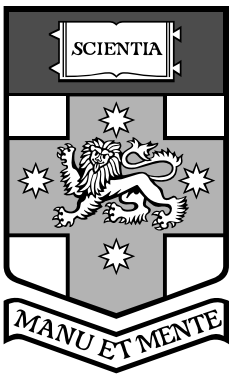
Efficient Ordering for XML Data

Damien K. Fisher Franky Lam William M. Shui Raymond K. Wong
School of Computer Science & Engineering
University of New South Wales
Sydney, NSW 2052, Australia
{damienf,flam,wshui,wong}@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-0316
June 2003

SCHOOL OF COMPUTER SCIENCE & ENGINEERING
THE UNIVERSITY OF NEW SOUTH WALES



Abstract

With the increasing popularity of XML, there arises the need for managing and querying information in this form. Several query languages, such as XQuery, have been proposed which return their results in document order. However, most recent efforts focused on query optimization have disregarded order. This paper presents a simple yet elegant method to maintain document ordering for XML data. Analysis of our method shows that it is indeed efficient and scalable, even for changing data.

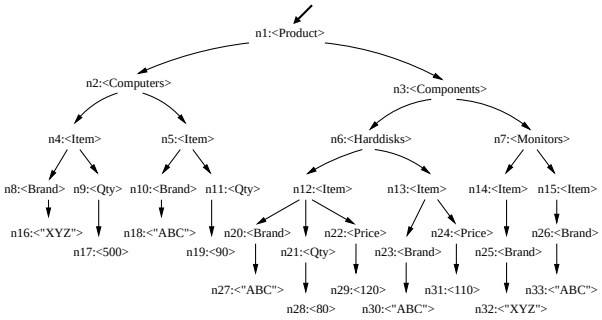


Figure 1: A small XML database

1 Introduction

In recent years XML [3] has emerged as the standard for information representation and exchange on the Internet. Since XML data is self-describing, XML is one of the most promising means to define semistructured data [1]. Although XML and semistructured data are similar, there are some differences [6, 12], and the most significant of these concerns data ordering [15]. In fact, researchers have already addressed the issue of order at the data model and query language level [6, 12] when adapting their work on semistructured data to XML. Although emerging standard XML query languages (e.g., XPath 2.0 [9] and XQuery [10]) require the output of queries to be in document order by default, little research work (from optimizing XML queries [23] to publishing data in XML [11]) has focused on efficiently maintaining results in document order.

To produce results in document order, without an efficient sort operator which will sort a set of nodes into document order, each query operator involved in query processing will have to preserve order. This limits the kinds of indexes that can be used, and hence the number of ways in which a query can be evaluated. Furthermore, methods from query optimization for unordered, semistructured data, e.g., [23], cannot be re-used to handle ordered data efficiently.

For example, consider the XML database shown in Figure 1 (where each node is represented by `oid:`$\langle$`value`$\rangle$) and the following XPath query on it:

```
//Harddisks/Item[Price<200][Brand="ABC"]
```

Ignoring order for the moment, we may employ indices and optimization techniques such as those proposed in the Lore database system [22, 23]. For example, we assume that a Tindex (a hash index on the text values) and a Vindex (a B+tree index on numeric values) have been created. Then optimal query plans may include the following two possibilities:

```
1. hash("ABC")/parent::Brand/
   parent::Item[parent::Harddisks][Price<200]

2. bptree(<,200)/parent::Price/
   parent::Item[parent::Harddisks][Brand="ABC"]
```

Let us look at the second plan in detail. A typical implementation of a B+tree would return a list of nodes based on the order of data values (i.e., $\langle n_{28}, n_{19}, n_{31}, n_{29} \rangle$) instead of their document order (i.e., $\langle n_{19}, n_{28}, n_{29}, n_{31} \rangle$). While this is not a problem in XPath 1.0, which has set semantics, it is a problem in XPath 2.0, which has list semantics. Without a sort function that can sort the output into document order, the output will be incorrect, as the XPath 2.0 specification states that all output must be returned

in its original document order. In other words, the result obtained by evaluating the second query plan would be $\langle n_{13}, n_{12} \rangle$ instead of the correct result $\langle n_{12}, n_{13} \rangle$, which is in document order.

However, there are no known algorithms to efficiently determine document order for changing data. Given any two nodes of an XML document, the worst case complexity of the naive ordering algorithm is $O(n)$ where n is the number of nodes in the document. As a result, sorting a set of nodes into document order will be very expensive. This may then mean that plans based on a simple top-down traversal may actually be more efficient than plans utilizing indices but requiring a sort, an outcome which is clearly undesirable.

As described above, an efficient sort operator increases the number of possible query plans significantly. For instance, for a particular plan, sorting could be performed on intermediate results if this were cheaper than performing the query in an unsorted fashion and then sorting the result at the end. In order for the query processor to select the optimal plan, it must be able to accurately estimate the cost of sorting a set of nodes. Therefore, we present empirical results for our algorithms in order to facilitate such estimation. We also discuss the application of our ideas to query optimization in Section 8.

To sort data into document order efficiently, this paper proposes two approaches. The first is the naive, obvious approach based on the pre-order traversal of the database, with extensions to handle dynamic databases. Our second approach employs a relabelling strategy which uses a degree of randomization to achieve good practical performance.

The rest of this paper is organized as follows. Section 2 provides a brief survey of work related to document ordering. Section 3 defines the terms used throughout this paper and provides some simple, but expensive, algorithms to determine document ordering. Section 4 defines a first approach to ordering, which is based on the pre-order traversal of a database. Section 5 discusses Bender's algorithm, which has good theoretical properties, in detail. Section 6 defines our second algorithm, which is based on an element of randomization. Section 7 presents empirical tests of our algorithms, and Section 8 discusses the application of our results to an important XML query optimization technique. Section 9 concludes the paper and presents some applications of our research to determining ancestor-descendant relationships.

2 Related Work

There is a dearth of research about ordered XML. The most recent work is that of Tatarinov et al [25], who considered storing ordered XML in a relational database system. They used a variety of techniques, essentially all of which are described below in other publications. Unfortunately, every one of the methods proposed suffers from potentially very poor update performance.

