L-Tree: a Dynamic Labeling Structure for Ordered XML Data

Yi Chen[1], George A. Mihaila[2], Rajesh Bordawekar[2], and Sriram Padmanabhan[2]

[1] University of Pennsylvania, yicn@seas.upenn.edu
[2] IBM T.J. Watson Research Center, {mihaila,srp,bordaw}@us.ibm.com

Abstract. With the ever growing use of XML as a data representation format, we see an increasing need for robust, high performance XML database systems. While most of the recent work focuses on efficient XML query processing, XML databases also need to support efficient updates. To speed up query processing, various labeling schemes have been proposed. However, the vast majority of these schemes have poor update performance. In this paper, we introduce a dynamic labeling structure for XML data: L-Tree and its order-preserving labeling scheme with O(log n) amortized update cost and O(log n) bits per label. L-Tree has good performance on updates without compromising the performance of query processing. We present the update algorithm for L-Tree and analyze its complexity.

1 Introduction

With the advent of XML as a data representation format, we see an increasing need for robust, high performance XML database management systems which support efficient queries and updates processing. There has been great interest in storing XML data in RDBMS [11, 14, 1, 15, 3], in order to leverage the power of RDBMS for data management. However, since XML data is fundamentally different from relational data encountered in typical business applications, there are several challenges for storing XML data into relational database.

First, XML is the successor of earlier document markup languages such as SGML and HTML, primarily a *document* format. The implicit order among data elements, the so called *document order*, is important. An XML database needs a mechanism to record the relative position of data elements. Recently [15] presented how to store XML data in RDBMS preserving the document order. However, how to maintain the order upon updates is not clear.

Second, an XML database must be able to efficiently retrieve XML fragments by some XML query language, like XPath or XQuery. The edge table approach [11] treated an XML document as a tree, and generated a tuple for every XML node with its parent node identifier in the relation. To process queries with structural navigation, one self-join is needed to obtain each parent-child relationship. [14, 1] proposed to inline the information of leaf nodes into the tuple for their parents, such that the joins between a node and its leaf children are eliminated. However, to answer descendant-axis "//" or ancestor-axis in XML query, many self-joins are needed.

One popular method for maintaining the document order, which assigns ordered labels to data items, turns out be very helpful to answer ancestor-descendant queries.

Specifically, an XML document, treated as an ordered tree, is traversed in depth-first order and ordered labels are assigned to element nodes. Each node x receives two numbers, the first one, B_x, when it is first visited, and the second one E_x, when it is exited. For example, Figure 1 shows an XML tree where every node is labeled by two numbers. Using this scheme, a navigation query can be converted to an interval containment test by using the following observation: for any two nodes x and y, x is an ancestor of y if and only if the interval (B_x, E_x) includes the interval (B_y, E_y), or equivalently $B_x < B_y$ and $E_y < E_x$. Now, to answer a query "book//title" over the example, one only needs to find the nodes with tag "book" and the nodes with tag "title", then test their labels to check the ancestor-descendant relationship. When XML data is stored in RDBMS, the ancestor-descendant queries can be processed by exactly one self-join with label comparisons as predicates, which is as efficient as child-axis. The effectiveness and efficiency of XQuery processing with the labeling scheme in comparison with other XQuery implementations is discussed in [7].

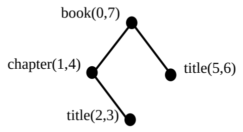


Fig. 1. An example of an XML labeling scheme

While very advantageous for queries and preserving the document order, most of the proposed labeling schemes [12, 13, 17, 2] incur large relabeling costs. Consider the labeling scheme in Figure 1 which assigns labels from the integer domain, in sequential order. This leads to relabeling of half the nodes on average, even for a single node insertion. Alternatively, one can leave gaps in between successive labels to reduce the number of relabelings upon updates. As proved in [5], an order-preserving labeling scheme without any relabelings upon updates requires $O(n)$ bits per label, which leads to large space requirements and costly label comparisons during query processing. It is not clear how to assign the gaps between labels such that we can find a good trade-off between the number of bits used to encode the labels and the number of node relabelings each update will cause. The paper addresses this problem.

