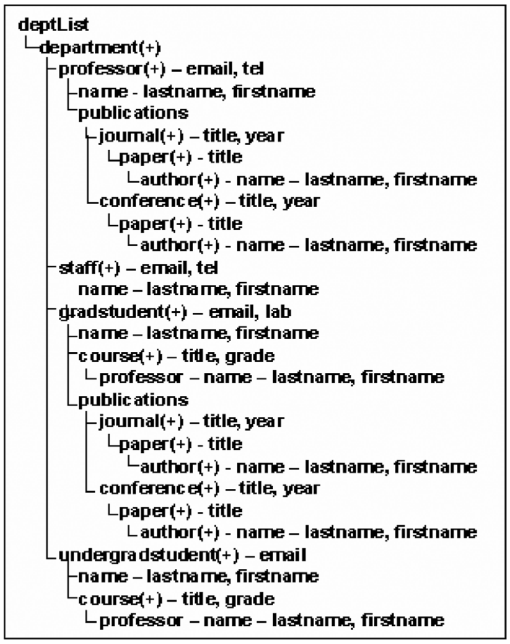
Algorithm 2. P-QEval.

Table 1
Query set

Q1	/deptList/department/professor/name (SPE: only P/C rel.)
Q2	//dept List/department//professor/name (SPE: two A/D rel.)
Q3	//deptList//department//professor//name (SPE: four A/D rel.)
Q4	//paper[title]/author/name (TPE: only P/C rel.)
Q5	//paper[title]/author//name (TPE: two A/D rel.)
Q6	//paper[//title]//author//name (TPE: four A/D rel.)

algorithm, and P-QEval represents a query processing algorithm using P-Filter and HTJ (step 13 in Algorithm 2).

Fig. 5(a) shows the performance of these algorithms when the data size is fixed to 60.3 Mbytes. As shown the figure, HTJ shows the worst performance. Meanwhile, the performance of BLAS-PU is very similar to that of P-QEval when all the elements in a query hold only P/C relationships. But, as the number of elements with A/D relationship increases, the performance of P-QEval is much better than that of BLAS-PU. Especially, when all the elements in a query have A/D relationship, the performance of BLAS-PU is similar or lower than that of HTJ.



```
deptList
└department(+)
  ├professor(+) – email, tel
  │ ├name - lastname, firstname
  │ └publications
  │   ├journal(+) – title, year
  │   │ └paper(+) - title
  │   │   └author(+) - name – lastname, firstname
  │   └conference(+) – title, year
  │     └paper(+) - title
  │       └author(+) - name – lastname, firstname
  ├staff(+) – email, tel
  │ name – lastname, firstname
  ├gradstudent(+) – email, lab
  │ ├name – lastname, firstname
  │ ├course(+) – title, grade
  │ │ └professor – name – lastname, firstname
  │ └publications
  │   ├journal(+) – title, year
  │   │ └paper(+) - title
  │   │   └author(+) - name – lastname, firstname
  │   └conference(+) – title, year
  │     └paper(+) - title
  │       └author(+) - name – lastname, firstname
  └undergradstudent(+) – email
    ├name – lastname, firstname
    └course(+) – title, grade
      └professor – name – lastname, firstname
```

Fig. 4. deptList.dtd.

To check the scalability of the algorithms, we prepared 4 data sets with incrementally different sizes and evaluated the query Q2. Fig. 5(b) shows the result. As shown the figure, the response times of HTJ and BLAS-PU linearly increase but that of P-QEval is nearly constant. This is because HTJ and BLAS-PU perform join operations on all elements or some elements with A/D relationships, however, P-QEval do not perform any join operation for a simple path query irrespective of the existence of A/D relationships.

Meanwhile, the size of UPET is an important factor to determine whether P-QEval makes use of P-Filter algorithm. If the size of UPET is large, P-Filter may incur much I/O and CPU cost. However, in our experiments, we confirmed that the size of UPET is negligible and it can be managed in main memory. Although the structure of an XML data can be very complex and irregular, generally the number of label paths in an XML data is extremely smaller than that of data nodes. Thus, the cost of performing P-Filter algorithm is negligible in the aspect of the overall cost to process a query.

Cost analysis. Let a query Q be composed of n elements, $e_1, e_2, \ldots, e_n$, and $|e_i|$ is the number of nodes in an element $e - i$ $(1 \leqslant i \leqslant n)$. To evaluate Q, HTJ

