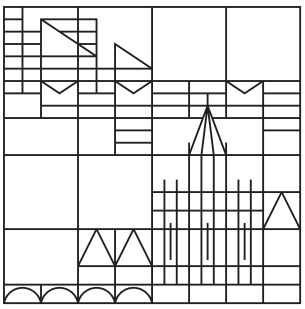
Polishing Structural Bulk Updates in a Native XML Database

Master thesis presented by

Lukas Kircher

Universität Konstanz



Submitted to the Department of Computer and Information Science at the University of Konstanz

Reviewers

Prof. Dr. Marc H. Scholl

Prof. Dr. Marcel Waldvogel

October 2013

Abstract

BaseX is a native XML database on the foundation of a fixed-length, sequential document encoding. Built with a strong focus on reading performance, this thesis shows that the *Pre/Dist/Size* encoding is yet perfectly capable of handling massive bulk update transactions. Despite its theoretical limitations regarding structural changes of the tree, we show that performance is indeed restricted by the document order and disk access patterns. During tests with the XQuery Update Facility (XQUF), we delete 1.8 million nodes in 22.4 seconds, evenly distributed over a 1.1GB XMark document instance. Compared to the prior approach, this equals a reduction of processing time by 99.99%. To achieve this, we extend the obligatory implementation of the XQUF pending update list with an additional low-level layer, that pre-calculates tree structure adjustments in-memory. This layer adds little overhead and further enables us to merge update operations and curb fragmentation that finds its origin at the user level. To not violate the document order, the XQUF is only arduously brought together with the concept of efficient bulk updates. A method is introduced that imposes an order on update primitives to finally get a ready-to-apply sequence of atomic updates. The reviewed implementation is fully consistent with the XQUF specification and has already proven rock-solid efficiency in production use. A few theoretical paragraphs on alternative approaches, disk access patterns and memory consumption highlight sleeping potential and prepare further progression.

Contents

1 Introduction

1.1 Motivation

Relational, object-oriented, NoSQL, XML-enabled, etc. - although flooded with numerous alternatives, there is still a place in the market for native XML databases (NXD). Different problems require different measures. As a highly-efficient open-source XML database engine and query processor, BaseX [1] fills one of the niches successfully. With the foundation of a company that evolves around the product and data-related consulting services, the project underlies constant development. And naturally, use cases get more challenging.

Naively, we thought that the initial implementation of the XQuery Update Facility (XQUF) is capable of handling even large bulk updates. Where, for instance, deleting ten thousand single nodes from a database could still be finished in reasonable time, it soon became obvious that the number of updated locations easily ventures into the hundreds of thousands. Updates of this size either take days to complete, or have to be split up over several sub queries to circumnavigate the quadratic increase in complexity.

To be highly regarded in the community of database users, being efficient only gets you so far. Reliability, standard-conformity and a rich feature set reside just as high on the list of most wanted qualities. Yet, they often interfere with feasibility. We are therefore most proud of BaseX, seeing that it yields superb reading performance, meeting all the requirements of a full-fledged database system. Grün sums up the theories and implementation in his 2010 thesis [Gr0]. Being processed ok, we are now eager to track down and eliminate the current limits of structural bulk updates as well. Why? Being slow never helps - and we also think the fight for an efficient architecture is neither lost nor won on paper.

[1] http://www.basex.org, *A scalable and high-performance, yet lightweight XML database engine and XPath/XQuery processor*

1.2 Contribution

BaseX is based on an encoding that caters primarily to an efficient evaluation of reading queries. Yet, in the course of this thesis, we extend it in a way that updating queries are evaluated just as well. The work especially focuses on the speed-up of massive bulk queries, that alter the structure of the XML tree in multiple locations.

There have been several attempts in the last years to get hands on the Holy Grail of a native XML encoding that is compact and yields both, excellent reading and writing performance. ORDPATH [OOP+04] is one of the well-known and repeatedly enhanced representatives. Despite some proposals for such encodings, we do not know of a single full-scale NXD based on a dynamic labeling scheme, that offers high performance as well as complete support for the XQUF standard.

While the general contribution might be considered limited as it is strongly tailored to BaseX, we think it is only partially so. Potential future adaptations can find clear answers to the two following problems:

1. How dreaded structural bulk updates can be evaluated efficiently on a fixed-length sequential encoding.

2. How XQUF queries are processed to leverage efficient bulk updates.

While question one is conclusively answered, the second point requires some background knowledge that is found in Kircher's bachelor thesis [Kir10].

1.3 Overview

In chapter 2, we get started with the current state of affairs regarding BaseX and bulk update transactions after introducing some preliminaries. After this, we will develop a theoretical approach that eliminates the main performance drawbacks. A detailed discussion of the implementation and further optimizations is followed by recommendations for future work. Chapter 3 takes a closer look at adaptions to the XQUF module, that are necessary in order to speed up bulk updates. An in-depth look at the performance of the actual implementation is presented in chapter 4 where we also identify the remaining weak points and briefly propose solutions. Having introduced the relevant background, we add a few words on related work in chapter 5 to put different approaches into perspective.

2 Efficient Structural Bulk Updates

2.1 Introduction

In order to show the general problem with bulk updates and to develop a solution, it is necessary to first take a look at the basics. Section 2.2 starts the discussion with a brief description of the mapping schema in BaseX. Being regularly referenced to describe navigation in a document, section 2.2.2 reviews the most important XPath axes. After this, we close in on the actual problem we want to solve by taking a look at the general effects of updates on the table (section 2.2.3) and particularly at the costs of structural bulk updates (sections 2.2.4 and 2.2.5).

Section 2.3 is all about a theoretical approach to efficient structural bulk updates. It first introduces a specific structure that caches atomic updates and reports on its use to speed up the process, as well as its restrictions. Details and pseudo-code of the actual implementation are given in section 2.4. The additional layer enables us to add a few optimizations conveniently (section 2.5) and provides room for future undertakings (2.6).

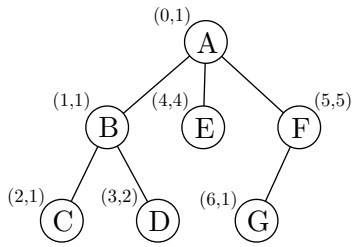
2.2 Preliminaries

2.2.1 XML Encoding in BaseX

In his thesis [Gr0], Grün explains the underlying encoding and its origin in great detail. We consequently introduce only the facts that are important for the following discussion. A proper encoding for a native XML database maps input and output in both directions. On the one hand, a given document should be stored in an efficient manner and on the other, it must be possible to restore the original document with all its details from the information in the database. In-between, typical queries should be evaluated efficiently. Regarding XML, an important property is the document order which is equivalent to a preorder traversal of the XML tree. Whenever queries are performed, the order of the result sequence adheres to the document order.

```
<A>
  <B><C/><D/></B>
  <E/>
  <F><G/></F>
</A>
```

Figure 2.1: XML sample document



(a) abstract tree incl. *(pre,dis)* 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

(b) sequential table

Figure 2.2: Relational encoding of sample document.

XML has been introduced first and foremost to store textual content in a human-readable fashion. Storing a book with all its chapters, headings and paragraphs in XML is a good example for the importance of the document order. With XQuery as the most important interface for access, a suitable encoding maps the properties of individual nodes in a way that the main XPath axes can be evaluated efficiently. BaseX features a fixed-length, sequential encoding that combines all these characteristics. Figure 2.1 shows a small XML snippet that we use to visualise the way from document to database, whereas figure 2.2 shows the mapping in detail. During the parsing step, the input document is traversed in preorder. For each visited node, a tuple of the form *(pre, dist, size, val)* is created that reflects the document order, tree-structure and the size and value of a node. Each tuple is then added to a sequence or sequential table.

