BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data*

Adam Silberstein Hao He Ke Yi Jun Yang
Department of Computer Science, Duke University, Durham, NC 27708, USA
{adam,haohe,yike,junyang}@cs.duke.edu

Abstract

Order-based element labeling for tree-structured XML data is an important technique in XML processing. It lies at the core of many fundamental XML operations such as containment join and twig matching. While labeling for static XML documents is well understood, less is known about how to maintain accurate labeling for dynamic XML documents, when elements and subtrees are inserted and deleted. Most existing approaches do not work well for arbitrary update patterns; they either produce unacceptably long labels or incur enormous relabeling costs. We present two novel I/O-efficient data structures, W-BOX and B-BOX, that efficiently maintain labeling for large, dynamic XML documents. We show analytically and experimentally that both, despite consuming minimal amounts of storage, gracefully handle arbitrary update patterns without sacrificing lookup efficiency. The two structures together provide a nice tradeoff between update and lookup costs: W-BOX has logarithmic amortized update cost and constant worst-case lookup cost, while B-BOX has constant amortized update cost and logarithmic worst-case lookup cost. We further propose techniques to eliminate the lookup cost for read-heavy workloads.

1. Introduction

XML has become a widely popular standard for representing and exchanging data over the Internet. Conceptually, an XML document consists of an ordered hierarchy of properly nested tagged elements. Elements can be labeled according to the structure of the document in ways that facilitate query processing. Many labeling schemes have been proposed in the literature (see Section 2 for a survey). Among the most popular and effective of them is an order-based labeling scheme that assigns a pair of numeric labels to each element based on the document order of its start and end tags. Figure 1 shows an example XML document with elements labeled using this scheme. This labeling scheme lies at the core of many fundamental XML operations such as containment join [21] and twig matching [5], because it supports efficient checking of ancestor-descendant relationships among elements: An element e_1 is an ancestor of another element e_2 if and only if the interval formed by e_1's labels contains that of e_2. There are alternative order-based labeling schemes, such as those based on pre- and post-order traversals of the XML tree, which work in similar ways. Because of their vital role in query processing, these order-based labels are often used as element identifiers and in various indexes [13, 12].

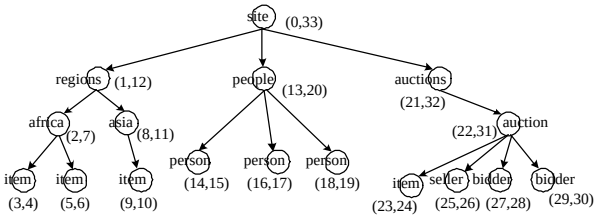

Figure 1. XML tree with order-based labels.

An important issue with any order-based labeling scheme is the ability to handle dynamic XML documents. When a document is updated (e.g., when elements or subtrees of elements are inserted or deleted), how do we ensure that the ordering among labels remain consistent with the document order? Ideally, we would want to keep all existing labels, but in [7] a negative result establishes that any *immutable* labeling scheme requires $\Omega(N)$ bits per label, where N is the size of the document. Such long labels not only incur high storage overhead, but are also less useful in query processing because they are more expensive to process on than shorter labels, especially when they cannot be accommodated by native machine words.

The alternative is to use a *dynamic* labeling scheme where existing labels can change with document updates. Most systems take a rather naive approach, which basically leaves gaps between adjacent labels in advance. Whenever this scheme runs out of values to assign to new labels because a gap has been filled by previously inserted labels, it relabels everything to leave equally sized gaps between adjacent labels. Unfortunately, this scheme is easily broken by an adversary that repeatedly inserts into the currently smallest gap. Even if we start with a gap of length 2^k, which requires k bits extra to encode each label, it would only take the adversary $k+1$ insertions to trigger relabeling. This worst case is perhaps not uncommon, since consecutive insertions into an XML docu-

* This work was supported by a National Science Foundation CAREER Award under grant IIS-0238386.

ment usually happen in nearby locations. Obviously, more robust solutions are needed.

Our contribution is a collection of data structures and techniques for maintaining order-based labeling for a dynamic tree-structured XML document. We propose two I/O-efficient data structures, *W-BOX* (*W*eight-balanced *B*-tree for *O*rdering *X*ML) and *B-BOX* (*B*ack-linked *B*-tree for *O*rdering *X*ML). W-BOX reduces the relabeling cost by limiting each relabeling operation to within a subrange; it uses a B-tree keyed on labels and piggybacks relabeling on tree balancing operations. B-BOX, on the other hand, avoids storing—and therefore updating—any label explicitly; it uses a keyless B-tree with *back-links* from children to parents, allowing labels to be reconstructed quickly on demand. The two structures together provide a nice tradeoff between update and lookup costs: W-BOX has logarithmic amortized update cost and constant worst-case lookup cost, while B-BOX has constant amortized update cost and logarithmic worst-case lookup cost. Both structures take linear space and use $O(\log N)$ bits per label. Both support efficient bulk loading and subtree insert/delete operations. We experimentally evaluate their performance and demonstrate their advantage over the naive approach.

In addition, we show how to adapt our data structures so that they can also return the *ordinal* labels of an element (defined formally in Section 3), which are the exact ordinal positions of its start and end tags within the document. Labels shown in Figure 1 happen to be ordinal; there are no gaps between adjacent labels. Ordinal labels contain the minimum number of bits per label, and are more efficient than non-ordinal labels for certain queries. However, they are more expensive to maintain: Both lookup and update costs become logarithmic for both W-BOX and B-BOX.

Finally, note that whenever a label changes value, all occurrences of the value in the database (e.g., in various indexes) must be updated, resulting in potentially unbounded update cost. This problem is inherent for any dynamic labeling scheme (including the naive approach) and can be solved by a level of indirection. However, this solution introduces an extra dereferencing cost which hurts query performance. We propose a combination of caching and novel logging techniques that can very effectively reduce this dereferencing cost.

2. Related Work

Many XML labeling schemes have been proposed in recent years to support efficient processing of path expression queries, which are the basic building blocks of XPath [19]. *Path-based labeling schemes* assign a code to each element, and the label of an element is simply the concatenation of the codes associated with the elements on its incoming path. With these labels, ancestor/descendant and parent/child axis steps can be processed by prefix matching. The main advantage of such path-based labeling schemes is that they can handle dynamically changing XML documents easily: When a new element is added, its label can be generated without modifying any of the existing labels. However, space overhead is a major concern, especially when the XML tree is tall, since the length of the label of an element is proportional to the length of its incoming path, and the majority of the elements in an XML tree are leaves with long incoming paths. Prefix matching is also more costly with these long labels. Examples of path-based labeling schemes include the two proposed by Cohen et al. [7] that do not use clues: Neither scheme handles tall XML trees well because label length grows linearly with tree height; furthermore, neither scheme maintains document ordering of siblings (only the insertion order of sibling can be recovered). One novel approach [20], which also tries to encode an element's incoming path, is to assign a prime number to each element, and the label of an element is formed by multiplying together the prime numbers associated with the elements on its incoming path. Ancestor/descendant axis steps then can be processed by checking if one label exactly divides the other. However, this approach still suffers from the same problem of long labels as other path-based schemes, as the resulting products of primes can become quite big.

The other class of popular labeling schemes includes the order-based interval and pre- and post-order labeling schemes, e.g., [21, 14, 11, 13, 12]. These schemes assign a pair of numeric start and end labels to each element, such that element e_1 is element e_2's ancestor if and only if the start label of e_1 precedes that of e_2, and the end label of e_2 precedes that of e_1. These schemes have several advantages over the path-based schemes. First, these schemes maintain document order. Second, each label only requires $O(\log N)$ bits, which is asymptotically minimum. Third, comparing numeric labels can be faster than prefix matching. Finally, fixed-size labels that fit in machine words are efficient and easy to implement. However, making such order-based labeling schemes dynamic remains a challenging problem. In contrast to the path-based schemes, repeated insertions will inevitably fill up the gaps between adjacent labels, necessitating a relabeling of part or all of the elements.