The contributions and the structure of this paper are:

1. We introduce a dynamic structure called L-Tree to maintain an order-preserving labeling scheme for XML data in the presence of updates in Section 2.
2. We analyze the amortized update cost of an L-Tree and the label size. We derive exact functions for the update complexity and the label size, and discuss how the optimal results can be achieved by choosing different tree parameters in various application settings. This is presented in Section 3.
3. We discuss how an L-Tree can accommodate XML subtree updates to achieve better performance compared to single node updates as well as a variant of the main labeling scheme in Section 4.

Related work is discussed in Section 5 and Section 6 concludes the paper.

2 L-Tree and XML Labeling Scheme

When designing an XML labeling scheme, we need to answer the following two questions: 1) how to assign labels to elements in an XML document D to reflect the document order? and 2) when a new node is inserted, what label should be assigned to it, and which existing labels need to change to preserve the order?

For the purpose of the discussion, it is helpful to view the XML document in its textual representation as a linear ordered list of begin tags, end tags, and text sections, $L_D = (a_1, a_2, \ldots, a_n)$. Our problem is similar to the maintenance of an ordered list. A label from an integer interval $[0, M)$ is assigned to each tag to reflect its order in the list. Intuitively, we would like to distribute these n labels over the whole interval evenly. However, random updates will cause some areas in the interval to become much more dense than others. If a new tag is inserted at a position where the difference between the labels of its neighbors is 1, we have to redistribute some labels to make room for the newly inserted tag.

The basic idea of our algorithm is that we divide the whole interval $[0, M)$ into many intervals of equal size, each of which are further divided to smaller intervals, and so on. Then we set a limit on the number of labels in each interval such that we can control the density of each interval. In this way relabelings upon an update are localized. The nested relationship of intervals and subintervals suggests a tree structure. So we build a tree and associate each interval to an internal node in the tree to maintain the labels for XML data. This tree is called a *L-Tree* (short from *label tree*).

2.1 Labeling Scheme of the L-Tree

An L-Tree is an ordered balanced tree with n leaves. We attach the tags in the XML document to these n leaves in order, starting from the leftmost leaf. The shape of the L-Tree is determined by two parameters f and s, which control the number of leaf descendants of internal nodes. Specifically, the maximal fanout of any internal node in an L-Tree is $f - 1$, and the minimal fanout is f/s. We will discuss how to choose the values of these parameters in Section 3.

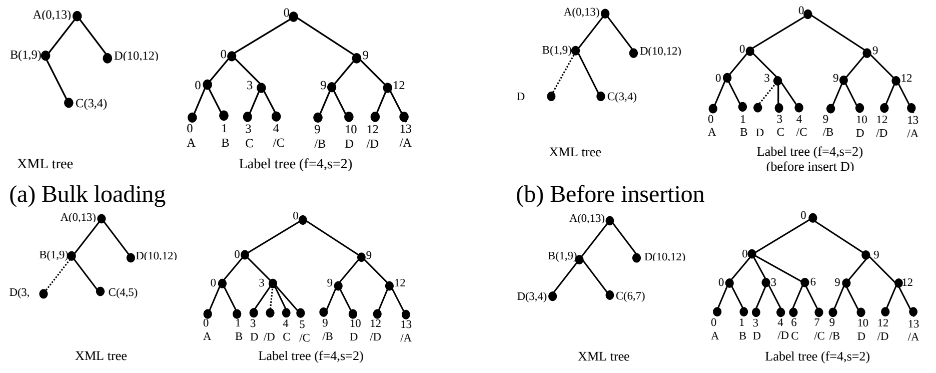
For any node x in the L-Tree, we assign a number $N(x)$, recursively in a top-down fashion as follows:

1. $N(root) = 0$
2. Let x be the i^{th}, $(0 \leq i < f - 1)$ child of y, $N(x) = N(y) + i \cdot (f - 1)^{h(x)}$

Here the height $h(x)$ of any node x is defined as the number of edges on the longest path from x to any leaf node in the subtree rooted in x (in particular, leaf nodes have height 0). Let us also denote by h the height of the L-Tree ($h = h(root)$). The number $N(x)$ is the smallest integer in the interval corresponding to x, and the difference between the numbers of two siblings are the space reserved for future insertions.

Finally, the labels of the XML tags are the labels assigned to their corresponding leaves in the L-Tree. It is easy to see that the above labeling scheme preserves the order of the XML tags.

