

# Labeling and Querying Dynamic XML Trees

Jiaheng Lu , Tok Wang Ling

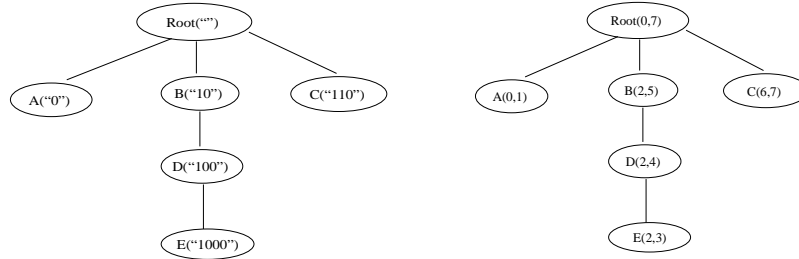
School of Computing, National University of Singapore  
3 Science Drive 2, Singapore 117543  
{lujiaheng, lingtw}@comp.nus.edu.sg

**Abstract** With the growing importance of XML in data exchange, much research tends to provide a compact labeling scheme and a flexible query facility to extract data from dynamic XML trees. In this paper, we first propose GRP, namely GRoup based Prefix labeling scheme. Compared to the previous labeling schemes, the total size of labels in GRP is much shorter. Experiment results with synthetic and real life data show that the size of labels with GRP is about 2%-10% of that with the previous labeling scheme. Based on GRP, we further propose GRJ (GRoup based structural Join), a structural join algorithm. GRJ is similar to the hash join algorithm in RDBMS and needs to scan the join data only twice. Furthermore, unlike other structural join algorithms, GRJ can perform efficiently without dependency on the join data sorted or indexed, for the data in the dynamic XML trees are usually unsorted. Finally our experiments show that GRJ is efficient in supporting structural joins on the context of dynamic XML trees.

## 1 Introduction

With the growing importance of XML in data exchange, much research has been done to provide an effective query mechanism to extract data from XML document [3,4,6,8]. Typical queries over XML documents amount to finding nodes with particular tags having certain *ancestor-descendant* relationship between them. XML query engines often process such queries by using an index structure, typically a big hash table, whose entries are the tag names in the indexed documents. To allow structural queries, each node in the XML trees is given a unique label, and every entry (tag name) in the hash table is associated with the list of those labels of the relevant nodes. The labels are designed such that given the labels of two nodes we can determine whether one node is an ancestor of the other. Thus, structural queries can be answered by only using the index, without access to the actual documents.

In order to making XML into a full-featured data exchange format, clearly, we should allow users of XML data to query not only the current values of documents, but also the changes in the content over time [1,5]. For example, users may be interested to know the price of a particular book in the previous time or ask for the list of new books. To support such queries, XML databases should design a *persistent labeling scheme* to mark each node in the XML tree and to connect the various versions of



(a) Simple prefix labeling scheme                      (b) Range labeling scheme

**Figure 1 An example of an XML tree**

a particular item through time. In a *persistent* labeling scheme, when a new node is inserted, this node is assigned a unique label and this label cannot be changed later on. We use this unique label to trace this node through different versions.

Much research has been done to propose a persistent labeling scheme. The state-of-the-art research is done by Edith Cohen(2002) [5]. In this paper, they propose a *simple prefix labeling scheme*. They label the root with an empty string. The first child of the root is labeled with “0”, the second child with “10”, and the third with “110”, etc. For any node,  $L(v)$  denotes the label of  $v$ . Then the first child of  $v$  is labeled with  $L(v)$ “0”, the second child of  $v$  is labeled with  $L(v)$ “10”, and the  $i^{\text{th}}$  child with  $L(v)$ “(1..1) <sup>$i-1$</sup> 0”. It is a correct prefix scheme, namely for all pairs of nodes  $v, u$ ,  $L(v)$  is a *prefix* of  $L(u)$  iff that  $v$  is an ancestor of  $u$ . See Figure 1(a), for example, B is an ancestor of E, for “10” is a prefix of “1000”.

