Lazy XML Updates: Laziness as a Virtue of Update and Structural Join Efficiency

Barbara Catania
University of Genova, Italy
catania@disi.unige.it

Beng Chin Ooi
National University of Singapore, Singapore
ooibc@comp.nus.edu.sg

Wenqiang Wang
National University of Singapore, Singapore
wangwq@comp.nus.edu.sg

Xiaoling Wang
Fudan University, China
wxling@fudan.edu.cn

ABSTRACT

XML documents are normally stored as plain text files. Hence, the natural and most convenient way to update XML documents is to simply edit the text files. But efficient query evaluation algorithms require XML documents to be indexed. Every element is given a unique identifier based on its location in the document or its preorder-traversal order, and this identifier is later used as (part of) the key in the index. Reassigning orders of possibly a large number of elements is therefore necessary when the original XML documents are updated. Immutable dynamic labeling schemes have been proposed to solve this problem, that, however, require very long labels and may decrease query performance. If we consider a real-world scenario, we note that many relatively small ad-hoc XML *segments* are inserted/deleted into/from an existing XML database. In this paper, we start from this consideration and we propose a new *lazy* approach to handle XML updates that also improves query performance. The lazy approach: (i) completely avoids reassigning existing element orders after updates; (ii) improves query processing by taking advantages from segments. Experimental results show that our approach is much more efficient in handling updates than using immutable labeling and, at the same time, it also improves the performance of recently defined structural join algorithms.

1. INTRODUCTION

XML is currently the most widely accepted standardization effort in the area of document representation through markup languages, and it is rapidly becoming a standard for data representation and exchange over the Internet. Most research works in the area focus on how to efficiently query XML documents, and structural join is nowadays considered a core operation in optimizing XML path queries. Many index schemes have been proposed to efficiently evaluate structural joins. Most of them model an XML document as an ordered tree, and every element/attribute is given a unique identifier (label) based on its location in the XML document or its order in the preorder traversal of the tree. This label is later used as the key or part of the key in the index. The results returned from structural join algorithms are typically pairs of element/attribute labels, which are later used to evaluate other path query expressions.

Obviously, to fully evolve XML into a universal data representation and exchange standard, we must not only provide users with an efficient way to evaluate path queries, but we must also provide an efficient way to update XML documents. In this paper, we consider structural update, where a new element is inserted into (or removed from) an XML document. Attributes can be considered as subelements of an element and treated accordingly. The major problem of structural update is that, in order to maintain the correctness of query results, we need to update the labels of possibly a large number of elements when the original XML document has been updated, which makes the update operation very inefficient.

Previous attempts to solve this problem basically rely on various labeling schemes. [7] is an extended interval-based scheme where additional space is reserved for future insertion. This scheme fails if the space required to hold inserted nodes has exceeded the reserved space. Prefix labeling [4, 8, 11] allows each node to inherit its parent's label as the prefix of its own label so that inserting new nodes does not affect the labels of existing nodes, i.e., labels are immutable. Unfortunately, results presented in [4] establish that any immutable labeling scheme requires $\Omega(N)$ bits per label, where N is the size of the document, thus incurring in high storage overhead. Moreover, structural join algorithms using a prefix labeling scheme are less efficient than those using an interval-based labeling scheme because determining the containment relationship between two elements using prefix comparison is slower than using simple integer comparison. The prime number labeling scheme [12] overcomes some problems of prefix-labeling by assigning to each node a product of a prime number as its label and the containment relationship of two elements can be determined by the properties of prime numbers. The order of each element is preserved by maintaining a table of simultaneous congruences of element label sets and element order sets. Heavy computations are required when inserting a new element since computing simultaneous congruences is costly.

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA

A different approach to cope with updates while guaranteeing good query performance has been proposed in [9], where a dynamic, thus mutable, labeling scheme is used together with specific data structures that provide a good trade-off between query and update costs. However, it is not clear what the real overhead is in using the proposed data structure for structural join computation since no experimental results for that have been reported.

The problem of achieving at the same time XML update and structural join efficiency has therefore not been completely solved until now. Motivated by such observation, in this paper, we present a different solution to the problem of efficiently updating and querying XML documents in very dynamic environments. The proposed solution relies on the consideration that, in real world scenarios, XML document updates tend to be done in batch manner, i.e., multiple XML elements are inserted (or removed) together. As an example, consider the DBLP XML database. It contains many articles, books and proceedings and almost each day new articles and proceedings need to be added into the DBLP database. Due to the high frequency of update operations, updating the database after each single request of element insertion/deletion is not a feasible solution. Another example is represented by an on-line registration system. In such a system, once a user submits a registration form, an automatically generated XML document containing information about the user's identification, name, occupation, etc., is inserted into the system. In this case, multiple XML elements are inserted instead of a single element. In both examples, instead of inserting/deleting each element when requested, it seems more reasonable to generate XML segments corresponding to a set of elements that must be inserted (deleted) into (from) the whole database and then update the database once for each segment.

In this paper we present a new approach to dealing with both updates and queries in an efficient way, based on the usage of segments. We call it *lazy* since segments are used to avoid computations during both updates and queries. The whole XML database is modeled as a single *super document*, by simply adding a dummy root to all the existing XML documents, and update operations correspond to inserting (or removing) XML "segments" into (or from) the super document. Note that, assuming to start from an empty database, any XML database corresponds to a single XML document composed of several segments, each of them being an XML document by itself.

In the considered model, each element has two positions. The first one is its *local position* with respect to the XML segment it belongs to. The second one is its *global position* in the super document. The local label will never change once it is assigned to an element, but it is not unique. On the other hand, the global label is unique but it will change if an update occurs. From these considerations it follows that if a local label is used as the key (or part of the key) in the element index, after an update we can avoid updating the existing element labels. However, since local labels are not unique, they cannot be directly used in structural join.

The key observation here is that the number of inserted (or removed) segments is likely to be significantly less than the number of XML elements these segments contain. For example, an XML document corresponding to a registration form may contain 20-30 XML elements. This gives us the inspiration to build an in-memory *update log* to record the information of every segment. The information recorded must be sufficient to support structural join between segments. We are aware that, after many insertions, the size of the update log could grow too large to be held inside memory. However, as we will show in our experiments, the size of the update log is small enough not to pose a problem for modern machines. Moreover, the database administrator can rebuild the index for the whole XML database during maintenance hours, and therefore the update log can be periodically cleared for further update operations.

Another difficulty about updating XML documents is that in real world scenarios, XML documents are very large and normally stored as plain text files. Therefore, inserting (removing) a segment is done by simply text editing, e.g., only the start location in the super document and the length of the inserted (removed) segment are available to us. The update log must therefore allow us to identify the structural information about the segments given only these two values.

Finally, we note that the usage of segments does not require the definition of specific XML query processing techniques. Rather, the update log can be easily integrated in existing structural join algorithms. However, segment-aware query processing techniques can be defined to reduce query processing costs. For example, segments can be used for parallelizing query processing or defining new segment-aware structural join algorithms. In this paper, we present one of such algorithms and we show that it improves query performance compared to non-segment based algorithms.

The contributions of this paper are the following:

- We propose a new *lazy* approach to handle XML document update operations using a novel in-memory update log.
- We present update algorithms for the proposed update log, assuming that each operation takes as input the position in the super document where the segment has to be inserted/deleted and the length of the segment.
- We present a structural join algorithm that works with our *lazy* approach. The algorithm has been obtained by extending the stack-based structural join algorithm proposed in [1] to deal with segments.
- We conduct experimental study on the update and structural join operations. The results show that our approach is significantly more efficient than using existing dynamic labeling approaches for updates and, additionally, it improves query processing performance.