In a classical paper [7], Dietz and Sleator proved constant worst case time bounds for the order maintenance problem, building on previous results [8]. The order maintenance problem requires the maintenance of the total order upon elements of a list, subject to insertions, deletions, and comparisons. A substantially more elegant formulation of this result has recently been obtained [2]. It is obvious that the maintenance of the document order on an XML document corresponds to the order maintenance problem, and hence this result gives the best possible theoretical bounds on our problem. However, the $O(1)$ constant time algorithm presented in [7] is complicated; moreover, in database applications, even constant worst case performance can be unsatisfactory, due to excessive disk I/O. Additionally, there is an upper bound on the size of the database for which the results hold; [2] estimates this upper bound at approximately $430,000$ elements for a particular parameter selection. Hence, this algorithm is only an incomplete answer to the question of document ordering in large databases, where the number of nodes can easily run into the millions. However, we do examine in detail the amortized time algorithms of [2] in Section 5.

To the best of our knowledge, the closest other related work to this paper are efforts on determining ancestor-descendant relationships. For example, in [18], a document tree is viewed as a complete k-ary tree, where k is the maximum number of children of any node of the tree. "Virtual nodes" are used to make the tree complete. The identifier of each node is assigned according to the level-order tree traversal of the complete tree. It is then a simple matter to find the ancestors and children of a node using just the identifier. The problem with this approach is that as the arity and height of the complete tree increase, the identifiers may become huge. Also, if the constant k changes due to the insertion of new nodes, then all identifiers have to be recalculated from scratch. This makes the approach unrealistic for large, dynamic XML documents [20]. In [17], a labelling scheme is used, such that the label of a node's ancestor is a prefix of the node's label. The idea is similar to the Dewey encoding [4] that can be used to check parent-child relationships easily. Using this method takes variable space to store identifiers, and the time to determine the ancestor-descendant relationship is no longer constant, but linear in the length of the identifier. The lack of a bound on the identifier makes it difficult to guarantee that such an index will be practically useful on large databases.

A recent work has proposed the use of the position and depth of a tree node for indexing each occurrence of XML elements [26]. For a non-leaf node, the position is a pair of its beginning and end locations in a depth-first traversal order. The containment properties based on the position and depth are very similar to those of the extended preorder proposed in [20]. The performance and results of these approaches based on labelling schemes are consistent with the theoretical properties of labelling dynamic XML trees presented by [5]. This work proved that any general tree labelling scheme which answers the ancestor-descendant question must in the worst case have identifiers linear in the size of the database. As the ancestor-descendant problem can be related to the document ordering problem, this theorem also applies to the problem we address in this paper.

In contrast to the efforts above, our method is on sorting and maintaining data in document order. However, it can also be applied to the ancestor-descendant problem (as discussed in Section 8). Moreover, our method sacrifices a constant time bound for lower space bounds. Thus, while the theoretical limitations imposed by [5] still apply, we have attempted to minimize the appearance of the worst case. This is in contrast to other schemes, where the worst case can occur frequently.

Furthermore, in contrast to most of the labelling schemes above, our method does not impose significant performance overhead to frequent updates on the database.

Other work has been done in addressing or utilizing order information from schema or type information. [21] proposed a technique to specify and optimize queries on ordered semistructured data using automata. It uses automata to present the queries and optimize the query using query typing and automata unnesting. On the other hand, in response to the ordering issue addressed in [6, 12], [15] extended dataguides [14] and proximity search [13] to take order into consideration.

3 Formal Definitions

3.1 Data Model

We will follow a common convention in the literature and model an XML document by a labelled, ordered, unranked tree. Of course, the order of the attributes of an element is undefined in XML; we will adopt the convention that the attributes come before the other children of the element, in some arbitrary, but fixed, order. This will have no impact on the results of this paper, as the document ordering between attribute nodes of an element is by definition arbitrary anyway. As we do not need to distinguish between elements, attributes, processing instructions, and other kinds of XML nodes in this paper, this model is suitable for our purposes; in fact, we do not even need to make the tree labelled, as labels are irrelevant when considering document order.

Accessor	Description
PARENT(x)	Parent of x
NEXT-SIBLING(x)	Next sibling of x
PREV-SIBLING(x)	Previous sibling of x
FIRST-CHILD(x)	First child of x
PREORDER-PREVIOUS(x)	Node before x in document order
PREORDER-NEXT(x)	Node after x in document order

Table 1: Constant time accessor functions

The document ordering on an XML document is the total ordering defined by a pre-order traversal of the corresponding tree [10]. In this paper, we will denote the document ordering by $<$. As the document ordering between attribute nodes of an element is implementation defined, for our purposes we can simply choose an arbitrary ordering amongst the attributes in our ordered tree representation, and use this as the document ordering. Figure 1 gives the tree representation of a sample XML document. As some examples of document ordering, in this figure we have $n_2 < n_8$, $n_{28} < n_{22}$, and $n_{10} < n_6$.

Throughout this paper, we impose a specific physical data model on our XML database, which gives a set of accessor functions which take constant time to run. We have carefully chosen this set of accessors so that it is likely that any reasonable native XML database would need to be able to implement these accessors in constant time. The accessors needed are summarized in Table 1. Of these accessors, PREORDER-PREVIOUS and PREORDER-NEXT can easily be implemented in terms of the others, although in worst cast time linear in the depth of the database. In practice, however, the depth of an XML database is extremely small, and we can assume that these accessors will essentially run in constant time. In our implementation, we do not maintain these accessors explicitly, instead relying on the observed properties of real XML documents.

We assign to each node a unique identifier, the *object identifier*, or *oid*. Throughout this document, object identifiers will be represented by integers of word size (32 bits on many modern machines). We stress that an ordering on the object identifiers of two nodes x and y does not necessarily correspond to the document ordering on x and y.

This paper deals with document ordering in dynamic XML databases. For simplicity, we assume that each insertion or deletion only adds or removes a single leaf node. The insertion or deletion of entire subtrees can be modelled as a sequence of these atomic operations.

3.2 Naive Sorting Algorithms

Algorithm 1 is the obvious naive algorithm for determining the relative ordering of two nodes in an XML database $\mathcal{D}$. This algorithm has worst case time complexity linear in the number of nodes in the database. When comparing nodes x and y, the algorithm finds nodes a, b, and c, such that a and b are children of c, a is an ancestor of x, and b is an ancestor of y. Then, one can determine whether $x < y$ by determining whether a comes before b in the list of children of c.

Suppose we have a set of nodes S from a database $\mathcal{D}$ that we wish to sort into document order. If we use a standard sorting algorithm with the comparison function given by Algorithm 1, we would have worst case time complexity $O(|S||\mathcal{D}| \log |S|)$. However, it is possible to generalize Algorithm 1 to handle n nodes at once, in which case the complexity drops to $O(|S||\mathcal{D}|)$. The reason for this drop in complexity is because examining the common ancestors of all nodes in S simultaneously can save operations. Due to space constraints, the algorithm has been omitted from the paper.