Simple prefix (SP) labeling scheme is a persistent labeling scheme, where, in order to identify the ancestor-descendant relationship between any two nodes, this labeling scheme does not need to be renumbered for any arbitrarily heavy update, but the size of labels in SP is often too large. For most real-world XML data, the total length of labels scales up quadratically with the increase of the size of XML tree.

In order to shorten the labeling size, Cohen *et al* also propose other shorter labeling schemes, but most of them must use special “clues”, which provide estimations on possible future insertions. For example, one of clues is to estimate the minimal and maximal number of the descendants for any new inserted node. Unfortunately, those “clues” are usually too strict to be satisfied in practice and the labels size may be as large as the previous algorithm when the estimations are wrong. Therefore, designing a shorter persistent labeling scheme for the dynamic XML tree is still an open challenge.

Once an XML tree has been labeled, those labels then should be used to evaluate structural queries. Since any complex structural queries can be decomposed into the basic *ancestor-descendants* relationship queries between the element sets, *structural join*[12,13] are recently considered as the core operation for XML structural queries. Previously proposed structural join algorithms [3,6] are only efficient for static XML setting, but not suitable for querying XML trees which are subject to frequent insertions and deletions of nodes. To understand this, note that (1) most previously efficient algorithms are based on range labeling scheme (detailed discussion of range

labeling scheme will be in next section), which is not a persistent labeling scheme; (2) their algorithms suppose that the join elements are either sorted or indexed with special indices, such as B+ tree. But in the dynamic XML context, the elements are usually unsorted and the creation of many indices will drastically worsen the updating performance. Therefore, designing an efficient structural join algorithm for querying dynamic XML trees is also an important research issue.

In this paper, we propose a novel labeling scheme, called as G<sub>R</sub>oup based Prefix (GRP) labeling scheme, and then, based on GRP, we also propose a novel structural join algorithm, called as G<sub>R</sub>oup based structural join (GRJ) algorithm. Our objective is to answer the above two challenges simultaneously by presenting a shorter labeling scheme and an efficient structural join algorithm for processing dynamic XML trees.

## 2 Related Work

XML data is commonly modeled by a tree structure, where nodes represent elements, attributes and text data, while parent-child pairs represent nesting between XML elements. The structural relationship between two elements can be quickly determined by the labels of two nodes. Two particular types of labeling scheme have been proposed in the previous work [5,7,11,12,15]. They are *range* labeling scheme and *prefix* labeling schemes.

### 2.1 Range labeling scheme

In the *range* labeling scheme, the label of a node  $v$  is interpreted as a pair of numbers  $\langle a_v, b_v \rangle$ :  $a_v$  is called as the *start* position, while  $b_v$  is the *end* position. A node  $v (\langle a_v, b_v \rangle)$  is an ancestor of  $u (\langle a_u, b_u \rangle)$  iff  $a_v \leq a_u \leq b_u \leq b_v$ . In other words, range  $\langle a_u, b_u \rangle$  is contained in range  $\langle a_v, b_v \rangle$ . In Figure 1(b), for example: E is a descendant of B, for  $2 \leq 2 < 3 < 5$ . This approach lacks flexibility. When a new node is inserted, the labels probably need to be recomputed for some of the existing tree nodes. Therefore, the range labeling scheme is not a *persistent* labeling scheme.

### 2.2 Prefix labeling scheme

In the *prefix* labeling scheme, the label of a node  $v$  is interpreted as a *binary string*  $L(v)$ . A node  $v$  is an ancestor of  $u$  iff  $L(v)$  is a prefix of  $L(u)$ . As stated before, Cohen *et al* [5] present a *simple prefix* (SP) labeling scheme. *Simple prefix* labels also can be generated by a *preorder* traversal of the tree. More specifically, for any new node  $v$ , the label  $L(v) = L(u) (1 \dots 1)^{i-1} 0$ , where  $u$  is the parent of  $v$ , and  $i$  is the number of children of  $u$ . Compared to the *range* scheme, for the purpose of identifying the ancestor-descendants relationship between two nodes, SP scheme need not be renumbered even under heavy update. Therefore, SP is a *persistent* labeling scheme.