The paper is organized as follows. Section 2 presents related work. Section 3 introduces the structure of the update log together with update algorithms, and suggests how to build an element index with an update log. Section 4 presents a structural join algorithm based on the element index and the update log usage. Experimental results are then discussed in Section 5 and concluding remarks are presented in Section 6.

2. RELATED WORK

Compared to path query evaluation, the topic of updating XML documents has received much less attention from the research community. In [10], a set of basic update operations for XML data is proposed and the XML query language,

XQuery, is extended to incorporate these update operations. In [6, 13], efficient update algorithms for summary indexes have been proposed. The work in [6], based on the notion of graph bisimilarity, analyzes two kinds of updates – the addition of a subgraph, intended to represent the addition of a new document to the database, and the addition of an edge, to represent a small incremental change. In [13], algorithms for maintaining a minimal index are provided.

Several labeling schemes have been proposed recently to solve the update problem. Among the proposed immutable labeling schemes (all incurring in high storage overhead according to [4]), [11] proposed an integer-base prefix labeling scheme where each label inherits its parent's label as its prefix and the n^{th} child of a node is labeled with integer n. Obviously, this scheme fails if there are more than 10 child nodes. Adding some sort of delimiter in the label solves this problem, but it also brings significant storage and query process overhead since the delimiter must be stored with the label and time to indicate the parent-child relationship is no longer constant. The binary-based dynamic labeling scheme proposed in [4] encodes every label as a binary string. The label of the $(i+1)^{th}$ child is generated by adding one to the label of the i^{th} child and, if the label of the $(i+1)^{th}$ child consists of all ones, its length is doubled by adding a sequence of zeros. Obviously, when the fan-out of the XML tree is large, the size of the label could become very large. Moreover, the proposed labeling scheme does not maintain sibling ordering. In [8], a dynamic variant of the Dewey order is provided. Similarly to the other schemes, the size of generated labels can be very large. The prime number labeling scheme proposed in [12] makes use of a property of prime numbers: if an integer A has a prime factor which is not a prime factor of another integer B, then B is not divisible by A. Every node is first given a prime number as its self label. Its label is then assigned as the product of its self label and the label of its parent. Therefore, a node X is an ancestor of another node Y if and only if $label(Y)\ mod\ label(X) = 0$. The order of the nodes is maintained by a table of simultaneous congruence values which maps nodes' self labels to order numbers. To keep the value of the simultaneous congruence small, which is easier to compute, the number of simultaneous congruences may be large. Therefore, when new nodes are inserted, recomputing large number of simultaneous congruences may be required. In order to solve space problems of labeling schemes and to guarantee at the same time good query performance, in [9] a dynamic, thus mutable, labeling scheme is used together with specific data structures (W-BOX and B-BOX) that provide a good trade-off between query and update costs. W-BOX uses weight-balanced B-trees to reduce the relabeling overhead, obtaining a logarithmic amortized update cost and constant worst-case lookup cost, whereas B-BOX further reduces update costs, resulting in a constant amortized update time and logarithmic worst-case lookup cost, by avoiding the storage of labels, that can however be reconstructed starting from the proposed data structure, a variant of B-tree. The reported theoretical and experimental results are very good but no experimental results for queries are reported.

Structural join, which is now considered a key issue in optimizing XML queries, has been intensively studied in recent years. Various techniques have been proposed to efficiently perform structural join. The first two approaches [7, 14] are based on the application of some variations of the relational merge-join algorithm to lists of elements. More recent approaches have improved these basic techniques with a stack mechanism [1, 2], and they are further optimized in [2] to reduce the size of the generated intermediate results by using an holistic approach. Structural join algorithms are based on the containment relationship between elements. They use labeling schemes to check ancestor-descendant, as well as parent-child, relationships.

Indexing techniques are used to improve the efficiency of structural join algorithms. The approach presented in [7] relies on the use of three main indexes: the element (attribute) index, for indexing elements (attributes) with respect to their name, and the structure index, for indexing elements and attributes with respect to the document they belong to. In [3], the approach presented in [7] is further improved by inserting additional information (like sibling pointers) in the indexes. The new data structure, based on B^+-trees, allows the structural join algorithm to skip descendants that do not match the considered structural relationship. The XR-tree, proposed in [5], is a further improvement of the idea presented in [7] to skip not only descendants but also ancestors. It supports the detection of ancestor-descendant relationships in logarithmic time. In [2], XB-trees are introduced as a variant of B-trees for indexing the positional representation of elements in the XML tree to be used in the proposed holistic algorithm.

3. THE DATA STRUCTURE

In this section, we introduce the in-memory update log and corresponding update operations. We then present the element index we use together with the update log.

3.1 Preliminaries

As we have mentioned in Section 1, by adding a dummy root, the whole XML database, whether it has been organized with a tree or many sub-trees, can be considered as one *super document* and XML update operations are therefore modeled as inserting (or removing) XML segments into (or from) the *super document*. It is obvious that every segment inserted (or removed) must be a valid XML document itself so that the validity of the whole XML database is preserved, which also implies that every segment, except the dummy root, is included in at least one other segment. As part of the *super document*, every segment s has a unique starting position in the *super document*, which we refer to as its *global position*, denoted by $s.gp$, with respect to the *super document*. Each segment has a length, which is denoted by $s.l$. Based on global positions and length, we can then define a segment *containment* relationship.

DEFINITION 1. *The global position of a segment s, denoted by $s.gp$, is the offset of the first element of s inside the super document. The length of s, denoted by $s.l$, is the number of characters in s. A segment s_1 contains a segment s_2 if and only if $s_1.gp < s_2.gp$ and $s_1.gp + s_1.l > s_2.gp + s_2.l$. s_1 is the ancestor and s_2 is the descendant segment. If there exists no other segment s_3 such that s_1 contains s_3 and s_3 contains s_2, s_1 directly contains s_2. In this case, s_1 is the parent and s_2 is a child segment of s_1.* □

In Figure 1(a), rectangles represent XML segments and dashed lines the direct containment relationships among different segments. We can see that segment 3 is directly contained in segment 2, which is its parent. Segments 1 is its

ancestor. Segments 4 and 5 are child segments of segment 3 and segment 6 is a descendant of segment 3.

Besides a *global position*, we also assign a *local position* to every segment (except the root), denoted by $s.lp$. The local position of a segment s_2 with respect to its parent s_1 is simply the number of characters in s_1 preceding s_2 and not contained in any left sibling of s_2, at the time when s_2 is being inserted.

DEFINITION 2. *Let segment s_1 be the parent of segment s_2. The local position of s_2, denoted by $s_2.lp$, is defined as* $s_2.lp = s_2.gp - s_1.gp - \bigcup_{(s \text{ is a left sibling of } s_2)} s.l.$ □

Local positions, once assigned to a segment, never change. Indeed, the only updates that may vary the local position of a segment s_2 are insertions and deletions of left siblings of s_2. However such insertions (deletions) increase (decrease) the global position of s_2 while the global position of its parent does not change; thus, according to Definition 2, $s_2.lp$ does not change too. Moreover, local positions are not unique whereas global positions are.

Figure 1(b) represents the super document corresponding to Figure 1(a), pointing out segment length, global and local positions, assuming, for the sake of simplicity, each element is a dummy element which contains no character content and each tag requires n characters for its storage.

3.2 Structure of Update Log

The update log consists of two data structures. The first consists of a B$^+$-tree to easily access segment information. The second is the tag-list, which is a simple inverted list that maps element tags to segments in the super document.