Algorithm 1 Relative document ordering of two nodes n_1 and n_2, using no indices.

```
NAIVE-ORDER-CMP(n_1, n_2)
 1 if n_1 = n_2 then
 2     return n_1 = n_2
 3 end if
 4 A_1 ← [n_1, PARENT(n_1), PARENT(PARENT(n_1)), ..., ROOT]
 5 if n_2 ∈ A_1 then
 6     return n_2 < n_1
 7 end if
 8 A_2 ← [n_2, PARENT(n_2), PARENT(PARENT(n_2)), ..., ROOT]
 9 if n_1 ∈ A_2 then
10     return n_1 < n_2
11 end if
12 Find the smallest i such that
   A_1[|A_1| − i] ≠ A_2[|A_2| − i]
13 m_1 ← A_1[|A_1| − i]
14 m_2 ← A_2[|A_2| − i]
15 Determine the ordering between the siblings
   m_1 and m_2 by traversing through all
   their siblings.
16 if m_1 < m_2 then
17     return n_1 < n_2
18 else
19     return n_2 < n_1
20 end if
```

4 A Naive Approach: Refactoring

In this section, we define an obvious strategy for handling document order. The basic idea of this approach is to label the nodes as in a pre-order traversal. While this is trivial on a static database, it is not immediately obvious how to extend this algorithm to handle changing data, particularly data that changes frequently. We will first describe the basic idea, and then present refinements which allow the average case to execute more quickly.

4.1 Basic Idea

We associate with each node a numeric identifier (the *document ordering identifier* or *docid*). In practice, we make the size of the docid equal to the word size of the machine, although the amoount of storage needed depends on both how many nodes are in the database, and the quality of the document ordering index algorithm (better algorithms should handle more nodes with less storage). Given a node n, we define a function `DOCID` which returns its docid. For simplicity, we will ignore any disk reads necessary to fetch the docid for a given node. If this information is stored directly in the record for each node, then this assumption makes sense, as they will be loaded into memory whenever the corresponding node is.

Let us first consider the simple case of a static database $\mathcal{D}$. In this case, the document ordering index is initialized by performing a pre-order traversal of the database, and assigning successive docids to successive nodes. Then, to compare two nodes x and y, we merely need to compare the relative order of their docids.

This method can be easily extended to the case of a database in which all nodes being inserted are inserted at the end of the database (in document order). We assign to each new node the next docid after the docid of the last node in the database. When a node is deleted from the database, we do nothing (this results in gaps being left between docids).

However, this approach breaks down when nodes can be inserted anywhere in the database. Suppose we insert a new node n between two sorted nodes x and y. If there is a gap between the docids of x and y (due to a previously deleted node), we can reuse that docid for n. If there is no gap, we instead set the docid of n to that of x. Algorithm 2 summarizes this procedure. As discussed in Section 3, the worst case of `PREORDER-PREVIOUS` and `PREORDER-NEXT` is linear in the depth of the database. In the latter case, for bulk insertions we can reduce the overall cost by using only one traversal for the entire set of nodes being inserted. Table 2 tabulates the worst case time complexities for frequent database operations.

Algorithm 2 Maintenance of the document ordering index during the insertion of a new node n.

```
INSERT-MAINTAIN(n)
 1 x ← PREORDER-PREVIOUS(n)
 2 if x is the last node in document order then
 3    DOCID(n) ← DOCID(x) + 1
 4 else
 5    y ← PREORDER-NEXT(n)
 6    if DOCID(y) > DOCID(x) then
 7       DOCID(n) ← ⌈(DOCID(x)+DOCID(y))/2⌉
 8    else
 9       DOCID(n) ← DOCID(x)
11    end if
12 end if
```

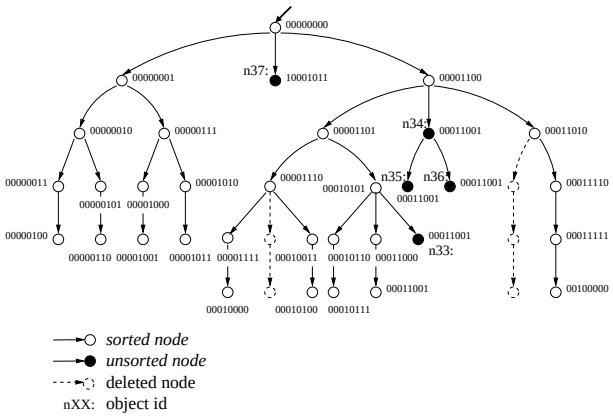


Figure 2: An instance of the document ordering index

operation	*overhead*
inserting one node	$O(d)$
inserting a subtree of n nodes	$O(n+d)$
moving a leaf node	$O(d)$
updating a node's value	$O(1)$
deleting node	$O(1)$

Table 2: Overhead while updating nodes

Figure 2 demonstrates the state of the document ordering index on the database in Figure 1, after several insertions and deletions have been performed. Deleted nodes are represented using dotted edges, and newly inserted nodes are represented using solid circles. We call a subtree consisting entirely of nodes with the same docid *unsorted subtree*; in the figure, the subtree rooted at node n_{34} is an unsorted subtree.

4.2 Comparing document order between two nodes

Algorithm 3 Find the relative document order of nodes n_1 and n_2 using the document ordering index

```
ORDER-CMP(n1, n2)
 1 if DOCID(n1) < DOCID(n2) then
 2     return n1 < n2
 3 elif DOCID(n2) > DOCID(n1) then
 4     return n2 < n1
 5 else
 6     return NAIVE-ORDER-CMP(n1, n2)  (see note)
 7 end if
```

Note: In line 6, we can improve on the call `NAIVE-ORDER-CMP`. For instance, consider Figure 2. If we sort nodes n_{33} and n_{36}, once we find that n_{35} is an ancestor of n_{33} we can terminate the search, as $\texttt{DOCID}(n_{35}) < \texttt{DOCID}(n_{36})$.

