Erschienen in: Proceedings 31st IEEE International Conference on Data Engineering (ICDE), 2015 / IEEE (Hrsg.). - Piscataway, NJ : IEEE, 2015. - S. 447-458

Efficient Structural Bulk Updates on the Pre/Dist/Size XML Encoding

Lukas Kircher, Michael Grossniklaus, Christian Grün, and Marc H. Scholl
Department of Computer and Information Science, University of Konstanz
P.O. Box 188, 78457 Konstanz, Germany
firstname.lastname@uni.kn

***Abstract*—In order to manage XML documents, native XML databases use specific encodings that map the hierarchical structure of a document to a flat representation. Several encodings have been proposed that differ in terms of their support for certain query workloads. While some encodings are optimized for query processing, others focus on data manipulation. For example, the Pre/Dist/Size XML encoding has been designed to support queries over all XPath axes efficiently, but processing atomic updates in XML documents can be costly. In this paper, we present a technique, so-called *structural bulk updates*, that works in concert with the XQuery Update Facility to support efficient updates on the Pre/Dist/Size encoding. We demonstrate the benefits of our technique in a detailed performance evaluation based on the XMark benchmark.**

I. Introduction

The XQuery Update Facility (XQUF) [4] introduces data manipulation capabilities to the XML query language XQuery [14] by extending both the syntax and the processing model of XQuery. Syntax extensions consist of a set of update operations to insert, delete, replace, and copy nodes in an XML document. When used in a query, each individual update operation leads to one or more update primitives. To manage these update primitives during query execution, the XQuery processing model is enriched with a data structure called the *Pending Update List* (PUL). The XQUF specification defines the PUL as "an unordered collection of update primitives [...] that have not yet been applied" (§2.1).

The main objective of the PUL is to realize atomicity, consistency, and isolation by caching all update operations that are to be executed within a transaction or snapshot. Only after checking whether the application of the pending update list leads to a consistent state of the database, all updates are applied in one single bulk operation at the end of the query or snapshot. This processing model mainly serves the following purposes. First, changes introduced during a snapshot are only visible in succeeding queries. Second, update primitives can be applied in a specific order defined by the XQUF to resolve ambiguities. Finally, insertion sequences (for insert, replace, etc.) are cached to avoid dirty reads.

Since the structural order of the XML document has to be maintained when it is modified, processing individual updates can be costly. A major factor that determines the exact cost of an update operation is the XML encoding that maps the hierarchical structure of the document to a flat representation. Several XML encodings have been proposed that all balance the trade-off between query and update runtime performance slightly differently. In general, these encodings fall into one of two categories. Prefix-based encodings use variable-length labels that represent the position of a node, whereas interval or region-based encodings physically store nodes in order, typically based on a pre-order traversal of the document tree.

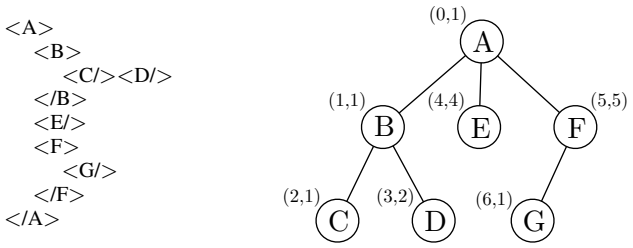
Since the encoding used in a given native XML database is fixed, the trade-off between query and update performance cannot be influenced for individual update operations. However, we argue that one major advantage of the PUL is the fact that it provides the opportunity to tailor the processing of bulk updates to the underlying encoding scheme. Our hypothesis is that analyzing the characteristics of the bulk update and optimizing the sequence of atomic updates can amortize part of the cost that would be incurred by executing atomic updates naïvely one after another. In this paper, we test this hypothesis in the setting of the Pre/Dist/Size encoding. The contributions of the work presented in this paper are as follows.

1) Optimization technique for bulk updates to reduce processing time with respect to a series of atomic updates.
2) XQUF implementation that leverages this technique.
3) Quantification of the benefit based on bulk update processing times with and without optimization.

Correspondingly, the paper is structured as follows. Section II gives an overview of the Pre/Dist/Size XML encoding that is used in this work. In Section III, we introduce efficient bulk updates and in Section IV, we present related optimizations that are enabled by bulk updates. Section V discusses how bulk updates are leveraged to implement XQUF. We evaluate our work in Section VI and discuss related work in Section VII. Finally, concluding remarks are given in Section VIII.

II. The Pre/Dist/Size XML Encoding

The work presented in this paper is situated in the context of the Pre/Dist/Size XML encoding [7]. In this section, we review this encoding, highlight its advantages with respect to querying and discuss its limitations with respect to updating. Pre/Dist/Size belongs to the family of interval or region-based XML encodings that use partitions of the pre/post plane [8] to represent the hierarchical document structure. The XML encodings in this family have two major advantages. First, each node in an XML document can be mapped to a fixed-length record and, second, these encodings support efficient queries over the XPath axes [5]. A drawback shared by these encodings is the fact that updates to the XML document often require several changes throughout the mapping table.



(a) Example document

(b) Tree with *(pre,dist)* tuples

pre	dist	size	val
0	1	7	A
1	1	3	B
2	1	1	C
3	2	1	D
4	4	1	E
5	5	2	F
6	1	1	G

(c) Sequential table

pre	dist	size	val
0	1	**6**	A
1	1	**2**	B
2	**1**	1	D
3	**3**	1	E
4	**4**	2	F
5	1	1	G

(d) Table after update

Fig. 1: Relational Pre/Dist/Size encoding of example XML document.

Figure 1 explains the Pre/Dist/Size encoding. A sample XML document is given in Figure 1a with its tree representation shown in Figure 1b. Pre/Dist/Size uses three values to encode the document structure. The *pre* value is the position of the node in the tree pre-order traversal. As shown in Figure 1c, pre values determine the position of records in the sequential table. The *dist* value encodes the parent of a node as the relative distance in the sequential table. For example, the parent of <F> is <A>, since $pre(F) - dist(F) = pre(A)$. Finally, the *size* value denotes the number of descendant elements of a node (including the node itself).

A. Querying the Pre/Dist/Size Encoding

In XQuery [14], queries over the structure of an XML document are typically expressed in terms of XPath axes. We show in the following how queries over each of these axes can be efficiently evaluated based on the Pre/Dist/Size encoding.

- As seen above, a **parent** node p of a node n has the pre value $pre(p) = pre(n) - dist(n)$.
- The first **child** node c_1 of a node n is found at $pre(c_1) = pre(n)+1$. All other child nodes c_i can be found by iteratively adding the size of the current child to its pre value, i.e., $pre(c_i) = pre(c_{i-1})+size(c_{i-1})$, until $pre(c_i) \geq pre(n) + size(n)$.
- The **ancestors** of a node n are computed by iteratively calculating the parent of the current node n_i, until $pre(n_i) = 0$.
- All **descendants** of a node n are located in the interval $[pre(n) + 1, pre(n) + size(n) - 1]$ and can be read sequentially from the table.
- To find the **preceding-siblings** of a node n, the parent node p is first determined and then all child nodes c_i are returned until $c_i = n$.
- Similarly, the **following-siblings** of a node n are found through its parent p by starting with $c_i = n$ and iterating over all child nodes until $pre(c_i) \geq pre(p) + size(p)$.
- The set of **preceding** nodes of a node n is calculated as all nodes in the interval $[0, pre(n) - 1]$ minus all ancestor nodes of n.
- The set of **following** nodes of a node n is given by all nodes the interval $[pre(n) + size(n), |T| - 1]$, where $|T|$ is the cardinality of the sequential table T.

All other axes are either trivial (e.g., *self*) or combinations of already presented axes (e.g., *ancestor-or-self*, *descendant-or-self*, etc.) and can thus be evaluated accordingly.