Figure 2 illustrates the structure of the B$^+$-tree. Keys are segment identifiers (*sid*), which are unique identifiers generated by the system when a new segment is inserted. The B$^+$-tree, called Segment B$^+$-tree (SB-tree), associates *sid*'s with the following information: the segment global position *gp*, the segment length *l*, the segment local position *lp*, a pointer to its parent, pointers to its children, sorted by global positions in ascending order. Every node in the B$^+$-tree corresponds to a segment in the super document. Since the leaf level is organized as a tree-like data structure, we refer to it as the ER-tree (sEgment-Relationship tree) in the remainder of this paper. The ER-tree is a segment based representation of a document, according to the sequence of insertions/deletions that have been executed. The root node represents the dummy root of the super document. As we will see in Sections 3.3 and 4, the ER-tree simplifies update operations whereas the B$^+$-tree is needed for query execution. Figure 3 illustrates the structure of the ER-tree for the segments shown in Figure 1.

Obviously, the size of the SB-tree is *O(N)*, where N is the number of segments contained in the super document. We claim that N will be quite small compared to the size of the document, thus it is reasonable to assume that the SB-tree resides in main memory.

Figure 4 illustrates the tag-list structure of the segments shown in Figure 1, where we suppose the tag ids for tag A and B are tid-1 and tid-2 respectively. Each list stores not only the segment id but also a *path* for every segment in the ER-tree. The *path* of a segment is actually the concatenation of the segment ids of all its ancestor segments plus its own segment id. Tag ids are sorted by ascending order, and within the path lists attached to the tags, the paths are

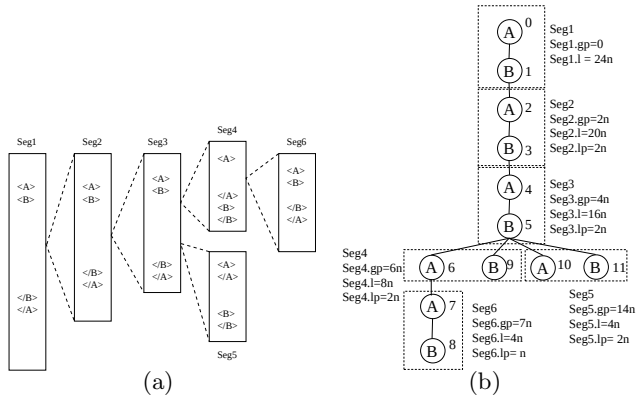


Figure 1: (a) Segment containment relationship; (b) The corresponding global document

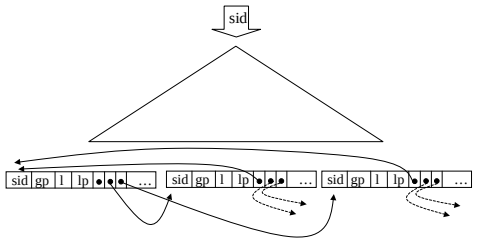


Figure 2: SB-tree (Segment B$^+$-tree)

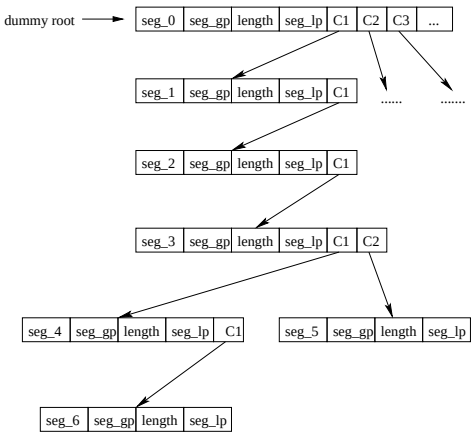


Figure 3: ER-Tree (sEgment Relationship tree)

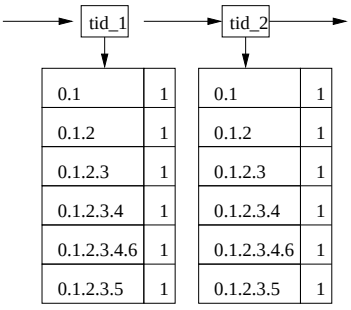


Figure 4: Tag-list

sorted by global positions of the corresponding segments. The reason for storing the path is that it allows us to more efficiently perform operations needed by the structural join algorithm, as we will see in Section 4. The path is computed when the segment is inserted into the super document and the length of the path is at most *O(N)*, which occurs in the most highly nested case where every segment has at most one child segment, i.e., the ER-tree is reduced into a single linked list. We also associate the numbers of element occurrences for each tag id in each segment together with the paths, which helps us to determine if a path should be removed from the path list when we remove segments from the super document. We will describe this in detail in Section 3.3. The size of the tag-list is $O(TN^2)$ where T is the number of different tag ids and N is the number of segments. This is the worst case estimation when every segment contains all tags and when the segments are most highly nested as we described above. We are aware that there could be more compact ways to represent the tag-list, but this approach is easier to maintain and since all the ids are simply numbers, the total size of the tag-list is still small enough to fit into the memory of modern machines.

PROPOSITION 1 (SPACE COMPLEXITY). *Let N be the number of segments and T the number of element tags. The space complexity of the SB-tree and tag-list is $O(N)$ and $O(TN^2)$, respectively.* □

3.3 Updating the Update Log

The assumptions under which we define update operations are: (i) for each insertion/deletion of a segment, we assume to know only its global position and its length; local positions are transparent to the application and are detected during update execution; (ii) only segment information can be modified; elements are only inserted or deleted but never modified; (iii) inserting a segment into the super document results in adding a new node into the SB-tree, but removing a segment from the super document does not necessarily mean deleting a node from the SB-tree.

```
AddNewSegment_Start(new)
1.   For each node m in ER-tree s.t. m.gp > new.gp
2.      m.gp = m.gp + new.l
3.   AddNewSegment(ER_root,new,empty_path)

AddNewSegment(root,new,path)
1.   path := path + root.sid
2.   root.length := root.length + new.l
3.   If a child node k of root is an ancestor of new
4.   Then
5.        AddNewSegment(k,new,path)
6.   Else
7.      insert new.sid into the child list of root
8.      path := path + new.sid
9.      lengthSum := 0
10.     for each left sibling n of new
11.        lengthSum := lengthSum + n.l
12.     new.lp = new.gp - root.gp - lengthSum
13.     insert new.Node into the SB-tree
14.  Endif
```

Figure 5: Adding a segment into the SB-tree

In case of insertion of a segment into the super document, the system will automatically generate a segment id for it. Then, both the SB-tree and the tag-list need to be updated. More precisely, in case of insertion, we need to: (1) update the global positions of relevant nodes in the ER-tree; (2) compute the needed information for the segment from the ER-tree; (3) add a node corresponding to the segment into the SB-tree; (4) update the tag-list accordingly.

Figure 5 gives the algorithm for Steps (1), (2), and (3). Function `AddNewSegment_Start` first increases global positions greater than that of the new segment by the length of the new segment. Then, it calls function `AddNewSegment`, that recursively transverses the ER-tree and adds the new node into the proper location in the child list of its parent node. The path variable is initially empty. The root parameter is initially set to the root node of the ER-tree. We first append the segment id of the current root node into the path variable and increase the length of the current root node by the length of the segment represented by the new node. Then, according to Definition 1, we check if any of the children of the current root fully contains the inserted node. If there exists such a child node, the current root is not the parent node of the inserted node and we recursively call the `AddNewSegment` function, setting this child node as the new root. If there is no such child node, which means that the current root is the parent of the inserted node, we simply insert the node into the child list of the current root. Of course, the new node must be inserted into a proper location in the child list such that the order in global positions of the child nodes is still preserved. New segment's local position is then computed according to Definition 2 and the node inserted into the SB-tree. Although child lists could be long after many insertions, we can search or update a child list of size K in *O(log(K))* time because the SB-tree resides in memory and efficient algorithms like binary search can be used. It is obvious that the overall cost of adding a node into the ER-tree is bounded by *O(N)*, where N is the number of segments. This worst case also occurs in the most highly nested case we described earlier. Concerning global position changes, again, although the cost of this propagation process is *O(N)*, it is still efficient in most cases because the SB-tree is memory resident. Since the cost of inserting a node into the SB-tree is $O(log(N))$, the overall time to update the SB-tree is $O(N)$.