Pre (pre)

The pre value serves as a quasi node identifier and reflects the number of nodes that have been visited before. Counting starts at zero and the maximum *pre* value equals the number of tuples in the table minus one. If nodes are deleted or inserted, following tuples are shifted accordingly so that this condition remains always true and the document order unviolated. Pre val-

ues are stored implicitly as they equal the row number. The actual cost of shifting pre values is due to shifting of tuples on disk.

Distance (dist)
The distance value keeps track of the tree structure by encoding the parent node in a relative manner. As the distance value is equal to the number of nodes between a node and its parent, it follows that the parent of a node n is equal to its pre value minus its distance, or *n.par = n.pre - n.dist*. Having a distance of one, the parent of the root node is *-1*.

Size (size)
The size value encodes the size of a node, that is the number of nodes in its subtree plus itself.

Value (val)
For now we use the value field to keep additional information on the node which is most often the element name, if not stated otherwise.

The actual real-world encoding contains some more information that we leave out for now, with the node kind being the most prominent one.

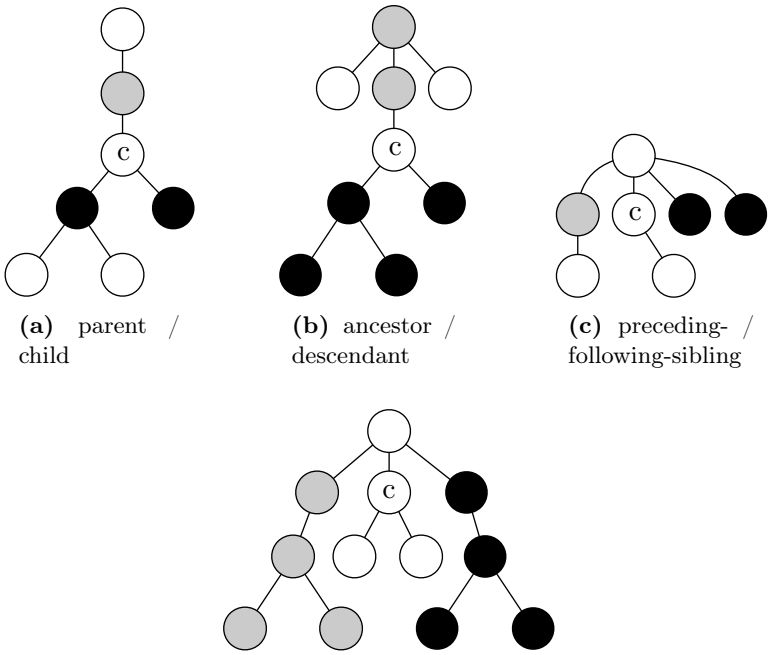
2.2.2 XPath Axes

Navigation in an XML tree can easily be described by means of the XPath axes [CD99]. Each axis describes a set of nodes relative to the given context node. As axis steps are frequently referred to during the argument we shortly introduce the most important ones (figure 2.3) and show how they can be evaluated based on our specific numbering scheme.

Parent
As seen before, the parent is encoded relatively via the distance and has the pre value $par.pre = c.pre - c.dist$.

Child
The first child n resides at position $c.pre + 1 = n.pre$. The remaining child nodes are determined iteratively by adding the size to the pre value of the current child, hence $next.pre = current.pre + current.size$. Iteration stops once $next.pre \geq c.pre + c.size$.

Ancestor
The ancestor axis is evaluated by calculating the parent of the current node iteratively until $current.pre = 0$.



(a) parent / child

(b) ancestor / descendant

(c) preceding- / following-sibling

(d) preceding / following

Figure 2.3: XPath axes relative to the context node (c).

Descendant
All descendants of node c lie in the interval $[c.pre + 1, c.pre + c.size - 1]$. Starting at $c.pre+1$ it is straightforward to determine the set of descendants.

Preceding-Sibling
An extra step is necessary to determine the set of preceding-siblings. First the parent node par is determined via $par.pre = c.pre - c.dist$. Starting at $par.pre$ the child axis of par is evaluated until context node c is hit.

Following-Sibling
Accordingly to the preceding-sibling axis, the parent of c has to be determined. Starting at c the child axis of par is iterated until $current.pre \geq par.pre + par.size$.

Preceding
The preceding axis includes all nodes n with $n.pre < c.pre$, with the exception of ancestor nodes. In course of this work we never really access it but it can be calculated simply by choosing pre values in the interval $[0, c.pre - 1]$ and subtracting the ancestor nodes (as well as attributes and namespace nodes).

pre	dist	size	value
0	1	3	A
1	1	1	**B**
2	2	1	C

(a) table before update

pre	dist	size	value
0	1	3	A
1	1	1	**X**
2	2	1	C

(b) table after update

Figure 2.4: An atomic value update renames node B to X.

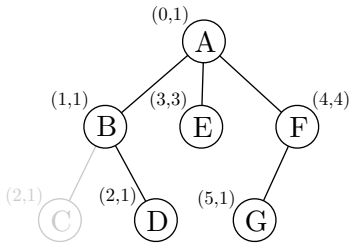
Following
The set of following nodes consists of all nodes in the document after the closing tag of the context node. Pre values lie in $[c.pre + c.size, T.size - 1]$, where $T.size$ equals the number of tuples in the table.

Other Axes
There are a few more axes with some of them being conglomerates of the presented ones *(ancestor-or-self, descendant-or-self)* and others that are either trivial *(self)* or not important for the ongoing discussion *(namespace, attribute)*.

2.2.3 Effects of Updates on the Table

Of course, a database system without the possibility to modify its content is of limited use. In course of this work, the term *atomic updates* will occur regularly. An atomic update is the most basic form of an update that can be applied to a database. We distinguish between two groups of atomic updates. Value updates are the ones easy to deal with, as they only induce changes to the value field of a tuple. An example would be the renaming of an element node. In figure 2.4 element B is given the new name X. The number of tuples in the table remains untouched, as well as the overall structure. Changes are exclusively applied to the value field of node B which has a pre value of 1. We take away from this example that each atomic value update results in a value or reference update of a single tuple which can be carried out quickly.

Among the structural updates there are (for now) two distinct types: insert and delete. Structural atomic updates lead to more profound changes of the table as new nodes are inserted or existing ones deleted. The number and sequence of tuples changes which is directly reflected in the pre, dist and size values. Deleting node C from our earlier example shows the consequences (figure 2.5). Updated values in the table are printed bold.



(a) tree after update

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

(b) table after update

Figure 2.5: Deleting a node from the table.

At least for the given example we can see that structural changes can indeed be very expensive. If node D is deleted instead, node C would remain untouched, but in this specific case each tuple of the table has to be changed in one way or another. The deleted node C had a pre value of 2. As pre values of the tuples are continually numbered, all tuples with pre greater 2 are shifted by -1, the number of deleted tuples (nodes *D, E, F, G*). If the number of tuples between a node and its parent changes the distance value must be updated, which is the case for nodes *D, E, F*. The size values of nodes A and B are reduced by 1, as they both loose a node in their subtree. In the end, the total costs depend highly on document structure and location of the update.

2.2.4 Costs of Structural Updates

This paragraph provides a more general view on update costs and how pre, dist and size values are influenced by structural updates in total. We insert a document A under the parent a at location p into our table. The number of inserted nodes is s, the size of the destination table equals n. The individual columns of the table are updated as follows:

Pre
All tuples t_i, where $t_i.pre$ is in the interval $[0, p-1]$, remain untouched. For all $t_i.pre$ in the interval $[p, n-1]$, $t_i.pre$ is recalculated as $t_i.pre = t_i.pre + s$. In the worst case, if $p = 1$ all tuples except the first one have to be shifted. As mentioned before, pre values are implicitly given by the row number. To avoid excessive disk access the table in BaseX is divided into pages and the order of pages is stored in a directory. To save I/O by avoiding tuple shifting, additional tuples are stored consequently at the physical end of the table on disk. Updating pre values is therefore not considered cost-prohibitive and depends mostly on p, the current page structure and fragmentation.