Proposition 1. *Let x be a leaf in an L-Tree corresponds to an XML tag a_i and y corresponds to a a_j, a_i appears before a_j in the XML document if and only if $N(x) < N(y)$.*



(a) Bulk loading (b) Before insertion (c) After the insertion of “D” (d) After the insertion of “/D”

Fig. 2. An example of L-Tree

The label of an XML element node is composed by a pair: the numbers of two leaves in the L-Tree which correspond to that XML node’s begin tag and end tag, respectively. The order-preserving property of this labeling scheme allows us to convert the XML navigation queries into interval containment tests as illustrated in Section 1.

2.2 Bulk Loading

Initially, we build an L-Tree for some existing XML document in a bulk loading mode. To maximize the capability to accommodate further insertions, we build a complete f/s-ary tree initially (the height of this tree is the smallest number h for which $(f/s)^h \geq n$). Figure 2(a) shows an XML tree and its corresponding L-Tree ($f = 4$, $s = 2$).

2.3 Incremental Maintenance

Now let us examine how to maintain the balance of an L-Tree in the presence of insertions and assign labels to the inserted XML tags (see Algorithm 1).

For example, we would like to insert an XML node with tag “D” as the preceding sibling of the node tagged by “C” in the XML tree in Figure 2(a). We need to insert two leaves into L-Tree, corresponding to the begin tag and end tag of the XML node, respectively. Next we illustrate how to insert two leaves into L-Tree one after another.

For every internal node t in the L-Tree, denote by $c(t)$ the number of children of t and by $l(t)$ the number of leaves in the subtree rooted at t. In order to keep the labels distributed in a balanced manner, we impose a limit $l_{max}(t) = s \cdot (f/s)^{h(t)}$ on the maximum number of leaves that each internal node t may have in its subtree.

When we insert a leaf x in the L-Tree, $l(z)$ increases by one for every ancestor z of x. We look for the highest ancestor t satisfying $l(t) = l_{max}(t)$.

If no such t exists, we relabel x and its right siblings. In our insertion example, assuming that we maintain the links between an XML node to its corresponding leaves in the L-Tree, we can get the label of the XML node tagged by “C”: (3,4). First we insert a leaf to the L-Tree, corresponding to the begin tag “D”, before the leaf with

Algorithm 1 Label Tree Incremental Maintenance Algorithm

```
function $insert-after(x, y)$
$z = parent(y)$
insert $x$ as the right sibling of $y$
while $z \neq NULL$ do
  $l(z)++$
  if $l(z) = s \cdot (f/s)^{h(z)}$ then
    $t = z$
  end if
  $z = parent(z)$
end while
if $t = NULL$ then
  let $x$ be the $k^{th}$ child of its parent
  call $label(parent(x), N(parent(x)), k)$
else
  let $t$ be the $k^{th}$ child of its parent
  split $t$ into $s$ complete $f/s$-ary trees with the same sequence of leaves
end if
if $t = root$ then
  create a new root with the $s$ top-level nodes as children;
  call $label(root, 0, 0)$
else
  replace subtree rooted at $t$ with the $s$ subtrees
  call $label(parent(t), N(parent(t), k)$
end if
return
function $label(x, num, k)$
$N(x) = num$
if $x$ is not a leaf then
  for $i = k$ to $c(x)$ do
    $y = i$-th child of $x$
    call $label(y, num + i \cdot (f-1)^{h(y_i)}, 0)$
  end for
end if
return
```

number “3”. Figure 2(b) shows the intended insertions as dotted lines. The L-Tree after the insertion is shown in Figure 2(c).

Otherwise, if such a node t does exist, we need to rebalance the L-Tree. We split t into s nodes and replace t with s complete f/s-ary subtrees of the same leaf sequence. This causes the relabeling of the subtrees of these s nodes as well as their right siblings.

As an example, now we insert another L-Tree leaf, which corresponds to the end tag “/D”, right after the leaf which corresponds to “D”. The intended insertion is represented by dotted lines in Figure 2(c). This insertion results in a split of the node numbered “3” of height 1 as shown in Figure 2(d).

The key idea behind the L-Tree splitting and subsequent relabeling upon an insertion is the following: if the insertion causes the number of leaves of some subtree to increase to a large number, this means that the labels of these leaves have become very dense. To remedy the situation we split the subtree and relabel it to provide more slack for this portion, in order to better accommodate further insertions in this portion. Since the number of leaves of any subtree is controlled and the density of the labels is also controlled, the number of nodes involved in relabelings amortized over several insertions will also be controlled.