As mentioned before, we require a near constant time method for comparing the document order of two nodes. Algorithm 3 determines the relative document ordering of two nodes. The most expensive

case in this algorithm is when both nodes to be compared are nodes with the same docid, as this falls back on a slightly faster variant of the naive algorithm. In all other cases, we get the comparison almost for free. If we let the maximum depth of an unsorted subtree (that is, a subtree of nodes with the same docid) be d, and the maximum breadth of an unsorted subtree be b, then in the worst case we need to execute $2d + b$ operations. Thus, this algorithm performs extremely well when the size of the unsorted subtrees in the database are reasonably small — all further refinements to this algorithm focus on ensuring that this is so.

4.3 Enhancements

We present here some enhancements which can make the above algorithm practical.

4.3.1 Gaps

One simple enhancement is to leave gaps between docids during construction or reconstruction of the ordering index to allow room for nodes to be inserted. However, the gaps cannot be too large, or else we will soon hit the upper limit on docids.

A sensible way to leave gaps is to use structural information to determine the gap size. In particular, a DTD or XML Schema will place restrictions on the structure of the document, which can assist us in determining the size of the gaps. For instance, if we know that a node n always has only one child, which must be a leaf node, then we immediately know that we only have to leave space for one node under n.

Apart from precise structural hints as mentioned above, we can utilize schema information to estimate the gap size. For instance, if we know that a certain type of element has zero or more children, which must all be leaf nodes, we can keep track of the mean number of children each element of this type has and use this as the gap size.

4.3.2 Regions

During the process of document index reconstruction, the whole database needs to be locked, and this will have a significant impact on the performance of the database system. To minimize the impact of the index reconstruction, we can add a *region identifer* into the document ordering index, where the maximum rumber of regions is a database parameter. Essentially, before comparing the docids of nodes x and y, we first check whether they are in the same region. If they are not, then we can use the ordering on regions to answer the query. Thus, if our docid is an integer, then a region can be thought of as the most significant bits of this integer. When the index is rebuilt, it is only rebuilt on a particular region; this helps to amortize the cost of rebuilding indices over a longer period of time, and increases database availability.

4.3.3 Refactoring

To reduce the chance of having large unsorted subtrees, we can refactor unsorted subtrees into several smaller unsorted subtrees by shifting the docid of neighbouring sorted nodes into the unsorted area. Figure 3 gives an example of how this strategy works.

Suppose we are comparing the document order of two unsorted nodes n_1 and n_2, such that $\texttt{DOCID}(n_1) = \texttt{DOCID}(n_2)$. We scan, in document order, to the left and right of n_1 and n_2 in exponentially increasing ranges, until we find nodes n_1' and n_2' such that $\texttt{DOCID}(n_1') < \texttt{DOCID}(n_1) = \texttt{DOCID}(n_2) < \texttt{DOCID}(n_2')$. We then relabel this range of nodes so that they are evenly distributed, i.e., if there are n nodes, we set the docid of the i-th node in the range equal to $\texttt{DOCID}(n_1) +$

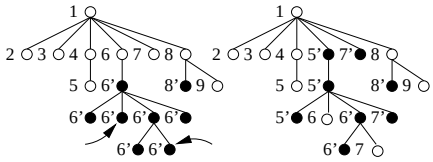


Figure 3: Example of Refactoring

$\lfloor \frac{i}{n}(\texttt{DOCID}(n_2) - \texttt{DOCID}(n_1)) \rfloor$. We note that this does not assign a unique docid to each node, but it does minimize the number of nodes on each docid, if we restrict the docids to those in the range.

The advantage of this algorithm is its simplicity. It is also, in a primitive sense, dynamic, because relabelling will only happen in areas where document ordering comparisons are actually occurringd. Its most significant shortcoming, however, is that it changes read operations into write operations; this means that a transaction which would otherwise be read-only may be exscalated to a write transaction. This obviously could have a significant impact on overall database performance. However, it is still possible that this algorithm could be used in single user environments.

5 Bender's Algorithm

In this section, we provide a brief overview of the algorithm of Bender et al [2]. Theoretically, this algorithm has excellent time complexity, but in practice there are some limitations. The basic idea of this algorithm is to assign to each node of the tree an integral identifier, which we call its *tag* (or docid, in keeping with the previous section's terminology), such that the natural ordering on the tags corresponds to the document ordering on the nodes. During insertions and deletions, it is obviously necessary to at some point relabel surrounding nodes, if there is no space to assign a tag for the new node. The algorithm guarantees that such relabellings cost only constant amortized time.

Let $u \in \mathbb{N}$ be the tag universe size, which we assume to be a power of two, and consider the complete binary tree $\mathcal{B}$ corresponding to the binary representations of all numbers between 0 and $u-1$. Thus, the depth of the tree is $\log |\mathcal{U}|$, and the root-to-leaf paths are in one-to-one correspondence with the interval $\mathcal{I} = [0, u-1] \subseteq \mathbb{Z}$; more generally, any node of the tree corresponds to a sub-interval of $\mathcal{I}$. When our database has n nodes, this tree will have n leaf nodes used, corresponding to the tags assigned to the nodes in the database. For a node $n \in \mathcal{D}$, we write $\texttt{DOCID}(n) \in \mathcal{B}$ for its numeric identifier. For a node in the identifier tree, we define its density to be the proportion of its descendants (including itself) which are allocated as identifiers.

When inserting a new node n between two nodes x and y, we proceed as follows. First, if $\texttt{DOCID}(x) + 1 \neq \texttt{DOCID}(y)$, we set $\texttt{DOCID}(n) = \frac{1}{2}(\texttt{DOCID}(x) + \texttt{DOCID}(y))$. Otherwise, we consider the ancestors of x, starting with its immediate parent and proceeding upwards, and stop at the first ancestor a such that its density is less than T^{-i}, where T is a constant between 1 and 2, and i is the distance of a from x. We then relabel all the nodes which have identifiers in the sub-range corresponding to a.

Bender et al [2] prove that the above algorithm results in an $O(\log n)$ amortized time algorithm. We omit the proof, but quote the following results. Firstly, for a fixed T the number of bits used to represent a tag is $\log u = \frac{\log n + 1}{1 - \log T}$. Intuitively, then, we would expect that as T decreases, the amortized cost of insertions decreases, because more bits are used to represent the tags, and hence there are larger gaps. This can be verified from the fact that the amortized cost of insertions is $(2 - \frac{T}{2}) \log u$.

Practically speaking, of course, we wish to fix $\log u = W$, where W is the word size of the machine. In this case, there is a trade-off between the number of nodes that can be stored and the value T. Another practical difficulty is that as more nodes are inserted into the database, the average gap size decreases. At some point, thrashing will occur due to the fact that many nodes are frequently

relabeled, and the theoretical properties of the algorithm fail. To alleviate this problem, Bender et al make the following small refinement: instead of making T constant, at each point in the algorithm we can can take T as the smallest possible value that causes the root node to not overflow. They show experimentally that this modification yields good results.