Size
The size value of a tuple t_i has to be updated if $t_i.pre < p < t_i.pre + t_i.size$. That means if the number of nodes in the subtree of a tuple t_i changes, it follows that $t_i.size = t_i.size + s$. With regards to the updating location p, all ancestor tuples have to be updated. The number of ancestors of a node equals its level (if the root node has a level of zero). For a 'sane' document, updating the appropriate size values is not a problem even if there is some overlap to be expected, which is especially true for the root.

Dist
The costs of updating distances is harder to predict. Formally, for a tuple t_i, if $(t_i.pre - t_i.dist) < p <= t_i.pre$ then $t_i.dist = t_i.dist + s$. That means if the number of nodes between a node and its parent changes, its distance has to be adjusted accordingly. Or, looking at it from the update location perspective, nodes to update lie on the following-sibling axis of p and on the following-sibling axis of all ancestors of p. By using size and distance values this set of nodes can be computed efficiently. It is obvious that the size of this set highly depends on the structure of the document. Whereas the levels of a document are limited, the number of siblings is not. Figure 2.6 shows the worst-case that is easily extended to replicate real-life scenarios. If node m with $m.pre = p$ is deleted, all following siblings have to be updated. In contrast to Pre, where updating costs are moderate, distance values are updated explicitly, which in this case takes $n-2$ steps. For bigger documents it can be estimated that each of these steps or distance updates results in accessing another page on disk or a full I/O.

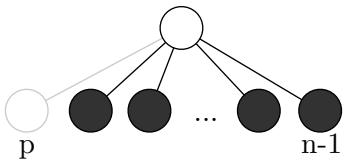


Figure 2.6: Locations of distance updates upon deletion in a flat document (marked black).

In general, insert and delete behave similar with the difference being that an insert at p shifts the existing tuples starting with $t_i.pre = p$ to the back by the size of the inserted subtree. Deletion of the tuple r affects the tuples t_i starting at $t_i.pre = r.pre + r.size$, as the deleted node is no longer part of the table. It is clear that distance adjustment is the most expensive part of the updating process at the moment. Besides our theoretical analysis, numerous profiling sessions with BaseX confirmed us that the practical situation is just as serious.

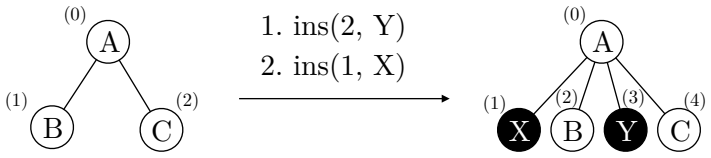


Figure 2.7: Bulk update example incl. order of updates.

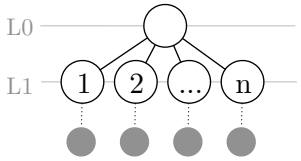


Figure 2.8: Bulk insertion leading to $O((n-1)^2)$ number of distance updates on level 1.

2.2.5 Bulk Updates

It is important to note that, due to the language specification of XQuery Update, one is obligated to cache all updates that are to be executed within a transaction or snapshot in a structure called the *pending update list*. After checking whether the application of the pending update list leads to a consistent state of the database, only then can updates be applied. In general, this concept helps to realize atomicity and consistency. We also use it to tailor the complete updating process to our encoding schema. Let us consider we want to perform a bulk update consisting of two insert operations (figure 2.7). The first insert adds node Y at position 2 and the second insert adds X at position 1. As we have seen before, inserting or deleting nodes in the table leads to a shift of pre values for all following nodes. By applying the two inserts in the given order, from the highest to the lowest pre value, we avoid recalculating the insert positions. Traversing the result in document order yields the node sequence *(A, X, B, Y, C)*. Changing the order by inserting X at position 1 first followed by insertion of Y at 2 would yield the sequence *(A, X, Y, B, C)* which is not the desired result. Cached updates are always applied in reverse document order (with regards to the location) to avoid exactly this issue.

Clearly, if a single atomic update can be expensive than a sequence of updates is even more so. Yet, there can be a significant overlap between the sets of distance values that have to be adjusted. Figure 2.8 shows such an example. The root node has n children. Into each of these children additional nodes are inserted, which leads to necessary distance updates on level 1 for all following nodes. If nodes are inserted into node 1, $n-1$ distances have to be adjusted. In general $n-i$ distance updates are performed for an insertion

into node i which leads to $\sum_{i=0}^{n} n - i$ for n nodes. At least for the given example, distance adjustments are in $O((n-1)^2)$. Real-world use cases are often based on a similar combination of document structure and update strategy. Consequently, looking at the characteristics of a bulk update before touching the table might save us a considerable amount of work. Performance is not an issue for a single atomic update and we cannot reduce the number of distance updates without changing the encoding schema anyway. Optimizing a sequence of updates is therefore the main objective.

2.3 Theory behind Efficient Structural Bulk Updates

In the last sections we took a closer look at the costs of structural updates and, consequently, at the costs of a sequence of updates. It has been shown in theory that distance adjustments are cost-dominating. Especially as the same distances are touched repeatedly, finding a method that avoids redundant access without adding excessive overhead could help to reduce overall processing time by magnitude. This assumption is also supported by practical tests.

From now on, a structure named *Atomic Update Cache* or *AUC* holds all atomics of a bulk update in document order, depending on their location. In contrast to the naive approach, where distances are *iteratively* adjusted with each atomic update, the proposed method adjusts distances *explicitly* after all updates have been applied. In the following passages we develop that method with the aid of a few simple examples. Saving time during bulk updates essentially boils down to the given facts:

- Distance updates can be delayed if updates are applied in reverse document order (high-to-low pre value wise).
- The AUC serves as a bi-directional mapping of pre values before and after updates.
- A new distance can be calculated explicitly for any node based on its original state and the mapping.

2.3.1 Delaying Distance Adjustments

As it has already been shown, a sequence of updates is executed from the highest to the lowest pre value to avoid re-computation of the individual update locations. A tuple t is only shifted if the number of nodes changes in the interval $[0, t.pre]$. Similarly, distance values are adjusted if the number of nodes changes between a child and its parent. Inserting or deleting a tuple invalidates only the distances of following tuples. In the course of a

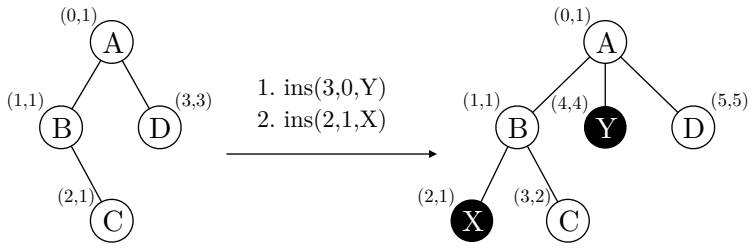


Figure 2.9: State of the document tree before and after bulk update.

bulk update, the part of the table, accessed by consecutive atomic updates, remains always valid. This fact plays an important role as it enables us to delay distance updates altogether. To apply atomic updates we proceed as follows:

1. Traverse atomic updates in the AUC back-to-front, which is equivalent to a traversal in reverse document order.

2. Insert or delete the corresponding nodes for each atomic update, implicitly shift pre values of the following tuples and adjust the size values of the ancestors. Distances remain untouched.