B. Updating the Pre/Dist/Size Encoding

In XML databases, two types of (atomic) updates can be distinguished. First, *value updates* change the value of an element in the document and, second, *structural updates* change the document structure itself. In the Pre/Dist/Size encoding, value updates can be implemented efficiently by simply updating the *val* value in the sequential table. Structural updates, however, can be costly. For example, suppose that element <C> is to be deleted in the example document shown in Figure 1. The resulting sequential table, which highlights the required changes in bold, is shown in Figure 1d. We now examine the effects of inserts and deletes on the Pre/Dist/Size encoding in more detail as this understanding is the basis for the structural bulk update technique presented in this paper.

Let us assume that we insert (or delete) a document A of size s at position l in the sequential table T. The pre values of all tuples $t_i \in T$ with $pre(t_i) \in [0, l - 1]$ remain unchanged, whereas all pre values $pre(t_i) \in [l, |T|]$ have to be recalculated as $pre(t_i) = pre(t_i) \pm s$, depending on whether the update is an insert $(+)$ or a delete $(-)$. In order to obtain a compact encoding, pre values are not represented explicitly in the table, but implicitly by the (physical) row number of the record. In the worst case, this requires shifting $O(|T|)$ tuples on disk.

Using an (in-memory) logical page directory to map the first pre value (*fpre*) of each page p_i to its physical address, pre values can be updated by shifting the tuples in $O(1)$ pages plus updating the subsequent *fpre* values as $fpre(p_i) = fpre(p_i) \pm fpre(p_{i-1})$. Figure 2 illustrates the logical paging mechanism. Initially, all pages are filled to their capacity of 256 records. On the left-hand side, 100 records have been deleted from the first page. Records in subsequent pages are not shifted, but their *fpre* value is decremented by 100. On the right-hand side, 100 records are inserted into the first page, which is already full. Instead of shifting all records, a new page is allocated to hold the records and the page directory is updated accordingly.[1]

The dist value of all tuples $t_i \in T$ for which $(pre(t_i) - dist(t_i)) < l \leq pre(t_i)$ needs to be recalculated as $dist(t_i) = dist(t_i) \pm size$. The actual cost of updating the distances is

[1]Note that this technique is similar to the use of the Pos/Size/Level table in MonetDB/XQuery [3], which is based on the Pre/Size/Level XML encoding.

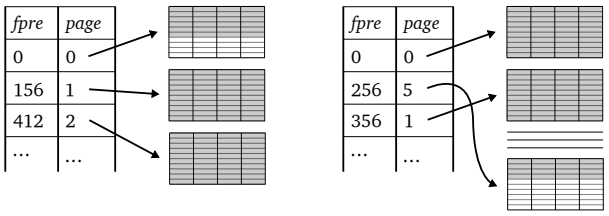


Fig. 2: Deletion and insertion using logical pages

hard to predict as it highly depends on the document structure. As the nodes to update lie on the following-sibling axes of A and of all its parents, the worst case requires $O(|T|)$ updates. Since dist values are represented explicitly in the table, these update costs cannot be reduced as in the case of pre values.

Finally, the size value has to be updated as $size(t_i) = size(t_i) \pm s$ for all tuples $t_i \in T$ for which $pre(t_i) < l < pre(t_i) + size(t_i)$, i.e., all ancestors of A. Therefore, the cost of updating the size values is bounded by the height of the document $O(log|T|)$ in general and $O(|T|)$ in the worst case.

III. Efficient Structural Bulk Updates

As explained in the previous section, distance adjustments are the dominating factor in the cost of atomic structural updates in the Pre/Dist/Size encoding. Additionally, the same distances are often adjusted multiple times in a sequence of updates. In order to reduce overall processing time of such bulk updates, the goal of this work is to avoid redundant distance adjustments without adding excessive overhead.

In contrast to the naïve approach, where distances are *iteratively* adjusted with each atomic update, the proposed technique adjusts distances *explicitly* after all updates have been applied. This approach is enabled by a data structure, named *Atomic Update Cache* (AUC), which holds all atomic updates of one bulk update. The AUC is organized as a table that stores atomic updates in document order of their location. For each atomic update, the pre value of the first affected tuple in the sequential table is recorded. Additionally, the number of tuples shifted by each individual update as well as the accumulated number of tuples shifted by the update and all its preceding updates is stored. In this section, we present a technique for efficient bulk updates that is based on this AUC.

A. Avoiding Redundant Distance Adjustments

Our method to avoid repeated and therefore redundant distance adjustments is based on four observations that apply to bulk updates in the presence of the described AUC. Based on examples, we motivate each observation and demonstrate how it contributes to supporting efficient bulk updates.

Observation 1: If updates are applied in reverse document order, adjusting distance values can be delayed until the very last step of the updating process.

If a sequence of updates is executed from the highest to the lowest pre value, re-computation of the individual update locations is avoided. A tuple t is only shifted if the number of nodes changes in the interval $[0, pre(t)]$. Similarly, distance values are only adjusted if the number of nodes changes between a child and its parent. Inserting or deleting a tuple only invalidates the distances of following tuples. Therefore, the part of the sequential table that is accessed by consecutive atomic updates during a bulk update always remains valid. Based on this fact, distance adjustments can be delayed altogether by applying atomic updates as follows.

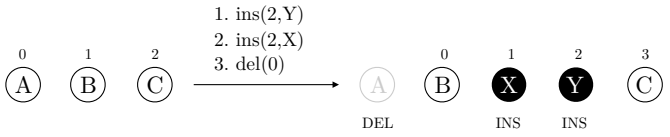
1) Traverse atomic updates in the AUC back to front, i.e., in reverse document order.
2) Insert or delete the corresponding nodes, implicitly shift pre values of the following tuples, adjust the size values of the ancestors, but leave distances.
3) After applying all atomic updates, restore the tree structure by adjusting distances in an *efficient* manner.

We now focus on the third step as it determines the overall performance of the proposed technique.

Observation 2: The contents of the AUC serve as a bi-directional mapping $pre^{old}(t_i) \leftrightarrow pre^{new}(t_i)$ between the pre value of a tuple before and after the bulk update.

To substantiate this observation, we discuss how the above-mentioned information on shifts and accumulated shifts contained in the AUC is initialized and used. Figure 3a shows the effect of a bulk update consisting of two inserts and one delete on a document that contains nodes <A>, <B>, and <C> as siblings. The pre values of the nodes in the sequential table are displayed above the nodes. Note that the atomic updates of this bulk update are applied in reverse document order. As a consequence, the repeated insertion at position 2, first <Y> then <X>, yields the intended sequence. The corresponding AUC in document order is given in Figure 3b. The *shifts* column records the number of tuple shifts and can be calculated based on the size of the inserted or deleted tree. Since we insert and delete single nodes in our examples, the number of shifts is always 1 or -1, respectively. Column *accum. shifts* lists the accumulated number of individual tuple shifts in document order. Finally, the *first affected tuple* column contains the lowest pre value that is shifted as a consequence of the corresponding atomic update. Atomic deletes affect the first pre value on the following axis, whereas inserts affect the pre value at their insert location.

Based on the information contained in the AUC, distance values can be adjusted explicitly as follows. The mapping



(a) Bulk update consisting of two insertions and one deletion

atomic	first affected tuple	shifts	accum. shifts
del(0)	1 (0)	-1	-1
ins(2,X)	2 (2)	1	0
ins(2,Y)	2 (3)	1	1

(b) Corresponding AUC in document order

Fig. 3: Mapping pre values before and after bulk update

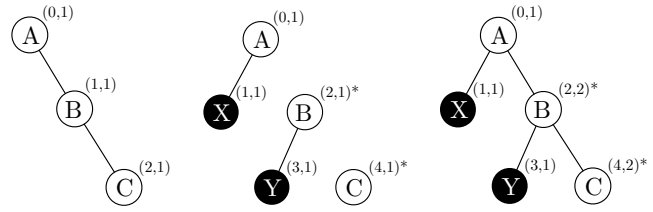
$pre^{old}(t_i) \rightarrow pre^{new}(t_i)$, which gives the new pre value of a node in the unaltered table, is derived by identifying the update at the highest index in the AUC that still affects this node. For example, the mapping $pre^{old}(B) \rightarrow pre^{new}(B)$ is determined by the delete, which is the operation with the highest index that still affects the pre value. The AUC gives an accumulated shift of -1 for this delete and therefore the mapping is $1 \rightarrow (1 + (-1))$, i.e., the new pre value of B is 0. Note that in the case of node A, there is no mapping $pre^{old}(A) \rightarrow pre^{new}(A)$ as the first affected tuple points to A itself, which is deleted. However, as mappings are only applied to existing nodes, this is not a problem.