From the above $O(\log n)$ amortized time algorithm, we can obtain an amortized $O(1)$ algorithm using a standard technique (see, for instance, [7]). We partition the list of nodes into $\Theta(n/\log n)$ lists of $\Theta(\log n)$ nodes, and maintain ordering identifiers on both levels. When one of the sub-lists overflows, we split it into two sub-lists, and insert the new sub-list into the list of lists. It is easy to show that this removes the logarithmic factor.

While this algorithm obviously has very desirable theoretical properties, in the context of disk-bound lists there are several problems. Firstly, in order to get amortized constant time worst case bounds, we need to maintain quite a bit of extra information for the two-level list structure. At a minimum, we must maintain the top level linked list, and for each node we must store a pointer to the sub-list it belongs to. Additionally, to perform ordering between two nodes one would have to lookup the tags of their sub-lists, which is an unavoidable indirection. This can have an adverse impact on paging, and possibly incur many expensive disk reads. Our experimental results show that it is this last effect that has the most serious impact on the constant time algorithm.

6 A Randomized Algorithm

In this section we present an alternative probabilistic algorithm which performs very well in practice. To illustrate how the algorithm works, suppose we have an ordered list of objects $x_1, x_2, \ldots$, and to each x_i we assign a tag (as in previous algorithms) to determine relative ordering. We define $g_i = \texttt{DOCID}(x_{i+1}) - \texttt{DOCID}(x_i)$ to be the gap between the i-th node's tag and its successor's tag.

Suppose we wish to insert a new node x_0 at the beginning of the list. We initialize x_0's tag to $\lfloor \frac{\texttt{DOCID}(x_1)}{2} \rfloor$. We then iterate through $x_1, x_2, \ldots$, adjusting the gap sizes as follows. We draw a random number g from some fixed discrete probability distribution ranging over the positive integers. If the gap we are currently considering (say g_i) is smaller than g, then we set $\texttt{DOCID}(x_{i+1}) \leftarrow \texttt{DOCID}(x_i) + g$. We continue with this procedure on successively higher values of i until we find a gap larger than the random number we sample. This handles the case where insertions happen at the beginning of the list. Insertions in the middle are handled by two traversals, one forward through the list (as above), and one backwards through the list, in a completely symmetric fashion.

While it is clear that this algorithm will preserve the document ordering properties of the tags, it is not at all clear why this algorithm should work quickly. Suppose that upon the insertion of a new node, the algorithm relabels n nodes. Then it is easy to see that the tag of x_n will be the sum of n random numbers from our probability distribution, because g_i for $i < n$ will have been drawn from this distribution. However, we cannot say anything about g_n. Nevertheless, we make the assumption that, once the algorithm has run for some long length of time, it will be the case that the tag of the i-th node x_i will be the sum of i random numbers from our probability distribution.

Of course, this assumption is not valid in the general case. To alleviate this problem, we will, as described below, choose a probability distribution which favors small gap sizes. This means that after the first n nodes have been relabelled, even though g_n will not have been sampled from the distribution, it will still be small and hence one of the more likely values from the probability distribution. This means that the effect of these "unsampled" gaps will have a negligible impact on the rest of this analysis.

Thus, with the above assumption in mind, we can now restate the algorithm as follows. Upon the insertion of a new node, we progressively choose increasing values of i, and for each i we choose a new $\bar{x}_i$, sampled from the cumulative probability distribution. We terminate the search if $\bar{x}_i \geq x_i$. We now must show that this algorithm terminates in a reasonable amount of time.

Suppose that X and Y are independent and identically distributed random variables. Then it is clear that $P(X \geq Y) = P(Y \geq X)$, by independence. Hence:

$$\begin{aligned} P(X \geq Y) + P(X < Y) &= 1 \\ P(Y \geq X) + P(Y > X) &= 1 \\ P(Y \geq X) &= \frac{1 + P(X = Y)}{2} \\ P(X \geq Y) &\geq \frac{1}{2} \end{aligned}$$

Thus, at the i-th step of the algorithm, there is at least a 50% chance that the algorithm will terminate. Hence, the probability of the algorithm *not* terminating after i steps is at most 2^{-i}. Therefore, in practice, the algorithm should terminate fairly quickly. In fact, it is easy to see that on average we would expect at least four relabellings. In practice, the number of relabellings will be higher due to the failure of our assumption; however, our experiments show that the algorithm still has good performance.

The question remains as to what probability distribution we choose to use. We choose to use the exponential probability distribution, given by probability density function:

$$f(x) = \lambda e^{-\lambda x}$$

Of course, this is a continuous distribution, whereas we require a discrete distribution, because gap sizes must be integral and non-negative. Hence, we actually use the distribution defined as:

$$P(g = i) = \int_{i-1}^{i} f(x)dx$$

For our experiments, we used $\lambda = \ln 2$. We chose the exponential distribution (and this value of λ) because while the above algorithm works well in theory, it assumes implicitly that there is no upper bound on the size of tags. Of course, in practice we want tag values to remain small. Hence, we do not want a probability distribution which yields large gaps with high probability. Additionally, the assumption we made in the above analysis can only be satisfied by a distribution such as the exponential distribution, which generates small values with very high probability.

The algorithm is given in pseudo-code in Figure 4. The function `GET-GAP` obtains a random sample from the gap distribution we defined above. We note one potential problem with our algorithm, which does not seem to be significant in practice. It is possible that during the relabelling process, the algorithm will hit the greatest or least possible tag value. In this case, we simply allow multiple nodes to have the same tag value, and use Algorithm 1 in this case to determine ordering. This case is unlikely to occur in practice, because the number of nodes present in the database would have to approach the total number of docids available. On the other hand, the fact that the algorithm makes only one pass of the range that is relabelled (as opposed to the two passes of Bender) will make a significant practical difference in a disk-bound data structure such as a database, as can be seen in our experimental results.