### 3. Group Based Prefix Labeling scheme

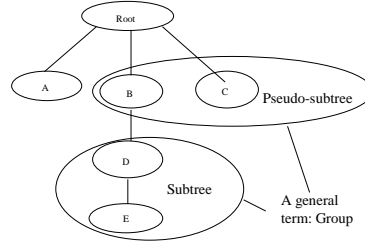


Figure 2. Subtree, Pseudo-subtree and Group

Given an XML tree  $T$ , a *subtree* is the tree which is a child of one node in  $T$ . If we delete the root node of any *subtree*, then we call the left *forest* as a *pseudo-subtree*. In Figure 2, for example, we call  $\{B,C\}$  as a *pseudo-subtree*, for they are a *forest* formed by deleting the *root* node in the subtree  $\{root, B,C\}$ . Both *subtrees* and *pseudo-subtrees* have the following important property:

**Property 1.** Given an XML tree  $T$  and a *subtree* or a *pseudo-subtree*  $S$ , for any node  $n \in T$ , but  $n \notin S$ , one of the following two conditions must be satisfied: (1)  $n$  is an ancestor of all nodes in  $S$  (2)  $n$  is not an ancestor of any node in  $S$ .

By using this property, we can reduce unnecessary comparisons in structural join. For example, if we know a node  $n$  is an ancestor of the *subtree* or the *pseudo-subtree*  $S$ , then without any further comparisons, it is clear that all nodes in  $S$  are the descendants of  $n$ . Our structural join algorithm described in Section 4 will apply this feature to offer the improvement of join performance.

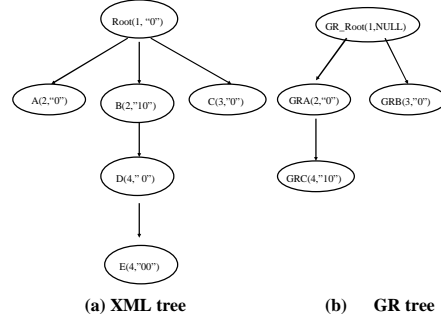
For simplicity, we give *subtree* and *pseudo-subtree* a general term: **group**. That is, whenever we refer in the following to *group*, it may be either a *subtree* or a *pseudo-subtree* (see Figure 2).

#### 3.1 GRP labeling scheme

In GRP labeling scheme any label consists of a *group ID* and a *group prefix label*, where group ID is an integer and a group prefix label is a binary string. All nodes in the same group have the same *group ID*, and are distinguished by their prefix labels. *Root* node has a fixed label: (1, "0"). In order to avoid the unlimited growth of the number of nodes in one group, we define that the maximal number of nodes in group  $i$  to be  $i$ , that means group 1 contains at most one node, group 2 contains at most two nodes, ... and the maximal number of nodes in group  $i$  is  $i$ . In addition, if the number of nodes in one group reaches the maximal value, we call that this group is *full*.

Now we introduce how to assign the labels by using GRP labeling scheme. For any new inserted node  $n$ , GRP first inquires whether the group of the parent of  $n$  is full, if not full, then node  $n$  is labeled with the same group of its parent. And then within that group, the above-mentioned SP (Simple prefix) labeling scheme[5] is used to

decide the prefix label of node  $n$ . Otherwise, if the group of the parent of  $n$  is full, then GRP asks whether the group of  $n$ 's youngest (the latest labeled) sibling is full, if not full, then  $n$  is labeled with the group of its youngest sibling. Otherwise,  $n$  should be labeled with a new group and its prefix label is "0".