Once we have obtained the path of an inserted segment, we can update the tag-list accordingly. If the inserted segment contains the tag with tag id T, then its path (computed by the `AddNewSegment` function) is inserted into the path list associated with tag id T according to global ordering. The cost of locating a tag id is *O(log(T))*, where T is the number of different tags. The cost of inserting a path into the path list associated with a tag id is bound by *O(log(N))*. Therefore the total cost of updating the tag-list is *O(p(log(T)+log(N)))*, where p is the number of tags contained in the inserted segment.

Compared to inserting a new segment, removing a segment from a super document is a bit more complicated. Indeed, the removed segment may not be any of the existing segments recorded in the SB-tree. To clarify the problem, let *seg* be the segment to be removed. We formalize the relationship between the removed segment and a segment k in the ER-tree (except the dummy root) into three main cases:

1. *seg* is *contained* in k, i.e., $k.gp < seg.gp$ and $k.gp + k.l > seg.gp + seg.l$. The length of k is reduced by *seg.l*.
2. *seg contains* k, i.e., $seg.gp < k.gp$ and $seg.gp + seg.l >$

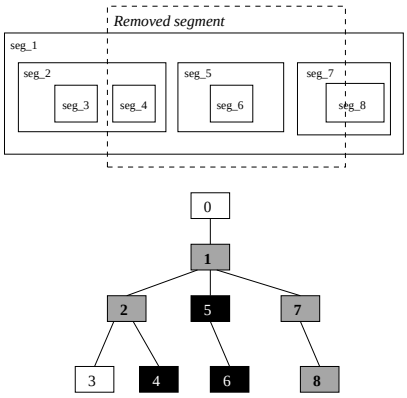


Figure 6: Removing a segment

k.gp + k.l. *k* and all its descendants are deleted from the ER-tree in this case.

3. *seg* intersects *k*. If *k.gp* < *seg.gp* < *k.gp + k.l* < *seg.gp + seg.l*, it is a *left intersection*; if *seg.gp* < *k.gp* < *seg.gp + seg.l* < *k.gp + k.l*, it is a *right intersection*. The length of *k* is reduced by *k.gp + k.l - seg.gp* (for left intersection) or *seg.gp + seg.l - k.gp* (for right intersection).

Besides the previous cases, the global position of segments starting after *seg* ending position is reduced by *seg.l*.

The relationship between the removed segment and the super document is a combination of the cases listed above. In Figure 6, the removed segment (dashed box) is contained in segment 1, contains segments 4, 5 and 6, left intersects segment 2 and right intersects segments 7 and 8. In the corresponding ER-tree, black nodes refer to those segments that are to be completely deleted from the tree, gray nodes refer to those segments that are affected by the removed segment in terms of segment length and global position, white nodes refer to those segments that are not affected when deletion occurs. Node 0 is the dummy root, whose global position never changes.

Figure 7 gives the algorithm for updating the SB-tree when a segment is removed from the super document. Function `RemoveSegment_Start` takes the current segment to be removed as parameter. We first reduce the global position of segments starting after *seg* ending position by the length of the current removed segment. Then, we call the recursive function `RemoveSegment` with the root of the ER-tree and the segment to be removed as parameters. In calling such function, the root segment will always contain the segment to be removed. `RemoveSegment` first updates the root length, then it checks the relationship between the current removed segment and the child segments of the current root. If the removed segment is contained in a child node, we just call function `RemoveSegment` recursively. If a child node is contained in the current removed segment, we remove the child node from the child list of the current root and from the SB-tree, together with all its descendants. If the removed segment left intersects a child node, we update the length of the removed segment by using an auxiliary segment, as indicated in lines 12-13, and call function `RemoveSegment` recursively, setting the child node as new root. If the removed segment right intersects a child node, we update the length and the global position of the removed segment by using an

```
RemoveSegment_Start(seg)
1.  For each node m in ER-tree s.t. m.gp > seg.gp+seg.l
2.     m.gp = m.gp - seg.l
3.  RemoveSegment(ER_root,seg)

RemoveSegment(root,seg)
1.     root.l := root.l - seg.l
2.     For every child node k of root
3.        If seg is contained in k
4.        Then
5.           RemoveSegment(k,seg)
6.        Else if k is contained in seg
7.        Then
8.           remove k from the child list of root
9.           remove k and its descendant nodes from SB-tree
10.       Else if seg left intersects k
11.       Then
12.          seg_aux.gp = seg.gp;
13.          seg_aux.l := k.gp + k.l - seg_aux.gp
14.          RemoveSegment(k,seg_aux)
15.       Else if seg right intersects k
16.       Then
17.          seg_aux.gp = k.gp;
18.          seg_aux.l := seg_aux.gp + seg_aux.l - k.gp
19.          RemoveSegment(k,seg_aux)
20.          k.gp := k.gp + seg_aux.l
21.       Endif
22.    Endfor
```

Figure 7: Removing a segment from the SB-tree

auxiliary segment, as indicated in lines 17-18, we call function `RemoveSegment` recursively, and we update the global position of the child node. The usage of an auxiliary segment allows the algorithm to maintain the correct information associated with the segment to be removed when checking the other child nodes. During the recursive removing process, we also record the information about those segments in the ER-tree which are affected. If a segment is completely contained in the removed segment, we simply record its segment id. If only part of a segment is contained in the removed segment, we record its segment id and the start/end position of that part. We do this for the convenience of removing corresponding element records from the element index, which we will describe in Subsection 3.4. The cost of recursively updating the ER-tree is bounded by *O(N)*. The worst case happens when the segments are most highly nested and the removed segment intersects all of them. Since the cost of deleting a node from the SB-tree is $O(log(N))$, the total cost for updating the SB-tree is $O(Nlog(N))$.

The tag-list is updated after updating the element index, as described in the next subsection. To update the tag-list when a segment is removed, we need to know the tag name and the number of elements actually removed from the super document, since a path has to be deleted only if no more elements with that tag are contained in the segment after the deletion. The information concerning the type and the number of elements removed is computed when we actually perform the delete operation in the element index. The cost of updating the tag-list for one tag id is bound by *O(log(T)+mlog(N))*, where m is the number of segment paths to be removed from the path list, T is the number of different tags in the super document, and N is the number of segments. *log(T)+log(N)* is the worst case cost of locating a single path in the tag-list for a given tag id. The worst case occurs when the tag id is contained in all seg-

ments and a path is to be removed from the path list associated with this tag id. Therefore, the total cost of updating the update log when a segment is removed is *O(Nlog(N) + p(log(T)+mlog(N)))*, where p is the number of distinct tag names the removed segment contains.