7 Experimental Results

We performed our experiments using the DBLP database. All experiments were run on a dual processor 750 MHz Pentium III machine with 512 MB RAM and a 30 GB, 10000 rpm SCSI hard drive. We tested both Bender algorithms (the $O(\log n)$ and $O(1)$ variants), the simple refactoring

Algorithm 4 Updating the document ordering tags using the randomized algorithm, upon inserting a node n.

```
RANDOM-UPDATE(n)
 1 n_1 ← PREORDER-PREVIOUS(n)
 2 n_2 ← PREORDER-NEXT(n)
 3 DOCID(n) ← ⌊(DOCID(n_1)+DOCID(n_2))/2⌋
 4 n' ← n
 5 while n_1 ≠ NIL do
 6    g ← GET-GAP()
 7    if DOCID(n) − DOCID(n_1) < g then
 8       DOCID(n_1) ← max{DOCID(n) − g, 0}
 9    else
10       break
11    end if
12    n ← n_1
12    n_1 ← PREORDER-PREVIOUS(n_1)
13 end while
14 n ← n'
15 while n_2 ≠ NIL do
16    g ← GET-GAP()
17    if DOCID(n_2) − DOCID(n) < g then
18       DOCID(n_2) ← min{DOCID(n) + g, |U| − 1}
19    else
20       break
21    end if
22    n ← n_2
23    n_2 ← PREORDER-NEXT(n_2)
24 end while
```

algorithm of Section 4, and the randomized algorithm of Section 6.

For each algorithm, we inserted 100, 1000, and 10000 DBLP records into a new database. The insertions were done in two stages. The first half of the insertions were appended to the end of the database, and hence simulated a bulk load. The second half of the insertions were done at random locations in the database, that is, if we consider the document as a linked list in document order, the insertions happened at random locations throughout the list; this stage simulated further updates upon a pre-initialized database. While the inserts were distributed over the database, at the physical level the database records were still inserted at the end of the database file. This resulted in a database which was not clustered in document order, which meant that traversing through the database in document order possibly incurs many disk accesses. We hypothesize that while many document-centric XML databases will be clustered in document order, *data-centric* XML databases will not be, as they will most likely be clustered through the use of indices such as B-trees on the values of particular elements. Hence, our tests were structured to simulate these kinds of environments, in which the document ordering problem is more difficult.

At the end of each set of insertions, there were n elements in the database, where $n \in \{100, 1000, 10000\}$. We then additionally performed $10n$ and $100n$ reads upon the database. Each read operation chose two random nodes from the database and compared their document order. The nodes were not chosen uniformly, as this does not accurately reflect real-world database access patterns. Instead, in order to emulate the effect of "hot-spots" commonly found in real-world database applications, we adopted a normal distribution with mean $\frac{n}{2}$ and variance $\frac{n}{10}$.

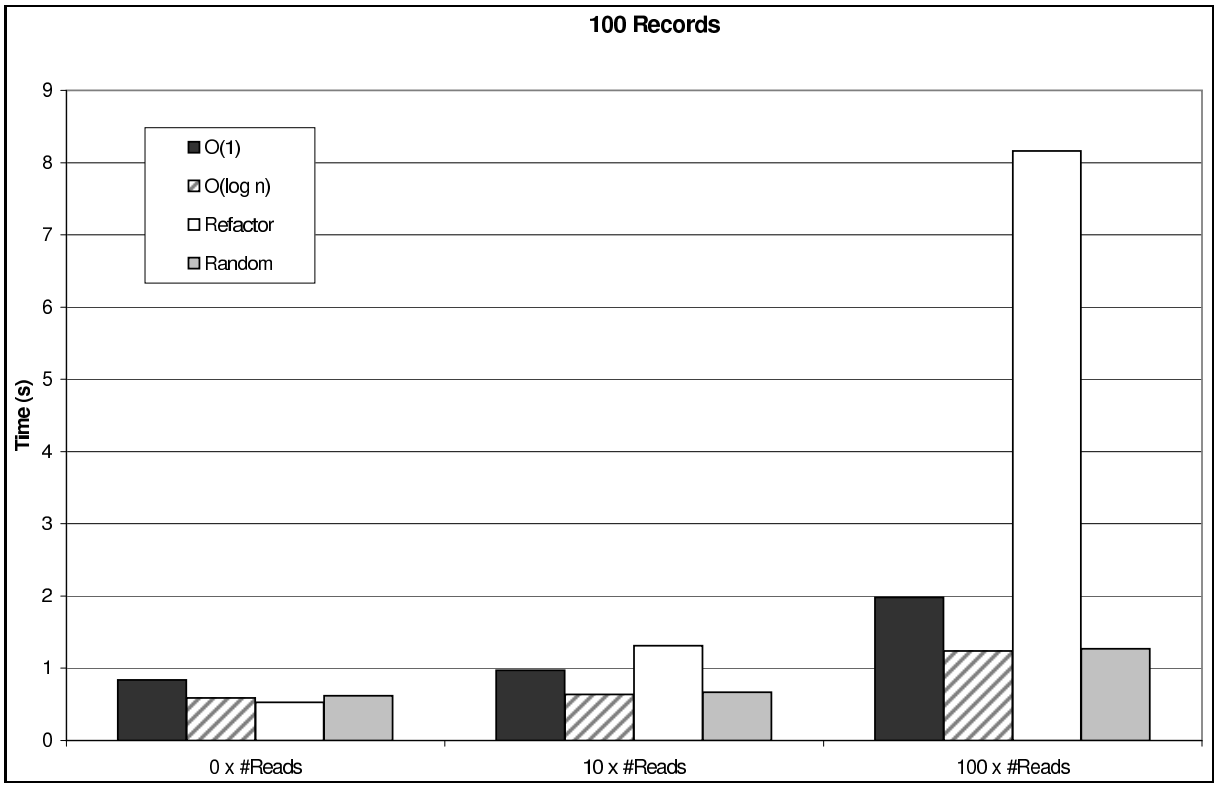


Figure 4: Results for database of 100 records

Figures 4 through 6 show the results from our experiments. There are several interesting things to note from our experiment. Firstly, the $O(1)$ algorithm of Bender is easily slower than the $O(\log n)$ algorithm. The relative performance gap becomes more noticeable as the number of reads increases, and hence is due to the extra level of indirection imposed in the comparison function by the $O(1)$