3. After all atomic updates have been applied, restore the tree structure by adjusting distances in an efficient manner.

Figure 2.9 shows a document before and after the execution of a transaction with two atomic insert operations. The first insert is performed at position 3 where the new node Y is inserted into the parent 0/A, whereas the second insert is performed at location 2 and inserts X as a child node of 1/B.

Figure 2.10 shows the different states of the table, where changes to the preceding table are printed bold.

(b) Between the original state (a) to state (b), the node Y is inserted at position 3 incl. a valid parent reference ($dist = 3$) and the correct size. As a consequence, node D shifts to the back and its pre value changes from 3 to 4, whereas the already existing distance value remains untouched. The size value of A is incremented by 1.

(c) Step three shows the table after the insertion of X at position 2 with a new distance value of 1 (as it's a child of B). Once again the following tuples are shifted and the respective pre values incremented (C, Y, D). The ancestor's size values are incremented by 1 (A, B). Again, existing distances remain untouched. At this stage we can see that nodes (C, Y, D) still have their original distance values. Based on these, distances are adjusted at a later point (which will be shown shortly).

pre	dist	size	val
0	1	4	A
1	1	2	B
2	1	1	C
3	3	1	D

(a) original state

pre	dist	size	val
0	1	**5**	A
1	1	2	B
2	1	1	C
3	**3**	**1**	**Y**
4	3	1	D

(b) after first insert

pre	dist	size	val
0	1	**6**	A
1	1	**3**	B
2	**1**	**1**	**X**
3	1	1	C
4	3	1	Y
5	3	1	D

(c) after second insert

pre	dist	size	val
0	1	6	A
1	1	3	B
2	1	1	X
3	**2**	1	C
4	**4**	1	Y
5	**5**	1	D

(d) after distance updates

Figure 2.10: States of the table during a bulk update.

(d) After updates are finished, distances have to be adjusted. State (d) shows the final table state.

This discussion allows us to record the observation:

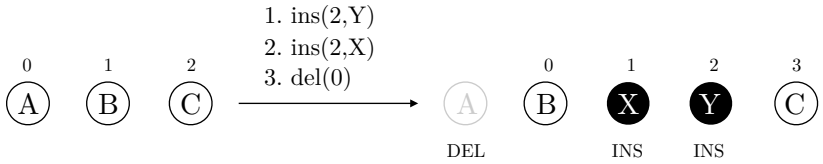
Observation 1 *Adjusting distance values can be delayed until the very last step of the updating process.*

2.3.2 Mapping Pre Values Before and After Updates

To adjust distance values explicitly we need to be able to map respective pre values before and after structural changes take place. We want to know the pre value of a specific tuple before structural changes have been applied and v.v.. Based on the AUC, that consists of a sequence of atomic updates in document order, this can be achieved easily.

Let us consider we want to insert node Y at position 2, insert node X at position 2 and we want to delete node A/0 (figure 2.11). Nodes A, B and C are siblings. Again, updates are applied in reverse document order. The final table after updates contains the node sequence *(B/0, X/1, Y/2, C/3)*. Note that the repeated insertion at position 2, starting with Y and following with X, yields the desired sequence. (b) shows the state of the AUC before execution and includes some additional information. 'Shifts' marks the number

of tuple shifts and equals the size of the deleted or inserted tree respectively. 'Accumulated shifts' simply accumulates the shifts in the direction of document order. The 'first affected tuple' column contains the lowest pre value that is shifted as a consequence of the corresponding atomic update. The value in brackets marks the first affected tuple including accumulated shifts, which is needed to calculate $pre^{new} \rightarrow pre^{old}$. Atomic deletes affect the first pre value on the following axis whereas inserts affect the pre value at their insert location (as the tuple that resides there is shifted backwards).



(a) insertion / deletion in a node sequence

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

(b) corresponding AUC (in document order)

Figure 2.11: Mapping pre values.

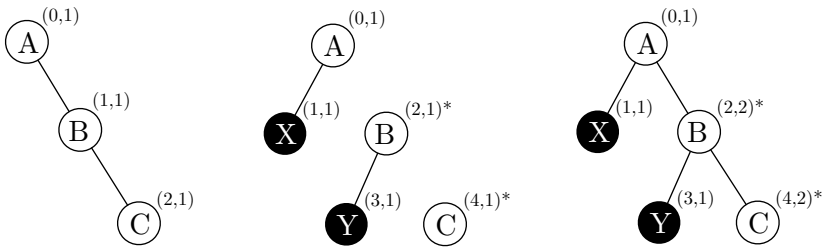
The new pre value or $pre^{old} \rightarrow pre^{new}$ for a node in the unaltered table is derived by identifying the update at the highest index in the AUC that still affects this node. For example:

$A^{old} \rightarrow A^{new}$**:** As the lowest 'first affected tuple' equals 1, there is no mapping for node A. While the result is fine here, it can happen that a node is deleted and the AUC still suggests a mapping. Yet, as we only apply the map to existing nodes this is not a problem.

$B^{old} \rightarrow B^{new}$**:** B has a pre value of 1. The atomic delete is the operation with the highest index that still affects this pre value. Hence, the AUC predicts an accumulated shift of -1, or $1 \rightarrow (1 + (-1))$.

$C^{old} \rightarrow C^{new}$**:** With a pre value of 2, the insertion of Y is the last update that affects node C, it follows $2 \rightarrow (2 + 1)$.

Calculating the original, untouched pre value via $pre^{new} \rightarrow pre^{old}$ for an already shifted tuple works similarly. Eventual tuple shifts have to be taken into account as the updates have already been applied. We are now looking



(a) original state

(b) after inserts w/ invalid distances (*)

(c) after distance adjustments (*)

Figure 2.12: Distance adjustments after node insertion.

for the pre value of the first affected tuple in brackets, as it already includes the number of accumulated shifts.

$B^{new} \rightarrow B^{old}$ With a pre value of 0, the delete operation is the one we're looking for. As the 'accumulated shifts' value states the changes that take place when updates are applied, the information has to be reversed. It follows that $0 \rightarrow 1$ as $0 - (-1) = 1$.

$X^{new} \rightarrow X^{old}$ With a pre value of 1, node X is only affected by the delete, it follows $1 \rightarrow 1 - (-1)$. This case shows that the mapping covers nodes that are inserted within the process, too. Prior to insertion, node X has not been a part of the table, yet it is inserted at position 2.

$Y^{new} \rightarrow Y^{old}$ Node Y is affected by the deletion of node A and the insertion of node X. As the amount of shifts accumulates to 0 it follows that $2 \rightarrow 2$. The explanation is similar to $X^{new} \rightarrow X^{old}$.

$C^{new} \rightarrow C^{old}$ Node C is finally affected by all updates, hence the accumulated shifts are found at the highest index of the AUC which results in $3 \rightarrow (3 - 1)$.

Having gone through the examples we can make the following observation:

Observation 2 *A bulk update serves as a bi-directional mapping between the pre value of a tuple before and after the application of the bulk update.*

2.3.3 Adjusting Distance Values Explicitly

In contrast to the naive approach, where the appropriate distances are adjusted after execution of each atomic update, it has been shown that delaying these adjustments would save a considerable amount of operations and is indeed possible. A bi-directional map between pre values before and after

updates in form of the AUC is the foundation of explicit distance value adjustments. In section 2.3.1 it has been shown, that at some point, all atomic updates have been carried out but the table is in a state that reflects all the original distance values. As shown before, the parent of a node n is calculated in the manner $n.par = n.pre - n.dist$. We cannot find out directly how the number of tuples changed between a node and its parent as we no longer know the parent node. Consequently, we compute the updated distance for any given node of the table via the original distance value and the AUC. A simple example shows the way (figure 2.12).