Hybrid labeling schemes that combine path- and order-based approaches are also possible. For example, Dewey-order encoding [18] labels each element by combining the local (sibling) order of each element on its incoming path. ORDPATH [15, 16] makes Dewey-order encoding dynamic using a clever "careting-in" scheme to support insertions. However, as an immutable labeling scheme, ORDPATH cannot escape the lower bound of $\Omega(N)$ bits per label established in [7]. Even for shallow XML documents, certain insertion sequences (such as the *concentrated* sequence we experiment with in Section 7) can result in $\Omega(N)$-bit labels.

The naive approach to order maintenance mentioned in Section 1 is to relabel all elements to make equally spaced labels when we run out of usable labels. This approach has been suggested in many existing systems, e.g., [13, 12]. However, this scheme is easily broken by an adversary that continuously inserts into the smallest gap. This worst case may indeed arise when, for example, a large number of elements (in an XML fragment) are inserted into one location in the document. Us-

ing floating-point numbers instead of integers (e.g., [1]) does not circumvent the problem: Although floating-point numbers have a larger range of values, the number of distinct values is still limited by the number of bits used in representation.

Maintaining all tags in the desired order under insertions and deletions is an instance of the well-known problem of maintaining ordered lists. The classic paper by Dietz [8] gives an algorithm that relabels $O(\log N)$ tags per insertion, amortized. With one extra level of indirection, the cost can be brought down to $O(1)$ [9]. The $O(\log N)$ cost can also be made worst-case, although the techniques are rather complicated and are primarily of theoretical interest. In [4], Bender et al. give a simplified version of the algorithm from [9], which is also easier to implement.

The database community has recently begun applying the above results to maintaining order-based XML labeling. Fisher et al. [10] use Bender's algorithm, and also provide a randomized algorithm which in practice performs slightly better in their experiments, although there is no theoretical analysis to guarantee its performance. Chen et al. [6] propose *L-tree*, which is parameterized to allow performance tuning and is also easier to implement than the algorithm from [9]. However, none of these structures are disk-based. In contrast, our BOXes are designed to be I/O-efficient, and we also develop techniques for avoiding the extra level of indirection necessitated by dynamic labeling schemes.

3. Preliminaries

For the purpose of this paper, we assume that an XML document can be modeled as a tree of elements. Each element e has a pair of start and end tags. In a well-formed XML document, e's start tag always precedes all tags of e's descendants, while e's end tag always succeeds all of them. An *order-based labeling scheme* (or *labeling* for short) is a function that assigns each element e a pair of integers $(l_<(e), l_>(e))$, where $l_<(e)$ is called the *start label* of e or the label of e's start tag, and $l_>(e)$ is called the *end label* of e or the label of e's end tag. A *valid* labeling is one that is consistent with the document order; that is, if a tag t_1 precedes another tag t_2 in the XML document, then the label of t_1 is less than that of t_2. Note that our proposed structures also work for other definitions of order (e.g., one based on pre-order and post-order traversals of the tree of elements), but for ease of presentation, we choose to use tag ordering within the document.

The use of order-based labeling in XML query processing has been discussed extensively in literature, so we will not elaborate it here; instead, we give an illustrative example. To see if element e_1 is a descendant of e_2, we can simply check if $l_<(e_2) < l_<(e_1) < l_>(e_2)$. It is usually much cheaper to evaluate this condition than traversing the element tree to check the ancestor-descendant relationship, which may take many steps.

The *ordinal* labeling is one that assigns label i to the i-th occurring tag in the document, for all $i \geq 0$ (assuming the ordinal is 0-based). Since the ordinal labeling leaves no gaps between consecutive labels, it makes the most efficient use of bits to encode labels. Furthermore, some queries are easier to answer with the ordinal labeling. For example, to see if e_1 is e_2's last child, we can simply check if $l_>(e_1) + 1 = l_>(e_2)$. With a non-ordinal labeling, we would need to check if there does not exist any label between $l_>(e_2)$ and $l_>(e_1)$, which is more expensive to evaluate. However, as we will see, the ordinal labeling is more difficult to maintain when the document changes.

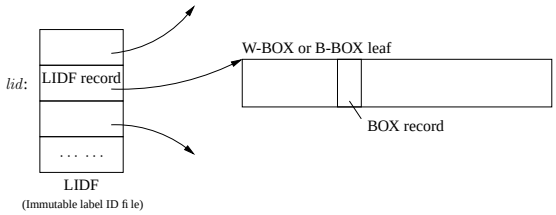

Figure 2. Immutable label ID file.

From immutable LIDs to dynamic labels Both W-BOX and B-BOX utilize a level of indirection to associate dynamic labels with *immutable label IDs* (or *LID* for short). Again, as motivated in Section 1, this indirection allows labels to be reassigned without disturbing references to them. We use a simple heap file called the *immutable label ID file* (or *LIDF* for short) to implement this indirection.

When a new XML element e is inserted into the document, we allocate two new records in the LIDF: one for e's start label and the other for e's end label. These are the *(start and end) LIDF records* of e. Their record numbers (or physical disk locations) serve as e's LIDs, which allow direct access to the LIDF records. Once LIDs are assigned, they are immutable, so they can be freely used in other XML indexes or even as XML element IDs. There is no need to keep LIDs in any order (although an obvious optimization is to allocate start and end LIDF records next to each other, so that a single I/O retrieves both records). When an element is deleted, its LIDF records can be reclaimed and allocated to a new element, allowing the LIDF to be stored compactly.

W-BOX and B-BOX both maintain two leaf entries for each XML element e: one for e's start label and the other for e's end label. We call them the *(start and end) BOX records* of e. As illustrated in Figure 2, e's LIDF records store pointers to the blocks containing corresponding BOX records. Thus, given a LID, we can retrieve the corresponding LIDF record with one I/O, and then the block containing the corresponding BOX record with another I/O. In Sections 4 and 5, we will see how to obtain the actual label from the block containing the BOX record. In Section 6, we discuss how to avoid the dereferencing cost.

Supported operations Here we briefly outline the operations on LIDF and W-BOX/B-BOX. The element/label operations include:

- `lookup`(lid): Return value of the label identified by lid.
- `insert-element-before`(lid): Insert a new element so that it immediately precedes the element tag whose label is identified by lid; return the two LIDs assigned to the new element's start and end labels. If lid identifies an element

e's start label, this operation effectively makes the new element the previous sibling of e. If lid identifies e's end label, this operation effectively makes the new element the last child of e. These two versions are sufficient for inserting any atomic XML element.

This operation is implemented using a low-level operation `insert-before`(lid_{new}, lid_{old}), which inserts a new BOX record (identified by lid_{new}) before an existing one (identified by lid_{old}) and writes the block address of the new BOX record to the corresponding LIDF record. We implement `insert-element-before`(lid) by first allocating two new LIDF records for the new element with LIDs (lid_1, lid_2), and then calling `insert-before`(lid_2, lid) and `insert-before`(lid_1, lid_2) in order. Thus, discussion of insertions in the rest of this paper will focus on `insert-before`.

- `delete`(lid): Remove the label identified by lid. To remove an element e, we need to call `delete` with the LIDs of both start and end labels of e. After the deletion, children of e, if any, effectively become children of e's parent.

In addition to the element/label operations described above, W-BOX and B-BOX also support bulk loading and subtree insertion and deletion operations. Details of these operations will be discussed in the next two sections.

