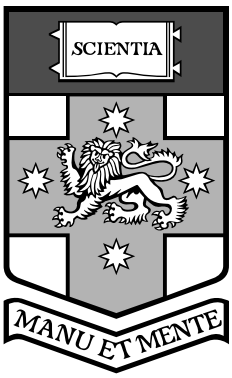
Fast Ordering for Changing XML Data

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-0317
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 either disregarded order or proposed a static labelling scheme in which update issues are not addressed. Based on the preliminary results from our previous work, this paper presents a fast method to maintain document ordering for changing XML data. Analysis of our method shows that it is more efficient and scalable than our previously proposed method as well as other related work, especially under various scenarios of updates.

1 Introduction

In recent years XML [4] 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 [8, 14], and the most significant of these concerns data ordering [17]. In fact, researchers have already addressed the issue of order at the data model and query language level [8, 14] when adapting their work on semistructured data to XML. Although emerging standard XML query languages (e.g., XPath 2.0 [26] and XQuery [27]) require the output of queries to be in document order by default, most research work, from optimizing XML queries [24] to publishing data in XML [12], has ignored the issue of 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., [24], cannot be re-used to handle ordered data efficiently.

While different XML labeling schemes (mainly for determining ancestor-descendant relationship) have been proposed to annotate XML nodes with their start positions or preorder numbers (e.g., [2, 6, 22, 28]), 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 linear in 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 mentioned 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.

Our previous work [13] examined three algorithms which sort data into document order efficiently. The first two are those of Bender *et al* [3], which have the advantage of good worst case theoretical performance. We have found some practical problems with the algorithms, discussed later in this paper. We also presented a simple randomized algorithm which appears quite fast in practice, but suffers from the drawback of a potentially very expensive worst case, which we demonstrate in our experiments in this paper.

The contribution of this paper is twofold. Firstly, we extend the theoretically sound algorithm of Bender *et al* with a practical refinement allowing the cost of updates to decrease, in exchange for an increase in comparison costs. Secondly, we demonstrate that the use of type information can substantially decrease the overhead of document ordering maintenance algorithms, which appear to have quite a high overhead for disk-bound structures in practice.

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, which are used in later sections. Section 4 describes an algorithm of Bender et al, and our improvements to it. Section 5 describes a system which utilizes type information to speed up ordering indices. Section 6 presents empirical tests of our algorithms, and finally, Section 7 concludes the paper.

2 Related Work

The problem of maintaining order in an XML database is equivalent to a well-studied problem, the *order maintenance problem*. The order maintenance problem is to answer queries regarding the relative order of elements of a linked list, which may change due to insertions and deletions. In a classical paper [9], Dietz and Sleator proved constant worst case time bounds for the order maintenance problem, building on previous results [10]. Unfortunately, their constant time solution is quite complicated, and difficult to implement; they also provide an amortized $O(1)$ algorithm which works well in practice. A substantially more elegant formulation of their work has recently been obtained by Bender et al [3], who show empirically that their much simpler algorithm outperforms that of Dietz and Sleator. While the order maintenance problem is theoretically solved, there are still several issues when applying previous results to large XML repositories. These problems are discussed in more detail in Section 4.

To the best of our knowledge, the closest other related work to this paper are efforts on determining ancestor-descendant relationships. For example, in [20], 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 [22]. In [18], 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 [5] 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 [28]. 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 [22]. The performance and results of these approaches based on labelling schemes are consistent with the theoretical properties of labelling dynamic XML trees presented by [6]. 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. However, their work assumes that once a label is assigned, it is never changed, whereas our work changes the value of the label frequently. Hence, the theoretical limitation imposed by [6] is not relevant to our work.

Whilst all the efforts described above were focused on ancestor-descendant relationships, our method is on sorting and maintaining data in document order. However, our method can also be easily extended and applied to the ancestor-descendant problem (as presented in [19]). Furthermore, different from 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. Liefke [23] 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 [8, 14], [17] extended dataguides [16] and proximity search [15] to take order into consideration.

In our previous work on this topic [13], we presented a simple randomized algorithm which seems to have good practical performance. Whereas the algorithms of Dietz and Bender require two passes

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

over some section of the database in order to relabel, our algorithm only required one pass. The trade-off for the second pass was that the worst case update cost is potentially linear in the size of the database. The worst case appears during frequent insertions into the middle of the list — we suspect that this case does not happen too frequently in real life, because such insertions will use a specialized bulk insertion strategy to minimize the number of updates. Nevertheless, for applications where a low worst case bound is required, the algorithm is insufficient. The work in this paper provides fast algorithms which have guaranteed good worst case performance, although we find that in many cases our earlier algorithm, which will be analysed in detailed in this paper, is still the fastest in practice.

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. The document ordering on an XML document is the total ordering defined by a pre-order traversal of the corresponding tree [27]. In this paper, we will denote the document ordering by $<$. Throughout this paper, we impose a specific physical data model on our XML database, which gives a set of accessor functions (as summarized in Table 1) which take constant time to run. We have carefully chosen this set of accessors so that it is likely that any reasonable XML database would need to be able to implement these accessors in constant time. 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.

We assign to each node a unique identifier, the *object identifier*, or *oid*. 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 (with further optimisations possible).