The mapping $pre^{new}(t_i) \rightarrow pre^{old}(t_i)$, which gives the original pre value of an already shifted tuple, is calculated similarly. Eventual tuple shifts have to be taken into account as the atomic updates have already been applied. Therefore, this mapping is calculated based on the values for the first affected tuple that includes accumulated shifts, which are given in brackets in Figure 3b. For example, for the mapping $pre^{new}(B) \rightarrow pre^{old}(B)$, the delete operation determines B's new pre value as 0, which equals the first affected tuple including accumulated shifts. To calculate the old pre value from the new, the effect of the accumulated shifts has to be reversed. The AUC gives an accumulated shift of -1 for the delete operation and therefore the mapping is $0 \rightarrow (0-(-1))$, i.e., the old pre value of B is 1. Note that this mapping also covers nodes that are inserted during the update process. For example, with a (new) pre value of 1, node X is only affected by the delete and it follows that the mapping $pre^{new}(X) \rightarrow pre^{old}(X)$ is $1 \rightarrow (1-(-1))$, i.e., the old pre value of X is 2, which is the position where it was inserted.

Observation 3: The new distance value of a node can be explicitly calculated based on its original state and the bi-directional mapping contained in the AUC.

Recall that the dist value of a node n gives the number of nodes that are stored between the node and its parent p in the sequential table. The pre value of the parent is then calculated as $pre(p) = pre(n) - dist(n)$. However, after performing all atomic updates of a bulk update, we cannot determine the number of tuples changed between a node and its parent directly, since the parent node is no longer known. Therefore, the updated distance for any given node of the table has to be calculated based on its original distance value and the bi-directional mapping contained in the AUC. We demonstrate how this explicit calculation of distance values can be achieved using the simple example given in Figure 4.

The original document including $(pre, dist)$ tuples is shown in Figure 4a. Figure 4b shows the state of the document after nodes X and Y have been inserted at positions 1 and 2, which shifts nodes B and C to the back. As distance updates are delayed, the distances of B and C still represent the original state and are therefore invalid after this first step. Based on the mapping $pre^{new}(C) \rightarrow pre^{old}(C)$, the old pre value of C (2) can be obtained. Together with C's original distance, this value can now be used to calculate the pre value of the original parent of C (B in Figure 4a) as $(2-1) = 1$. Using the mapping in the other direction $pre^{old}(B) \rightarrow pre^{new}(B)$, i.e., $1 \rightarrow 2$, gives us the pre value of C's new parent (B in Figure 4c). Finally, it follows that the updated distance of node C is $dist^{new}(C) = (pre^{new}(C) - pre^{new}(B))$ or $2 = (4-2)$.



(a) Original state (b) After inserts with invalid distances (*) (c) After distance adjustments (*)

Fig. 4: Distance adjustments after node insertion.

In general, the distance of any node n in the database can be adjusted explicitly by starting with its new pre value pre^{new} as follows, where p is the parent of node n.

$$\begin{aligned} pre^{new}(n) &\rightarrow pre^{old}(n) \\ pre^{old}(p) &= pre^{old}(n) - dist^{old}(n) \\ pre^{old}(p) &\rightarrow pre^{new}(p) \\ dist^{new}(n) &= pre^{new}(n) - pre^{new}(p) \end{aligned}$$

Observation 4: The distances that have to be adjusted can directly be determined by the sequential table and the corresponding bulk update.

Based on the AUC, all first tuples that are affected by a structural update are known and the remaining distances can be determined through ancestor-or-self and following-sibling axis steps. Using an additional set to keep track of nodes with already adjusted distances avoids repetition. As the number of distance adjustments is minimal, the impact of the order in which the first affected tuples are visited is negligible.

In a static setting using the ancestor-or-self and following-sibling axes to describe the sequence of nodes for which distances are affected relative to the first affected tuple of an update is valid. In reality, this sequence needs to be determined dynamically while distance adjustments are being carried out. The next node is then either calculated via the following axis relative to the current node or by switching to the next first affected tuple if the set of following nodes is empty. As a consequence, it is no longer necessary to access the parent axis, which saves a few operations (see Figure 5).

A simple example illustrates how the set of nodes for which distances need to be adjusted is determined. Figure 6a shows a document with nodes labelled with $(pre, dist)$ tuples. Distances that are invalid after the insertion of nodes X and Y are marked with an asterisk. The corresponding AUC is shown in Table I. The starting points for distance adjustments are found by traversing the AUC in document order and checking the first affected tuple entry. In our example, we identify the nodes with pre values 3 and 7, i.e., C and F, as starting points. Let S be the set of nodes that have been adjusted already. We begin by adjusting the distance of C as described above and add it to S. The pre value of the following node n is computed as $pre(C) + size(C)$, $3 + 1 = 4$, which identifies node D. This process is repeated for the nodes D, E, and G. Since $pre(G) + size(G)$ equals the document size, the iteration ends. S now contains the nodes $\{C, D, E, G\}$ as their distances

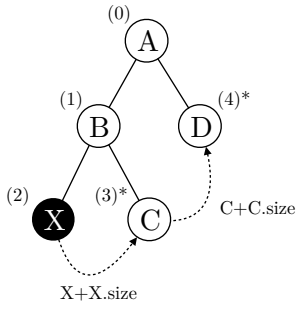
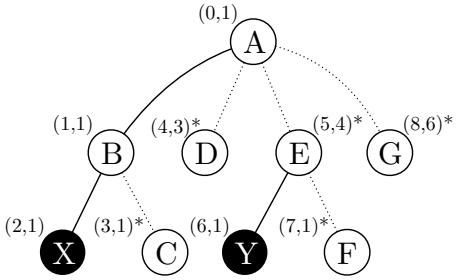
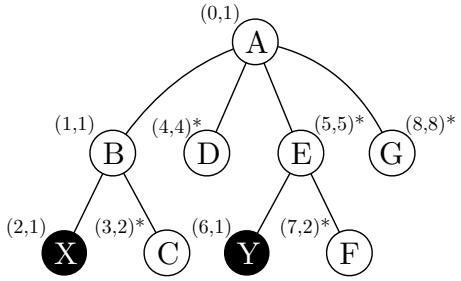


Fig. 5: Accessing distances to update on-the-fly



(a) Invalidation of distances after insertion (*)

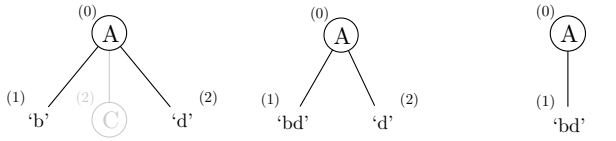


(b) Final table w/ adjusted distances (*)

Fig. 6: Calculating the set of invalid distances.

atomic	first affected tuple	shifts	accum. shifts
ins(2,X)	2 (3)	1	1
ins(5,Y)	5 (7)	1	2

TABLE I: Corresponding update cache in document order



(a) Step 1 (b) Step 2 (c) Step 3

Fig. 7: Text node merging after delete

have been adjusted. The other node identified by the AUC is F, which is the first node affected by the insertion of Y. Its distance is updated and F is added to S. Calculating the next node as $pre(F) + size(F)$ again yields G. As G is already contained in S and there are no more unprocessed atomic updates in the list, the adjustment of distances is finished. The document with all distances adjusted is shown in Figure 6b.