PROPOSITION 2 (UPDATE COMPLEXITY). *Let N be the number of segments, T the number of distinct element tag ids, p the average number of distinct element tag names in a segment, and m the average number of paths to be removed from a path list in the tag-list. The segment insertion cost is $O(N + p(log(T) + log(N)))$ and the segment deletion cost is $O(Nlog(N) + p(log(T) + mlog(N)))$.* □

3.4 Element Index

The element index is simply a B$^+$-tree. Every record in the index represents an element and the key of the B$^+$-tree index is the tuple (*tid*, *sid*, *start*, *end*, *LevelNum*), where *tid* is the tag id of the element, *sid* is the segment id the element belongs to, *start* is the starting position of the element in the segment identified by *sid*, i.e., the *local position* of the element; *end* is the local ending position of the element; *LevelNum* is the depth at which the element appears in the document. According to the proposed numbering, each element is univocally identified by the pair $(sid, start)$.

Search and insertion operations for the element index are the same as those for standard B$^+$-trees. But when element records are removed from the element index, we need to record the number of removed elements with the same tid and sid, which is necessary when we decide if a segment path should be removed from a path list as we mentioned when we discussed the impact of deletion on the tag-list in Subsection 3.3.

4. QUERY EVALUATION

Under the lazy XML update approach, existing structural join algorithms can still be used to compute pairs of ancestor/descendant or parent/child elements. As we discussed in Section 3.4, elements in the element index are identified by the id of the segment in which they appear and their local position. In order to compute structural joins, we first need to access the SB-tree to get the global position of the segments in which elements are contained. From that, element global starting and ending positions can be generated and structural joins computed by using any existing algorithm. The cost of this approach is comparable to that of the algorithm we use since the update log from which we get segment information is contained in main memory. However, information concerning segments can be used to reduce the number of elements to be checked, thus improving the processing. In the following, we first present some results concerning containment of elements and segments; then we show how to perform structural join by using the element index and the update log.

4.1 Preliminaries

The containment relationship between XML segments is closely related to the containment relationship between XML elements, which is the foundation of structural join. In general, we distinguish between *cross-segment join*, i.e. join between elements contained in distinct segments, and *in-segment join*, i.e. join between elements contained in the same segment.

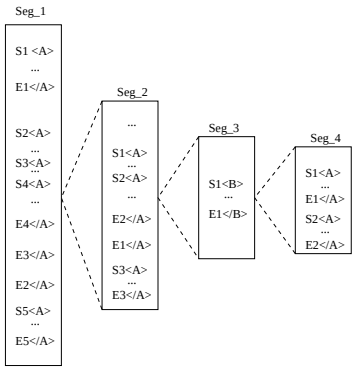


Figure 8: Cross-segment join between segments

In the following, we present two properties of cross-segment joins that will be useful in defining our structural join algorithm. Their proof trivially follows from Definition 1. The first property specifies that, in order to be related by an ancestor-descendant relationship, elements must be contained in pairs of ancestor-descendant segments, thus providing a necessary condition for pairs of segments to generate cross-segment joins; the second provides a sufficient and necessary condition for an element to generate cross-segment joins. In presenting such properties, given two segments S and T such that S contains T, P_T^S denote the local position of a segment L containing T and directly contained in S. For example, if S is identified by path 0.1.2 and T by 0.1.2.3.4.6, P_T^S is the local position of segment 3 with respect to S. If S directly contains T, we just set P_T^S to be the local position of T in S. Moreover, we call *X-element* an element with X as tag name.

PROPOSITION 3. *Let S and T be two distinct segments. Let a be an A-element in S and b a B-element in T.*

1. *If a is an ancestor (parent, descendant, child) of b then S contains T (S directly contains T, T contains S, T directly contains S).*
2. *a is an ancestor of b if and only if a contains T and $a.start < P_T^S$ and $a.end > P_T^S$.* □

EXAMPLE 1. *Consider Figure 8, where S_n and E_n represent the starting and ending position of element n. We see that the* A*-element S_2 in segment 2 contains segment 3 and so is its ancestor* A*-element S_1. Therefore, according to Proposition 3(2), we have two join results: (2:S_1,3:S_1) and (2:S_2,3:S_1). Segment 2 is contained in the* A*-element S_4 in segment 1, hence, the* A*-element S_4 in segment 1 contains segment 3 as well. We get another three results from* A*-element S_4 in segment 1 and its ancestor* A*-elements, which are S_2 and S_3. The three pairs are (1:S_2, 3:S_1), (1:S_3, 3:S_1) and (1:S_4, 3:S_1). We can finally see that* A*-element S_3 in segment 2 does not produce any result with* B*-element in segment 3 since it does not contain segment 3. The same happens with* A*-elements S_1 and S_5 in segment 1.* □

4.2 The Lazy-Join Algorithm

In the following, we present a structural join algorithm that uses segment information to improve the processing. It is a variation of the stack-based algorithm proposed in

```
Algorithm Lazy-Join(SL_A,SL_D)
1.  sa = SL_A.firstNode; sd = SL_D.firstNode; OutputList = NULL; stack = empty_stack();
2.  While (stack is empty) /* the stack is empty*/
3.    If (sa.gp < sd.gp)
4.      stack.push(sa);
5.      sa = SL_A.nextNode;
6.    Else if (sa.gp = sd.gp)
7.      Append the result of Stack-Tree-Desc(sa,sd) to OutputList;
8.      sd = SL_D.nextNode;
9.    Endif
10. Endwhile
11. While ((SL_A and SL_D are not empty) /* both lists are not empty */
12.   If (sd.gp > stack.top.gp + stack.top.l) stack.pop();                    /* Step 1 */
13.   Else if (sa.gp < sd.gp)                                                 /* Step 2 */
14.     If (sa contains sd)
15.       remove from stack.top() elements e such that e.lep < P_sa^stack.top();
16.       stack.push(sa);
17.     EndIf
18.     sa = SL_A.nextNode
19.   Else if (sa.gp ≥ sd.gp)                                                 /* Step 3 */
20.     For every segment sa1 in stack, starting from stack bottom
21.       For every element a1 in sa1 such that a1.start < P_sd^sa1, starting from the lowest local position
22.         If (a1.end > P_sd^sa1)
23.           For every element d1 in sd
24.             Append (sa.sid,a1.lp,sd.sid,d1.lp) to OutputList;
25.     If (sa.gp = sd.gp) Append the result of Stack-Tree-Desc(sa,sd) to OutputList;
26.     sd = SL_D.nextNode
27.   Endif
28. Endwhile
```

Figure 9: Algorithm Lazy-Join

[1], called `Stack-Tree-Desc`. We consider this algorithm because it is quite efficient and, at the same time, it is very easy to implement. The algorithm we propose, called `Lazy-Join` to highlight that it relies on a lazy XML update approach, returns pairs of ancestor/descendant elements first sorted with respect to descendant positions.

`Lazy-Join` differs from traditional structural join algorithms in two aspects: (i) it computes the result starting from two lists of segment identifiers, instead of two lists of element identifiers; (ii) it relies on Proposition 3 to improve cross-segment join computation. Any traditional structural join algorithm can be used to generate in-segment joins.

Suppose the path expression is $A//D$. The algorithm starts from two lists of segment identifiers SL_A and SL_D, the first containing A-elements, the second containing D-ones, sorted by global position. These lists are extracted from the taglist (see Section 3.2). The basic idea of the algorithm is to merge the two lists of segments (thus, each segment in the lists is accessed just once), according to their global position, using a stack. The stack at all times contains a sequence of segments from SL_A. Each segment in the stack is a descendant of the segment below it. For each segment s, we push: (i) its identifier, global position, and length (retrieved from the SB-tree); (ii) the local starting and ending positions of A-elements in s (retrieved from the element index).