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

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
```

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.

4 Improving Bender's Algorithm

In this section, we describe an improvement to Bender's document ordering algorithm. Briefly, this improvement allows the database to increase the speed of the ordering algorithm, in exchange for an increase in the worst case bound of comparisons.

4.1 Overview of Bender's Algorithm

We now provide a brief overview of the algorithm of Bender *et al* [3]. 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*, 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 $id(n) \in \mathcal{B}$ 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 $id(x) + 1 \neq id(y)$, we set $id(n) = \frac{1}{2}(id(x) + id(y))$. 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* [3] 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. The pseudo-code for this algorithm is given in Algorithm 2.

From the above $O(\log n)$ amortized time algorithm, we can obtain an amortized $O(1)$ algorithm using a standard technique (see, for instance, [9]). 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. In fact, in our implementation, we found that the $O(1)$ variant required approximately 5.5 bytes per node in the database, whereas the $O(\log n)$ variant required 4 bytes.

Additionally, to perform ordering between two nodes one must lookup the tags of their sub-lists, which is an unavoidable indirection. If the data is not stored in document order (which is likely in a mature database product, where the data will be clustered along common access paths instead of document order), this can have an adverse impact on paging, and possibly incur many expensive disk reads. Finally, if we choose to use only an additional word of information per node, this algorithm can only handle a small number of nodes before the constant time performance is lost. Thus, for use in a native XML database the constant time algorithm may be impractical in many situations, although as we show in the experiments, its performance is good enough that the extra space utilization may well be worth the benefit.

An Improvement

As described above, while the $O(1)$ algorithm is very fast, it does have some weaknesses that make it inappropriate in some common scenarios. Thus, it is worth investigating whether the practical performance of the $O(\log n)$ algorithm can be sped up. The key idea to improving the practical performance of the algorithm is noting that we wish to minimize the *average* cost of ordering comparisons, not the

worst case cost. Suppose we insert a new node n between two nodes x and y. If there is a gap between the tags of x and y, we can easily find a unique tag for n. If there is no gap, then the algorithm above will always relabel some (potentially large) portion of the database. When we compare the relative order of two nodes in the database, we lookup their tag values, and compare them numerically.

Suppose, however, that in the database, the nodes that are compared are generally a large distance apart in document order. In this case, if adjacent nodes were allowed to share the same tag, then we would reduce the number of tags in use in the database, and thus hopefully leave larger gaps for insertions. When two nodes are compared, and they have the same tag value, we must of course revert to using the naive comparison algorithm of Algorithm 1. However, if we bound the number of nodes that share the same tag value, then we also bound the cost of this naive comparison.

Suppose we generalize Bender's algorithm to take an additional parameter, c, which gives the number of nodes that should share the same tag value. It is straightforward to generalize the $O(\log n)$ algorithm to this case: instead of defining the density in a range of tag values as $\frac{n}{2^i}$, where n is the number of nodes in that range, we use $\frac{n}{2^i c}$. Unfortunately, this straightforward generalization is not terribly useful, as the amortized maintenance cost rises from c $(2 - \frac{T}{2})c \log \frac{n}{c}$. Thus, we have simultaneously increased both the maintenance cost and the comparison cost!

Instead of a deterministic modification of Bender's algorithm, we adapt the idea of the previous paragraph by incorporating an element of randomization. Suppose we have a node n to insert into the database. With probability $\frac{1}{c}$, we follow the algorithm described in the previous paragraph. With probability $1 - \frac{1}{c}$, we instead arbitrarily choose either $id(x)$ or $id(y)$, where x is the element to the left of n and y is to the right of n, and use this as the tag value of n. This algorithm is presented in pseudocode in Algorithm 2.

Algorithm 2 includes the comparison function for this document ordering algorithm. We note that, in the event that a call to `NAIVE-ORDER-CMP` is made, there is a small practical improvement that can be made. In lines 4 and 8 of Algorithm 1, we can compare the tag of each of the parents with n_1 or n_2. In the event that the tags differ, we can terminate the naive traversal early, because we have enough information to deduce their relative ordering.

The expected worst case of this algorithm is easy to show:

Theorem 1 *The expected worst case time of Algorithm 3 is $O(c)$.*

Proof: Clearly, the expected worst case time of Algorithm 3 is equal to the expected worst cast number of nodes per tag. After a tag has been used in a relabelling, exactly c nodes share it. After a relabelling, in the worst case there is a sequence of node insertions at that tag; for each of these insertions, another node is added to the tag with probability $1 - \frac{1}{c}$, and a relabelling is triggered with probability $\frac{1}{c}$. This is clearly a geometric distribution, and hence the expected number of nodes that will be added before a relabelling is c. Hence the expected worst case number of nodes for the tag is $c + c = O(c)$. □

Theorem 2 *The expected worst case time of Algorithm 2 is $O(\log \frac{n}{c})$.*

Proof: As described above, the worst case time of the deterministic variant is $(2 - \frac{T}{2})c \log \frac{n}{c}$. As this algorithm is triggered with probability $\frac{1}{c}$, and the reuse of a tag, costing $O(1)$ time, occurs with probability $1 - \frac{1}{c}$, the expected worst case time is clearly $O(\log \frac{n}{c})$. □

Thus, we now have a parameterized family of algorithms (indexed by c), which tradeoff comparison cost and update cost. The actual choice of c must be made by the database designer, as the optimal choice clearly depends on many aspects of the application domain, especially characteristics of the query workload and also the implementation of the native XML database system. We note that the randomized variant we have specified only works with the $O(\log n)$ algorithm, not the $O(1)$ extension.

Algorithm 2 Maintenance of the document ordering index during the insertion of a new node, using a randomized variant of the algorithm of Bender et al [3].

```
BENDER-INSERT(n, c)
 1 x ← PREORDER-PREVIOUS(n)
 2 y ← PREORDER-NEXT(n)
 3 if x = NIL and y = NIL then
 4    id(n) ← 2^(W−1)
 5 end if
 6 Randomly choose p ∈ [0, 1] uniformly
 7 if p ≥ 1/c then
 8    if x = NIL then
 9       n ← id(y)
10    else
11       n ← id(x)
12    end if
13 elif y = NIL and id(x) ≠ 2^W − 1 then
14    id(n) ← ⌈(id(x) + 2^W)/2⌉
15 elif x = NIL and id(y) ≠ 0 then
16    id(n) ← ⌈id(x)/2⌉
17 elif id(x) + 1 ≠ id(y) then
18    id(n) ← ⌈(id(x) + id(y))/2⌉
19 else
20    T ← (|D| / (2^W c))^(−1/W)
21    for i ∈ {1, ..., W} do
22       l ← 2^i ⌊id(x)/2^i⌋
23       u ← 2^i (⌊id(x)/2^i⌋ + 1)
24       S ← {m ∈ D : l ≤ id(m) < u} ∪ {n}
25       d ← |S| / (2^i c)
26       if d < T^(−i) then
27          j ← 0
28          for m ∈ S in ascending document order do
29             id(m) ← l + ⌊2^i j / |S|⌋
30             j ← j + 1
31          end for
32       end if
33    end for
34 end if
```

In practice, we find that in many cases the $O(1)$ algorithm has good performance. However, our new variant still has many uses, due to the disadvantages of the $O(1)$ algorithm mentioned previously, i.e., increased space usage, and an indirection on each comparison which may incur a disk access.

5 Utilizing Schema Information

All algorithms that have been described so far can work on any XML document, and can in fact work on any list structure. Of course, a significant portion of XML documents in real life come with type information, typically in the form of a DTD [4], XML Schema [11], or a schema written in some other standard schema language. This type information is used for verification purposes and provides a primitive form of constraint checking. Even for untyped documents, there is generally a high degree of regularity, and a large body of research on schema inference can be used in these cases to generate a schema.

In this section, we provide an automated system which can utilize type information to optimize the

Algorithm 3 Comparing the relative document order of two nodes using the randomized variant of Bender's algorithm.