1. In the first step (b), nodes Y and X are inserted at position 2 and 1, which shifts nodes C and B to the back. After the first step, the distances of B and C are invalid as distance adjustments are delayed.

2. The distances of B and C represent the original state. With the mapping $pre^{new} \rightarrow pre^{old}$, we get the old pre value 2 for C. Together with C's original distance, this also gives us the original parent B/1 of C as $(2-1) = 1$. Using the mapping in the other direction $pre^{old} \rightarrow pre^{new}$ gives us the new parent B/2. It follows that the updated distance of node C is $dist^{new} = (pre^{new} - par^{new})$ or $2 = (4-2)$.

Generally, the distance of any node in the database is adjusted starting with the new pre value pre^{new}:

$$\begin{aligned} pre^{old} &= pre^{new} \rightarrow pre^{old} \\ par^{old} &= pre^{old} - dist^{old} \\ par^{new} &= par^{old} \rightarrow par^{new} \\ dist^{new} &= pre^{new} - par^{new} \end{aligned}$$

Observation 3 *The new distance value of a node can be explicitly calculated based on its original state and a bi-directional mapping of pre values before and after updates.*

2.3.4 Adjusting the Appropriate Distance Values

The last piece missing is to determine the set of distance values that have to be adjusted after the application of atomic updates. This doesn't work without knowledge of the tree structure. As all the 'first' tuples that are affected by a structural update are known thanks to the AUC, the remaining distances can be determined from there via ancestor-or-self- and following-sibling- axis-steps. An additional set keeps track of nodes with already adjusted distances to avoid repetition. As the number of distance adjustments is minimal, the order of visiting the first affected tuples is more or less irrelevant.

Until now, we have used the ancestor-or-self and following-sibling axes to describe the sequence of nodes which distances are affected relative to

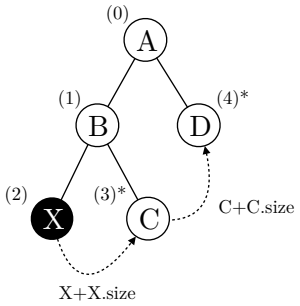
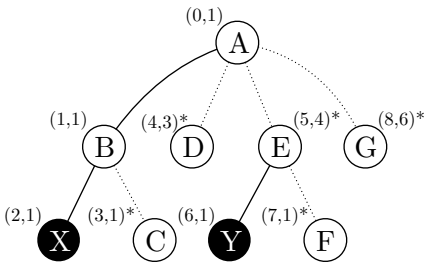
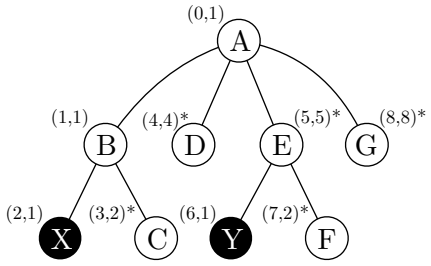


Figure 2.13: Accessing distances to update on-the-fly



(a) invalidation of distances after insertion (marked *)



(b) final table w/ adjusted distances

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

(c) corresponding update cache in document order

Figure 2.14: Calculating the set of invalid distances.

the first affected tuple of an update. In a static environment, this is correct. However, this sequence is determined on-the-fly as distance adjustments are 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. It is consequently no longer necessary to access the parent axis, which should save a few operations (see 2.13). Again, a small example illustrates the complete process in figure 2.14. To adjust distances, the AUC is traversed in document order. A quick look at the 'first affected tuple' column shows that the starting point are nodes 3/C and 7/F. Let S be the set of nodes that have been adjusted already.

3/C This is the first node for which the distance is adjusted (section 2.3.3). From this point on the following node n is calculated directly via

$n.pre+n.size$, which is repeated for the nodes (D, E, G). $G.pre+G.size$ equals the document size and ends the iteration. S now contains the nodes $\{C, D, E, G\}$ as their distances have been adjusted.

7/F Node F is the first node affected by the insertion of Y. Its distance is updated and $S = S \cup \{F\}$. $F.pre + F.size$ yields G as the next candidate. As G is already contained in S and there are no more unprocessed atomics in the list, the adjustment of distances is finished.

Again, we can record another observation:

Observation 4 *Distances that have to be adjusted are determined directly via the table and the corresponding bulk update.*

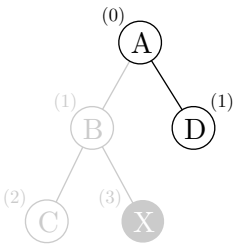
2.3.5 Constraints of the AUC

The AUC describes a sequence of atomic operations and is, due to the pre references, tightly coupled with the corresponding table. For example, for two insert operations at the same pre location, the application order directly affects the document order. It is therefore necessary to define the sequence of updates in a way that their application in reverse document order leads to the desired result. This is kind of a soft constraint and left to the implementation. Also, dealing with sequences of atomics that lead to ambiguous results (like two renames of the same node, etc.) falls into this category.

Tree-Aware Updates
However, there are certain setups of the cache that interfere with the overall concept of efficient distance adjustment. The given scenario (2.15) pictures such a case. Node X is inserted into the subtree of B, hence incrementing B.size to 3. The AUC is consequently no longer valid as it states a shift of -2 for the deletion of node B. This is a general problem of the AUC but can be solved conveniently:

1. Traverse the AUC in document order.
2. For each atomic delete, remove all structural updates that take place in the subtree of the target node (B). As node B is deleted, the changes in the subtree have no effect anyway.

After applying these steps to our example, the cache contains only the delete operation and is again in a valid state. The procedure is called *tree-aware updates* and has been introduced in [Kir10] to save superfluous operations in a slightly different context. 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 in section 2.3.2.



(a) tree with insert, delete

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

(b) update cache

Figure 2.15: Invalidation of shifts due to update sequence.

Order of Insert and Delete
Another setup of the cache can lead to unwanted results due to the application in reverse document order. If we consider an update sequence *{del(p), ins(p, Y)}* and apply insert followed by delete, the inserted node Y would be deleted straightaway. Also a soft constraint as it doesn't break the overall concept, sequences like this should be avoided which is easily realized in preprocessing.

2.4 Implementation of Efficient Structural Bulk Updates

Given a theoretical solution for efficient structural bulk updates, it is relatively simple to follow this with an implementation. An additional layer in BaseX serves as update cache and provides a low-level interface to the underlying table. We give the same name to this dedicated layer as to the theoretical structure. The AUC also makes it possible to realize further optimizations that would have been very hard to implement beforehand. To leverage efficient structural bulk updates, the process of an updating transaction consists of the following steps:

1. Filling the AUC with a sequence of updates.
2. Making sure that AUC constraint are satisfied.
3. Preparation of the cache with *tree-aware updates* and accumulation of shifts etc.

4. Application of the update sequence with delayed distance adjustments.
5. Adjustment of distances.
6. Resolution of text node adjacency and processing of the additional AUC that results from text node deletions.

An actual AUC instance consists foremost of the two separate lists for structural updates (insert, delete) and value updates (rename, ...) sorted by location in document order. Individual atomics have to be added in the proper order which avoids ambiguities and is left to the layers using the AUC. Atomics are implemented as objects and have the following fields:

field name	description
location	actual pre value where the update is applied
fpre	pre value of the first tuple which distance is affected
shifts	number of introduced shifts
accum	number of accumulated shifts over all atomics in document order including **shifts**
insertion	insertion sequence for insert atomics, stored as a separate table instance

Filling of the AUC is very flexible and simple, therefore we won't talk about it in detail. The actual code examples start with the two lists for structural and value updates already being filled. One thing noteworthy is the fact that a few tasks can be carried-out on-the-fly if the cache is filled in document order. Most of the cache preparation (accumulating shifts, tree-aware updates), as well as the implementation of some constraints (order of delete and insert, avoidance of ambiguous renames and replaces) fall into this category. While the real-world implementation makes use of this, it is not reflected in the presented pseudo code as it only saves a handful of linear traversals and can be derived easily from the given solution. There is a number of other problems that are taken care of in the real-world implementation which includes i.e. the treatment of attribute nodes and the realization of a few other types of atomic updates like replace, etc. The code examples form the base of the actual implementation and provide enough insight to derive the remaining details.