At each step, the ancestor-descendant relationship between the current segment in SL_A - say sa - and the current segment in SL_D - say sd - is checked. Based on the result of this comparison, the stack is manipulated, pointers in the lists advanced, and results produced, according to Proposition 3. More precisely, three distinct operations can be executed, until SL_A or SL_D become empty:

1. *Pop segments*: $sd.gp > stack.top.gp + stack.top.l$. This means that sd is not a descendant of the top segment in the stack, thus no future segment from SL_D will be a descendant of the current top of the stack (since segments are sorted by their global position). Therefore, we can pop the stack. No result is generated.

2. *Push segments*: $sa.gp < sd.gp$. Due to Step 1, we know that sd is a descendant of the top segment in the stack. Since $sa.gp < sd.gp$, sa is a descendant of the top segment, too. If sa contains sd, sa is pushed into the stack since, due to Proposition 3(1), it may generate joins with sd. Then, we advance SL_A. No result is generated.

3. *Join generation*: $sa.gp \geq sd.gp$. Due to Steps 1 and 2, we know that sd is a descendant of all the segments in the stack, thus, according to Proposition 3(1), all segments in the stack may generate cross-segment joins with sd. However, differently from [1], not all elements in the stack will generate joins with elements in sd. Thus, for each segment in the stack, say $sa1$, we generate cross-segment joins with D-elements in sd only if condition 2 of Proposition 3 is satisfied. No condition has to be checked for elements in sd.

   If $sa.gp = sd.gp$, in-segment joins are then generated. Elements are joined based on their local positions by using any structural join algorithm (e.g., `Stack-Tree-Desc` [1]). Note that if $sa.gp > sd.gp$, sa and sd cannot generate joins due to Proposition 3(1). In both cases, according to Proposition 3(1), due to the ordering of SL_A, any future segment in SL_A cannot generate joins with sd. Thus, SL_D is advanced.

If SL_D becomes empty before SL_A, no more joins can be generated and the algorithm stops. On the other hand, if

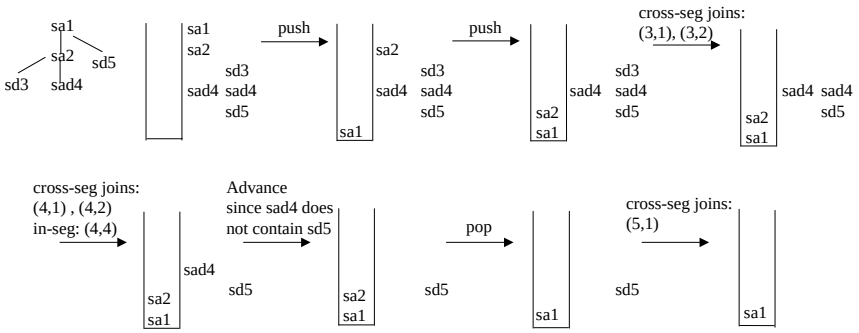


Figure 10: `Lazy-Join` algorithm processing: segments sa_i contain A elements and not D ones, segments sd_i contain D elements and not A ones, segments sad_i contain both D and A elements

SL_A becomes empty before SL_D, we just process the remaining segments in SL_D according to Steps (1) and (3), until SL_D or the stack become empty.

According to the algorithm above, elements in the stack do not represent a chain of ancestor-descendant relationships. On the other hand, this property is satisfied by the algorithm presented in [1]. In order to reduce the overhead, due to the fact that more elements than required are inserted in the stack, the `Lazy-Join` algorithm can be optimized by inserting in the stack only elements that can *potentially* generate cross-segment joins. Since the stack contains a chain of ancestor-descendant segments, according to Proposition 3(2), those elements are such that they contain at least one child segment. We get this behavior by modifying Step (2) as follows: (i) we push only elements containing at least one segment; this information can be checked by using child information associated with each leaf value in the SB-tree. Note that, since in-segment joins are computed before a segment from SL_A is pushed into the stack, no pairs are lost; (ii) before pushing a segment sa into the stack, we remove from the top segment the elements ending before sa starts, since they will not contain any future segment from SL_A. Figure 9 presents the optimized version of the algorithm, assuming the length of SL_A is not shorter than SL_D. Figure 10 presents an example of its application.

We finally observe that the `Lazy-Join` algorithm can be easily extended to compute parent-child relationships. In this case, according to Proposition 3(1), segments must share a parent-child relationship. Thus, at Step 3, cross-segment joins can be generated only from the pair of segments $(stack.top, sd)$. For each pair of elements $(a1, d1)$, the join is generated if $d1.LevelNum = a1.LevelNum + 1$.

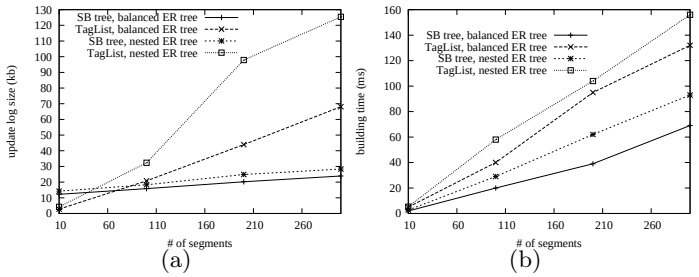
4.3 Analysis of Lazy-Join algorithm

Correctness of the proposed algorithm follows from Proposition 3 and [1]. Concerning time complexity, the algorithm implements a merge of two lists. Let p_A (p_D) be the average number of $A(D)$-elements in one segment. The operations performed by the algorithm are the following:

- *Push segments.* Each segment in SL_A is pushed at most once. Since for each pushed segment sa we need to access the SB-tree to get necessary information about sa and the element index to retrieve A-elements from sa, the cost of pushing segments is $O(|SL_A|*(log(N)+log(NE)+p_A))$, where N is the total number of segments and NE is the total number of elements.
- *Pop segments.* Each segment can be popped at most once. Thus, the cost is bounded by $O(|SL_A| * p_A)$.
- *Join generation.* In Step 3, for each segment in the stack, at most all its A-elements are checked. Each of them either generates cross-segment joins with all D-elements in sd or with none of them (lines 21-22-23-24). Thus, the cost of accessing A- and D-elements is amortized by the cost of returning join results in output. Moreover, according to the applied optimization, only in the top segment more than one element may generate no join with elements in sd (all elements ending before sd starts). Since SL_D is ordered, such elements will generate no join with any next segment from SL_D and therefore can be removed from the stack. By applying this additional optimization, each top segment is completely analyzed once during the overall join computation. Thus, the complexity of generating cross-segment joins is $O(|SL_D|(log(NE) + log(N)) + |SL_A| * p_A + OutputList)$, where $log(NE)$ is due to the element index access needed to retrieve D-elements contained in sd, $log(N)$ is due to the computation of P_{sd}^{sa1} for the top segment in the stack (for the others, it can be computed after each push operation and stored in an auxiliary data structure), and $OutputList$ is the number of returned pairs. Concerning in-segment joins, since we use the algorithm proposed in [1], the cost is $O(S_{AD}(p_A + p_D + log(NE)) + OutputList)$, where S_{AD} is the number of segments containing both A- and D-elements.