```
BENDER-ORDER-CMP(n_1, n_2)
 1 i_1 ← id(n_1)
 2 i_2 ← id(n_2)
 3 if i_1 = i_2 then
 4     return NAIVE-ORDER-CMP(n_1, n_2)
 5 else
 6     return i_1 < i_2
 7 end if
```

use of a document ordering algorithm. There are several interesting aspects to our system. Firstly, the system should work with any of the numerous schema languages available, even though we frame our discussion in terms of the theoretically simplest, DTDs. Secondly, the algorithm can be used to improve the practice of *any* algorithm for the document ordering problem, and hence has wide applicability.

Our algorithm works in two stages. In the first stage, we *linearize* the DTD, extracting as much information relating to document ordering as possible from the DTD. This information is represented in the form of a graph, which we can use to sometimes infer in $O(1)$ time, without the help of a document ordering algorithm, the relative ordering of two nodes. Unfortunately, in many cases the schema can yield little information after this first phase. Hence, the system also examines a small sample of real data in order to infer additional useful information. Using the information gathered, the system then produces a set of rules defining how a given document ordering algorithm should be used, and how ordering comparisons should be made between two nodes.

5.1 Background

While our algorithm should work with any schema language, it is framed in terms of DTDs for simplicity, and because they are one of the most commonly used schema languages. The only phase of the algorithm that is schema specific is the linearization phase of Section 5.2.

It is well-known that DTDs correspond to regular tree grammars [25]. Therefore, we will represent a DTD as a set of rules $\{n = t_n \mid n \in N\}$, where N is the set of all element names used in the DTD (note this is *not* the same as $\mathcal{N}$, the set of all possible element names). We will denote by R_N the set of all regular expressions, and write $\tau(n) = t_n$ for the map between N and R_N. For ease of presentation, we will not consider attributes. Also, as we do not care about character data, we will simply ignore PCDATA. Then, for each $n \in N$, t_n is a regular expression over over the set of rules. We follow the standard notation for regular expressions, that is:

- $r_1 r_2$ is the concatenation of r_1 and r_2
- $r_1 | r_2$ is the disjunction of r_1 and r_2
- $r*$ is the Kleene closure of r
- ϵ is the empty rule
- n for an atom (in this case, $n \in N$).

We implicitly assume that, given a node n from the database, we can in $O(1)$ time identify its type. This assumption should be valid in all database systems. Also, in order to handle the ANY feature of DTDs, we define a special element name, α, which stands for all possible element names that are not

declared in the DTD, that is, α represents all names in $\mathcal{N} - N$. We then use the regular expression $(n_1|\ldots|n_m|\alpha)*$, where $N = \{n_1, \ldots, n_m\}$, to represent ANY. This is sufficient for our purposes, even though it is not an exact representation of this feature of DTDs.