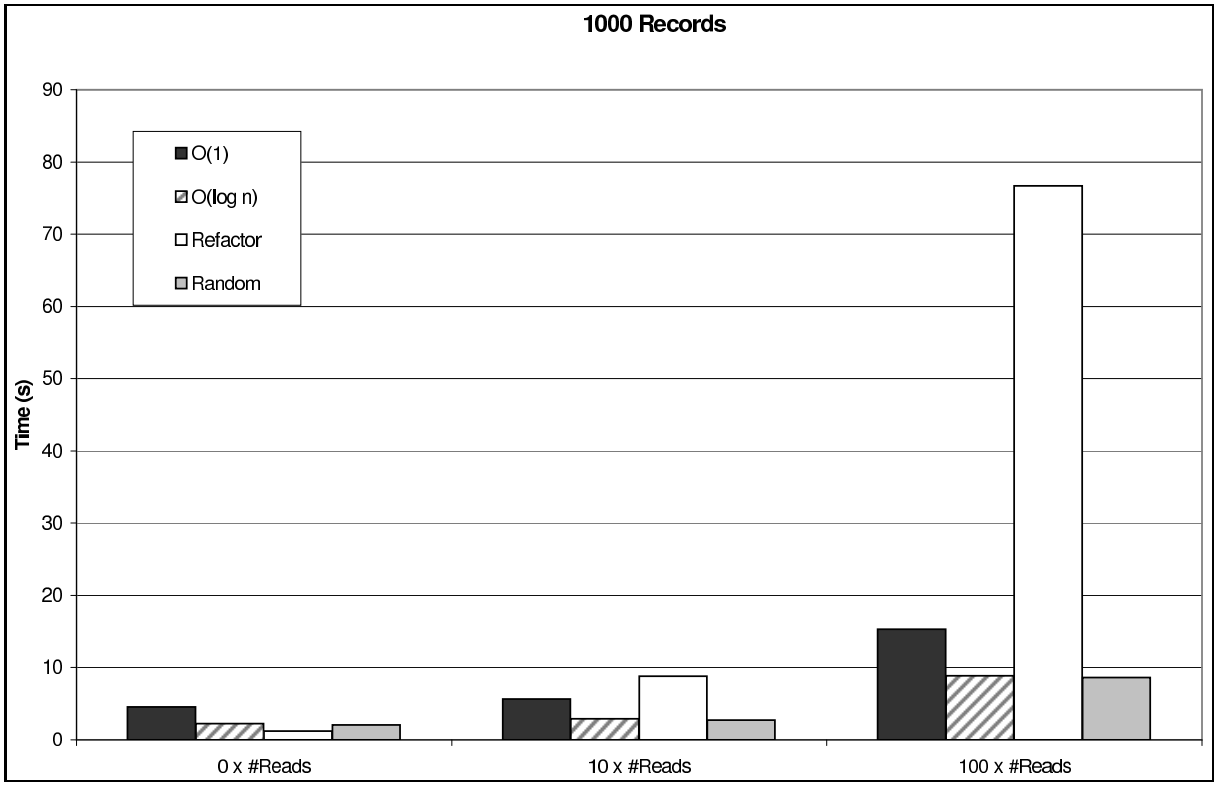


Figure 5: Results for database of 1000 records

algorithm. Secondly, it is clear that the refactoring algorithm is of little use in high read scenarios. Finally, the performance of our randomized algorithm is clearly more encouraging, as it is ahead of all the other algorithms by a comfortable amount in all tests. The performance gap between the other algorithms and the randomized algorithm also increases as the number of records increases, which indicates that our algorithm will perform better on huge databases than the others.

8 An Application: Ancestor-Descendant Relationships

As mentioned Section 1, our method can be applied to efficiently determine ancestor-descendant relationships. The key insight is due to Tarjan [24], who noted that the *ancestor query problem* can be answered using the following fact: for two given nodes x and y of a tree $\mathcal{T}$, x is an ancestor of y if and only if x occurs before y in the preorder traversal of $\mathcal{T}$ and after y in the postorder traversal of $\mathcal{T}$.

We note that while we have framed our discussion in terms of document order (that is, preorder traversal), our results could be equally well applied to the postorder traversal as well. Therefore, by maintaining *two* indices, one for preorder traversal, and one for postorder traversal, which allow ordering queries to be executed quickly, we can determine ancestor-descendant relationships efficiently.

It is well-known that ancestor-descendant relationships can be used to evaluate many path expressions, using region algebras. In fact, some native XML databases, such as TIMBER [16], use the above trick by storing numerical identifiers giving the relative preorder and postorder position for each node. However, to the best of our knowledge, this is the first work to address the efficient maintenance of these identifiers in the context of dynamic databases. As an example of how structural and range queries can be answered efficiently, consider the XPath:

```
//Item[.//Price > 200]
```