Notations and metrics We use N to denote the total number of labels (including both start and end), which is twice the number of elements. We assume N to be a power of 2 for simplicity of presentation; our approach does not have this restriction. The minimum length of a label is thus $\log N$ bits. We define B, the size of an I/O block, as the number of bits per block divided by $\log N$, i.e., the number of minimum-sized labels that a block can hold. We also assume B to be a power of 2 for simplicity of presentation.

We assume that a block pointer takes $\log N$ bits, which should be more than enough because the number of blocks we need address is far less than N. Assuming that the LIDF is kept compact, we can also encode a LID using $\log N$ bits; thus, the space taken by the LIDF is $O(N/B)$.

We evaluate the performance of a labeling scheme using three metrics: (1) length of a label in bits, (2) total space used by all data structures, (3) number of block I/Os required for each operation. The last two metrics are standard in the analysis of I/O-efficient data structures. The first metric is also extremely important because shorter labels are faster to operate on by queries. In particular, fixed-length integer labels that fit in a machine word are easy to implement and have efficient hardware support.

4. W-BOX

The idea behind W-BOX is to store the labels using a balanced search tree, and leverage the tree-balancing operations to redistribute labels when they become too dense for a range. B-tree is one of the simplest I/O-efficient balanced search trees. Unfortunately, a regular B-tree results in too many relabeling operations. Thus, we use a *weight-balanced B-tree* [3] as the basis for our W-BOX.

Background on weight-balanced B-tree In a normal B-tree, each internal node must have between $\lceil b/2 \rceil$ and b children, where b is the maximum fan-out dictated by the block size. In a weight-balanced B-tree, constraints are imposed on the weight of each node rather than its fan-out. The *weight* of a node u, denoted $w(u)$, is defined to be the number of leaf entries stored in the subtree rooted at u. Given a *branching parameter* a and a *leaf parameter* k, we require the following: (1) All leaves are at the same depth. (2) A node at level i (assuming that leaves are at level 0) has weight less than $2a^ik$. (3) A node at level i (except for the root) has weight greater than $a^ik - 2a^{i-1}k$. (4) The root has more than one child. These properties are slightly different from those in [3]; the changes are intended to make the weight-balanced B-tree more efficient for our purpose.

Lemma 4.1 [1] *The number of children of a non-root internal node in a weight-balanced B-tree is between* $\lfloor \frac{a}{2} \rfloor$ *and* $2a + 3 + \lceil \frac{8}{a-2} \rceil$. □

Let b be the maximum internal fan-out dictated by the block size. By Lemma 4.1, for the weight constraints to be consistent with the maximum fan-out requirement, we may choose a to be the maximum value that satisfies $2a + 3 + \lceil \frac{8}{a-2} \rceil \leq b$, or equivalently $a = b/2 - 2$ (assuming $a \geq 10$). Accordingly, the minimum fan-out is $\lfloor \frac{a}{2} \rfloor = b/4 - 1$, which is lower than the requirement imposed by a regular B-tree, but is still $\Theta(b)$. We choose k such that $2k - 1$ is the maximum number of leaf entries that can be stored in a block.

If a node u at level i violates its weight constraint because $w(u) = 2a^ik$, we split it into two nodes u_1 and u_2 with roughly equal weights. More precisely, u_1 gets the leftmost m children of u, and u_2 gets the rest of the children of u, where m is the largest value for which $w(u_1) \leq a^ik$.

Next, we show that a node u will not be split again until there are $\Omega(w(u))$ new leaf entries inserted below u. This low rate of splits is crucial for the W-BOX to achieve its low amortized update cost. As we will see later, splitting u in the worst case causes all leaves below u's parent to be rewritten, which involves $O(w(u))$ I/Os. The low rate of splits implies that the amortized cost of splitting u will be only $O(1)$; therefore, overall, the amortized update cost will still be bounded by the height of the tree.

Lemma 4.2 *After a split of node u on level i into two nodes u_1 and u_2, more than $a^ik - 2a^{i-1}k$ insertions have to pass through u_1 (or u_2) to make u_1 (or u_2) split again. After a new root is created in a tree containing N records, at least $(a - 1)N$ insertions have to be done before the root is split again.* □

1 Because of space constraint, we omit the proof for this lemma as well as other proofs in this paper; they can be found in the full version of this paper [17].

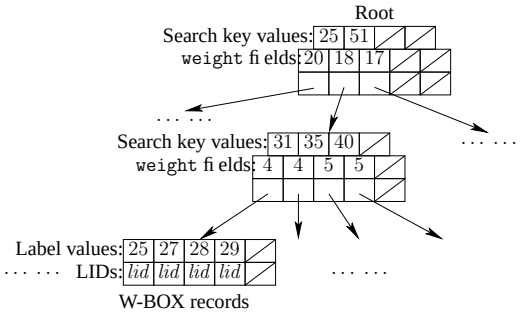

Figure 3. W-BOX nodes.

Data structure W-BOX, as its name implies, is a weight-balanced B-tree of W-BOX records with label values as search keys. A W-BOX leaf contains an ordered list of W-BOX records, each of which stores the value and the LID of a label. A non-leaf W-BOX node contains a list of child pointers separated by search key (label) values; each child pointer is associated with a `weight` field that stores the weight of the child. Figure 3 illustrate these two types of W-BOX nodes.

Conceptually, each node of the W-BOX is associated with a range of permissible values for labels stored in the subtree rooted at this node. Assuming M is the maximum integer that can be used for all labels, the root of the W-BOX is associated with the full range $[0, M]$. This range is then subdivided into b subranges of equal length. Each child of the root is assigned one of these subranges. We ensure the ordering among children is consistent with the ordering of their assigned subranges; however, it is acceptable to skip some subranges if the number of children is less than b. This process is carried on recursively down the tree. We require the range associated with a leaf to have length of at least $2k-1$. We maintain the invariant that all labels stored below a node u are within the range associated with u.

W-BOX inherits the space complexity of the weight-balanced B-tree. Since each individual field in a W-BOX node requires $O(\log N)$ bits, both the maximum fan-out (b) and the minimum fan-out ($b/4 - 1$) are $\Theta(B)$. Therefore, the W-BOX takes $O(N/B)$ total space and has a height of $O(\log_B N)$. The number of bits required for a label is determined by the size of the full range $[0, M]$, which is bounded roughly by b^h, where h is the height of the W-BOX. Therefore, the number of bits is roughly $\log b^h = h \log b = O(\log_B N \cdot \log B) = O(\log N)$. In fact, we show below that $\log N + 1 + \lceil \log(2 + 4/a) \cdot \log_a(N/k) + \log b \rceil$ bits are enough for a label in the W-BOX. For example, if we use 32-bit integers as labels, assuming $a = k = 64$, then the W-BOX can support at least 2.58 millon labels.

Lemma 4.3 *The height of a weight-balanced B-tree with N records is at most $1 + \lceil \log_a \frac{N}{k} \rceil$.* □

Theorem 4.4 *A W-BOX takes $O(N/B)$ space, and a W-BOX label takes no more than $\log N + 1 + \lceil \log(2 + 4/a) \cdot \log_a(N/k) + \log b \rceil$ bits, where $\log(2 + 4/a) < 1.3$ assuming $a \geq 10$.* □

Lookup The `lookup` operation of W-BOX, which returns the label of a given LID, is very straightforward. Following the pointer in the LIDF record identified by the given LID, we retrieve the W-BOX leaf u containing the W-BOX record we need. We then scan u looking for the W-BOX record with matching LID, and return the corresponding label value.