```
<!ELEMENT dblp (article|inproceedings|...)*>
<!ENTITY %field "author|editor|...">
<!ELEMENT article (%field;)*>
<!ELEMENT inproceedings (%field;)*>
<!ELEMENT author (#PCDATA)>
<!ELEMENT editor (#PCDATA)>
...
```

Figure 1: An excerpt from the DBLP DTD.

$$\begin{aligned} dblp &= (article|inproceedings|\ldots)* \\ article &= (author|editor|\ldots)* \\ inproceedings &= (author|editor|\ldots)* \\ author &= \epsilon \\ editor &= \epsilon \end{aligned}$$

Figure 2: The regular grammar for the DTD fragment of Figure 2.

Figure 2 shows the regular grammar for a fragment of the DBLP DTD, given in Figure 1. We will use this as a running example throughout the paper.

5.2 Linearizing a DTD

Given a DTD, we must extract as much information as possible about the relative ordering between types from it. We call this process *linearization*, because we are essentially constructing from the DTD a schema for the XML document when considered as a linked list in document order.

We represent the linearized DTD as a node-labelled, directed graph $G = \langle V, E \rangle$, where V is the set of vertices of the graph, in one-to-one correspondence with the types in the DTD. There is an edge in G between n_1 and n_2 if and only if an element of type n_2 may follow an element of type n_1 (in document order) in any document validated by the given DTD.

We linearize the DTD by providing a set of recursive transformation rules, which act on the rules of the DTD. We first define two useful functions, $f : N \to 2^N$ and $l : N \to 2^N$, such that $f(n)$ gives the entry nodes of the graph, and $l(n)$ gives the leaf nodes of the graph corresponding to the node n. We will utilize these functions when combining the graphs for two sub-rules of a rule, in order to construct its graph. The functions are defined as:

$$f(n) = \begin{cases} f(\tau(n)) & \text{if } \tau(n) \neq \epsilon \\ \{n\} & \text{if } \tau(n) = \epsilon \end{cases}$$

and:

$$l(n) = \begin{cases} l(\tau(n)) & \text{if } \tau(n) \neq \epsilon \\ \{n\} & \text{if } \tau(n) = \epsilon \end{cases}$$

In the above, we have used functions $f : R_N \to 2^N$ and $l : R_N \to 2^N$, which are defined using structural recursion as:

- $f(r_1 r_2) = f(r_1) \cup \begin{cases} f(r_2) & \text{if } e(r_1) \\ \emptyset & \text{otherwise} \end{cases}$

  and

  $l(r_1 r_2) = l(r_2) \cup \begin{cases} l(r_1) & \text{if } e(r_2) \\ \emptyset & \text{otherwise} \end{cases}$

- $f(r_1|r_2) = f(r_1) \cup f(r_2)$ and $l(r_1|r_2) = l(r_1) \cup l(r_2)$
- $f(r_1*) = f(r_1)$ and $l(r_1*) = l(r_1)$
- $f(\epsilon) = \emptyset$ and $l(\epsilon) = \emptyset$
- For any $n \in N$, $f(n)$ and $l(n)$ recurse on the formulas defined above.

The above definitions made use of a predicate $e(r)$, which evaluates to $true$ if r may match the empty string. More precisely, using structural recursion:

- $e(r_1 r_2) = e(r_1) \wedge e(r_2)$
- $e(r_1|r_2) = e(r_1) \vee e(r_2)$
- $e(r_1*) = true$
- $e(\epsilon) = true$
- $e(n) = false$

Using these functions, we may now define a function $G : N \to \mathcal{G}$, where $\mathcal{G}$ is the set of all graphs $\langle V, E \rangle$ as above. The function $G(n)$ constructs the linearization corresponding to the rule n, and is defined as:

$$G(n) = \begin{cases} \langle V \cup \{n\}, E \cup \{n \to n' | n' \in f(\tau(n))\}, n \rangle \\ \qquad \text{if } \tau(n) \neq \epsilon \\ \langle \{n\}, \emptyset, n \rangle \text{ if } \tau(n) = \epsilon \end{cases}$$

Where $\langle V, E \rangle = G(\tau(n))$, and $G : R_N \to \mathcal{G}$, defined using structural recursion as:

- For $r_1, r_2 \in R_N$, let $\langle V_1, E_1 \rangle = G(r_1)$ and $\langle V_2, E_2 \rangle = G(r_2)$. Then $G(r_1 r_2) = \langle V_1 \cup V_2, E_1 \cup E_2 \cup \{n_1 \to n_2 \mid n_1 \in l(r_1), n_2 \in f(r_2)\} \rangle$.
- $G(r_1|r_2) = \langle V_1 \cup V_2, E_1 \cup E_2 \rangle$.
- $G(r*) = \langle V, E \cup \{n_1 \to n_2 \mid n_1 \in l(r), n_2 \in f(r)\} \rangle$.
- $G(\epsilon) = \langle \emptyset, \emptyset \rangle$.
- For any $n \in N$, $G(n)$ recurses on the formula defined above.

The function G defined above translates the DTD into a graph. We define the *linearization* of a DTD to be $G(r)$, where r is the root rule of the DTD. We note that a DTD does not, in fact, have a root rule. However, we have found in practice that there is almost always a single rule which is implicitly assumed to be the root. In any case, we require the existence of such a rule, and have found that this has not been a significant burden in practice. We then made $G(r)$ a rooted graph $\langle V, E, r \rangle$, with root r.

The correctness of the above transformation is summarized in the following theorem, the proof of which is omitted due to space considerations.

Theorem 3 *The graph $G = \langle V, E, r \rangle$ constructed as above satisfies the following property: for any $t \in V$, for any document $\mathcal{D}$ validated by the DTD, let $x \in \mathcal{D}$ have type t. If x is not the last element in document order, then its successor in document order, y, must be of type t', where $t \to t' \in E$.*

Moreover, G is the unique smallest graph for the DTD satisfying this property, in the sense that if $G' = \langle V', E', r \rangle$ is another graph satisfying this property, then $V \subseteq V'$ and $E \subseteq E'$.