ProcessBulkUpdate, algorithm 2.1
With the AUC already being filled, value updates are carried out first. This avoids recalculating the update locations conversely to applying them afterwards. Order is not important, yet in some cases it could be beneficial, with regards to page access, to proceeded strictly in (reverse) document order.

Algorithm 2.1: Processing a Bulk Update

```
1: ProcessBulkUpdate(S: Structural updates, V: Value updates)
2:     for v ∈ V do
3:         ApplyAtomic(v)
4:     end for

5:     TreeAwareUpdates(S)
6:     ApplyStructuralUpdates(S)
7:     AdjustDistances(S)
8:     ResolveTextAdjacency(S)
9: end
```

Structural updates are given in document order depending on the location of the update. As stated before, some preparation of the AUC is necessary before updates can be applied and distances adjusted. Resolving eventual text node adjacency is an additional step which benefits from delayed distance adjustments but must be adapted accordingly.

TreeAwareUpdates, algorithm 2.2
Removing superfluous atomic updates not only reduces the size of the AUC, but is necessary in some cases to re-validate it after all. The list of structural updates is again traversed in document order. Once we come across a delete atomic there is potential for superfluous updates in the subtree of the to-be-deleted node. To identify and get rid of these updates, one has to distinguish between insert and delete atomics. *Case 1* shows that an insert can be removed from the list if its insert location does not exceed the directly following location after the end of the subtree and the new parent of the inserted node resides on the descendant-or-self axis of the to-be-deleted node. A delete operation can only be removed if its location is part of the subtree *(Case 2)*. If neither is the case, the search for the next delete atomic is continued *(Case 3)*.

ApplyStructuralUpdates, algorithm 2.3
To support mapping of old and new pre values, the amount of accumulated shifts for each atomic update has to be determined. This is carried out in document order and can be done on-the-fly if the cache is also filled in document order. After preparation, the actual updates are finally applied in reverse document order by the function *ApplyAtomic()*. Tuples are inserted or deleted and resulting pre value shifts and size updates are taken care off. Distance adjustments are delayed and carried out later by *AdjustDistances()*.

Algorithm 2.2: Preparation of the AUC (*tree-aware updates*)

```
1:  TreeAwareUpdates(S: Structural updates)
2:      i ← 0
3:      while i < S.size do
4:          if s_i is delete atomic then
5:              start = s_i.location
6:              end = start+TABLE.getSize(start)
7:              i ← i + 1
8:              deleting ← TRUE
9:              while i < S.size and deleting do
10:                 l ← s_i.location
11:                 if s_i is insert atomic and l ≤ end and start ≤ TA-
                        BLE.getParent(l) < end then
                                                          ▷ Case 1
12:                     drop s_i
13:                     i ← i + 1
14:                 else if l < end then                  ▷ Case 2
15:                     drop s_i
16:                     i ← i + 1
17:                 else                                  ▷ Case 3
18:                     deleting ← FALSE
19:                 end if
20:             end while
21:         else
22:             i ← i + 1
23:         end if
24:     end while
25: end
```

Algorithm 2.3: Application of structural updates

```
1:  ApplyStructuralUpdates(S: Structural updates)
2:      a ← 0
3:      for i ← 0, S.size do          ▷ calculate accumulated shifts
4:          s_i.accum = a + s_i.shifts
5:          a ← a + s_i.accum
6:      end for
7:      for i ← S.size, 0 do
8:          ApplyAtomic(s_i)
9:      end for
10: end
```

Algorithm 2.4: Adjusting distance values

```
1:  AdjustDistances(S: Structural updates)
2:    updated ← ∅                          ▷ set of already updated pre values
3:    for s ∈ S do
4:      npre ← s.fpre + s.accum
5:      while npre < TABLE.size do
6:        if npre ∉ updated then
7:          odist ← TABLE.getDistance(npre)
8:          opre ← MapPre(S, npre, true)
9:          opar ← opre − odist
10:         npar ← MapPre(S, opar, false)
11:         ndist ← npre − npar
12:         TABLE.setDistance(npre,ndist)
13:         updated ← updated ∪ {npre}
14:         npre ← npre+TABLE.getSize(npre)
15:       end if
16:     end while
17:   end for
18: end
```

AdjustDistances, algorithm 2.4
To avoid touching the same distance values multiple times, the pre values of the corresponding tuples are stored in a set. Adjusting distances now starts with a traversal of the list of structural updates. Each structural update has the field *fpre* that holds the pre value of the first tuple which distance is affected. As updates have been applied already, the number of accumulated shifts has to be taken into account, which gets us the new pre value *npre*. Based on the old distance value of the tuple with pre *npre* and the old pre value, the old parent is calculated. For the starting value of *npre*, the old pre value equals *s.fpre*, hence accessing the mapping could be omitted (which is not reflected in the given code). The old parent and the pre value mapping gives the new distance value for *npre*, which is then propagated to the table via *TABLE.setDistance()*. After adding *npre* to the set of already updated nodes, the algorithm continues with the following node to cover all distances that require adjustment.

MapPre, algorithm 2.5
The pre value mapping works in both directions: a new pre value, after updates have been applied, can be passed as an argument to receive the old pre value without the effects of shifting (*isnew* = *TRUE*). It can also be used to get the new pre value if the argument *isnew* = *FALSE*. As shown before, the pre mapping is based on the list of structural updates and the

Algorithm 2.5: Mapping pre values before and after updates

```
1:  MapPre(S: Structural updates, pre: Integer, isnew: Boolean)
2:      i ← Find(S, pre, isnew)
3:      if isnew then                                  ▷ pre^new → pre^old
4:          val ← s_i.fpre + s_i.accum
5:          i ← i + 1
6:          while i < S.size and s_i.fpre + s_i.accum = val do
7:              i ← i + 1
8:          end while
9:          return pre − s_{i−1}.accum
10:     else                                           ▷ pre^old → pre^new
11:         val ← s_i.fpre
12:         i ← i + 1
13:         while i < S.size and s_i.fpre = val do
14:             i ← i + 1
15:         end while
16:         return pre + s_{i−1}.accum
17:     end if
18: end
```

pre values of the first affected tuple *fpre* of each atomic. The mapping is divided into two steps: a rough binary search that stops once it hits an atomic update where $fpre = pre$ and returns the index of it *(Find())*. If there is no such atomic, it returns the index of the one with the biggest $fpre$ while $fpre < pre$. *Find()* of course takes into account whether it is given a new or old pre value. If given a new pre value, comparison is based on $fpre + accum$ for each atomic. For an old pre value, comparison is based on $fpre$ only. The rough search is followed by fine-tuning the initial result, as it may be possible that a sequence of atomic updates affects the same tuple and therefore has the same value for $fpre$ (imagine i.e. a sequence of inserts at the same location). The atomic to find is the one with the highest index, as this reflects the accumulated changes. It is found by linearly scanning the list. The two cases for mapping directions are distinguished in the same manner as for *Find()* and shown separately for clarity. In both cases, once the appropriate atomic update is found, the pre value is recalculated accordingly by either subtracting or adding the number of accumulated shifts.