Theorem 4.5 *Given a LIDF record, the cost of retrieving the label from a W-BOX is one I/O.* □

Insert and delete To process `insert-before`(lid_{new}, lid_{old}), we start with the W-BOX leaf u pointed to by the LIDF record identified by lid_{old}. We create a new W-BOX record for lid_{new} right before the W-BOX record identified by lid_{old}, and we record the address of u in the new LIDF record for lid_{new}. To reflect the effect of this new insertion on node weights, we increment the `weight` fields for all child pointers that directly or indirectly point to u. These child pointers can be found by performing a regular B-tree lookup for any label stored on u. If no weight constraint is violated, then there must be some unused label value in the range associated with u, which allows us to assign a label to the new record (possibly requiring some existing records in u to be relabeled).

However, if the weight constraint is violated at some nodes, we must split them to enforce the constraint. We now discuss the steps involved in splitting a node u. Let $parent(u)$ denote u's parent. If u is the root, a new root is created as u's parent, and the height of the W-BOX grows by one. The new root extends u's range by a factor of b, and u's range becomes its first subrange.

- First, we check the two subranges within $parent(u)$ adjacent to the subrange associated with u. If either one is currently unassigned, we create a new sibling of u called v and assign it the unused range. We then relocate some entries in u to v such that v's weight is roughly one half of the original weight of u. Those entries that remain in u require no further processing. However, the relocated entries must be further processed. If the entries are leaf records, we relabel them with values within v's assigned range, and update their corresponding LIDF records to point to v. If these entries point to W-BOX subtrees, we subdivide v's range and assign them to these subtrees. This process proceeds recursively down the tree to the leaves, where the records in each leaf are relabeled with values within the leaf's assigned range.
- In the worst case, both subranges adjacent to u's are unavailable. Then, to make space for v, we reassign all children of $parent(u)$ with equally spaced subranges, and relabel all records in the subtree rooted at $parent(u)$. We note here that by the properties of the W-BOX, $parent(u)$ is guaranteed to able to accommodate v as an additional child (see the full version [17] for a detailed explanation).

Recall that the weight-balanced B-tree has the following property: As long as we can split a node u using $O(w(u))$ I/Os, the amortized update cost of the weight-balanced B-tree will be $O(\log_B N)$. In the worst case described above,

relabeling the entire subtree rooted at $parent(u)$ requires $O(w(parent(u))/b) = O(w(u) \cdot b/b) = O(w(u))$ I/Os, exactly as needed to establish the $O(\log_B N)$ bound.

We use the global rebuilding technique to handle deletions. To process `delete`(lid), we retrieve the W-BOX leaf u pointed to by the LIDF record identified by lid, and simply mark its W-BOX record as "deleted." We do not decrement the `weight` fields of any nodes. When a future insertion comes to a leaf, we first check if the leaf has any existing "deleted" W-BOX records. If so, one such record is reclaimed to make room for the new label (we might also need to relabel other labels in this leaf), again not changing any of the `weight` fields (hence no splitting). If the leaf has no existing "deleted" W-BOX records, we handle the insertion in the normal way as described previously. After we have collected $N/2$ deletions, we rebuild the whole structure. As will be shown, bulk loading the W-BOX takes $O(N/B)$ I/Os, so the amortized cost of a deletion is $O(1)$.

Theorem 4.6 *The amortized cost to insert and delete a label in a W-BOX is $O(\log_B N)$ and $O(1)$, respectively.* □

Note here that had we used a regular B-tree instead of a weight-balanced one, we would have been unable to provide the same low amortized update bounds. In general, a regular B-tree node u at level i can split every $\lceil b/2 \rceil^{i+1}$ insertions. On the other hand, there can be close to to b^{i+1} leaves below u's parent, yielding an amortized cost of 2^{i+1} I/Os per insertion, which is exponential in i. In contrast, since a weight-balanced B-tree imposes constraints on weights, the number of leaves below a node cannot vary by more than a constant factor, allowing us to bound the amortized relabeling cost to a constant.

For any algorithm designed to maintain ordered lists that has a logarithmic update cost, there is a well known technique [9] that can bring the update cost down to $O(1)$, while preserving the asymptotic storage and lookup cost. This technique can also be applied to W-BOX, though we do not recommend doing so for reasons we discuss in [17].

Bulk loading and subtree insert/delete Bulk loading a W-BOX from an XML document is extremely efficient because it requires no sorting. Simply scanning the document in order would produce all W-BOX records in exactly their intended order. Thus, with a single scan of the document, we can construct the LIDF and all W-BOX leaves in parallel. As each W-BOX leaf becomes full, we insert it into the W-BOX. When an internal node becomes full, instead of splitting it, we simply allocate an initially empty new sibling to its right; this strategy avoids the cost of relabeling by never relocating any entries. During the construction process, we always keep the rightmost node of each level in memory, so that insertions of leaves can be performed without additional I/Os. At the end of this process, we are left with a W-BOX whose only underflow nodes are those on the rightmost root-to-leaf path. We repair these underflow nodes by borrowing from or merging them with their left siblings. Overall, bulk loading costs $O(N/B)$.

To insert an entire subtree of XML data with N' tags, we first locate the W-BOX leaf u containing the insertion point. For $i = 0, 1, 2, \ldots$, we check whether v_i, the ancestor node of u at height i, has enough empty space to accommodate N' labels, i.e., $a^i b - w(v_i) > N'$. If so, we simply rebuild the subtree rooted at v_i to incorporate the new labels. The rebuilding process keeps all existing leaf entries in their original blocks, except those in u. This technique minimizes the cost of updating the LIDF for any W-BOX record that has relocated to a different block. In the worst case, all existing W-BOX records may have to relabeled, so the cost is $O((N + N')/B)$.

Deleting an entire subtree of XML data is similar. All N' labels to be deleted are clustered together in one continuous range. After deleting $O(N'/B)$ leaves and modifying up to two leaves, we look for the lowest common ancestor of these leaves with enough remaining weight to satisfy the weight constraint. We then rebuild the subtree rooted at this common ancestor, again trying to avoid relocating leaf entries. In the worst case, however, all existing W-BOX records may have to be relabeled, so the cost is $O(N/B)$. This bound also covers the cost of deleting LIDF records of deleted labels from the LIDF, which is of size $O(N/B)$.

Ordinal labeling support In order to support ordinal labeling, each non-leaf node entry needs to keep track of the total number of W-BOX records found within the subtree rooted at this entry. The `weight` fields almost fulfill this purpose, except that they also count records marked as "deleted" since we use the global rebuilding technique to handle deletions. Therefore, for a W-BOX with deletion support, we need to augment each non-leaf node entry with a `size` field that records the number of valid records found below the entry. This additional field does not alter the asymptotic space complexity of the W-BOX.

To retrieve an ordinal label given a lid, we first call `lookup`(lid) to find the regular label. We then perform a regular B-tree lookup using the regular label. We use a running counter initialized to 0. For each non-leaf node visited in this top-down traversal, we add to the counter all `size` fields located to the left of the child pointer leading to lid. Finally, in the leaf containing lid, we add to the counter the number of W-BOX records located to the left of lid. The value of the counter at the end of the traversal is the ordinal label. Therefore, the cost of looking up an ordinal label is dominated by that of the regular B-tree lookup, which is $O(\log_B N)$. For example, in Figure 3, the ordinal label for the non-ordinal label 28 is $20 + 0 + 2 = 22$, assuming that the `size` fields in this case happen to be equal to the `weight` fields.

To insert (or delete) a single W-BOX record, the `size` fields of all non-leaf node entries that lead to the inserted (or deleted) record need be incremented (or decremented) by 1. In the case of split, appropriate `size` fields must be updated too (details are straightforward and omitted). The I/O complexity of insertion is unaffected, but the amortized cost of deletion becomes $O(\log_B N)$, dominated by the cost of updating `size` fields. Bulk loading and subtree insert/delete operations can be modified in a straightforward manner to maintain the `size` fields. The extra cost does not affect the com-

plexity of these operations.