**Figure 3 An example of GR tree**

We use an XML tree in Figure 3(a) as an example to explain the above rule. First suppose that the inserting sequence is  $root, A, B, C, D, E$ . Firstly, in GRP labeling scheme,  $root$  has the fixed label (1, "0"). Secondly, since the node  $A$  is the child of the  $root$  and group 1 ( $root$ ) is full,  $A$  is labeled in a new group and its prefix label is "0", so the label of  $A$  is (2, "0"). The next is node  $B$ . Since the group of its youngest sibling  $A$  (2, "0") is not full,  $B$  is labeled in group 2 and its prefix label is "10" (by SP labeling scheme within group 2). So the label of  $B$  is (2, "10"). Next, when  $C$  is inserted, as the group of  $C$ 's parent ( $root$ ) and the youngest sibling (node  $B$ ) are full,  $C$  is assigned the label (3, "0"). According to the same spirit,  $D$  is labeled with (4, "0") and  $E$  with (4, "00").

We shall introduce another tree, called "Group Relationship (GR) tree", to summarize the structural information between different groups. Given an XML tree  $T$  labeled by GRP, in a group relationship (GR) tree, each node  $n$  presents one group in  $T$  and is labeled with  $\langle s, pa \rangle$ , where  $s$  is an integer and denotes the group ID,  $pa$  is a binary string and presents the prefix label of parent of the first labeled node with group  $s$ . Every edge in GR tree means the *child-parent* relationship between two groups.

Whenever an inserted node  $v$  is labeled with a new group in XML tree, we need correspondingly insert a new node  $gv$  in GR tree. Now we use an example to illustrate the creation of GR tree. Assume the insertion sequence in the XML tree of Figure 3(a) is  $Root, A, B, C, D$  and  $E$ . First, when  $root$  is inserted in the XML tree, node  $GR\_Root$  is inserted in GR tree. Since  $root$  is in group 1 and has not a parent node,  $GR\_Root$  is labeled with (1, NULL). Second, when  $A$  is inserted,  $GRA$  is inserted in GR tree. Since  $A$  is in group 2 and the prefix label of parent of  $A$  is "0",  $GRA$  is labeled with (2, "0"). With the same spirit, when  $C$  is inserted in XML tree,  $GRB$  is inserted in GR tree. And since the prefix label of parent of  $C$  (i.e.  $root$ ) is "0",  $GRB$  is labeled with (3, "0"). Finally, as  $D$  (4, "0") is inserted in XML tree,  $GRC$  is inserted as the child of  $GRA$  (2, "0") and is labeled with (4, "10").

## 4 Group Structured Join Algorithm

Once the whole XML tree is labeled by GRP labeling scheme, then we are ready to process structural join efficiently. In this section, we first present GGroup based structural Join (GRJ) algorithm, followed by the efficiency analysis of GRJ.

### 4.1 GRJ Algorithm

The main idea in GRJ is to divide the join operations into two classes, one is *intra-group* join, and the other is *inter-group* join.

- *Intra-group* join is to join the elements in the same group. This operation can be implemented by simply comparing the prefix labels of elements.
- *Inter-group* join is to join the elements in the different groups. This operation should use GR tree.

Algorithm 1 lists the routine of GRJ, and Algorithm 2 discusses the inter-group join process.

---

**Algorithm 1. Group based Structural Join algorithm**

---

**Input:**  $A$  is the ancestor list and  $D$  is the descendant list

**Output:** Pairs of ancestor-descendant elements

1. Scan  $A, D$  list once to assign every element to their respective group bucket;
  2. Initialize  $DgroupHash$  as a hash table, where keys are group IDs of  $Dlist$ , and each value is initialized as an empty set.  
*/\* Each element in the set  $DgroupHash(i)$  will be a label of the node of XML tree that is the ancestor of group  $i$ . In other words,  $DgroupHash(i)$  caches the ancestors of group  $i$ . \*/*
  3. **For**  $i:=2$  to max group do  
*/\*since group 1 only contains root node, here begin from group 2 \*/*
  4. Output all elements in  $Dlist$  of group  $i$  as the descendants of each element in set  $DgroupHash(i)$ ;
  5. Perform ***Intra-group*** Join for group  $i$ ;
  6. Perform ***Inter-group*** Join for group  $i$  by calling Algorithm 2
  7. **End For**
- 