ResolveTextAdjacency, algorithm 2.6

Up to now, the discussion evolved solely around the element node type. However, text nodes cannot be left out of the equation, as they require a non-trivial 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

Algorithm 2.6: Resolving text node adjacency

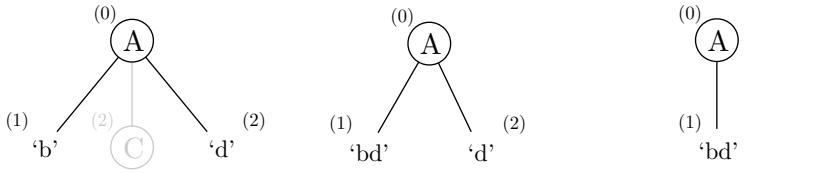
```
1:  ResolveTextAdjacency(S: Structural updates)
2:      deletes ← ∅                                  ▷ cached atomic deletes
3:      smallestVisited ← ∞                          ▷ tracks visited locations
4:      for i ← S.size, 0 do                         ▷ Stage 2
5:          l ← s_i.location + s_i.accum − s_i.shifts
6:          if s_i is insert atomic then
7:              a ← l + s_i.insertion.size − 1
8:              if a < smallestVisited then
9:                  deletes ← concat(MergeTexts(a), deletes)
10:                 smallestVisited ← a
11:             end if
12:         end if
13:         b ← l − 1
14:         if b < smallestVisited then
15:             deletes ← concat(MergeTexts(b), deletes)
16:             smallestVisited ← b
17:         end if
18:     end for

19:     ApplyStructuralUpdates(deletes)              ▷ Stage 3
20:     AdjustDistances(deletes)
21: end
```

Algorithm 2.7: Merging adjacent sibling text nodes

```
1:  MergeTexts(a: Integer)
2:      b ← a + 1
3:      if a or b leave the table boundaries or are no text nodes then
4:          return ∅
5:      end if
6:      TABLE.setText(a, concat(TABLE.getText(a), TABLE.getText(b))
7:      return atomic delete of b
8:  end
```

node is inserted as a sibling of an existing text node. In either case, texts have to be merged as the XQuery Data Model [FMM+07] forbids adjacency. With the implementation of the XQuery Update Facility in BaseX came an algorithm that solved this problem on a higher level. The values of two text nodes are concatenated in one of the two nodes and the other one is deleted. As this operation leads to structural changes, the concept of delayed distance updates can be applied as well. The algorithm to resolve text node adjacency has to be revised as a consequence. Three little examples are given for deletes,



(a) tree after deletion of C (Stage 1)

(b) tree after text concatenation (Stage 2)

(c) tree after deleting superfluous text nodes (Stage 3)

Figure 2.16: Example 1: Merging of text nodes after delete.

inserts, and for a combination of the two, to show how text concatenation is propagated. The algorithm can be broken down into three distinct stages:

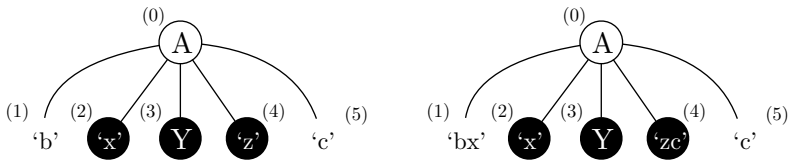
Stage 1 Application of atomic updates and distance adjustments.

Stage 2 Creation of final text nodes by concatenating adjacent texts.

Stage 3 Deletion of the left-over superfluous text nodes of stage 2 and adjustment of distances.

Eventual text node adjacency after deletes is discovered and resolved easily, see figure 2.16. As shown in algorithm 2.6, the update cache is traversed in reverse document order, as this shows us the locations for potential adjacency. A list *deletes* tracks the nodes that have to be removed during *stage 3*. The variable $smallestVisited$ tracks the lowest pre value that has already been tested for adjacency. In the given example, node C is deleted from the tree, the update cache consequently holds a single atomic s with *s.location = 2, s.fpre = 3, s.shifts = -1, s.accum = -1*. First the location l, where adjacency can occur, is directly derived from s, in this case $l \leftarrow 2 - 1 + 1 = 2$. As there is only one potential location for text adjacency directly before the deleted node, the function *MergeTexts()* has to be called once for position $b \leftarrow l - 1$. This function merges the node at the given position and the first following sibling if possible, by directly concatenating their texts in the correct order (*setText()*). It returns a delete atomic for the second node which is added to the first position of the *deletes* list to be executed later during *stage 3*. As the AUC holds one atomic, *stage 2* is already finished. Deleting the superfluous text node with text content *'d'* is carried out by the already presented function *ApplyStructuralUpdates()*. Distances are adjusted as usual. Some pre-processing steps for distance adjustment can be skipped. I.e. applying *TreeAwareUpdate()* isn't necessary as text nodes don't have any descendants.

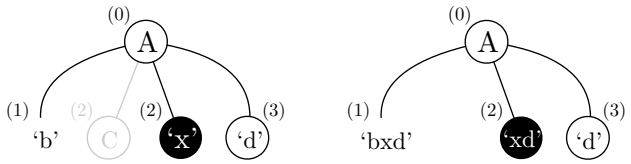
Figure 2.17 shows an example where a sequence of three nodes is inserted with a single atomic insert. Consequently, the AUC holds the atomic insert



(a) tree after insertion of the node sequence ('x',Y,'z'). (Stage 1)

(b) tree after text concatenation (Stage 2)

Figure 2.17: Example 2: Merging of text nodes after insert. Stage 3 not displayed.



(a) tree after ins(3,'x') and del(2). (Stage 1)

(b) tree after text concatenation (Stage 2)

Figure 2.18: Example 3: Merging of text nodes after combined delete and insert. Stage 3 not displayed.

s with *s.location = 2, s.fpre = 2, s.shifts = 3, s.accum = 3*. The size of the insertion sequence equals 3 and consists of a text node followed by an element, followed by a text node. Insertion sequences are always 'clean', which means no adjacent text nodes are to be expected as these would have been merged beforehand. The example focuses on the special case where two text node merges are necessary due to a single insert operation. Algorithm 2.6 covers this case by checking the position at the end of the insertion sequence for adjacency. In the given case, $l \leftarrow 2$, which leads to $a \leftarrow l + 3 - 1 = 4$. Calling *MergeTexts(4)* takes care of the first case of adjacency. It is important to check locations strictly in reverse document order to avoid incorrect concatenation and wrong order of the resulting delete atomics. The second check at location $l - 1 = 1$ follows and leads to another concatenation and delete atomic. The resulting and temporary AUC now holds the two atomics *(del(2), del(5))*. *Stage 3* is not explained here as it strictly follows the first example.

Example 3 (figure 2.18) shows, through a combination of insert and delete, how the algorithm propagates text concatenation to achieve the desired result. After stage 2, *deletes* contains the update sequence *(del(2), del(3))* and is processed accordingly.

2.5 Related Optimizations

2.5.1 Replaces

Up to now, we singularly talked about insert and delete atomics. Being a conglomerate of delete and insert, replaces are arguably not an 'atomic' type itself. Yet, implementation-wise they help to realize a few important optimizations. To explain them properly, it is important to know about some storage internals of BaseX. We cover the basics here and leave details to Grün's thesis [Gr0].

Logical Pages
In contrast to being stored in a contiguous file, the table in BaseX is divided into logical pages. A page currently holds up to 256 tuples. To keep track of the document, a main memory directory remembers the location and sequence of pages, the first pre value on each page and the number of tuples in a page. Free space is only allowed after all tuples on a page, hence no gaps between tuples or at the start of a page. In short, 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. Another reason is, that it allows for some basic buffering mechanisms, where a page is i.e. completely loaded and then altered in main memory before being flushed to disk.