Further optimization for start/end pairs The basic W-BOX stores an element's start and end labels in two different W-BOX records, possibly located on different leaves, which require two separate I/Os to retrieve. However, requests for both start and end labels of an element occur quite frequently in query processing. We propose a variant of W-BOX, called *W-BOX-O*, that is optimized for retrieving start/end labels in pairs. The tree structure of W-BOX-O is identical to that of the basic W-BOX. Their difference lies in the format of leaf entries. In W-BOX-O, each start record maintains a pointer to the block containing its corresponding end record, and vice versa. Furthermore, the start record also keeps a local copy of the value of the end label. Thus, both start and end labels are obtained from the start record, without an extra I/O for the end record.

While W-BOX-O improves the lookup performance, we have to pay the price of maintaining extra information in the leaf entries. Maintenance is required in two cases. In the first case, when a leaf splits, half of its entries move to a new block. As a result, pointers storing these entries' block addresses become invalid and need to be updated (through the pointers in the reverse direction). These updates require $O(B)$ I/Os. Since this leaf cannot split again until it receives at least $\Omega(B)$ insertions, the amortized cost for updating these pointers is $O(1)$. In the second case, when a non-leaf node splits, a continuous range R of labels need to relabeled. The cost of relabeling is $O(\log_B N)$ I/Os amortized, as discussed before. What remains to be bound is the cost of updating the local copies of end labels stored by those start records outside R (start records inside R are updated as part of the relabeling process). At the first glance, this cost can be huge—up to the total number of labels in R. Fortunately, the hierarchical nature of XML plays into our hands. Consider the start records that need to be updated, i.e., those start records outside R that are linked to the end records inside R: Elements with these tags must form a segment of a path in the element tree, because these elements all contain the left endpoint of R. Hence, the number of such elements is bounded by D, the depth of the XML tree. Therefore, regardless of the number of records in R, the overall amortized cost of insertion into the W-BOX-O is $O(D + \log_B N)$.

Theorem 4.7 *The amortized cost of inserting a label into the W-BOX-O is $O(D + \log_B N)$, where D is the depth of the XML document tree. The amortized cost of deleting an entry is $O(1)$.* □

5. B-BOX

The design of B-BOX is motivated by the observation that updating labels is costly. Therefore, instead of physically storing the actual labels, we ensure that they can be reconstructed efficiently from the data structure whenever needed. Thus, B-BOX goes even further than W-BOX in trading read performance for faster updates. With the techniques to be described in Section 6 for enhancing read performance, we believe it is reasonable to make this tradeoff.

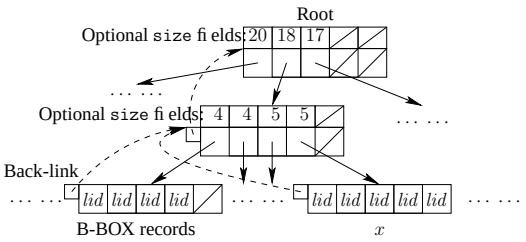

Figure 4. An example B-BOX.

Data structure B-BOX, as its name implies, is similar in structure to a regular B-tree constructed on the labels with normal balancing properties. Unlike B-tree, however, B-BOX do not keep any search key values in its nodes. A B-BOX leaf contains an ordered list of B-BOX records, each storing the LID for the label. A non-leaf B-BOX node contains an ordered list of child pointers. Every node except the root contains a *back-link* that points to the parent node. Figure 4 illustrates the structure of a B-BOX (ignore the optional `size` fields for now).

The label of a B-BOX record can be constructed by the path from the root to the leaf containing this record. Each B-BOX node on this path contributes to one component of the label. A non-leaf node contributes the (0-based) ordinal position of the child pointer that points to the next node on the path. The leaf at the end of the path contributes the (0-based) ordinal position of the B-BOX record. For example, in Figure 4, the label of the B-BOX record x is $(1, 3, 2)$. We will provide the details on how to obtain a label given its LID when discussing the `lookup` operation.

The multi-component labels of B-BOX somewhat resembles the Dewey-order encoding proposed in [18]. However, the crucial difference is that our labels are defined using a balanced B-BOX tree rather than the XML document tree, so our labels have a bounded length that is independent of the document structure.

B-BOX is a more compact structure than W-BOX. Each B-BOX leaf fits up to $B - 1$ B-BOX records, and each non-leaf B-BOX node has a maximum fan-out of $B - 1$. With standard B-tree analysis, it is easy to see that the B-BOX takes $O(N/B)$ total space and has a height of $O(\log_B N)$. B-BOX labels are also very compact. Because each component of a label takes at most $\log B$ bits and the number of components is equal to the height of the tree, the total number of bits in a label is $O(\log N)$. In fact, we show below that a B-BOX label never takes more than $\log N + 1 + \lfloor \frac{\log N - 1}{\log B - 1} \rfloor$ bits.

Theorem 5.1 *A B-BOX takes $O(N/B)$ space, and each label takes no more than $\log N + 1 + \lfloor \frac{\log N - 1}{\log B - 1} \rfloor$ bits.* □

Lookup The `lookup` operation, which returns the label for a given LID, cannot be performed in a top-down fashion as a regular B-tree, because there are no search key values in B-BOX nodes to guide the search. Even if there were, we would not know what key to search for—it is precisely what we are looking for in the first place. Instead, `lookup`(lid) proceeds bottom-up, starting from the leaf u containing the B-BOX record in question, which is obtained by following

the pointer in the LIDF record. We scan u looking for the B-BOX record containing lid; the ordinal position of this record within u gives us the last component of the label. Next, we follow the back-link to the parent of u. We then scan the parent looking for the entry that points to u; the ordinal position of this entry within the parent gives us the second-to-last component of the label. The process continues up the tree until reaches the root, where the first component of the label is determined.

Besides the extra I/O to obtain the pointer to the B-BOX leaf, the number of I/Os is equal to the the height of the B-BOX. Therefore we have the following theorem.

Theorem 5.2 *Given a LID, the cost of retrieving the label from a B-BOX is* $O(\log_B N)$. □

One of the most frequent operations used in XML query processing is the comparison of two labels. This operation can be performed in a B-BOX with potentially much fewer I/Os, especially if the two labels being compared are close to each other in document order. To carry out the comparison, we traverse the tree bottom-up in parallel starting from the two B-BOX records being compared. We stop as soon as their lowest common ancestor node is reached. The ordering of the labels is determined by the ordering of the two entries that lead to the corresponding B-BOX records.

Insert and delete Both `insert-before` and `delete` start with the B-BOX leaf pointed to by the LIDF record. The rest is similar to dynamic management of a regular B-tree, but with some additional bookkeeping involving LIDF records and back-links.

When a new B-BOX record is inserted before an existing record, we record the address of the leaf block in the corresponding new LIDF record. If the leaf overflows as a result of this insertion, we split the leaf into two: The first half of the B-BOX records remain on the old leaf while the rest move to a new leaf. For each B-BOX record relocated to the new leaf, we use its LID to access the corresponding LIDF record and update it to point to the new leaf. Finally, a pointer to the new leaf is added to the parent node, immediately after the pointer to the old leaf.

If the addition of this new pointer causes the parent u to overflow, a split of non-leaf node occurs. A sibling of u is created, and half of the entries relocate to this new sibling. For each relocated entry, we need to update the node that it points to, so its back-link points to u's new sibling. Finally, a pointer to the new sibling node is added to u's parent. In the worst case, the split can propagate all the way up to the root, causing the tree to grow.

If the deletion of a B-BOX record causes a leaf u to underflow, we first attempt to borrow a record from a sibling of u. If this attempt succeeds, in addition to relocating the borrowed B-BOX record, we must update the corresponding LIDF record to reflect the new block address of the borrowed record. If u's siblings do not have spare records, we merge a sibling into u by moving all records in the sibling to u. Again, corresponding LIDF records to be updated to point to u. Finally, the pointer to the sibling is removed from u's parent. An underflow non-leaf node is handled in a way similar to an underflow leaf, with the only difference being that we update back-links for relocated pointers (analogous to but instead of updating LIDF records for relocated B-BOX records).