In this paper we focus on the XML insertions since for deletions we can just mark as deleted the corresponding leaves in the L-Tree without any relabeling.

2.4 Structure Properties

In this section we examine the properties of an L-Tree and compare it with a B^+-tree.

Proposition 2. *For any internal node x, we have (1) $(f/s)^{h(x)} \leq l(x) < s \cdot (f/s)^{h(x)}$; (2) $f/s \leq c(x) < f$; and (3) All leaves are at the same level of the tree.*

Proposition 3. *Cascade splitting in an L-Tree is not possible, that is, one node splitting will not cause another node split.*

An L-Tree is similar to a B^+-tree in that it guarantees certain occupancy of the tree such that the tree is balanced dynamically and the height is bounded by $O(\log n)$, where n is the number of nodes in the tree. The differences are:

1. The goals of a B^+-tree and an L-Tree are different. The purpose of a B^+-tree is to enable fast lookups given XML labels. Though B^+-tree can adjust its structure dynamically if the labels change their values, it is not able to determine how the labels should be changed. An L-Tree is built to maintain the labels for XML nodes in the presence of updates. It helps us to determine what labels to be assigned to newly-inserted XML nodes and how the labels of existing nodes should be changed.
2. The splitting criterion of an L-Tree is based on the number of leaves of a node, rather than the number of children, therefore it will not cascade split as a B^+-tree.
3. For each internal node x in L-Tree, $f/s \leq c(x) < f$. For B^+-trees, $s = 2$.

3 Complexity Analysis

3.1 Query and Maintenance Costs

In this section we compute an upper bound on the amortized cost of queries/updates and the number of bits used per label as functions of an L-Tree parameters f and s. The query and maintenance cost of an L-Tree is measured as the number of disk accesses. Since the XML nodes are recommended to be clustered by their tags rather than labels [17], and we don't make any assumptions about nodes being cached, the cost is measured in terms of the number of nodes accessed for searching or relabeling.

For queries, an L-Tree does not incur any additional cost. In fact, if we store the label along with the XML node itself, we can retrieve the label of a given node for free.

Now we analyze the amortized cost of maintaining an L-Tree upon an insertion of leaf x using the accounting method. The cost consists of three parts: First, cost h to update $l(z)$ for every ancestor z of x. Second, if there are no nodes satisfying the splitting criterion, we pay at most f to relabel the right siblings of x. Otherwise, we split the highest one t into s nodes and relabel these s subtrees as well as t's right siblings. The number of nodes relabeled is bounded by the number of descendants of t's parent:$2s \cdot (f/s)^{h(v)}$. We charge the cost of relabeling to the $(s-1) \cdot (f/s)^{h(t)}$ insertions, which make $l(t)$ grow from $(f/s)^{h(t)}$ to $s \cdot (f/s)^{h(t)}$. So an insertion is charged $2f/(s-1)$ for each relabeling, and charged $2f \cdot h/(s-1)$ for all the relabelings.

Let n be the number of tags in the current XML tree, and M be the maximal number we use to label all the nodes in the corresponding L-Tree. Since each XML tag corresponds to a leaf in the L-Tree, we have $n = l(root) \geq (f/s)^h$.

The amortized cost for an insertion to an XML tree of size n is

$$cost(f,s,n) \leq \frac{(1+2f/(s-1))}{\log(f/s)} \cdot \log n + f$$

Since $M \leq (f-1)^h$, the maximum number of bits to encode a label is:

$$bits(f,s,n) = \log M = \frac{\log(f-1)}{\log(f/s)} \cdot \log n$$

Since f and s are some constant parameters, we can maintain the labels of XML data with $O(\log n)$ bits and $O(\log n)$ amortized insertion cost.

3.2 Tuning the L-Tree

As discussed in [10], $O(\log n)$ is the tight worst case lower bound for update cost to maintain an ordered list, if the query cost is 1. Since the cost is measured as the number of disk accesses even a reduction by a constant factor is helpful for a system implementation. Hence, we are interested in minimizing the precise insertion cost. We approximate the current XML tree size as the initial tree size n_0. We would like to set the values of parameter f and s according to different application needs to optimize the constant factors of the cost and bits.