We now go through Algorithm 1. Assume  $A$  and  $D$  are two element lists. Basically, in this algorithm, we first scan the  $A, D$  list once and assign them to respective group buckets in terms of their group IDs. Then in each bucket, *intra-group* join and *inter-group* join are performed. Detailedly, in step (2), a hash table  $DgroupHash$  is initialized, which will contain the ancestor nodes of any group. In step (4), we directly output *ancestor-descendant* pairs between the current group and its ancestor groups according to  $DgroupHash$ . Furthermore, in step (5), we compare the labels of any two nodes to determine whether they are the *ancestor-descendant* pairs. Finally, in step (6), Algorithm 2 is called to perform *inter-group* join.

---

**Algorithm 2: Inter-group join**

---

**Description:** The objective of this algorithm is to find all groups which are descendants of group  $i$ , the result is stored in hashtable  $DgroupHash$

**Input :** Ancestor list ( $A_i$  list) in group  $i$ , hashtable  $DgroupHash$ , GR tree

**Output:** Updated  $DgroupHash$

1. **For** each child  $c$  of group  $i$  in GR tree and each element  $a$  in  $A_i$  list  
/\*Assume  $c$  is labeled ( $c.groupID, c.parent-prefix-label$ ),  
 $a$  is labeled ( $i, a.prefix-label$ ) \*/
    - 1.1 **If**  $a.prefix-label$  is equal to or the prefix of  $c.parent-prefix-label$   
/\*This means element  $a$  is the ancestor of group  $c$  \*/
    - 1.2 **Then** { add the element  $a$  into the set  $Ancestor(c)$ ;
    - 1.3        add  $c$  into set  $N$ ; }/\*Any element in  $Ancestor(c)$  is the ancestor of group  $c$ \*/
  2.        **Traverse** the subtree of each node  $c \in N$  in GR tree
    - 2.1        **For** any visited node  $v$ ,  
/\* assume  $v$  is labeled ( $v.groupID, v.parent-prefix-label$ )\*/
    - 2.2        **If** hashtable  $DgroupHash$  contains the key  $v.groupID$
    - 2.3        **Then**  $DgroupHash(v.groupID) := DgroupHash(v.groupID) \cup Ancestor(c)$ ;  
/\* This means any element which is the ancestor of group  $c$  is also the ancestor of group  $v$ , since  $c$  is the ancestor of  $v$ .\*/
- 

#### 4.2 An analysis of GRJ algorithm

If we consider that the “group” of each node is a natural hash function, then group based (GRJ) join algorithm is much like hash join in relation database. The greatest difference between hash join and GRJ lies in that GRJ should deal with join among different group buckets. But when we perform *inter-group* join, we only need access GR tree and do not read other bucket data. Since the size of GR tree is small even for the large dataset (e.g. 90K bytes GR tree in memory vs. 167M bytes DBLP XML source document), GR tree can be fit in the main memory. So there is no extra I/O cost for inter-group join. As a result, GRJ is as efficient as hash join and the element lists are visited twice. Hence, we have the following result(interested of space, proofs are omitted here)

**Theorem 4.1** The I/O complexity of GRJ algorithm is  $2*S(Alist)/B + 2*S(Dlist)/B + S(Outputlist)/B$ , where  $B$  is the blocking factor,  $S(List)$  denotes the total size of labels in this list.  $\square$

**Theorem 4.2** The main memory time of GRJ algorithm is

$$\sum_{i=2}^{Maxgroup} (|AList(i)| * |DList(i)| + |AList(i)| * |ChildList(i)| + |Des\_AList(i)|)$$

where  $|ChildList(i)|$  is the cardinality of children of group  $i$  in GR tree and  $|Des\_AList(i)|$  is the cardinality of descendant groups of each element in  $AList(i)$ .  $\square$

## 5 Experimental Result

Comprehensive experiments were conducted to study the effectiveness of GRP labeling scheme and GRJ algorithm. In this section, we describe these experiments and present the results.