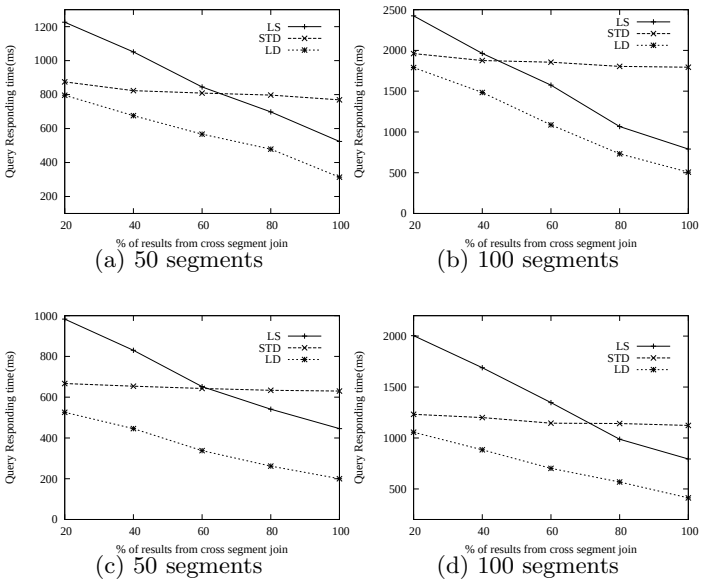
PROPOSITION 4 (TIME COMPLEXITY). *The time complexity of algorithm* `Lazy-Join` *is* $O(|SL_A| * p_A + S_{AD} * p_D + Seg\_overhead + OutputList)$ *where* $Seg\_overhead = (|SL_A| + |SL_D|) * (log(N) + log(NE))$. □

We note that $S_{AD} * p_D$ is the maximum overhead due to in-segment joins. Thus, by fixing the total number of joins and the number of segments, increasing the percentage of cross-segment joins improves the performance. This is due to the fact that, in this case, segments that do not satisfy Proposition 3(1) are skipped. With respect to the `Stack-Tree-Desc` algorithm, whose complexity is linear in the number of A- and D-elements [1], i.e., in $|SL_A| * p_A + |SL_D| * p_D$, we note that, since $S_{AD} * p_D \leq |SL_D| * p_D$, `Lazy-join` outperforms `Stack-Tree-Desc` depending on: (i) the percentage of in-segment joins; (ii) the segment overhead, which in turn depends on the number of segments.



(a) (b)

Figure 11: (a) Update log size; (b) Elapsed time for building the update log



(a) 50 segments (b) 100 segments

(c) 50 segments (d) 100 segments

Figure 12: Elapsed time for structural join over: (a)-(b) nested ER-trees; (c)-(d) balanced ER-trees

5. PERFORMANCE STUDY

5.1 Experiment Setup

We have implemented the update log and the related element index in C++, on an ultra450 machine with processor of 500MHz and 3Gbytes of main memory. For the experiments, we used both synthetic data sets, created by the IBM XML Generator [15] and XMark benchmark [16]. We used synthetic data sets in order to more easily get the characteristics we need for analyzing the properties of the proposed algorithms. On the other hand, by using XMark datasets, we can analyze the proposed techniques in more realistic situations. When using synthetic data sets, to simulate the real world scenario, we chopped the data sets into many small segments and inserted these segments into an initially dummy XML document, while maintaining the validity of the super document. Segment size varies from few kilo bytes to several hundred kilo bytes and the number of elements they contain varies from a few to several thousands.

Experiments have been conducted concerning update log space occupancy and building time, as well as time to execute structural joins and to update the structures. In the last two cases, we considered two different assumptions, resulting in different query/update times. Under the first assumption (that we call *lazy dynamic* (LD)) we assume to maintain incrementally updated the update log, thus at query time the update log is ready to be used; under the second assumption (that we call *lazy static* (LS)), we further reduce update time by assuming to maintain incrementally updated only the ER-tree and to keep the tag-list unsorted. Path lists are sorted and the B^+-tree generated from scratch just before querying the XML database.

5.2 Update Log Space and Building Time

The update log consists of both the SB-Tree and the tag-list. Figure 11(a) reports the size of each component and the total size of the update log, in term of kbytes, when the number of inserted segments varies. Each segment contains all element tags appearing in the tag-list (worst-case for tag-list update). We can see that the size of the tag-list increases much faster than that of SB-tree and the tag-list size contributes a large part of the total size of update log. This is because, as stated in Section 3.2, the size of the tag-list is $O(TN^2)$ while that of the SB-Tree is $O(N)$, where T the is number of distinct tag ids and N is the number of segments. We also note that in the nested case tag-list size increases faster, as discussed in Section 3.2. As we have claimed before, the size of the update log is considerably small. Its size is only about 95 kbytes for balanced ER-trees and 195 kbytes for nested ER-trees after over 300 insertions, which cannot be a problem for modern machines. Figure 11(b) reports similar results for update log building time.

5.3 Structural Join Processing

We compared the elapsed time of solving path expressions like $A//D$ by LS, LD, and the `Stack-Tree-Desc` algorithm proposed in [1] (denoted by STD in the following). Three main groups of experiments have been performed.

The aim of the first group is to analyze the impact of cross-segment joins in query performance. To this purpose, we fixed the number of segments and the number of A- and D-elements. Then, we varied the percentage of cross-segment joins. Since the structure of the ER-tree determines how many segments containing D-elements can be skipped, we considered two different ER-tree structures: a completely nested one (which corresponds to the worst case) and a balanced one, which corresponds to a more reasonable real situation. Figure 12 reports the results for the nested and balanced case, by considering 50 and 100 segments. We can see that when the number of cross-segment joins increases, the performance of LS and LD increases, since, as we saw in Section 4, the cost of performing cross-segment joins is lower than the cost of performing in-segment joins. On the other hand, the cost of STD is almost constant since, even if it can be influenced by the position of elements generating joins, it has to read all elements that may potentially generate joins even if some of them will not generate any result. From this experiment we also note that, as expected, LD is always more efficient than STD, due to the fact that by LD entire segments can be skipped and the segment processing overhead is very low. On the other hand, LS is more efficient than STD for high cross-segment join percentage (higher than 60% inthis experiment).

In the second group of experiments, we investigated the impact of the number of segments in structural join performance. To this purpose, we fixed a document (which

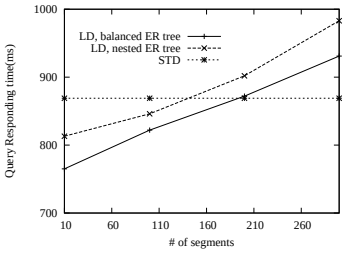


Figure 13: Elapsed time for structural join over the same document, with different ER-trees

Query	Xpath expression	Result cardinality
Q1	person//phone	413170
Q2	profile//interest	494240
Q3	watches//watch	879891
Q4	person//watch	1455040
Q5	person//interest	1074792

Figure 14: Xmark Queries

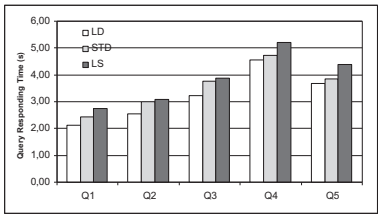


Figure 15: Elapsed time for structural join over XMark datasets

contains about 120000 elements and whose size is approximatively 10Mb), we changed the number of segments and the structure of the ER-tree, and we executed the same query over all situations. In all cases, the percentage of cross-segment joins is around 20%. Results for LD and STD are reported in Figure 13. We can see that the higher is the number of segments the higher is the processing time since the segment lists to be scanned are longer. We also note that, for more than 180 segments and balanced ER-tree, LD performs worst than STD. This is because, in this case, the overhead in segment processing is higher than the improvement got from cross-segment join computation.

Finally, in the third group of experiments, we analyzed the performance of LD on an XMark dataset, slightly modified to increase the number of cross-segment joins. The size of the dataset is 100Mb and it contains about 3 millions elements. Figures 14 and 15 present the considered queries and the obtained results. For the experiment, we chopped it into 100 segments and we considered a balanced ER-tree. The percentage of cross-segment joins is about 20% to 30%. We can see that for all the considered queries, LD, differently from LS, outperforms STD. These results are coherent with those presented in Figure 12.