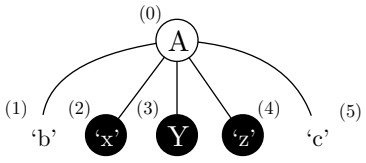
B. Resolution of Text Adjacency

Up to now, the discussion evolved solely around the element node type. However, text nodes also need to be considered as they require special treatment. Adjacent sibling text nodes can occur if a node that separates two text nodes with the same parent is deleted, or if a text node is inserted as a sibling of an existing text node. In both cases, texts have to be merged as the XQuery Data Model [6] forbids adjacency. A typical algorithm to implement this merge operation is to first concatenate the values of two adjacent text nodes in one of the two nodes and then to delete the other one. Since this operation leads to structural changes, our technique of delaying distance updates can be applied as well. As a consequence, we can revise the algorithm for resolving text node adjacency to perform the following three steps.

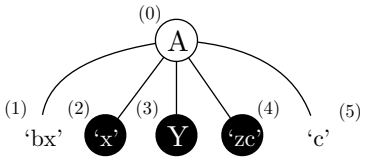
1) Apply atomic updates and distance adjustments.
2) Merge text nodes by concatenating adjacent texts.
3) Delete superfluous text nodes from Step 2 and adjust distances again.

In the example shown in Figure 7a, node C is deleted from the tree. Consequently, the AUC holds a single atomic update del(2, C) with first affected tuple 3, -1 shifts, and -1 accumulated shifts. First, the location l where adjacency can occur is directly derived from the AUC by adding the difference between accumulated shifts and shifts to the location targeted by the update, i.e., $l = 2-1+1 = 2$. In case of a delete the location of the node to merge is given by $l_< = l - 1 = 1$. The corresponding node is then merged with the following sibling, if possible, by directly concatenating their text values in the correct order. For the other node, an atomic delete is created and inserted into a temporary AUC to be executed in Step 3. The result of this step is shown in Figure 7b. Since the original AUC only contained one atomic update in this example, Step 2 is finished. Figure 7c shows the tree after Step 3 which executes all delete operations gathered in Step 2 and adjusts the distances as discussed above.

Figure 8 shows an example, in which a sequence of three nodes ('x', Y, 'z') is inserted with a single atomic insert. Consequently, the AUC holds the atomic insert ins(2, Y) with first affected tuple 2, 3 shifts, and 3 accumulated shifts. There can be no adjacent text nodes within insertion sequences as these would have been merged beforehand. The example focuses on the special case where two text node merges are necessary as a consequence of a single insert operation. This case is handled in our approach by checking the position at the end of the insertion sequence for adjacency. The location targeted by the update is calculated as above, i.e., $l = 2+3-3 = 2$. In contrast to the previous example, however, the atomic update is an insert. In this case, the position before and after the insert need to be checked for possible merges. As above, the before position is given by $l_< = l - 1 = 1$ and the after position is given by $l_> = l + size(\text{'x'}, Y, \text{'z'}) - 1 = 2 + 3 - 1 = 3$. It is important to check locations strictly in reverse document order to avoid incorrect concatenation and wrong order of the resulting delete atomics. Therefore, node 4 is first merged with its following sibling and a corresponding delete operation is inserted into a temporary AUC. Then node 1 is checked, which leads to another concatenation and atomic delete. The resulting and temporary AUC now holds the two atomics $\{\text{del}(2), \text{del}(5)\}$. Step 3 is not explained here as it strictly follows the first example. Finally, Figure 9 shows,

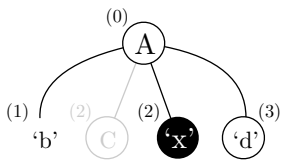


(a) Step 1

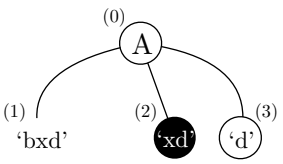


(b) Step 2

Fig. 8: Text node merging after insert



(a) Step 1



(b) Step 2)

Fig. 9: Text node merging after combined delete/insert

through a combination of insert and delete, how the algorithm propagates text concatenation to achieve the desired result. After Step 2, the temporary AUC contains the update sequence $\{\text{del}(2), \text{del}(3)\}$ and is then processed accordingly.

C. Constraint Checking

Due to its use of pre references, the AUC is tightly coupled to the sequential table. It is therefore necessary to define the sequence of atomic updates in a way that their application in reverse document order leads to the desired result.

1) Tree-Aware Updates: There are certain scenarios of the cache that interfere with the overall concept of efficient distance adjustment. For example, if a node X is inserted into an already deleted subtree rooted at B, the AUC is no longer valid as it gives incorrect shift and accumulated shift values. In general, this problem can be solved by so-called *tree-aware updates* that use the information contained in the AUC itself. Tree-aware updates check structural constraints by traversing the AUC in document order. For each encountered atomic delete, all updates that take place in the subtree of the target node are removed. As the subtree is deleted, these changes have no effect anyway. In our example, the AUC only contains the delete operation rooted at B after applying these steps and is again in a valid state. As the size of the AUC is eventually reduced, we not only save I/O but also reduce the complexity of the pre value mapping described earlier.

2) Order of Insert and Delete: Another setup of the AUC that can lead to unwanted results is due to the application of atomic updates in reverse document order. Consider the update sequence $\langle \text{del}(p), \text{ins}(p, Y) \rangle$. If the insert is applied followed by the delete, the inserted node Y would be deleted right away. Again, such sequences can be identified by checking constraints on the AUC during preprocessing and rewritten to obtain the intended effect.

D. Processing of Efficient Bulk Updates

Having discussed the key aspects of our technique for efficient structural bulk updates, we conclude this section by giving an overview of how these steps are combined to form the algorithm given below.

1) Fill AUC with a sequence of updates.
2) Check AUC constraints.
3) Perform tree-aware updates, shift accumulation, etc.
4) Apply updates with delayed distance adjustments.
5) Adjust distances directly.
6) Resolve text node adjacency.

Filling the AUC (Step 1) is straightforward. Note that tasks such as constraint checking (Step 2) and cache preparation (Step 3) can be carried out on the fly if the cache is filled in document order. Value updates are performed before structural updates to minimize recalculation of update locations. During the processing of structural updates distance adjustments are delayed (Step 4) and then adjusted directly (Step 5) as described in Section III-A. Finally, the method for resolving text node adjacency presented in Section III-B is applied (Step 6).

IV. Related Optimizations

We discuss two related optimizations, replace operations and merging of atomic updates, which are enabled by the caching of update operations. In order to illustrate the benefit of these optimizations, it is helpful to provide some details on how the Pre/Dist/Size encoding stores data physically.

A. Preliminaries

Recall from Section II that the sequential table is divided into logical pages, rather than being stored in a contiguous file. As shown in Figure 2, main memory directory keeps track of the document by recording the location and sequence of pages, the first pre value *fpre* on each page. Free space is only allowed after all tuples on a page, hence no gaps between tuples or at the start of a page. As mentioned before, the main purpose of this setup is to reduce I/O costs if tuples are inserted or deleted, as tuple shifts are restricted to the tuples on the same page. An additional reason is that it allows for some basic buffering mechanisms, where a page is completely loaded and then altered in main memory before being flushed to disk.

In order to guarantee a fixed-length encoding of the different node types, text and attribute values are only referenced by an offset and not directly stored in the table. The actual values reside in sequential files on disk. In case of frequent updates, the structure of these files degenerates, as new entries are only appended to the existing files and no overwriting takes place. If values are frequently removed, added or re-inserted, an increase in size and fragmentation is the consequence.