Minimize the Update Cost.
In some applications, our goal is to set the values of parameters f and s, such that the amortized cost of insertion is minimal. This is an optimization problem:

$$Object : min(cost)$$

We can find the solution by solving the following equations:

$$\frac{\partial cost}{\partial f} = 0 \text{ and } \frac{\partial cost}{\partial s} = 0$$

For a given n_0, we can solve the above equations to get the values of f_0 and s_0.
Minimize the Update Cost for Given Number of Bits.
If we are constrained on the number of bits we are allowed to use to encode a label as B, the math model we build is the following:

$$Object : \min(cost) \quad Subject\,to : bits \leq B$$

This is a problem of optimization under inequality constraints. First we minimize function *cost* unconstrained. If the minimum point (f_0, s_0) satisfies the inequality constraints, it's the minimum point in the interior of the region under consideration.

We also investigate the function on the boundary of the region. That is, we convert the optimization problem under inequality constraints to the optimization problem under equality constraints as follows:

$$Object : \min(cost) \quad Subject\,to : bits - B = 0$$

We solve this problem by introducing a Lagrange multiplier μ and form:

$$g(f, s, n, \mu) = cost + \mu \cdot bits$$

The values of f, s and μ which give the conditional minima of $cost$ can be found by solving the following equations:

$$\frac{\partial g}{\partial f} = 0 \text{ and } \frac{\partial g}{\partial s} = 0 \text{ and } bits - B = 0$$

We compare the solutions of the above equations with (f_0, s_0) to determine the values of f and s which result in minimal cost given label size B.

Minimize the Overall Cost of Query and Updates.
When the number of bits to encode a label is less than the machine word size, the label comparison for query can done by hardware, otherwise it must be done by software. In this case, the query cost is proportional to the number of bits used. For this situation, we want to find optimal f and s to get the minimal overall cost for queries and updates. To achieve it, we need to know the query/update workload and some characteristics of the document. Due space limitation, we defer the details to [4].

4 Discussion

4.1 Multiple Node Insertions

Usually, insertions to XML documents are subtrees. Although a subtree insertion can be implemented as a sequence of leaf insertions, the question is whether we can improve the update cost by inserting multiple leaves to a L-Tree at the same time.

Without loss of generality, let p be the number of leaves to be inserted to the L-Tree. To see how the subtree insertion affects the amortized update cost, we notice that the update cost consists of three parts:

1. The cost h for updating $l(z)$ for all the ancestors z of inserted nodes. This cost now is charged to p inserted nodes.
2. The cost f for relabeling right siblings if no nodes satisfy the splitting criterion. Now it is charged to p inserted nodes.
3. The amortized cost $2f/(s-1)$ for each insertion to relabel the subtrees rooted at the splitting node as well as its right siblings. Since it is an amortized cost, it makes no difference that nodes are inserted one after another, or at one time. However, a node which is inserted by itself may need to pay this cost for up to h ancestors' splits. Will this be affected if the insertion size is p?

For simplicity, assume $p = (s-1) \cdot (f/s)^{h_0}$, for some $h_0 \geq 1$. Each subtree insertion causes the split of an ancestor x whose height $h(x) = h_0$. Also, these inserted nodes may later on pay for the splits of other ancestors y with $h(y) > h_0$. The total amortized cost for all the ancestors' splits is bounded by $2f \cdot (h - h_0 + 1)/(s-1)$. So the amortized cost of each inserted node is bounded by:

$$cost(f,s,n,p) \leq \frac{\log n}{p \cdot \log(\frac{f}{s})} + \frac{f}{p} + \frac{2f}{s-1} \cdot (\frac{\log n - \log(\frac{p}{s-1})}{\log(\frac{f}{s})} + 1)$$

To generalize, if the bulk insertion size p is not exactly $(s-1) \cdot (f/s)^{h_0}$, we will have similiar result above. Due space limitation, we defer the details to [4].

As we can see, the larger the size of inserting subtree is, the lower the amortized cost each inserted node need to pay for. However, the decrease of the cost is roughly logarithmic in the increase of insertion size.