5.3 Using a Linearization

Once we have a linearization, it is an easy matter to extract document ordering information from it. We construct the graph of strongly connected components G' for the linearization G, using standard techniques [7]. It is clear that G' is a directed acyclic graph. The following lemma easily follows from Theorem 3:

Lemma 1 *Let G be the linearization of a DTD, and G' be the graph of strongly connected components of G. Then for any document $\mathcal{D}$ validated by the DTD, for any nodes $x, y \in \mathcal{D}$, if $[t(y)]$ is reachable from $[t(x)]$ in G', then $x < y$ in document order, where $t(x)$ is the type of x and $[v]$ is the strongly connected component of vertex v in G.*

Thus, ordering information can be easily computed at database creation time by traversing G', and storing the relative ordering between the strongly connected components in an array of size $O(|N|^2)$. More sophisticated techniques can be used if $|N|$ is large, although in practice it generally is not. For nodes within each strongly connected component, of course, we do not know their relative order. In this case, we can apply a document ordering algorithm such as Bender's algorithm — Theorem 3 shows that we can access the next and previous nodes in each component in $O(1)$ time (assuming this access is constant time in the underlying database).

Note that we can apply a *different* instance of the algorithm to each strongly connected component, which could possibly result in reducing both the size of the tag, or decreasing the cost of updates. As an extension, if we have selectivity information indicating the approximate sizes of each strongly connected component, then we can use this information to choose the optimal algorithm for each component. For instance, for very large components it may be suitable to choose the $O(1)$ variant of Bender's algorithm, but for smaller components we can choose an algorithm with less space overhead.

5.4 Annotating a Linearization

While the scheme described so far has been very general, it has some limitations in practice. We have found that many schemas used in the real world are significantly looser than they might ideally be, and that this inexactness propogates itself into the analysis above. For instance, consider the linearization in Figure 3. In this case, it is clear that there are only two strongly connected components, one consisting of the single node type `dblp`. Clearly, this is not particularly useful information.

However, we can still use type information in a heuristic manner which gives good practical performance in many cases. In Figure 3, we have annotated the linearization with dotted edges, which represent that, in our particular choice of native database system, from any node we can access its parent node in $O(1)$ time. In fact, we also have an accessor which allows any parent node to access its first child in $O(1)$ time as well. This accessor does not appear in Figure 3, of course, because the DTD does not give us enough information to determine what the first child is. For instance, consider a node of type `article`. From the DTD, we are unable to infer whether this node will even *have* a first child. On the other hand, we always know that any node (except `dblp`) will have a parent. We will now formalize the notion of an acceptable accessor:

Definition 1 *An* accessor *is a map from $\phi : N_1 \to N_2$, where $N_1, N_2 \subseteq N$, such that $\forall x \in N_1$, $\phi(x)$ can be retrieved from x in $O(1)$ time.*

Of course, not all accessors are particularly interesting in terms of document ordering. The interesting accessors are those that allow us to infer some partial information about the relative ordering between nodes. For instance, the parent accessor is useful, because if two nodes have different parents, then clearly their relative order is related to the relative order of their parents. Thus, we are interested in *order preserving* accessors:

Definition 2 *An* order preserving accessor *is an accessor* $\phi : N_1 \to N_2$ *such that* $\forall x, y \in N_1$, *if* $x < y$ *then* $\phi(x) \leq \phi(y)$.

The accessors available to us differ depending upon the underlying implementation. As an example, we define the parent accessor for a particular node type n. Let N_1 be the set of all nodes that appear in the regular expressions containing n in the DTD, and N_2 be the nodes with those regular expressions. Then in our implementation there is a map $\phi : N_1 \to N_2$ which yields the parent of any $n \in N_1$. This accessor is illustrated in Figure 3 — the regions surrounded by dotted areas are the domain and codomain of this accessor, with the dotted lines representing the movement through the accessor.

While we are interested in order preserving accessors, we are not interested in all of them. For instance, suppose it is possible to access the root node of the database in $O(1)$ time. Then it would be trivial to define an accessor from the set of all node types N which accessed the root. This would not be a terribly useful accessor in terms of document ordering, however, because $\forall x, y \in N$, the image under the map would be equal. In fact, we are only interested in order preserving accessors which not only allow constant time access, but also have a low probability of disk accesses.