In the worst case, split and merge could occur at every level of the tree; at each level, the cost is dominated by that of updating $B/2$ back-links or LIDF records. Therefore, the worst-case update cost of B-BOX is $O(B \log_B N)$. However, this worst-case scenario is extremely rare. Most of the time, an update affects only the leaf, without causing any reorganization across blocks or updates of LIDF records or back-links. In fact, the amortized update cost of B-BOX over a sequence of insertions can be shown to be $O(1)$: At worst, every $B/2$ insertions will fill up a leaf and force it to split at a cost of $O(B)$ I/Os; every $(B/2)^2$ insertions will fill up a parent of a leaf, causing additional $O(B)$ I/Os, and so on. Therefore, the amortized cost is $O(1) + O(B) \times (\frac{1}{B/2} + \frac{1}{(B/2)^2} + \frac{1}{(B/2)^3} + \cdots) = O(1)$.

It is also possible to obtain $O(1)$ amortized update cost over a sequence of updates containing both insertions and deletions, but we will need to relax the minimum fan-out requirement to $B/4$, which does not alter the asymptotic space complexity of the B-BOX. The standard B-tree minimum fan-out of $B/2$ is susceptible to frequent splits and merges caused by repeatedly inserting an entry into a full leaf and then deleting the same entry. However, with a fan-out of $B/4$, both split (of an overflow node with B entries) and merge (of an underflow node with $B/4 - 1$ entries and a node with $B/4$ entries) result in nodes with size of about $B/2$. Each such node then has to gain at least $B/2$ or lose at least $B/4$ entries before it will be split or merged again. While this smaller minimum fan-out requirement allows us to bound the amortized update cost for both insertions and deletions, it will result in a taller tree and longer labels (specifically, a label will take at most $\log N + 1 + \lfloor \frac{2(\log N - 1)}{\log B - 2} \rfloor$ bits). Therefore, the standard minimum fan-out requirement of $B/2$ is still recommended for workloads consisting of mostly insertions.

Theorem 5.3 *The worst-case cost of updating a B-BOX is* $O(B \log_B N)$*; the amortized update cost is* $O(1)$. □

Bulk loading and subtree insert/delete Bulk loading a B-BOX from an XML document is very similar to bulk loading a W-BOX. Again, no sorting is required. With a single scan of the document, we construct the LIDF and all B-BOX leaves in parallel. As each B-BOX leaf becomes full, we insert it into the B-BOX. When a non-leaf node becomes full, instead of splitting it, we simply allocate an initially empty new sibling to its right; this strategy avoids the cost of updating the back-links by never relocating any entries. Like bulk loading a W-BOX, we always keep the rightmost node of each level in memory to avoid additional I/Os. In the end we are left with a B-BOX whose only underflow nodes are those on the rightmost root-to-leaf path. We repair these underflow nodes by borrowing from or merging them with their left siblings; the number of additional I/Os is no more than $O(B)$ per level

(for updating back-links or LIDF records). Overall, bulk loading costs $O(N/B)$.

To insert an entire subtree of XML data into an existing B-BOX T, we first use bulk loading to construct a separate B-BOX T' for the data to be inserted (but instead of creating a new LIDF, we append to the same one used by T). Suppose that T' has h' levels. We "rip" T from the inserting point as follows. First, we split the leaf node u of T containing the insertion point right at that point, into u_1 and u_2. Then, we split u's parent node into two: One node contains all pointers up to and including the pointer to u_1, and the other node contains the pointer to u_2 and those following it. "Ripping" continues up T for a total of h' levels including the leaf. The result is a gap in which we can fit T' perfectly, thereby producing a combined B-BOX with all root-to-leaf paths having the same length. Finally, we repair underflow nodes (on the two sides of the gap) and overflow node (where we insert the root of T'). Overall, the cost is $O(N'/B + B \log_B(N + N'))$, where N' is total number of tags in the inserted XML subtree.

Conceptually, deleting a subtree of XML data simply reverses the steps involved in inserting it. Note that all labels to be deleted, say, N' of them, are clustered together in one continuous range. We "rip" the B-BOX starting from both endpoints of the range in parallel, until the two bottom-up processes meet at the same node. As a result, we have isolated the labels to be deleted into a number of subtrees in the B-BOX. We can then remove these subtrees and repair any remaining underflow nodes. Overall, the cost of updating the B-BOX is $O(B \log_B N)$. On the other hand, the cost of deleting corresponding LIDF records can be up to $O(N')$, as each deletion may result in a random I/O if these records are scattered across the LIDF. However, if the elements to be deleted were inserted at around the same time (either with bulk loading or subtree insertion), their LIDF records would be clustered and the cost of deleting them would be $O(N'/B)$.

Ordinal labeling support In order to support ordinal labeling, we augment each non-leaf node entry with a `size` field that keeps track of the total number of the B-BOX records found within the subtree rooted at this entry (Figure 4). This additional field does not alter the asymptotic space complexity of the B-BOX.

Looking up an ordinal label is similar to looking up a regular label, but uses a running counter. This counter is initialized with the number of B-BOX records located to the left of the B-BOX record in question on the same leaf. For each non-leaf node visited in the bottom-up traversal, we add to the counter all `size` fields located to the left of the entry that leads to the B-BOX record in question. The value of the counter at the end of the traversal is the ordinal label. For example, the ordinal label of x in Figure 4 is $2 + (4 + 4 + 5) + 20 = 35$. The complexity of the lookup operation remains $O(\log_B N)$.

To insert (or delete) a single B-BOX record, the `size` fields of all non-leaf node entries that lead to the inserted (or deleted) record need be incremented (or decremented) by 1. Thus, every update must go all the way to the root. In the case of split, merge, or borrowing from sibling, appropriate `size` fields must be updated too (details are straightforward and omitted). The worst-case update cost is unaffected, but the amortized cost becomes $O(\log_B N)$, dominated by the cost of updating `size` fields.

Bulk loading and subtree insert/delete operations can be modified in a straightforward manner to maintain the `size` fields. The extra cost does not affect the complexity of these operations.

6. Reducing the Cost of Indirection

As we have already pointed out in Section 1, the level of indirection that bridges the gap between immutable LIDs and dynamic labels introduces an extra dereferencing cost. Both W-BOX and B-BOX further trade off lookup performance for update performance. These additional costs come in the form of random I/Os, which neutralize the benefit of using order-based labeling in query processing, thereby making this approach unsuitable for a read-heavy workload. In this section, we address this issue using a combination of caching and logging techniques. We begin our discussion with the basic caching approach, and then show how logging can be combined to increase its effectiveness.

Basic caching approach Instead of using just LIDs to refer to dynamic labels indirectly, we augment each reference with the cached value of the label as well as a `last-cached` timestamp indicating when the cached value was obtained. The system also maintains a `last-modified` timestamp for each XML document being labeled, which tracks the time of the last modification made to the document that changed any existing labels. We assume that the `last-modified` timestamp is kept in main memory most of the time.

Given an augmented reference, a lookup operation first compares the `last-cached` timestamp stored in the reference with the `last-modified` timestamp associated with the document. If `last-modified` precedes `last-cached`, the cached label value in the reference is valid and is immediately returned without incurring any additional I/O. If `last-modified` is more recent than `last-cached`, the lookup operation starts with the LID and performs the normal steps as described in Sections 4 and 5. Then, it replaces the cached value with the label it obtained, and updates the `last-cached` timestamp. Here, the lookup operation pays the full cost of W-BOX or B-BOX `lookup`, but this cost, as we have shown in previous sections, is bounded and reasonably small.