B. Replaces

Up to now, we exclusively talked about insert and delete atomics. Being a combination of delete and insert, replaces are arguably not an atomic type itself. Yet, implementation-wise they help to realize a few important optimizations. Performance results characterizing the performance benefit of the described replace operations are given in Section VI.

a) Basic Replace: A *basic replace* operation r is carried out as follows. First, the node at the update location is deleted, which is consequently followed by a forward shift of all tuples with a pre value greater or equal to this location. Then, the replacing sequence is inserted at the updated location, which leads to a backward shift of the following tuples.

b) Rapid and Lazy Replace: In addition to the naïve basic replace, two more advanced approaches have been realized to limit fragmentation and I/O. The first approach, called *rapid replace*, directly overwrites entries in the sequential table. Following tuples and the according pages are then only touched once, which not only saves a considerable amount of I/O but also reduces fragmentation. The second approach, referred to as *lazy replace*, tries to substitute the structural replace operation with less costly value updates. This approach pays off if the replaced and replacing subtrees generally share the same structure. The lazy replace compares the node to be deleted with the replacing insertion sequence. If they are topologically identical, a sequence of value updates suffices. The implementation is straightforward as it simply requires a sequential and pair-wise comparison of the tuples in the source and destination table. In case the lazy replace fails because of structural differences, a rapid replace is applied instead.

C. Merging Atomic Updates

Reducing the number of structural atomics naturally reduces the complexity of bulk updates. However, there are also benefits to merging atomic updates that are less obvious. For example, there are more opportunities for the replace optimizations described above if neighboring inserts and deletes are merged. Similarly, merging neighboring inserts into one operation can reduce I/O due to buffering strategies at the page level. In general, two atomic updates o_1 and o_2 can be merged if they fulfill certain conditions.

- Targeted locations of o_1 and o_2 are directly adjacent.
- o_1 and o_2 are performed under the same parent node.
- o_1 and o_2 adhere to the order constraints of the AUC.

Based on these conditions, the following substitution rules to merge atomic updates have been defined. The two atomic updates that are replaced are given in document order with regards to the location.

1) $\langle ins(l, X), del(l)\rangle \rightarrow rep(l, X)$*:* The substituting replace is inserted into the AUC as follows. The first affected tuple is the same as the one of the original delete, the shift value is the sum of the shift values of the original updates, and the accumulated shift value is the one of the original delete as it already contains the correct value.

2) $\langle del(l), ins(l+1, X)\rangle \rightarrow rep(l, X)$*:* In this case, the new first affected tuple value is taken from the original delete, the shift values of the original updates are again summed up, and the accumulated shift value is directly derived from the original insert.

3) $\langle ins(l, X), ins(l, Y)\rangle \rightarrow ins(l, XY)$*:* The first affected tuple of the new insert is the same as the one of the first insert, the shift values can be summed up, and the new accumulated shift value is the corresponding value of the second insert. XY denotes the concatenation of X and Y.

4) $\langle rep(l, X), ins(l+1, Y)\rangle \rightarrow rep(l, XY)$*:* The resulting replace affects the same first tuple as the original replace, shifts tuples by the sum of the shift value of the original replace and insert operation, and has the same accumulated shift value as the original insert.

5) $\langle ins(l, X), rep(l, Y)\rangle \rightarrow rep(l, XY)$*:* The value of the original replace is used as the first affected tuple value of the new replace, the shifts are summed up and the new accumulated shifts correspond to those of the original replace.

Insertion sequences (Cases 3–5) must be merged with regards to the desired document order. Between the end of the first and the beginning of the second insertion sequence, there is potential for text node adjacency which has to be resolved. In Section III, we claimed that atomic updates can be merged on the fly during cache preparation. This claim can now be substantiated based on the transformation rules given above. As can be seen, all information required for the merge is already contained in the AUC.

V. Implementing the XQuery Update Facility

So far, we have discussed efficient bulk updates at the implementation level as a series of atomic insert, delete, and replace operations. However, at the interface level, data is manipulated in terms of the primitives of the XQuery Update Facility (XQUF) [4]. In this section, we present how these primitives are implemented by atomic updates and the AUC. As the AUC is motivated by the Pending Update List (PUL) introduced by XQUF, this implementation is relatively straightforward. Nevertheless, one issue that needs to be addressed is in which order the XQUF primitives have to be added as atomic updates to the AUC to produce correct results. This issue is due to the fact that the AUC imposes an order constraint, whereas the order of the primitives in the PUL is exchangeable.

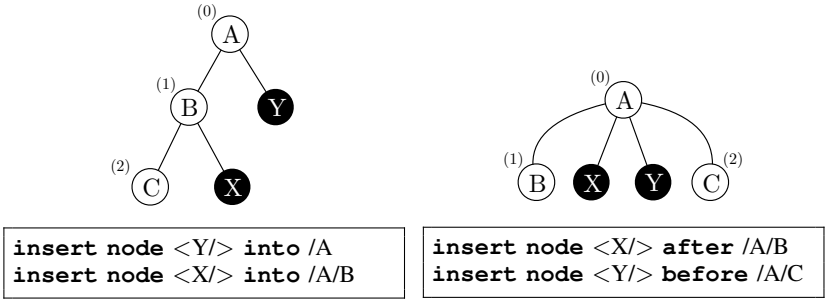
A. XQUF Update Primitives

Table II lists the update primitives defined by XQUF (§3.1)[2] together with their ranks and update location. Similar to the location field of an atomic update in the AUC, the location of an XQUF primitive identifies the node targeted by the update. For most update primitives, the location corresponds to the pre value of the target node. However, for some primitives, the location must be re-calculated as it is relative to the target value. For an *insert into as first* statement, the given insertion sequence is added directly after the attribute nodes of the target. The number of attributes must consequently be added to the target value to determine the appropriate location. The last three primitives add insertion sequences directly at the following position of the target node. The rank value is assigned based on the type of the primitive. In the case that multiple updates target the same node, the rank guarantees that these updates are applied in a way that the result of a query is always consistent with the XQUF specification. For example, if an *insert before* and an *insert after* have the same target, the *insert after* must be applied first due to the application order of the AUC. For the last three primitives, the order implied by the rank value is particularly important as they all access the same location.

[2]Note that *replace* stands for both *replace node* and *replace element content*, whereas *put* is not shown as it is outside the scope of this work.

primitive	rank	location
insert before	1	pre(n)
delete	2	pre(n)
replace	3	pre(n)
rename	4	pre(n)
replace value	5	pre(n)
insert attribute	6	pre(n)
insert into as first	7	pre(n) + attSize(n)
insert into	8	pre(n) + size(n)
insert into as last	9	pre(n) + size(n)
insert after	10	pre(n) + size(n)

TABLE II: Order of update primitives and calculation of location relative to target.



(a) Case 2 (b) Case 3

Fig. 10: Ordering of primitives

B. Ordering Update Primitives

Based on the rank and location of update primitives, they can be ordered in a way that not only satisfies the constraints of the AUC, but also yields the result specified by XQUF. To determine the order of primitives, pair-wise comparison is based on location, target, and rank. Unfortunately, it does not suffice to simply base the comparison on target, location or rank alone. First, the location of the two given update primitives has to be determined, as this is the most important property for ordering. Afterwards, there are four distinct cases that take care of all possible combinations of primitives.

1) Primitives operate at different locations in the sequential table: In this case, deciding on an order is simple. As the AUC is filled in document order, the primitive with the greater location value has to be applied first.

2) Primitives operate at the same location in the sequential table and one of them operates within the subtree of the other: Figure 10a gives an example of this case using two *insert into* statements. Assume that primitive a inserts <X> into <B> and primitive b inserts <Y> into <A>. Both primitives add the new nodes at the same location, hence $location(a) = location(b) = 3$. For each insertion sequence, primitive b must be applied first in order to end up under the appropriate parent. In the given case, the update of primitive a takes place in the subtree of $target(b)$ as it meets the conditions $location(b) > target(a)$ and $target(b) < target(a)$. Node Y has to end up after X and is therefore inserted first, which is achieved by the ordering $\langle b, a\rangle$. The example evolves around statements of type *insert into*, yet the same applies to the types *insert into as last* and *insert after*.

3) Primitives operate at the same location in the sequential table but on different target nodes: If the first two cases do not define the ordering of the primitives, the target value can be used, which is illustrated in Figure 10b. Primitive a inserts <X> after <B> and b inserts <Y> before <C>. As $target(b) > target(a)$, the final ordering is given as $\langle a, b\rangle$

4) Primitives of different types operate on the same target node: Finally, if all of the previous cases do not apply, the ordering is determined based on the rank. For example, if two primitives operate on the same target node with one of them being a *rename* and the other one a *delete*, their ranks imply that the delete has to be applied after the rename in order to adhere to the XQUF specification.