Therefore, we introduce a threshold parameter, controlled by the database creator, which specifies the maximum acceptable probability of a disk access along an accessor. For a given set of accessors, we assume we have the approximate probability of a disk access incurred by following that accessor. This information could come from one of several sources:

- The clustering subsystem of the database;
- The database creator; or
- An analysis of a small representative sample of the data to be stored in the database.

We do not believe that in practice it is difficult to produce these probabilities. For instance, for DBLP stored in document order it is fairly obvious that each record will generally lie on a single page, and hence for intra-record accesses the likelihood of a disk access is quite low.

5.5 Coalescing

Once we have a set of accessors, along with their corresponding disk access probabilities, we *coalesce* portions of the linearization. Informally, we merge nodes which have no accessors between them, and which lie in the same strongly connected component. The motivation is that as we are unable to infer any information about the relative ordering of nodes in the database corresponding to these types, there is no point in maintaining them as separate nodes in the linearization.

Given a set of accessors $\mathcal{A}$, we coalesce using a simply greedy heuristic. Firstly, we remove any accessor $\phi : N_1 \to N_2 \in \mathcal{A}$ such that N_1 or N_2 are not strictly contained in a strongly connected component of the linearization. We then iterate through the following steps until $\mathcal{A}$ is empty:

1. Remove the accessor $\phi : N_1 \to N_2 \in \mathcal{A}$ which has the smallest disk access probability, and annotate the graph with ϕ.
2. Merge the nodes in N_1 together, and the nodes in N_2 together.

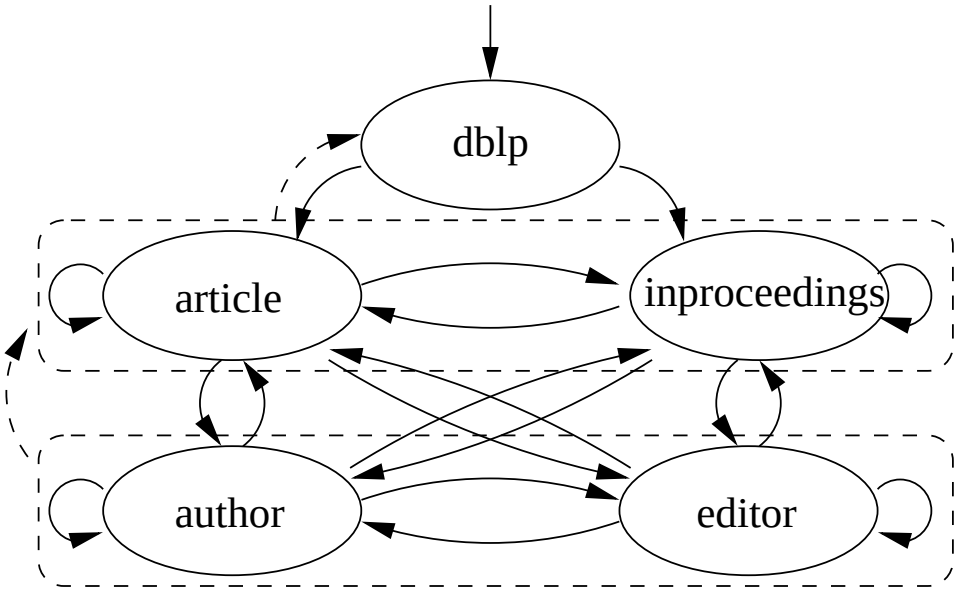


Figure 3: Linearization of the DTD fragment of Figure 1.

3. Remove any $\phi' : N_1' \to N_2' \in \mathcal{A}$ where either N_1' or N_2' have a non-empty intersection with either N_1 or N_2.

The process of running this procedure on the linearization of Figure 3 is shown in Figure 4.

5.6 Using an Annotated Linearization

Using an annotated linearization is similar to using a linearization. We use the techniques described in previous sections when comparing nodes belong to two different strongly connected components. When the nodes lie in the same strongly connected components, however, we now see if they lie in the same node (which may be the result of merging several types). If they are, and this node has an available accessor, we first traverse the accessor and determine the relative ordering under the image of the accessor. Only in the event that this does not resolve the relative order of the two nodes do we compare them using their own tag values.

The reason this scheme works is because we have further split portions of the strongly connected components, in such a way that we can again apply a different instance of the document ordering algorithm. For instance, in Figure 4, we can apply a separate instance of the document ordering algorithm to each node in the graph. In fact, for the `author|editor` node, we apply a separate instance of the ordering algorithm to each set of nodes S such that $\forall x, y \in S$, $\phi(x) = \phi(y)$. It is easy to see that such a set of nodes must be contiguous (in document order), from the definition of an order preserving accessor.

6 Experimental Results

6.1 Experimental Setup

We performed several experiments to determine the effectiveness of the various algorithms covered in this paper. All experiments were performed on a dual processor 750 MHz Pentium III machine with 512 MB RAM and a 30 GB, 10,000 rpm SCSI hard drive. We used as our data set the DBLP database.

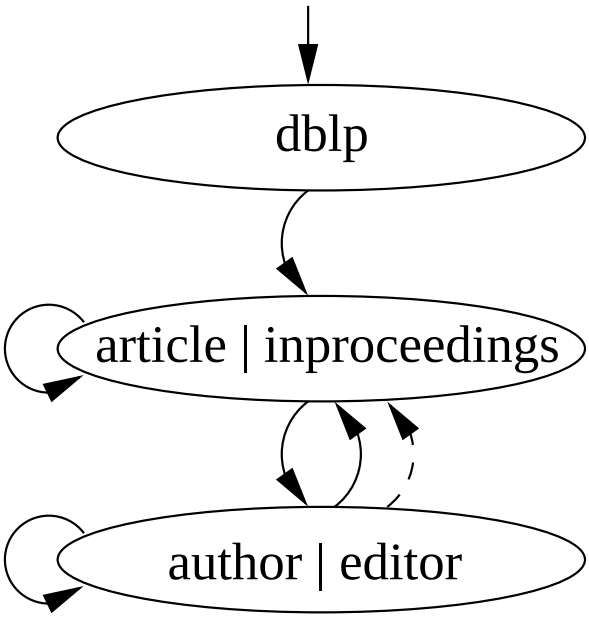


Figure 4: Linearization of the DTD fragment of Figure 1 after coalescing.

Due to the fact that there are, at the time of this writing, very few native XML databases available, we performed our experiments on a custom-written simple disk-bound database not supporting transactions or concurrency. While the database kernel code was not optimized, we believe that our results will give good indications of performance in other native XML databases.

Our experiments were designed to investigate the worst case behavior of each of the document maintenance algorithms implemented. We chose to focus on worst case for several reasons. Firstly, our previous work found an algorithm which has very good average case performance (for a particular definition of average), but very poor worst case performance. As our new work has good worst case performance, we emphasize this in our experiments. More practically, however, at the time of writing it is very difficult to determine a suitable "average" case for usage of document ordering indices, simply because XML databases are only beginning to be used in realistic projects.