From the first two groups of experiments, it follows that, when the number of segments is very high or when the percentage of cross-segment joins decreases, and therefore for the special case when one segment coincides with one element, the performance of LS and LD may decrease. In those cases, nested segments can be collapsed together in order to reduce the overall number of segments, increase their size, and improve query performance. Alternatively, traditional

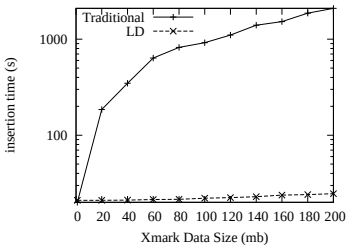


Figure 16: Elapsed time of inserting one segment

structural join algorithms can still be used. At the same time, as we will see in Subsection 5.4, the usage of segments is still useful since it always improves update performance.

5.4 Update Processing

In order to analyze update time of the lazy approach, we considered two different experiments. In the first experiment, we compared LD with a traditional approach, labeling elements by their starting and ending positions. For that, we inserted a segment into Xmark datasets of variable size and we reported the elapsed time of updating the element index (and the update log, for the lazy approach). We considered the average case in which the inserted segment causes half the elements to change their global position. Figure 16 reports the obtained results in logscale. We can see that, as the size of the XML document increases, the insertion time of the traditional approach increases dramatically, while that of LD remains low and almost constant. The reason is simple. For the traditional approach, whenever a new segment is inserted, many of the indexed element records have to be updated. However, for LD, we need only to insert a new segment into the in-memory log, and to insert element records of that segment into the element index. No update of existing element records is required. We note that for the lazy approach, global renumbering of segments is required. However, since the number of segments is usually much less than that of elements, the overhead of this step does not greatly impact the insertion cost.

The aim of the second experiment is to compare update performance of LD and LS with that of approaches based on immutable labeling schemes. To this purpose, we considered the prime numbering scheme recently proposed in [12] (denoted by PRIME in the following). Since the structure of the ER-tree influences the length of paths in the tag-list and the number of segments to be updated, we considered both the balanced and nested case. Figure 17 shows the elapsed time of inserting one element in a document chopped in 100 segments, by changing (a) the number of elements (maintaining fixed the number of distinct tag names) and (b) the number of distinct tag names (maintaining fixed the number of elements) in the inserted segment. Since our approach is based on segments, to determine the update cost of each single element (needed in order to compare LD and LS with PRIME), we divided the time required to insert the segment by the number of elements it contains. K value in the figure is the number of prime numbers that share the same simultaneous congruence in PRIME. We can see that LS and LD require much less time than PRIME. Indeed, PRIME requires recomputing at least one simultaneous congruence value in the table of simultaneous congruence values and this recomputation process contributes in large part of the

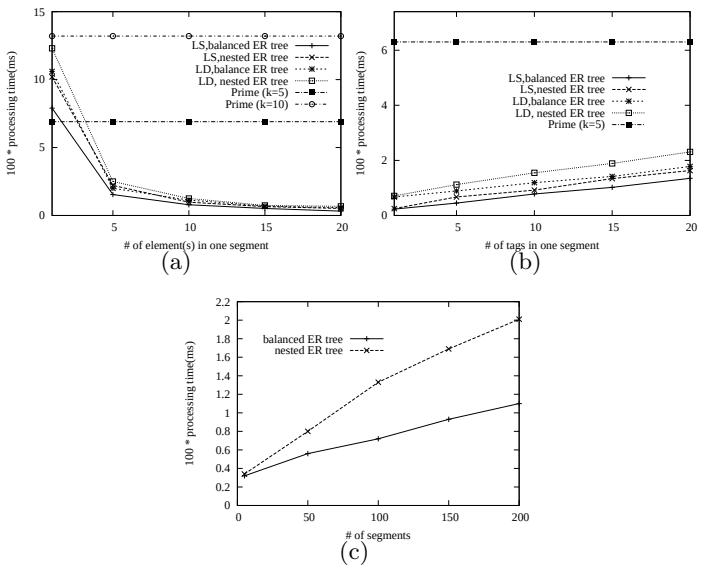


Figure 17: Elapsed time of inserting one element by varying the number of: (b) elements; (c) tag names; (d) segments

total processing time and it is also very costly according to the algorithm presented in [12]. On the other hand, under the lazy approach, no complicated computation is required. We see that by increasing the number of elements inside a segment, the insertion time decreases. This is due to the fact that, in order to obtain the element insertion time, we divide the segment insertion time, which is constant, by the number of elements one segment contains. On the other hand, costs increase when the number of tag names increases since more path lists must be updated. The structure of the ER-tree also influences costs. We can see that nested ER-trees require slightly higher costs since path lengths increase and tag-list update cost increases as well. This is even more evident from Figure 17(c), showing how insertion costs for LD change when varying the number of segments. As expected, insertion time varies almost linearly with respect to the number of segments. Moreover, LS is more efficient compared to LD, even if the gain in performance is very small. Since LD guarantees better query performance, it provides the best compromise between update and query time.

6. CONCLUDING REMARKS

In this paper we have presented a lazy approach to XML updates. Differently from all the other existing approaches for XML updates, under the lazy approach multiple XML elements (called XML segments) are inserted (deleted) into (from) the whole XML database without modifying element identifiers. Thus, no update to existing records in the element index is required. To support the proposed approach, specific data structures have been designed and a structural join algorithm relying on the usage of segments has been proposed. Experimental results show that our approach outperforms other existing solutions in handling updates and guarantees better performance compared with traditional structural join algorithms, such as the one proposed in [1]. As a future work, we plan to investigate how packing techniques and concurrency can be used to improve segment update and query processing. We also plan to compare the lazy approach with the one proposed in [9] and to investigate how it can be used for improving other XML data management techniques, such as query optimization and access control.

7. REFERENCES

[1] S. Al-Khalifa et al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. of Int. Conf. on Data Engineering*, page 141-152, 2002.

[2] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 310-321, 2002.

[3] S.Y. Chien et al. Efficient Structural Join on Indexed XML Documents. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 263-274, 2002.

[4] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Tree, In *Proc. of the ACM Int. Symp. on Principles of Database Systems*, pages 271-281, 2002.

[5] H. Jiang, H. Lu, W. Wang, and B.C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *Proc. of the Int. Conf. on Data Engineering*, pages 253-263, 2003.

[6] R. Kaushik, P. Bohannon, J. F. Naughton, and P. Shenoy. Updates for Structure Indexes. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 239-250, 2002.

[7] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 361-370, 2001.

[8] P. E. O'Neil et al. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 903-908, 2004.

[9] A. Silberstein, H. He, K. Yi, and J. Yang. BOXes: Efficient Maintenance of Order-Based Labeling for Dynamic XML Data. In *Proc. of the Int. Conf. on Data Engineering*, 2005. To appear.

[10] I. Tatarinov, Z.G. Ives, A.Y. Halevy, and D.S. Weld. Updating XML. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 2001.

[11] I. Tatarinov et al. Storing and Querying Ordered XML Using a Relational Database System. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 204-215, 2002.

[12] X. Wu, M.L. Lee, and W. Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *Proc. of the Int. Conf. on Data Engineering*, pages 66-78, 2004.

[13] K. Yi, H. He, I. Stanoi, and J. Yang. Incremental Maintenance of XML Structural Indexes. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 491-502, 2004.

[14] C. Zhang et al. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 425-436, 2001.

[15] IBM Alpha Works XML Generator. http://www.alphaworks.ibm.com/tech/xmlgeneratorhp

[16] The XML Benchmark Project. http://www.xml-benchmark.org