Observe that the four cases outlined above define a comparison function $c(x, y)$ that has the following properties, where $sgn(x)$ is the signum function.

$$sgn(c(x, y)) = -sgn(c(y, x)) \quad (1)$$

$$sgn(c(x, y)) = sgn(c(y, z)) = s \Rightarrow sgn(c(x, z)) = s \quad (2)$$

$$\forall z : c(x, y) = 0 \Rightarrow sgn(c(x, z)) = sgn(c(y, z)) \quad (3)$$

The first property ensures that the comparison order does not influence the resulting sort order of the primitives. The second property, transitivity, is guaranteed by the hierarchical decision process defined by Steps 1 to 4. The last property is only of theoretical interest as there can never be a comparison between two identical update primitives.

C. Correctness

To conclude this section, we sketch a proof of correctness for our XQUF implementation. Recall that duplicate updates (same target, same type) are prevented by the XQUF PUL.

An execution order is correct if every node is placed at the right position in document order relative to all other nodes. In our approach, the order of update primitives is determined by a score that is based on the pre value of the node and the rank of the primitive (cf. Table II). Since the score is used to order primitive using the comparison function $c(x, y)$, inconsistencies and wrong placements can only arise for primitives with the same score. We can distinguish the following two cases.

- If there are no ancestor/descendant relationships between the target nodes, location/target/rank is always unique. Since the pre value of the target is used as a tie-breaker, there cannot be two non-identical primitives that collide.
- The case of ancestor/descendant relationship has to be handled explicitly, but pair-wise comparison of two update primitives again suffices as only the relative order matters. As a consequence, this case can also be reduced to one of the four cases defined in the previous section.

factor	size	nodes	date elements	people size
0.01	1 MB	$3.3 \cdot 10^4$	$1.0 \cdot 10^3$	$5.2 \cdot 10^3$
0.10	11 MB	$3.2 \cdot 10^5$	$9.2 \cdot 10^3$	$5.0 \cdot 10^4$
1.00	116 MB	$3.2 \cdot 10^6$	$9.0 \cdot 10^4$	$5.1 \cdot 10^5$
10.00	1167 MB	$3.2 \cdot 10^7$	$9.0 \cdot 10^5$	$5.1 \cdot 10^6$
100.00	11,670 MB	$3.2 \cdot 10^8$	$9.0 \cdot 10^6$	$5.1 \cdot 10^7$

TABLE III: XMark benchmark documents statistics.

VI. Evaluation

The XQuery Update Facility based on structural bulk updates was implemented in the open-source native XML database BaseX[3]. Based on this implementation, we evaluated the proposed technique both in a quality improvement and in a feasibility study. Threats to the validity of these studies are discussed at the end of this section.

A. Experimental Setup

All measurements presented in this section are collected using commodity hardware. Specifically, a 2010 Apple iMac with an Intel Core i3 3.2 GHz 64-bit processor, 8 GB of main memory and a 1 TB hard-disk drive is used. Standalone versions of BaseX are run on OS X 10.8.4 using Oracle Java 7 (Update 12) with a 6 GB heap size (*-Xmx6G*).

The data sets used are generated by the XMark [16] benchmark for XML data management. Scaling factors, sizes, number of nodes in the table, number of date elements, and the size of the people element subtree are shown in Table III.

B. Quality Improvement Study

Even though we use the XMark documents in our evaluation, it was necessary to define our own queries as XQuery Update is not yet covered by the test suite. Bulk delete and insert as well as replace scenarios were run in a hot database state. First, a fresh and thus defragmented database was created for each query. Then, each combination of query and document was tested several times, until no significant change in processing time could be noted. The number of executed runs for measurements depends on the size of the document. For the smaller documents (1 MB, 11 MB) this equals twenty runs, for the 116 MB document ten runs and for the two biggest documents five and three runs. Although the answering time for the two tested versions of BaseX varies greatly, the number of runs remains the same to ensure equity. The fastest recorded processing times are reported here. For the 11.7 GB document the text and attribute indexes of BaseX are deactivated to not exceed the memory limits. No options are changed apart from this.

We use the `<date>` element as the target for bulk queries, as it is distributed over the entire XMark document. The element consists of an element node that contains a single text node as a child. We compare the performance of BaseX 7.3 (v73) and BaseX 7.7 (v77), since the latter implements the technique presented in this paper. An overview of the results is given in Table IV and in Figure 11a. Value updates are not shown in the figure as we focus on the efficiency of structural bulk updates.

[3]http://www.basex.org

Query	1 MB	11 MB	116 MB	1.1 GB	11.7 GB
Q1	8 ms	51 ms	0.52 s	8.6 s	1105 s
	9 ms	60 ms	0.61 s	8.1 s	414 s
Q2	80 ms	7577 ms	810.58 s	–	–
	10 ms	82 ms	0.96 s	22.4 s	1803 s
Q3	133 ms	9644 ms	1019.10 s	–	–
	17 ms	143 ms	1.96 s	145.9 s	17735 s

TABLE IV: Bulk update processing times of v73 (top) and v77 (bottom).

1) Value Updates: Although value updates are not affected by the optimization presented in this paper, their performance can be used to quantify the overhead, if any, introduced by our technique. In order to do so, we run the following query (Q1) on all five document instances.

```
for $d in //date/text() return
replace value of node $d with 99.99.9999
```

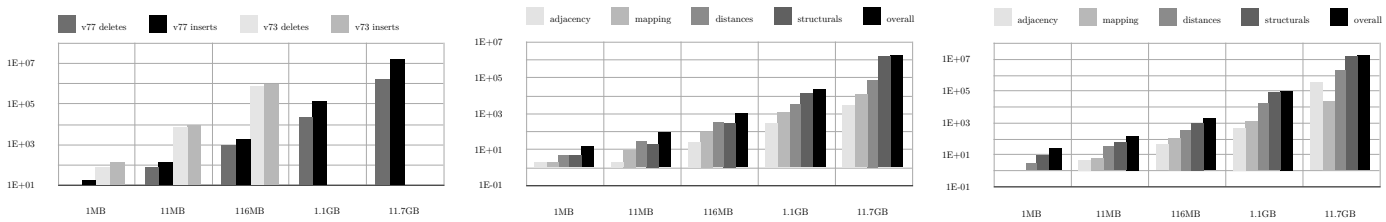
The comparison of processing times for v73 and v77 shown in Table IV indicates that there is no significant overhead due to efficient bulk updates. Throughout the documents 1 MB to 116 MB, the processing time scales almost linearly without noticeable difference between v73 and v77. Between the 1.1 GB and 11.7 GB document the processing time for both versions grows by a factor of 50 to 100. This super-linear increase is due to the fact that the actual text values are not directly stored in the Pre/Dist/Size table, but referenced through an offset. Alternating between the location of the currently accessed table block and the blocks of the file that holds the text values causes additional I/O in this scenario.

2) Deletes: To measure the performance of deletes, we run the following query (Q2) that deletes all `<date>` elements on each of the five document instances.

```
delete node //date
```

The total processing times reported in Table IV and plotted in Figure 11a clearly document the performance benefit of our technique. We can observe a reduction of processing time by at least an order of magnitude for all queries and document sizes. For the 1 MB to 116 MB documents, the processing time increases linearly with the document size for v77, whereas for v73 it grows exponentially by a factor of 100. This trend explains well why we aborted the tests for the two biggest documents with v73, as they would have taken several days to complete. At a rate of deleting around $1.8 \cdot 10^6$ nodes in 22.4 seconds and around $18 \cdot 10^6$ nodes in 30 minutes on these two documents, respectively, v77 still performs adequately. As seen with value bulk updates, we nevertheless observe a super-linear increase for the largest document size. In order to understand this effect, we break down the complete processing time of Q2 into its major components shown in Figure 11b.