For each experiment, we investigated the performance of four different algorithms:

1. The randomized algorithm of our previous work [13];
2. The $O(1)$ and $O(\log n)$ variants of Bender et al [3];
3. The randomized variant of Bender's algorithm, described in Section 4. For this algorithm, we used values of c from the set $\{50, 200, 1000\}$; and
4. The $O(\log n)$ variant of Bender's algorithm, but augmented with optimizations garnered from the DBLP DTD, using the methods of Section 5.

6.2 Experiment 1: Performance on a Bulk Insert

In this experiment, we evaluated the performance of the algorithms under a uniform query distribution. The experiment began with an empty database, which was then gradually initialized with the DBLP database. After every insertion, on average r reads were performed, where r was a fixed parameter taken from the set $\{0.01, 0.10, 1.00, 10.0\}$. Each read operation picked two nodes at random from

the underlying database, using a uniform probability distribution, and compared their document order. In this experiment, we measured the total time of the combined read and write operations, the number of read and write operations, and the number of relabellings. However, due to space considerations, we only include the graphs for total time.

This experiment was designed to test what is a worst case scenario for all the algorithms except for our randomized algorithm. We included this experiment because the extremely heavy paging incurred by the uniform query distribution demonstrates the practical problems with Bender's $O(1)$ algorithm. Unfortunately, due to this heavy paging, this experiment took an extremely long time to run, and we were unable to complete it for all the algorithms.

As can be seen from the results in Figure 5, the randomized algorithm is easily the best performer. In fact, in this case, the randomized algorithm performs the absolute minimum work possible for an ordering algorithm, and hence is a useful baseline to compare the other algorithms with. We note that, as the ratio of reads increases, the performance of both Bender's $O(1)$ algorithm and the schema based algorithm degrades. We attribute this in both cases to the extra indirection involved in reading from the index. Also, because of the extremely heavy paging, even the small paging overhead incurred by an algorithm such as the schema based algorithm, which only infrequently loads in an additional page in due to a read from the index, has a massive effect on the performance. Thus, although this experiment is slightly contrived, it does demonstrate that in some circumstances the indirection involved is unacceptable.

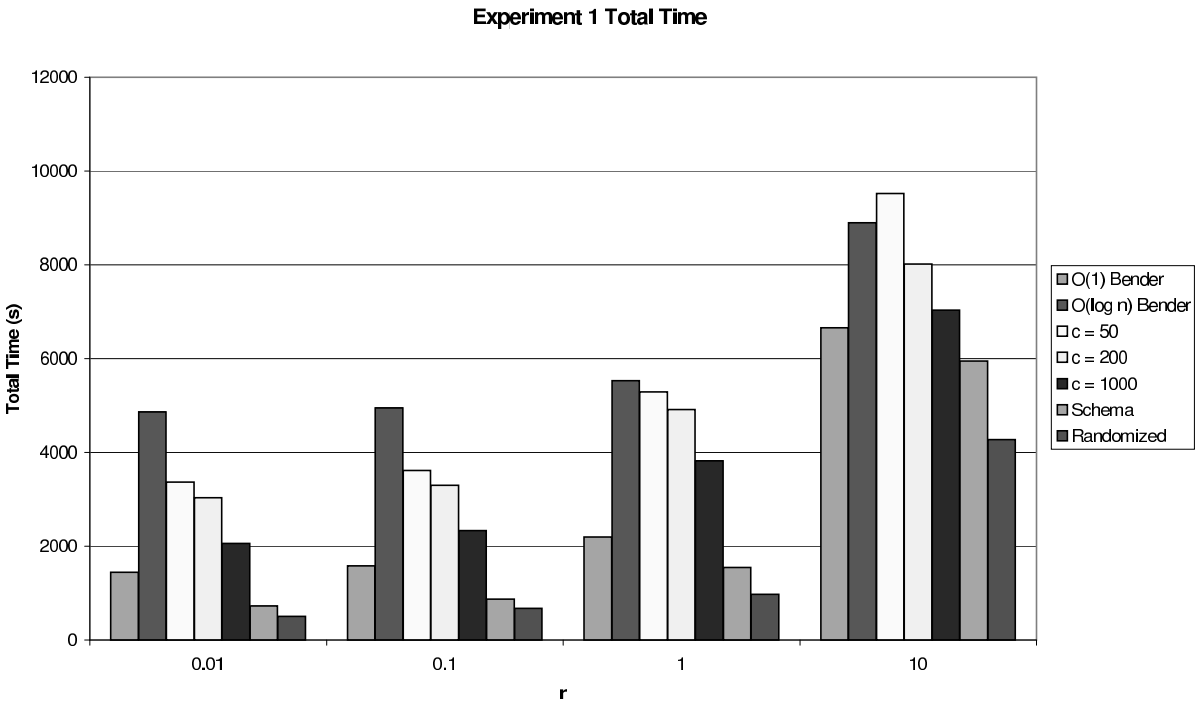


Figure 5: Experiment 1 Results

6.3 Experiment 2: Performance on a Non-Uniform Query Distribution

This experiment was identical to the first experiment, except that the reads were sampled from a normal distribution with mean $\frac{|D|}{2}$, and variance $\frac{|D|}{10}$. The idea was to reduce the heavy paging of the first experiment, and instead simulate a database "hot-spot", a phenomenom which occurs in practice.

As can be seen from the results of Figure 6, this experiment took substantially less time to complete than the first experiment. It can be seen that, apart from the randomized algorithm (which again performed the minimal possible work), the schema based algorithm is clearly the best algorithm. Indeed, it is impressive that it came so close to the randomized algorithm in performance.

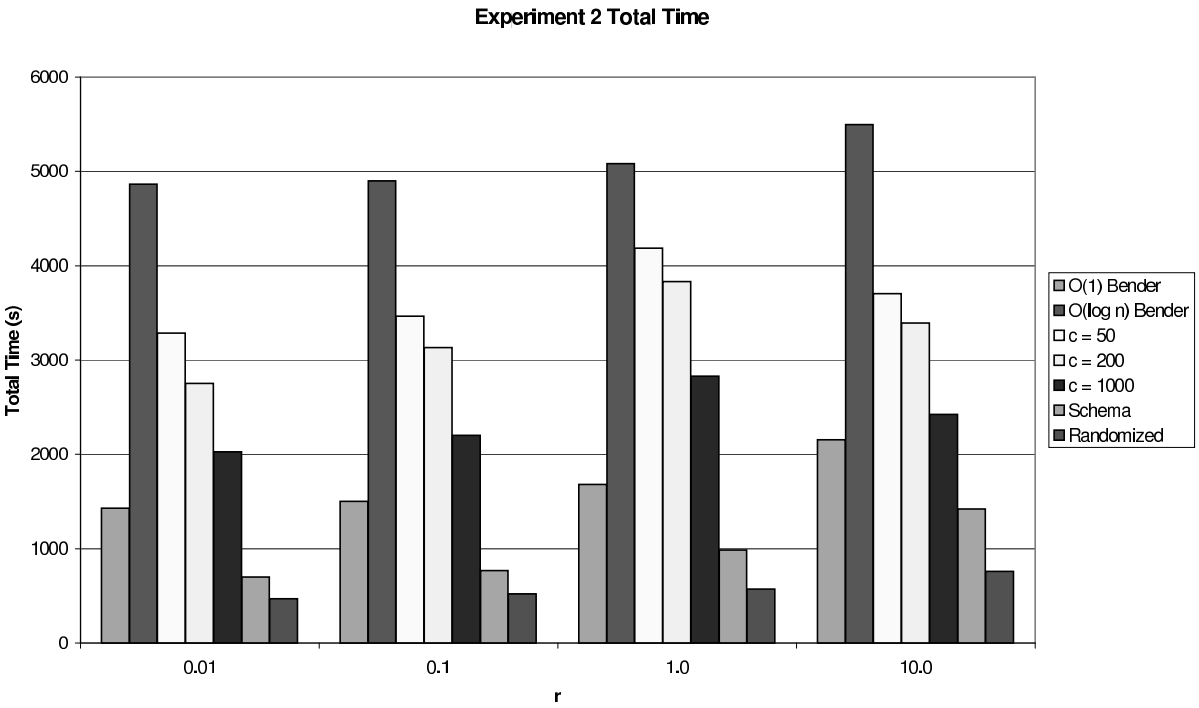


Figure 6: Experiment 2 Results

6.4 Experiment 3: Worst-Case Performance for the Randomized Algorithm

The previous two experiments showed that the randomized algorithm had very good performance. We demonstrate in this experiment that, in some cases, it has very bad performance, far worse than the other algorithms. In spite of this, we believe the randomized algorithm is worth considering, as in many situations it is almost unbeatable. This experiment was identical to the first experiment, except that instead of inserting DBLP records at the end of the database, we inserted them at the beginning. As the experiment would take an unreasonable amount of time to run on the full DBLP, and because the worst case performance of the randomized algorithm is so pronounced in this case, we only ran the experiment on a small subset of DBLP.

As can be seen from the results in Figure 7, all the other algorithms easily beat the randomized algorithm's performance. Hence, in situations where worst case bounds must be guaranteed, the randomized algorithm is not a good choice.

6.5 Experiment 4: Verifying the Theory

Our final experiment was performed to verify the theory of Section 4, in particular, the expected run-time cost of the randomized variant. For this experiment, we simply loaded DBLP into a database, maintaining ordering information using Algorithm 2, for values of c from the set $\{1, 5, 10, 50, 100, 200, 1000\}$. The total running time of each of these algorithms, as a proportion of the $c = 1$ running time, is found in Figure 8. The line of best fit was determined using a standard

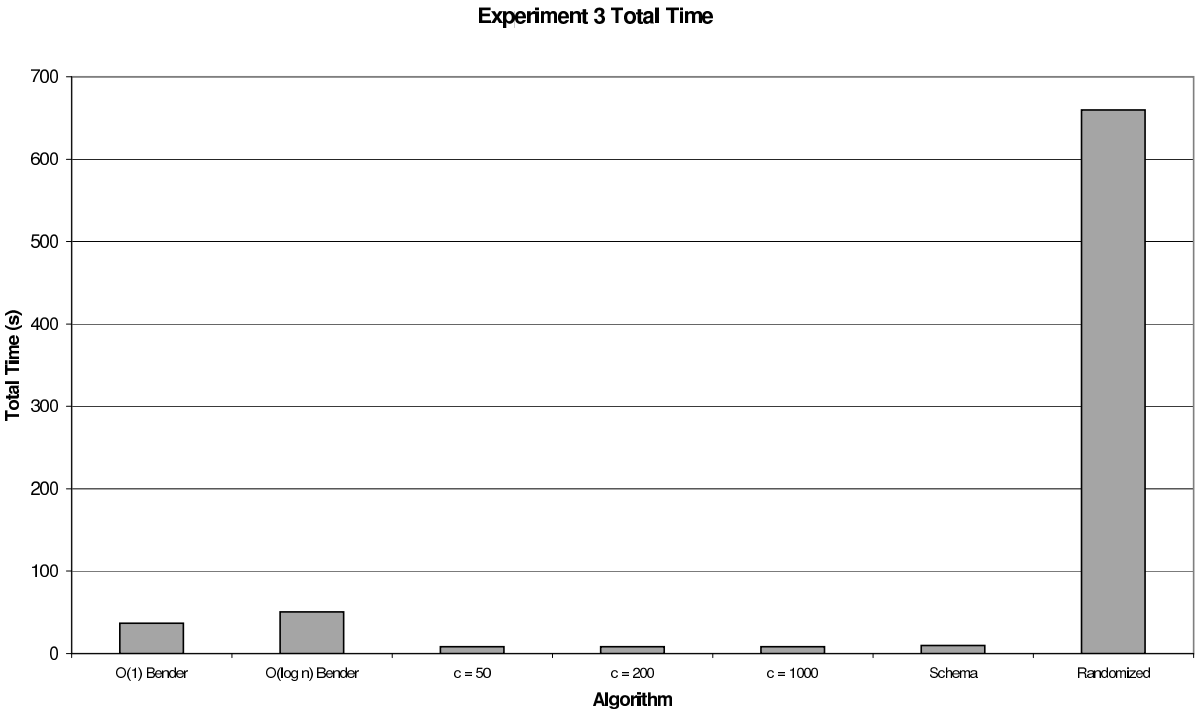


Figure 7: Experiment 3 Results