For workloads with few updates, this basic caching approach works predictably well. Its lookup performance is practically as efficient as an immutable labeling scheme, while avoiding the problems of an immutable labeling scheme when updates do occur. On the other hand, a single `last-modified` timestamp may be insufficient to mitigate the effect of a steady update stream on read performance; we discuss a more effective approach next.

Caching and logging approach Instead of a single `last-modified` timestamp, we log the last k modifications to the document. Each log entry contains the timestamp

of the modification and a description of its effect on existing labels. For efficiency, the log should be kept in main memory and maintained as FIFO queue: when a new entry is logged the oldest entry is dropped.

This approach works because, fortunately, for our data structures, effects of modifications can be described succinctly and applied to an existing label without any additional information. For example, consider an ordinal labeling: The effect of inserting an element before an existing element with start label 142857 is that all existing labels greater than or equal to 142857 are incremented by 2. This effect can thus be logged as a range update $[142857, \infty) : +2$. The full description of logging techniques, which is more complicated for non-ordinal labeling, is provided in the full version [17].

A lookup operation starts by comparing the `last-cached` with the earliest modification timestamp logged. If `last-cached` is earlier, the cached label is unusable and the full cost of lookup must be paid. Otherwise, we "replay" the effects of all modifications with timestamps later than `last-cached` on the cached label and return the result without additional I/Os. Finally, we replace the cached label with this result and update `last-cached`.

A log with k entries gives roughly a k-fold boost in the effectiveness of caching because it takes k subsequent modifications instead of one to make cached labels unusable. On the other hand, a larger k also increases memory requirements and computational overhead.

7. Experiments

Our experiments evaluate the naive relabeling scheme (introduced in Section 1), W-BOX, B-BOX, and their variants on their I/O performance. We have implemented all algorithms in C++. W-BOX, B-BOX, and their variants are implemented using TPIE [2], a library that provides support for implementing and evaluating I/O-efficient algorithms and data structures. For all experiments, the block size is set to 8KB. Performance is measured by the number of I/Os. We present results obtained with main-memory caching turned off. Turning off caching exposes the full costs of I/O and makes the results easier to interpret. In practice, and as we have observed in experiments with caching turned on, our structures perform better with caching, especially because the root tends to be cached at all times.

Our experiments compare the following dynamic labeling schemes: W-BOX, W-BOX-O (the variant of W-BOX optimized for reading start/end labels in pairs), B-BOX, B-BOX-O (the variant of B-BOX with ordinal labeling support), and naive-k (the naive relabeling scheme with k bits of extra storage per label). All schemes use the LIDF described in Section 3 to map immutable LIDs to dynamic labels. For BOXes, LIDF records point to index leaves containing BOX records. For naive-k, each LIDF record directly stores the label value and the length of the gap between this and the previous label value. Our implementation of naive-k requires sorting the LIDF for relabeling. We assume that there is enough memory devoted to naive relabeling such that sorting can be done entirely in memory without extra I/O passes; this assumption produces a lower bound on the cost of naive-k to compare with our BOXes. We will see shortly that, even with this unfair advantage, naive-k is still inferior to our BOXes in most experiments.

Concentrated insertion sequence We start with a two-level XML document with $2,000,000$ elements and bulk load our data structures. Then, we insert a two-level XML subtree with $500,000$ elements, one element at a time, into the base document. Specifically, we insert the root element of the subtree first, as a child of the root of the base document. Then, we insert the first and the last children of the subtree root, followed by the second and the second-to-last, then the third and the third-to-last, and so on. In effect, each subsequent pair of insertions are "squeezed" into the center of a growing list of siblings. This insertion sequence behaves in a similar way as the adversary described in connection with the naive labeling scheme in Section 1, and it also creates the (near) worst case for many other labeling schemes, such as ORDPATH [15]. We have specifically designed this insertion sequence to stress-test our dynamic labeling schemes.

Figure 5 shows the average cost of element insertion (which involves inserting two labels) for various dynamic labeling scheme over the entire insertion sequence. Basic B-BOX, with its compact structure and the advantage of not having to materialize actual labels, has the best performance, confirming the amortized $O(1)$ bound predicted by our analysis. B-BOX-O, with support for ordinal labels, incurs some additional I/Os (up to the height of the tree) in maintaining the `size` fields, but still provides excellent performance. W-BOX suffers a little more, because the worst-case insertion sequence triggers frequent relabeling which is unavoidable for any labeling scheme that materializes labels. W-BOX-O, by further trading update performance for lookup performance, understandably has a higher update cost. On the other hand, all naive schemes perform extremely poorly compared with BOXes. Even with 256 extra bits, each insertion still costs 100 I/Os. Further increasing the number of extra bits gives diminishing returns, because the space and manipulation overhead of long labels quickly become significant.

Figure 6 shows, for each I/O cost, the fraction of insertions in the sequence that incurred *higher* than this cost. Note that both axes have logarithmic scale. This figure provides information about the distribution of individual costs. The results in this figure largely confirm our analysis and reaffirms the effectiveness of BOXes. The "steps" in the figure do provide some interesting insights into the operational details of the data structure. The drop in the B-BOX curve around $1,000$ I/Os, for example, represents the fraction of insertions that cause splits of internal B-BOX nodes.

Scattered insertion sequence The next experiment is designed to contrast with the concentrated one. We start with the same document of $2,000,000$ elements and insert another $500,000$. In this case, though, the inserts are spread evenly throughout the document. As shown in Figure 7, the naive policies, as expected, particularly shine in this test. These

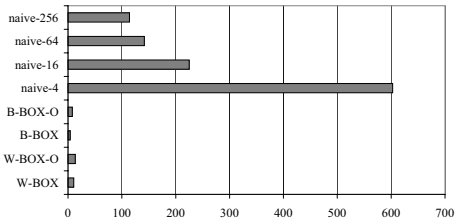

Figure 5. Amortized update cost, concentrated.

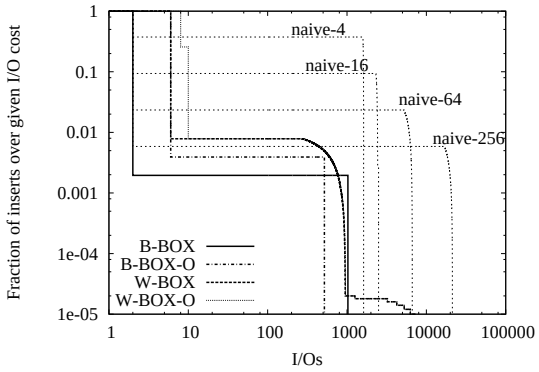

Figure 6. Distribution of update cost, concentrated.

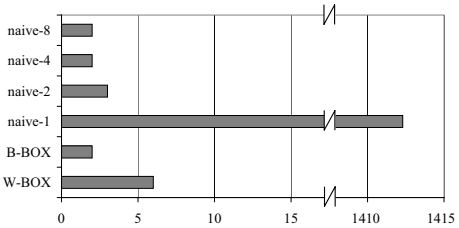

Figure 7. Amortized update cost, scattered.

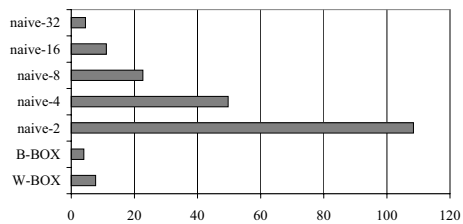

Figure 8. Amortized update cost, XMark.