- *Overall* describes the complete processing time. The overall value is always higher than the sum of the other parts, as some factors, e.g., query parsing, are not included.



(a) Total bulk update processing time (ms) (b) Breakdown of delete processing time (ms) (c) Breakdown of insert processing time (ms)

Fig. 11: Summary of the results of the quality improvement study

- *Structurals* accumulates the time for node deletion, updating of size values and shifting successor tuples on disk.
- *Distances* times distance adjustments, including the calculation of the appropriate node set, calculation of the new distances, and writing distances to disk. It also includes the time for computing the pre value mapping (see *Mapping*).
- *Mapping* accumulates time spent to map old pre values to new ones and vice-versa. The mapping operation is processed entirely in main memory.
- *Adjacency* captures the time spent for all text node adjacency-related operations, which includes checks and the actual resolution.

The processing times for *Distances*, *Mapping*, and *Adjacency* all increase linearly with the document size and do therefore not explain the overall deterioration. However, it can be observed that the time spent on *Structurals* increases and, towards the end, accounts for almost the complete processing time. Having addressed expensive distance adjustments, the new dominating factor in a structural bulk update is the actual deletion of nodes on disk and shifting tuples. Rather than being linked to the technique presented in this paper, this effect is due to the logical paging mechanism of BaseX, described in Section II. As BaseX simply leaves empty space at the end of pages in which nodes were deleted, the database does not shrink in size. Consequently, the probability of cache misses increases and performance drops disproportionally. Experiments have shown that, in the case of large databases, doubling the page size as a means to reduce the overall number of pages already pays off.

3) Inserts: The performance of inserts is measured using the following query (Q3) that inserts a new <ndate> element after every existing <date> element.

```
for $d in //date return
insert node <ndate>99.99.9999</ndate> after $d
```

The trend of the overall results reported in Table IV and plotted in Figure 11a follows the results for delete bulk update in Q2. Already for the smallest document size of 1 MB, v77 outperforms v73. Up to a document size of 116 MB, the processing time increases linearly. In contrast to the previous experiment, the performance deterioration for inserts already occurs at the step to the 1.1 GB document. Nevertheless, inserting about $1.8{\cdot}10^6$ nodes at approximately $9{\cdot}10^5$ locations in less than two and a half minutes is impressive. To isolate the reason for this performance drop, we again broke down the query processing time of Q3 into the components shown in Figure 11c. As observed with Q2, the insertion of nodes is by far the most expensive part. While *Mapping* scales well, all tasks that perform I/O operations become increasingly expensive. Again, the cause for this performance deterioration is not related to the technique presented in this paper, but due to the logical paging of BaseX. Upon database creation, BaseX fills logical pages to capacity to minimize the database size. If new nodes are added, a new logical page has to be appended after the sequence of all existing pages on disk and the page directory keeps track of document order. This is detrimental in two ways. First, a lot of partially filled pages are created, as there is no redistribution of existing tuples between existing pages. Second, the database is no longer contiguously stored on disk, but logically inserted nodes in the middle of the document are stored physically at the end of the table. These issues greatly increase the number of I/O operations as the document size grows. Adding a text node contributes in the same manner, as the inserted text has to be added to the appropriate file.

4) Replaces: Finally, we evaluate the different optimizations for replace operations that have been introduced in Section IV. Recall that the *rapid* replace aims at reducing processing time, whereas the goal of the *lazy* is to avoid fragmentation. To analyze these optimizations and to compare them, we have measured their run-time performance in a number of setups (including the worst-case) and in two distinct scenarios.

Even: The replacing and the replaced tree have the same size or node count. Therefore, successor tuples do not have to be shifted. We use the following XQuery expression for this comparison. The size of the *people* subtree for different documents is given in Table III.

```
replace node //people with //people
```

Uneven: We replace the target node <people> with the <europe> subtree, which is about half the size, using the XQuery expression below. As the two trees do not feature the same structure, *lazy* replace cannot be applied. Nevertheless, the *uneven* comparison can quantify the benefits of the *rapid* replace better.

```
replace node //people with //europe
```

In order to get comparable results, the individual replace approaches are explicitly activated in the code. A modified

	1 MB	11 MB	116 MB	1.1 GB	11.7 GB
basic	18 ms	144 ms	1630 ms	20.73 s	528.7 s
lazy	16 ms	133 ms	1496 ms	17.87 s	298.6 s
rapid	16 ms	141 ms	1601 ms	19.51 s	310.9 s
lazy/rapid	17 ms	155 ms	1724 ms	21.03 s	357.7 s
basic	17 ms	121 ms	1302 ms	14.62 s	267.9 s
rapid	13 ms	105 ms	1103 ms	11.87 s	169.9 s

TABLE V: Performance of different replace types in v77 for the even (top) and uneven (bottom) scenario.

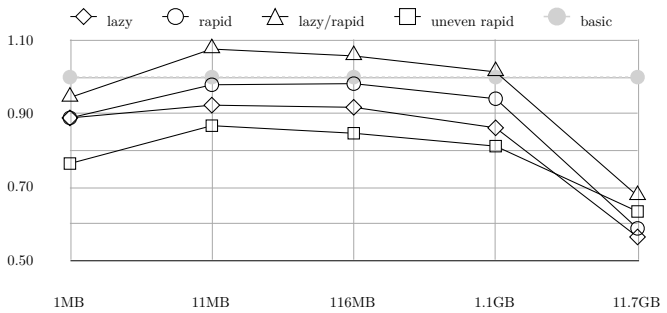


Fig. 12: Relative comparison of different replace scenarios

version of v77 is tested that includes the *lazy* replace. Table V summarizes the total processing times, whereas Figure 12 relates the different strategies to the *basic* replace.

Basic: The *basic* replace serves as the baseline for the relative comparison in Figure 12. The target node is first deleted, followed by the insertion of the replacing tree at the same location. For the two biggest documents, we observe a super-linear drop of performance. Nevertheless, replacing around $5 \cdot 10^7$ nodes in under 10 minutes yields an impressive rate of almost 100 node replaces per millisecond. The deterioration of the performance for larger document sizes can be explained following the same argument as for bulk inserts and deletes.

Lazy: A *lazy* replace can only be applied if the two trees are structurally equal. Traversing both trees completely for comparison is therefore mandatory. The resulting processing time consists of a linear traversal of the table, in addition to a single value update afterwards, which is negligible. Although the *lazy* approach is only marginally faster (about 10%) than the *basic* replace for smaller documents, its benefit increases with the document size, reaching about 43% for the 11.7 GB document. Nevertheless, the reduced rate of fragmentation is the major benefit of this approach.

Rapid: The results of the *rapid* replace are similar to the results of the *lazy* replace. While the difference at the beginning is even smaller ($< 5\%$), it clearly outperforms the naive approach on the 11.7 GB document with about 41% performance gains. Due to the paging strategies, overwriting the table in-place seems to pay off where performance sharply deteriorates otherwise.

Lazy/Rapid: Comparing a failed *lazy* replace with the *basic* approach is the most relevant setup. In order to do so, we provoked the worst-case by making sure that the *lazy* replace needs to sequentially scan the complete replaced tree, before aborting due to structural differences with the replacing tree. At that point, the *rapid* replace is used to perform the operation. For document sizes from 11 MB to 1.1 GB, even the *basic* replace shows better performance in this setup. However, the difference between a *basic* replace and a failed *lazy* replace stays well below 10% most of the time. In the case of the 11.7 GB document, the benefit of using a *lazy* replace, even if it might fail, over a *basic* replace can be quantified as a speedup of $> 30\%$.

Uneven scenario: In the last scenario, the replacing tree is only half the size of the replaced tree. For cases like this, the *lazy* replace yields no overhead, since the initial comparison of tree sizes immediately terminates the scan. However, in this scenario further performance benefits of the *rapid* replace can be observed. The reason is that the optimized replace overwrites tuples in-place and thus avoids random I/O access, whereas the *basic* approach first deletes some tuples and afterwards re-inserts them at the physical end of the table.