logarithmic regression. As can be seen, the expected logarithmic dependence on c is exhibited very closely.

7 Conclusions

The contributions of this paper are two-fold. Firstly, we have presented an improvement to the work of Bender et al, which yields a parameterized family of algorithms giving fine control over the cost of updates to the document ordering index versus querying this index. We believe this work is of particular relevance in XML databases, as we anticipate in the future a wide range of application domains will utilize such databases, and each of these domains will have different requirements.

Our second contribution is a general scheme to utilize type information to improve the speed of document ordering indices. The significance of this work is increased due to its wide generality, as it is not tied to any particular ordering algorithm. We have found that in practice many large XML repositories have a DTD or some other schema for constraint purposes, and hence we expect that this work will have great practical impact.

There is still a lot of future research in this area, which we believe is absolutely crucial to the success of XML-based data repositories. Firstly, we intend to extend work such as that of Lerner and Shasha [21] to native XML database systems, in order to produce a theoretically sound query optimization framework in the presence of ordering. Once XML query optimization is a more well understood problem, it may also be possible to automate the selection of the parameter c described in Section 4. Secondly, as our experiments have shown, it is clear that the maintenance of an ordering index has a substantial impact on the update cost for XML databases. We intend to investigate whether there are any theoretical lower bounds on this impact — clearly this analysis cannot be done in terms of asymptotic performance, as in that respect the ordering problem has already been solved.

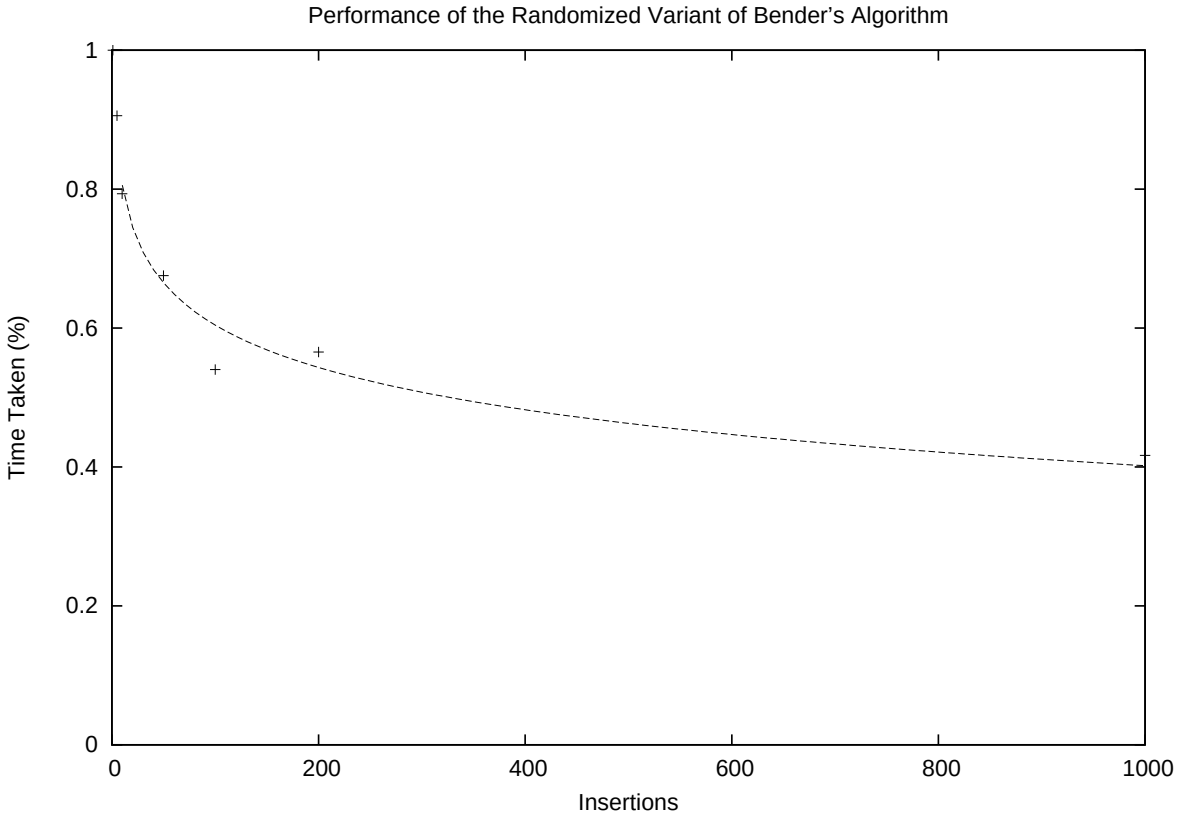


Figure 8: Experiment 4 Results

References

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

[2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, and J. M. Patel. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *ICDE*. IEEE Computer Society, 2002.

[3] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. 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.

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

[5] O. C. L. Center. Introduction to the Dewey Decimal Classification. `http://www.oclc.org/oclc/fp/about/about_the_ddc.htm`.

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

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[8] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(11–16):1155–1169, May 1999.

[9] 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.

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

[11] D. C. Fallside (Eds). "XML Schema Part 0: Primer". W3C Recommendation, May 2001. `http://www.w3.org/TR/xmlschema-0`.

[12] M. Fernández, Y. Kadiyska, D. Suciu, A. Morishima, and W.-C. Tan. SilkRoute: A framework for publishing relational data in XML. *ACM Transactions on Database Systems*, 27(4):438–493, Dec. 2002.
[13] D. Fisher, F. Lam, W. Shui, and R. Wong. Efficient Ordering for XML Data. Submitted for publication. also appeared as Technical Report UNSW-CSE-TR-0316, University of New South Wales, June 2003.
[14] 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.
[15] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity Search in Databases. In *Proceedings of VLDB*, pages 26–37. Morgan Kaufmann Publishers, 1998.
[16] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of VLDB*, pages 436–445, East Sussex - San Francisco, Aug. 1998. Morgan Kaufmann.
[17] R. Goldman and J. Widom. Summarizing and Searching Sequential Semistructured Sources. Technical Report, Stanford University, 2000.
[18] W. E. Kimber. HyTime and SGML: Understanding the HyTime HYQ Query Language. Technical Report Version 1.1, IBM Corporation, Aug. 1993.
[19] F. Lam, W. Shui, D. Fisher, and R. Wong. Efficient Structural Joins for Changing XML Data. Submitted to ICDE2004. also appeared as Technical Report UNSW-CSE-TR-0320, University of New South Wales, July 2003.
[20] Y. K. Lee, S.-J. Yoo, K. Yoon, and P. B. Berra. Index structures for structured documents. In *ACM International Conference on Digital Libraries*, pages 91–99, 1996.
[21] A. Lerner and D. Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *Proceedings of VLDB*, to appear, 2003.
[22] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of VLDB*, pages 361–370, 2001.
[23] H. Liefke. Horizontal query optimization on ordered semistructured data. In *WebDB (Informal Proceedings)*, pages 61–66, 1999.
[24] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of VLDB*, pages 315–326, 1999.
[25] M. Murata, D. Lee, and M. Mani. "Taxonomy of XML Schema Languages using Formal Language Theory". In *Extreme Markup Languages*, Montreal, Canada, Aug. 2001.
[26] W3C Working Draft. XML Path Language (XPath) 2.0. `http://www.w3.org/TR/2002/WD-xpath20-20021115`, Nov. 2002.
[27] W3C Working Draft. Xquery 1.0: An xml query language. `http://www.w3.org/TR/2002/WD-xquery-20021115`, Nov. 2002.
[28] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, 2001.