### 5.1 Experimental setup

We run experiments on synthetic data, such as the benchmark database XMARK[10], the IBM XML data generator[16] and real-world XML data set, such as DBLP[9] and Shakespeare’s Plays[14], but due to the space limitation, we only report results on XMARK(the results of other data were similar). In order to get a comprehensive experimental result, we use an XMARK dataset generated by xmlgen[10] with varied scaling factors, from 20K to 120M bytes of data. The number of nodes in XML document varies from 400 to 1,600,000. All the experiments were performed on the Pentium IV 1.7G Hz PC with 768M RAM, 30G hard disk, running windows XP with C++ language.

### 5.2 Size of labels

The first experiment was performed to study the comparative size of labels with two labeling scheme, including SP and GRP labeling schemes. For each XML document, we use the preorder traversal to label it. Each group ID is presented by 16 bits (MAX value= $2^{16}-1=65,535$ ). Furthermore, *variable-length representation* [7] is used to represent the value of label, where each label may have a different length, consisting of a *fixed-length prefix* stating the actual length of label followed by label itself. The size of the *fixed-length prefix* in this experiment is also 16 bits. For instance, the total length of label “01100000” in *variable-length representation* is 24 bits (16+8=24). Table 1 shows the results of our experiments.

**Table 1. Size of the labels with SP and GRP labeling schemes**

File	Size of XML tree(MBytes)	Number of nodes in XML tree (K)	Total length of labels (M bytes)	
			SP scheme	GRP scheme
1	1.2	17	1.36	0.64
2	2.3	33	4.64	1.20
3	3.5	50	10.24	2.00
4	4.7	68	17.92	2.72
5	5.7	84	27.04	3.44
6	7.0	100	39.20	4.24
7	8.1	118	54.08	5.04
8	9.4	134	69.12	5.92
9	10.5	151	88.00	6.80
10	11.6	167	107.2	7.60

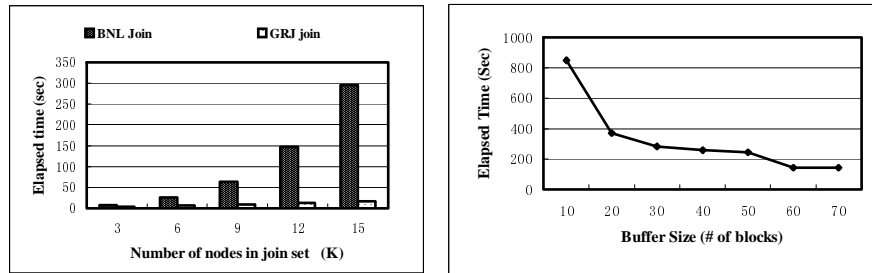
Several observations and explanations can be made from the Table 1.

- GRP labeling scheme has much smaller size of labels than SP scheme and the advantage margin gets larger with the increase of file size.
- The labeling size of SP scheme increases by about  $O(N^2)$ ,  $N$  being the number of nodes in XML tree, while GRP scheme scales up linearly with the increase of size of XML tree.

## 5.2 Query performance

Finally, the focus in the last experiment is to characterize the performance of structural join based on SP labeling scheme and GRP labeling scheme.

For GRP scheme, we use GRJ algorithm. For SP scheme, we have to use Block Nest Loop (BNL) algorithm. Because if the label of node is given directly according to their inserted order, they are usually unsorted, we cannot use more efficient algorithm. In addition, we find that the join selectivity has not a significant impact on the performance of both BNL and GRJ algorithm. So the following result comes from the same structural query “*item vs. name*”.



(a) BNL Vs GRJ

(b) Varied buffer size

**Figure 4 Join Performance**

We first perform the experiments with the *fixed* buffer size, then with the *varied* buffer sizes.