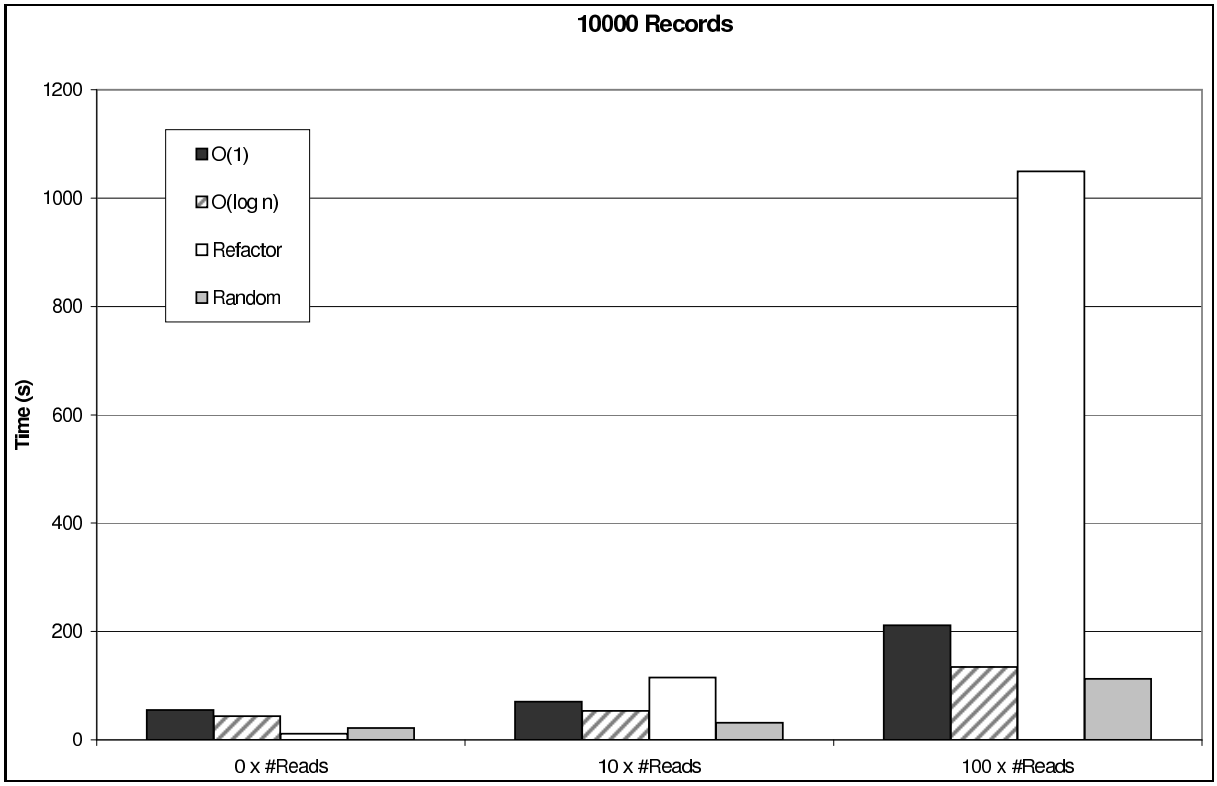


Figure 6: Results for database of 10000 records

This query can be answered by the following plan (where we adopt the definitions of Section 1). Taking:

$$\begin{aligned} \mathcal{S}_1 &\leftarrow \texttt{hash}(\text{"Item"}) \\ \mathcal{S}_2 &\leftarrow \texttt{hash}(\text{"Price"}), \text{ and} \\ \mathcal{S}_3 &\leftarrow \texttt{bptree}(>, 200) \end{aligned}$$

We then find $\mathcal{M}$ where:

$$\begin{aligned} &(\forall n \in \mathcal{M})(\mathcal{M} \subseteq \mathcal{S}_1)(\exists x \in \mathcal{S}_2)(\exists y \in \mathcal{S}_3) \\ &\quad (n <_{pre} x <_{pre} y \vee n >_{post} x >_{post} y) \end{aligned}$$

In the above, $<_{pre}$ is an ordering comparison in preorder traversal, and $>_{post}$ is an ordering comparison in postorder traversal. We can then sort $\mathcal{M}$ into document order using the results in this paper.

9 Conclusions

In this paper, we have presented the first analysis of practical algorithms for maintaining an index for document order in dynamically changing databases. Having such an index will prove invaluable in optimizing queries over XML databases. We have shown that the straightforward approach, refactoring, scales very poorly, but additionally that even theoretically good results can have surprising practical performance. This is best demonstrated by the relatively poor performance of the $O(1)$ time algorithm

of Bender. Taking into account practical considerations, we have developed a simple algorithm that performs better than all other known algorithms, and in particular scales in a significantly better fashion. Finally, we note that while we have couched our discussion in terms of native XML database systems, our results could be adapted to handling XML data in relational database systems.

There are many open research topics left in this area. We intend to investigate the utilization of schema information to reduce the number of nodes for which document ordering information needs to be obtained. A more significant topic would be to extend the work of Lerner and Shasha [19] to handle ordered XML data. This is now possible due to the fact that we have developed an efficient ordering operator.

References

[1] Serge Abiteboul. Querying semi-structured data. In *Proceedings of ICDT*, volume 1186 of *Lecture Notes in Computer Science*, pages 1–18, Delphi, Greece, 8–10 January 1997. Springer.

[2] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA 2002)*, volume 2461 of *Lecture Notes in Computer Science*, pages 152–164, Rome, Italy, September 17–21 2002.

[3] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (second edition). `http://www.w3.org/TR/2000/REC-xml-20001006`, 2000.

[4] Online Computer Library Center. Introduction to the Dewey Decimal Classification. `http://www.oclc.org/oclc/fp/about/about_the_ddc.htm`.

[5] Edith Cohen, Haim Kaplan, and Tova Milo. Labeling Dynamic XML Trees. In *Proceedings of PODS*, pages 271–281, New York, June 3–5 2002. ACM Press.

[6] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(11–16):1155–1169, May 1999.

[7] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 365–372. ACM Press, 1987.

[8] Paul F. Dietz. Maintaining order in a linked list. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 122–127, 1982.

[9] W3C Working Draft. XML Path Language (XPath) 2.0. *`http://www.w3.org/TR/2002/WD-xpath20-20021115`*, November 2002.

[10] W3C Working Draft. XQuery 1.0: An XML query language. *`http://www.w3.org/TR/2002/WD-xquery-20021115`*, November 2002.

[11] Mary Fernández, Yana Kadiyska, Dan Suciu, Atsuyuki Morishima, and Wang-Chiew Tan. SilkRoute: A framework for publishing relational data in XML. *ACM Transactions on Database Systems*, 27(4):438–493, December 2002.

[12] R. Goldman, J. McHugh, and J. Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *Workshop on the Web and Databases (WebDB '99)*, pages 25–30, 1999.

[13] Roy Goldman, Narayanan Shivakumar, Suresh Venkatasubramanian, and Hector Garcia-Molina. Proximity Search in Databases. In *Proceedings of VLDB*, pages 26–37. Morgan Kaufmann Publishers, 1998.

[14] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of VLDB*, pages 436–445, East Sussex - San Francisco, August 1998. Morgan Kaufmann.

[15] Roy Goldman and Jennifer Widom. Summarizing and Searching Sequential Semistructured Sources. Technical Report, Stanford University, 2000.

[16] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native XML database. *VLDB Journal: Very Large Data Bases*, 11(4):274–291, 2002.

[17] W. Eliot Kimber. HyTime and SGML: Understanding the HyTime HYQ Query Language. Technical Report Version 1.1, IBM Corporation, August 1993.

[18] Yong Kyu Lee, Seong-Joon Yoo, Kyoungro Yoon, and P. Bruce Berra. Index structures for structured documents. In *ACM International Conference on Digital Libraries*, pages 91–99, 1996.

[19] Alberto Lerner and Dennis Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *Proceedings of VLDB*, to appear, 2003.

[20] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of VLDB*, pages 361–370, 2001.

[21] Hartmut Liefke. Horizontal query optimization on ordered semistructured data. In *WebDB (Informal Proceedings)*, pages 61–66, 1999.

[22] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Wid om. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, September 1997.

[23] Jason McHugh and Jennifer Widom. Query optimization for XML. In *Proceedings of VLDB*, pages 315–326, 1999.

[24] R. Endre Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9(3):355–365, December 1974.

[25] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 204–215. ACM Press, 2002.

[26] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, 2001.