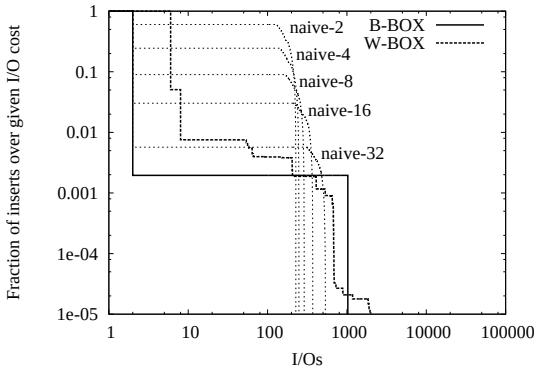

Figure 9. Distribution of update cost, XMark.

policies bank on no gap being overwhelmed with inserts; consequently, almost all inserts are done in constant time, and no relabeling is needed. The exception is naive-1, whose gap size is too small to accommodate even a single element. Therefore, relabeling is triggered constantly. The BOXes handle this case just as well. While they are designed to handle arbitrary insertion sequences gracefully, they too benefit from the evenly spread inserts.

XMark insertion sequence The next experiment is of the same flavor as the previous two, but now uses a document generated from the XMark benchmark with $336,242$ elements. We insert elements in a way to reflect how such a document might build up over time: Elements are added in document order of their start tags, one by one. As an example, for the document in Figure 1, we would first insert `site` (both its start and end tags), and then `regions`, `africa`, `item`, another `item`, `asia`, etc., in order. Note that this sequence—inserting all *elements* in document order—is not the same as inserting all *labels* in document order (which would behave like bulk loading), because end labels are inserted together with corresponding start labels without knowing subtree sizes in advance.

Our results represent the insertions taking place after the first $200,000$. This was done to "prime" the structures with an initial size. The results, shown in Figures 8 and 9, as expected, fall somewhere between the non-taxing scatter test and the adversarial concentrate test. No policies escape without doing any splits or reorganizations. The BOXes outperform the naive policies. The naive versions, relative to each other, perform the same way they did in the concentrate test.

Query performance At the end of each experiment, W-BOX and B-BOX heights were usually 3, but sometimes 2. As with B-trees, the number of elements must rise enormously to force the BOXes to grow in height. It is easy to see that with such low BOX heights, the logarithmic lookup costs for regular B-BOX and its ordinal version are, in practice, quite low (3–4 counting the indirection through LIDF, without caching the root). W-BOX, on the other hand, always looks up a label in two I/Os (again counting the indirection through LIDF), regardless of the tree height. If start and end labels are looked up together, W-BOX-O can do so in two I/Os total, two fewer than W-BOX. Finally, naive-k must also incur one I/O per label lookup because of the indirection through LIDF, which is unavoidable for any dynamic labeling scheme.

Other findings The previous experiments all insert one element at a time into the document, but the concentrate test in fact inserts a subtree of elements. In practice, this subtree, if known in advance, should be inserted using the bulk insert methods. The element-at-a-time test costs $5,401,885$ and $2,000,448$ total I/Os for W-BOX and B-BOX, respectively. With bulk insert methods, costs dramatically decreased to $11,374$ and 492, respectively.

It is an interesting exercise to determine which policies are hurt by the limit of machine word size, typically 32 bits. Our experiments use data sizes of $2,000,000$ elements, or $4,000,000$ labels overall. Labels for these keys can be differentiated with only 12 bits, far below machine word size. On

the other hand, the naive-k scheme requires additional k bits to maintain its gaps. In our experiments, the naive-32 scheme and those with even larger gap sizes all have labels that exceed machine word size. Therefore, aside from the I/O costs shown in the experiments, the naive policies also run slower because of inefficiencies in processing such long labels.

8. Conclusion

We have presented W-BOX and B-BOX, two novel structures for maintaining order-based labeling of XML elements. Most existing schemes fall prey to adversarial conditions that result in long labels or frequent expensive relabeling. Our structures temper the effects of any possible update pattern by trading off the costs of update and lookup, while providing good bounds for both. By basing the BOX schemes on formal, balanced tree structures, we are able to achieve provably good performance. Our experiments show that the BOXes indeed process updates, and especially adversarial updates, more efficiently than a naive gap-maintaining scheme. Currently, we are working on further improving the effectiveness of the caching and logging approach in Section 6, by using an efficient data structure for storing the log.

Acknowledgements We thank H. V. Jagadish and members of the Carolina Database Research Group for their insightful discussions of and related to this problem.

References

[1] T. Amagasa, M. Yoshikawa, and S. Uemura. QRS: A robust numbering scheme for XML documents. In *Proc. of the 2003 Intl. Conf. on Data Engineering*, pages 705–707, Bangalore, India, March 2003.

[2] L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient data structures using TPIE. In *Proc. of the 2002 European Symp. on Algorithms*, pages 88–100, Rome, Italy, September 2002.

[3] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. of the 1996 IEEE Symp. on Foundations of Computer Science*, pages 560–569, Burlington, Vermont, USA, October 1996.

[4] M. A. Bender, R. Cole, E. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proc. of the 2002 European Symp. on Algorithms*, pages 152–164, Rome, Italy, September 2002.

[5] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 310–321, Madison, Wisconsin, USA, June 2002.

[6] Y. Chen, G. Mihaila, S. Padmanabhan, and R. Bordawekar. L-Tree: a dynamic labeling structure for ordered XML data. In *Proc. of the 2004 Intl. Workshop on Database Technologies for Handling XML Information on the Web*, pages 31–45, Heraklion-Crete, Greece, March 2004.

[7] E. Cohen, H. Kaplan, and T. Milo. Labeling dynamic XML trees. In *Proc. of the 2002 ACM Symp. on Principles of Database Systems*, pages 271–281, Madison, Wisconsin, USA, June 2002.

[8] P. F. Dietz. Maintaining order in a linked list. In *Proc. of the 1982 ACM Symp. on Theory of Computing*, pages 122–127, San Francisco, California, USA, May 1982.

[9] P. F. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proc. of the 1987 ACM Symp. on Theory of Computing*, pages 365–372, New York, New York, USA, May 1987.

[10] D. K. Fisher, F. Lam, W. M. Shui, and R. K. Wong. Efficient ordering for XML data. In *Proc. of the 2002 Intl. Conf. on Information and Knowledge Management*, pages 350–357, New Orleans, Louisiana, USA, November 2003.

[11] T. Grust. Accelerating XPath location steps. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 109–120, Madison, Wisconsin, USA, June 2002.

[12] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. Viglas, Y. Wang, J. F. Naughton, and D. J. DeWitt. Mixed mode XML query processing. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, pages 225–236, Berlin, Germany, September 2003.

[13] 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. *The VLDB Journal*, 11(4):274–291, 2002.

[14] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, pages 361–370, Rome, Italy, September 2001.

[15] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-friendly XML node labels. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 903–908, Paris, France, June 2004.

[16] S. Pal, I. Cseri, O. Seeliger, G. Schaller, L. Giakoumakis, and V. Zolotov. Indexing XML data stored in a relational database. In *Proc. of the 2004 Intl. Conf. on Very Large Data Bases*, pages 1146–1157, Toronto, Canada, September 2004.

[17] A. Silberstein, H. He, K. Yi, and J. Yang. BOXes: Efficient maintenance of order-based labeling for dynamic XML data. Technical report, Duke University, Durham, North Carolina, USA, June 2004. `http://www.cs.duke.edu/dbgroup/papers/2004-shyy-xmlorder.pdf`.

[18] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 204–215, Madison, Wisconsin, USA, June 2002.

[19] World Wide Web Consortium. XML path language (XPath), November 1999. `http://www.w3.org/TR/xpath`.

[20] X. Wu, M. L. Lee, and W. Hsu. A prime number labeling scheme for dynamic ordered XML trees. In *Proc. of the 2004 Intl. Conf. on Data Engineering*, pages 66–77, Boston, Massachusetts, USA, March 2004.

[21] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 425–436, Santa Barbara, California, USA, June 2001.