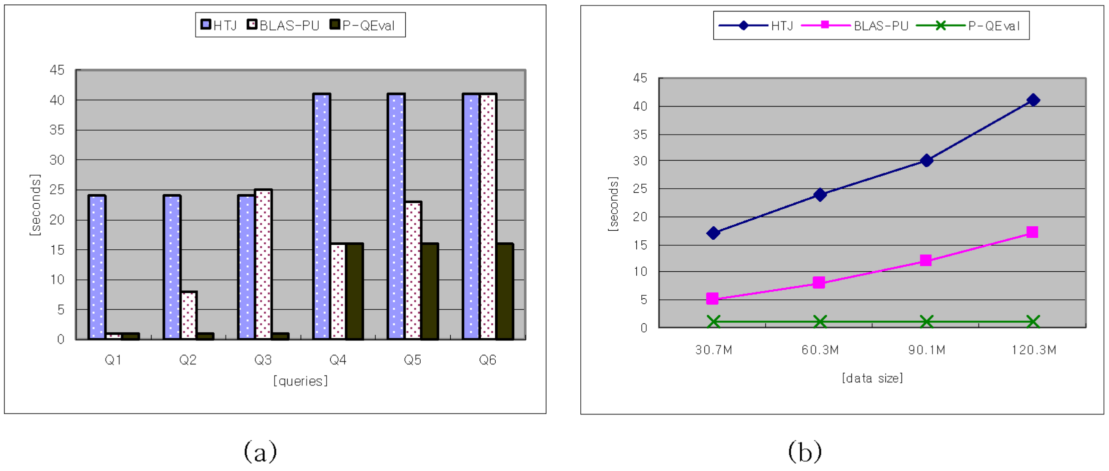


Fig. 5. Experimental results.

performs $(n-1)$ join operations. Thus, the cost is $O(|e_1|+|e_2|+\cdots+|e_n|)$. On the contrary, BLAS-PU and P-QEval transform Q into l linear path expressions and perform join operations on them. Thus, they require $(l-1)$ join operations. Since $l \leqslant n$, the costs of BLAS-PU and P-QEval are always smaller than or equal to that of HTJ. Meanwhile, P-QEval is different from BLAS-PU in its query transformation and node filtering algorithm. If b is the number of child elements in branch points and d is the number of descendant elements, BLAS-PU transforms Q into $(b+d)$ path expressions. On the contrary, P-QEval picks up only the elements in significant positions (branch, terminal, and retrieval) together with their structural predicates. If some elements are located in descendant axis and their locations are not significant positions, P-QEval does not select them as join elements. Thus, the number of elements selected by P-QEval is always smaller or equal to $(b+d)$ and the join cost may much smaller than BLAS-PU. In addition, P-QEval filters out the nodes that do not match to structural predicates on the selected join elements before join operations. Thus, when the elements are located in two or more label paths, P-QEval may access much smaller number of nodes than BLAS-PU if some label paths are filtered out by P-Filter.

6. Conclusions

In this paper, we proposed a filtering algorithm, P-Filter, which filters out all the nodes in the paths that do not match to the structural predicates of join elements. Also, we proposed a new query processing algorithm, P-QEval, which finds join elements and evaluates the query based on the elements and P-Filter algorithm. From our experimental results, we observed that our algorithm significantly outperforms existing state-of-the-art query processing algorithms, by filtering out large number of nodes and reducing the number of joins. We also confirm that our filtering algorithm can be easily combined with existing structural join algorithms, thereby significantly enhancing the performance of them.

References

[1] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, Y. Wu, Structural joins: a primitive for efficient XML query pattern matching, in: ICDE, 2002, pp. 141–153.

[2] N. Bruno, N. Koudas, D. Srivastava, Holistic twig joins: optimal XML pattern matching, in: SIGMOD, 2002, pp. 310–321.

[3] Y. Chen, S.B. Davidson, Y. Zheng, BLAS: an efficient XPath processing system, in: SIGMOD, 2004.

[4] S. Chien, Z. Vagena, D. Zhang, V.J. Tsotras, C. Zaniolo, Efficient structural joins on indexed XML documents, in: VLDB, 2002, pp. 263–274.

[5] B. Cooper, N. Sample, M.J. Franklin, G.R. Hjaltason, M. Shadmon, A fast index for semistructured data, in: VLDB, 2001, pp. 341–350.

[6] R. Goldman, J. Widom, DataGuides: Enabling query formulation and optimization in semistructured databases, in: VLDB, 1997, pp. 436–445.

[7] H. Jiang, H. Lu, W. Wang, B.C. Ooi, XR-tree: indexing XML data for efficient structural joins, in: ICDE, 2003, pp. 253–263.

[8] R. Kaushik, P. Bohannon, J.F. Naughton, H.F. Korth, Covering indexes for branching path queries, in: SIGMOD, 2002, pp. 134–144.

[9] Q. Li, B. Moon, Indexing and querying XML data for regular path expressions, in: VLDB, 2001, pp. 361–370.

[10] H. Wang, S. Park, W. Fan, P.S. Yu, ViST: a dynamic index method for querying XML data by tree structures, in: SIGMOD, 2003, pp. 110–121.

[11] C. Zhang, J.F. Naughton, D.J. DeWitt, Q. Luo, G.M. Lohman, On supporting containment queries in relational database management systems, in: SIGMOD, 2001, pp. 425–436.