C. Feasibility Study

The AUC presented in Section III is an additional data structure that is introduced by our technique for efficient bulk updates. The goal of this feasibility study is to investigate whether these additional memory requirements are prohibitive. For our study, we revisit the bulk insert and replace queries described in the previous subsection. During the bulk insert queries, up to $18 \cdot 10^6$ nodes are added to the database, which are all cached in memory in addition to the AUC and PUL data structures. Profiling the memory consumption of the Java virtual machine revealed that the assigned 6 GB were more than sufficient to execute all of the tested insert and replace queries. Since, additionally, the size of memory required by nodes from bulk insert and replace is much larger than the additional data structures introduced by AUC, we conclude that our approach is indeed feasible. The fact that the implementation of our approach is part of recent productive versions of BaseX further confirms its viability in practice.

D. Threats to Validity

The results presented in this section have been obtained within BaseX, which is based on the Pre/Dist/Size XML encoding. As a consequence, the validity of these results is limited to systems that use a similar encoding based on the pre/post plane with fixed-length records, e.g., the Pre/-Size/Level encoding used in MonetDB/XQuery.

However, a number of native XML databases use encodings that are based on dynamic labeling schemes, which will be discussed in more detail in Section VII. For example, Sedna[4] and eXist-db[5] both use an internal representation that is based on the Dewey encoding [9] and use B^+ trees to index nodes on disk. A performance comparison of the XQUF support presented in this paper with these native XML database would therefore be interesting and relevant. Unfortunately, neither Sedna nor eXist-db offer XQUF support at this time. Additionally, these systems do not cache bulk updates and apply each atomic update immediately. While this approach does not

[4] http://www.sedna.org
[5] http://exist-db.org

require any resources for caching, it might lead to undesired side-effects with respect to the XQUF semantics.

VII. Related Work

Updating XML is challenging as the structural order of the document has to be observed when it is manipulated. In order to address this problem, a number of XML encodings have been proposed to map hierarchical documents to a flat representation. Initially, the main focus of these encodings has been to support efficient querying of the XPath axes. As updating data in native XML databases gained importance, the focus of research shifted. For example, O'Connor and Roantree [12] surveyed existing encodings with respect to their suitability for processing updates.

XML encodings can generally be classified into two families. First, prefix-based encodings [15] use variable-length labels to denote the position of a node within an XML document. Among others, notable proponents of this approach are ORDPATH [13] and DeweyIDs [9]. As there is no need to re-label existing nodes when the document is updated, these encodings have a high updatability. However, as the length of the label is not fixed, overhead is introduced elsewhere. Accessing records on disk now requires an index and labels can get very bulky for deep documents or in the case of frequent insertions. Label compaction [1] methods have been proposed to address the latter problem.

Region or interval-based encodings store records ordered based on a tree traversal. The majority of approaches are based on a pre-order traversal. The additional information that is stored varies. To increase updatability, the Pre/Size [10] encoding uses "extended pre-order", which leaves gaps for future node insertions, and a size value that indicates the number of descendant nodes. In the context of MonetDB/XQuery [3], which is based on the Pre/Size/Level XML encoding, the use of an additional Pos/Size/Level table [2] has been proposed to process updates more efficiently. In the setting of the Pre/Level/Parent, Noonan *at al.* [11] have proposed to address updates by leveraging techniques used in relational databases to rebuild indexes. Finally, Thonangi [18] proposes a concise hybrid labeling scheme that attempts to unify the advantages of prefix and region/interval-based encodings. Note that the Pre/Dist/Size encoding is already more updatable than most other interval or region-based schemes as it encodes the parent of a node using a relative rather than a direct reference.

The benefit of exploiting bulk updates to optimize sequences of atomic updates has been demonstrated in the past, see e.g., Srivastava and Ramamoorthy [17]. To the best of our knowledge, however, the work presented in this paper is the first proposal to leverage this technique in native XML databases.

VIII. Conclusion

The semantics of XQUF and, in particular, the introduction of the PUL have opened up new optimization possibilities for updates in XML databases. In this paper, we proposed efficient structural bulk updates as a technique that exploits these opportunities in the setting of the Pre/Dist/Size XML encoding. Specifically, we demonstrated how the cost of maintaining the document order can be amortized by delaying distance adjustments for a series of atomic updates. Our evaluation of efficient structural bulk updates showed that this technique reduces the processing time by orders of magnitude. Moreover, large bulk updates that previously had prohibitive runtimes are now feasible. Our technique has been implemented in the open-source native XML database BaseX and is part of all releases starting from Version 7.7.

References

[1] R. Alkhatib and M. H. Scholl, "Compacting XML Structures Using a Dynamic Labeling Scheme," in *Proc. British National Conference on Databases (BNCOD)*, 2009, pp. 158–170.

[2] P. A. Boncz, J. Flokstra, T. Grust, M. van Keulen, S. Manegold, S. Mullender, J. Rittinger, and J. Teubner, "MonetDB/XQuery – Consistent and Efficient Updates on the Pre/Post Plane," in *Proc. Intl. Conf. on Extending Database Technology (EDBT)*, 2006, pp. 1190–1193.

[3] P. A. Boncz, S. Manegold, and J. Rittinger, "Updating the Pre/Post Plane in MonetDB/XQuery," in *Intl. Workshop on XQuery Implementation, Experience, and Perspectives (XIME-P)*, 2005.

[4] D. Chamberlin, M. Dyck, D. Florescu, J. Melton, J. Robie, and J. Siméon, "XQuery Update Facility 1.0," http://www.w3.org/TR/xquery-update-10, 2009.

[5] J. Clark and S. DeRose, "XML Path Language (XPath) Version 1.0, W3C Recommendation," http://www.w3.org/TR/1999/REC-xpath-19991116, 1999.

[6] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh, "XQuery 1.0 and XPath 2.0 Data Model (XDM)," http://www.w3.org/TR/xpath-datamodel, 2007.

[7] C. Grün, "Storing and Querying Large XML Instances," Ph.D. dissertation, University of Konstanz, 2010.

[8] T. Grust, "Accelerating XPath Location Steps," in *Proc. Intl. Conf. on Management of Data (SIGMOD)*, 2002, pp. 109–120.

[9] M. P. Haustein, T. Härder, C. Mathis, and M. Wagner, "DeweyIDs – The Key to Fine-Grained Management of XML Documents," *Journal of Information and Data Management (JIDM)*, vol. 1, no. 1, pp. 147–160, 2010.

[10] Q. Li and B. Moon, "Indexing and Querying XML Data for Regular Path Expressions," in *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 2001, pp. 361–370.

[11] C. Noonan, C. Durrigan, and M. Roantree, "Using an Oracle Repository to Accelerate XPath Queries," in *Proc. Intl. Conf. on Database and Expert Systems Applications (DEXA)*, 2006, pp. 73–82.

[12] M. F. O'Connor and M. Roantree, "Desirable Properties for XML Update Mechanisms," in *Proc. 2010 EDBT/ICDT Workshops*, 2010, pp. 23:1–23:9.

[13] 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. Intl. Conf. on Management of Data (SIGMOD)*, 2004, pp. 903–908.

[14] J. Robie, D. Chamberlin, M. Dyck, and J. Snelson, "XQuery 3.0: An XML Query Language," http://www.w3.org/TR/xquery-30/, 2014.

[15] V. Sans and D. Laurent, "Prefix Based Numbering Schemes for XML: Techniques, Applications and Performances," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1564–1573, 2008.

[16] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "XMark: A Benchmark for XML Data Management," in *Proc. Intl. Conf. on Very Large Databases (VLDB)*, 2002, pp. 974–985.

[17] J. Srivastava and C. V. Ramamoorthy, "Efficient Algorithms for Maintenance of Large Database," in *Proc. Intl. Conf. on Data Engineering (ICDE)*, 1988, pp. 402–408.

[18] R. Thonangi, "A Concise Labeling Scheme for XML Data," in *Proc. Intl. Conf. on Management of Data (COMAD)*, 2006.