4.2 Virtual L-Tree

As an alternative to storing the L-Tree on disk, we can store only the leaf labels (with the XML nodes) because all the structural information of the L-Tree is implicit in the labels themselves. Indeed, if we examine closely the way labels are computed, we see that any leaf label is of the form:

$$N(x) = i_0 + i_1 \cdot (f-1)^1 + \cdots + i_{h-1} \cdot (f-1)^{h-1}$$

where i_0 is x's relative position in its siblings list, i_1 is x's parent's position among its siblings, and so on. In other words, the base $(f-1)$ digits of $N(x)$ provide an encoding of all the ancestors of x. Based on this observation, we can run the L-Tree incremental maintenance algorithm without the L-Tree. For example, in order to check if an internal node y satisfies the splitting criterion, it suffices to count how many leaf labels are in the range $[N(y), N(y) + (f-1)^h(y))$. If the leaf labels are maintained in a B-tree whose internal nodes also maintain counts, such range queries can be executed efficiently (in logarithmic time). Furthermore, once a splitting (virtual) node has been identified, the leaf labels corresponding to the s complete f/s-ary (virtual) trees can be computed easily and updated in place, on the labels identified by the range query. There is clearly a tradeoff between the extra computation required by the range queries and the storage space necessary for materializing the L-Tree.

5 Related Work

The problem of order-preserving labeling of an ordered list in the presence of random updates has been studied previously [8, 9, 16]. Our work has been inspired by these works, and extends to parameterize the problem, and propose a solution that can adjust the parameters according to different application requirements. Furthermore, we support batch insertions in L-Tree to improve the performance. [6] proposed a multi-level labeling scheme, which trades query cost to get better update cost.

Recently Cohen et al. [5] addressed the problem of designing persistent labels upon updates, by studying the minimum number of bits required to encode such a label, without considering the order of siblings. We approach the problem in a different perspective: we minimize the number relabelings upon updates given a label size of $\Theta(\log n)$.

Recently, several researchers have investigated various approaches for using labeling schemes to facilitate XML query processing [13, 12, 17, 7]. None of these schemes consider label maintenance in presence of updates.

6 Conclusions

We have presented a labeling scheme for maintaining the order of data items of an XML document. An L-Tree is introduced to assign and update labels of data items. An L-Tree can automatically adapt to uneven insertion rates in different areas of the XML document: in the areas with heavy insertion activity, the L-Tree adjusts itself by creating more slack between labels to better accommodate future insertions. We analyzed the amortized cost of incremental updates and derived a cost formula that enabled us to tune the tree parameters to achieve optimal performance in various application settings.

References

1. P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML-Schema to Relations: A Cost-Based Approach to XML Storage. In *Proc. ICDE*, 2002.
2. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *ACM SIGMOD*, 2002.
3. Yi Chen, Susan Davidson, Carmem Hara, and Yifeng Zheng. RRXS: Redundancy reducing XML storage in relations. In *Proc. VLDB*, 2003.
4. Yi Chen, George Mihaila, Sriram Padmanabhan, and Rajesh Bordawekar. Labeling your XML. Technical report, 10 2002.
5. E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic XML trees. In *ACM PODS*, June 2002.
6. Shunsuke Uemura Dao Dinh Kha, Masatoshi Yoshikawa. An XML Indexing Structure with Relative Region Coordinate. In *Proc. of the 17th ICDE*, pages 313–320, 4 2001.
7. David DeHaan, David Toman, Mariano Consens, and M. Tamer Ozsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *ACM SIGMOD*, 2001.
8. P. F. Dietz. Maintaining order in a linked list. In *Proc. 14th ACM STOC*, pages 62–69, 1982.
9. P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th ACM STOC*, pages 365–372. Springer, 1987.
10. Paul Dietz, Joel Seiferas, and J. Zhang. A tight lower bound for on-line monotonic list labeling. In *In Proc. 4th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 131–142, 1994.
11. Daniela Florescu and Donald Kossmann. Storing and querying XML data using RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.
12. T. Grust. Accelerating XPath location steps. In *ACM SIGMOD*, 2002.
13. Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *VLDBJ*, pages 361–370, 2001.
14. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDBJ*, pages 302–314, 1999.
15. I. Tatarinov, E. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *ACM SIGMOD*, 2002.
16. Athanasios K. Tsakalidis. Maintaining order in a generalized link list. *Acta Informatica*, 21, 1984.
17. C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *ACM SIGMOD*, 2001.