**Fixed buffer size:** The fixed buffer pool size is 100 blocks and the size of each block is 8K. As can be seen from Figure 4(a), the performance of GRJ algorithm is better than BNL. The rationale is the twofold : (1) when the size of files increases, the data cannot fit in the buffer and BNL algorithm has to read the same block many times, which occupies most of join time. In contrast, GRJ algorithm scans the lists only twice. The first time is to assign every data into the respective buckets. The second one is to perform the join operation. (2) The size of labels with GRP is much smaller than SP. Thus, GRJ algorithm spends less time to scan the list once than SP.

**Varied buffer size:** We also measure the performance of different algorithms with the varied buffer size. The results show that the performance of BNL algorithm was essentially affected by varied buffer size, while GRJ algorithm was not affected. This result is not hard to be understood, for BNL algorithm is actually a kind of nested loop

join algorithm, while GRJ is similar to hash join. Figure 4(b) displays the elapsed time for BNL with the varied buffer pool sizes.

## 6 Conclusion

We have proposed a GRP labeling scheme for dynamic XML trees. This scheme uses the well-known *divide-and-conquer* algorithm design paradigm, where the problem is solved by solving smaller subproblem and using the solution to solve the original problem. In GRP, the whole big tree is divided to many smaller *groups* and be conquered(labeled) using the prefix labeling scheme. Moreover, we have also proposed a GRJ algorithm, which is similar to hash join in RDBMS. The extensive experimental evaluation has demonstrated that GRP labeling scheme and GRJ algorithm is efficient on the context of dynamic XML trees.

## References

- 1 Xyleme A dynamic data warehouse for XML data of the Web. <http://www.xyleme.com>.
2. Dao Dinh Kha, Masatoshi Yoshikawa and Shunsuke Uemura *A Structural Numbering Scheme for XML Data* In Proc. EDBT 2002 LNCS 2490 pages 91-108, 2002
3. Haifeng Jiang, Hongjun lu, Wei Wang and Beng Chin Ooi *XR-Tree: Indexing XML Data for Efficient Structural Joins* ICDE, March 2003
4. Haixun Wang, Sanghyun Park, Wei Fan and Philip S. Yu *ViST: A Dynamic Index Method for Querying XML Data by Tree Structures* ACM SIGMOD, June 2003.
5. Edith Cohen, Haim Kaplan, Tova Milo *Labeling Dynamic XML Trees* ACM PODS 2002
6. Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras and Carlo Zaniolo *Efficient Structural Joins on Indexed XML Documents* In Proc. 28<sup>th</sup> VLDB Conference 2002.
- 7 H. Kaplan, T. Milo, and R. Shabo. *A comparison of labeling schemes for ancestor queries.* In Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA), 2002.
8. R. Kaushik, P. Bohannon, J. Naughton, and H. Korth. *Covering index for Branching path queries.* In ACM SIGMOD, June 2002.
9. Michael Ley. DBLP database web site. <http://www.informatik.uni-trier.de/~ley/db/>, 2003.
10. XMARK: The XML-benchmark project. <http://monetdb.cwi.nl/xml> 2002.
11. Q Li and B. Moon *Indexing and querying XML data for regular path expressions* In Proc. 27<sup>th</sup> VLDB Conference 2001.
12. Yuqing Wu Jignesh.M Patel, H.V. Jagadish *Structure join order selection for XML query optimization* ICDE Conference March 2003.
13. D. Srivastava, S.Al-khalifa, H. V. Jagadish, N. Koudas, J. M. Patel and Yuqing Wu. *Structural Joins: A primitive for efficient XML query pattern matching*, in ICDE 2002.
14. Robin Cover. The XML Cover Pages. <http://xml-coverpages.org/xml.html> February 2001
15. S.Abiteboul, H.Kaplan and T.Milo. *Compact labeling schemes for ancestor queries* In Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA), January 2001.
- 16.IBM Corporation. *XML data generator* <http://www.alphaworks.ibm.com/tech/xmlgenerator>