Text Value Storage
To enable a fixed length encoding of the different node types in BaseX, 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. Currently, there is no structure that keeps track of empty space, hence an increase in size and fragmentation is the consequence if values are frequently removed, added or re-inserted.

Basic & Rapid Replace
A basic replace operation r is carried out in BaseX as follows:

- The node at $r.location$ is deleted, which is consequently followed by a shift of all tuples with a pre value greater or equal $r.fpre$.
- The replacing sequence is inserted at $r.location$, which again leads to a shift of the following tuples.

While this approach may look inefficient on paper, it is considered 'safe', as it reduces the complexity of a replace considerably if the source or target

table contain namespaces. Yet, the basic replace obviously leads to a number of problems:

- An excessive amount of I/O in general, as the same tuples are shifted twice. Costs are albeit bounded due to paging strategies.
- A potentially unnecessary increase in size and fragmentation of the files on disk that contain text and attribute values. In a real-world application, it might very well be the case that the replacing subtree differs very little from the deleted one and a simple and cheap value update might be enough to resolve this.
- Wearing out the space of possible IDs for future nodes due to the same reasons as above.

In addition to the naive *basic replace*, two more advanced approaches have been realized to fight fragmentation and a waste of I/O. The first is the *lazy replace*, which tries to substitute the structural replace operation with cheap value updates. This comes in handy if the replaced and replacing subtrees feature the same structure in general. The second we call *rapid replace*, which directly overrides entries in the table (see the section on future work in [Kir10] for the basic algorithm). 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.

Lazy Replace
The lazy replace compares the to-be-deleted node with the replacing insertion sequence. If they are topologically identical, a sequence of value updates suffices. The implementation is very straightforward as it simply requires a sequential and pair-wise comparison of the tuples in the source and destination table (algorithm 2.8), which shows the relevant snippet in *Apply-Atomic()*. The final protocol for applying replace atomics is now divided into three stages:

1. If there are no namespaces to be dealt with, it is first checked whether a *lazy replace* would be sufficient. In this case, value updates are collected and applied at the end (*Case 1*). They cannot be applied on-the-fly as the process might still be aborted.
2. In case the *lazy replace* fails because of necessary structural changes, a *rapid replace* is leveraged (*Case 2*).
3. If namespaces are contained in either the source or target table, a *basic replace* is applied by deletion of the target node and subsequent insertion at the same location (*Case 3*).

Algorithm 2.8: Application of atomic update

```
1:  ApplyAtomic(s: atomic update)
2:      ...                                  ▷ eventually process other atomic types

3:      if s is a replace atomic then
4:          if there are no namespaces in TABLE and s.insertion then
5:              insSize ← s.insertion.size
6:              if insSize ≠ TABLE.getSize(s.location) then
7:                  failed ← TRUE
8:              else
9:                  failed ← FALSE
10:             end if
11:             valueUpdates ← ∅
12:             i ← 0
13:             while i < insSize and not failed do
14:                 if tuples insertion.t_i, TABLE.t_i differ in either kind, dis-
                        tance or size then
15:                     failed ← TRUE
16:                 else if tuples insertion.t_i, TABLE.t_i only differ in value
                        then
17:                     add appropriate value update to valueUpdates
18:                 end if
19:                 i ← i + 1
20:             end while
21:             for v ∈ valueUpdates do                              ▷ Case 1
22:                 ApplyAtomic(v)
23:             end for

24:             if failed then                                       ▷ Case 2
25:                 run rapid replace ...
26:             end if

27:         else                                                     ▷ Case 3
28:             run basic replace ...
29:         end if
30:     end if
31: end
```

While the *lazy replace* looks expensive on paper, as a possibly huge subtree is first traversed and then potentially replaced anyway, performance tests showed that the actual worst-case overhead remains small. In case the *lazy replace* fails, a *rapid replace* makes up for it easily as it is naturally faster

than the basic approach. In the end, it is well worth the effort to reduce both fragmentation and file size on disk. A direct comparison is found in chapter 4.

2.5.2 Merging Atomic Updates

Reducing the number of structural atomics naturally reduces the complexity of updates. While this is more or less obvious, there is a number of advantages that don't stand out as clear. For example, replace optimizations are leveraged more often with an increasing number of replace atomics, which can be achieved by merging neighbouring inserts and deletes. Merging neighbouring inserts into one operation could additionally reduce I/O due to buffer strategies on page level. Going into details about buffer strategies is beyond the scope of this work. Therefore we limit this section to the important facts about merging of structural atomics. Two atomic updates a and b can be merged if they fulfill certain conditions:

- a and b are directly adjacent regarding their location. Merging i.e. an insert at location x with an insert at location $x + 1$ does not work as the inserted sequences are not adjacent.
- a and b work under the same parent node.
- a and b adhere to the order constraints of the AUC.

At the moment, there are two distinct cases where an insert and delete operation are substituted with a replace. The two atomics are given in document order with regards to the location.

(ins(x), del(x)) → rep(x)
For an insert i at location x and a delete d at the same location in the given order, a substituting replace is set-up as follows:

Replace: i.location, d.fpre, i.shifts+d.shifts, d.accum, i.insertion

The field *accum* is directly derived from the delete atomic as it already contains the correct value.

(del(x), ins(x+1)) → rep(x)
For a delete d at location x and an insert i at $x + 1$, a substituting replace is set-up as follows:

Replace: d.location, d.fpre, i.shifts+d.shifts, ins.accum, i.insertion

Again the field *accum* is directly derived from the insert, as it already contains the shifts of the delete. As stated before, merging atomics is part of

update cache preparation and can be carried out on the fly. Consequently, it does not add any considerable overhead.

A number of potential merges exist that are covered, but not yet implemented. The reasons are strictly implementation dependent and evolve around the merging of insertion sequences. An insert sequence for an insert or replace atomic in BaseX is stored in a temporary table. This table serves as a container for all insertion sequences that are currently in the update cache, yet sequences are unordered. While this is an efficient concept in general, it complicates the merging of insertion sequences. Reordering the sequences explicitly is out of the question, so the introduction of a mapping, to restore the order might be worthwhile to look into. It remains to find out whether the gains are worth the effort. If efforts are made in this direction, there are three potential merges to be implemented:

(ins(x), ins(x)) → ins(x)
For two inserts i_1 at x and i_2 at x substitution is set-up as follows:

Insert: i_1*.location,* i_1*.fpre,* i_1*.shifts+*i_2*.shifts,* i_2*.accum, concat(*i_1*.insertion,* i_2*.insertion)*

(rep(x), ins(x+1)) → rep(x)
For a replace r at x and an insert i at $x+1$ substitution is set-up as follows:

Replace: r*.location,* r*.fpre,* r*.shifts+*i*.shifts,* i*.accum, concat(*r*.insertion,* i*.insertion)*

(ins(x), rep(x)) → rep(x)
For an insert i at x and a replace r at x substitution is set-up as follows:

Replace: r*.location,* r*.fpre,* r*.shifts+*i*.shifts,* r*.accum, concat(*i*.insertion,* r*.insertion)*

Insertion sequences 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 somehow.

2.6 Future Work

2.6.1 Delayed Distance Updates Based On ID-PRE Mapping

Interestingly, the whole concept of efficient bulk updates focuses entirely on pre values and their tedious re-calculation, as they are used as identifiers but ultimately underly changes during the updating process. So why not base it all on a fixed identifier instead? Each node or tuple in BaseX features

indeed an id, that is both unique and static. One reason to work without it is that there has been no efficient mapping between id and pre values, which changed some time ago with the introduction of a structure tailored to the current table encoding by Popov [Pop12]. This structure tracks the changes of pre values after updates in an efficient manner. The following section sketches a solution for efficient bulk updates based on Popov's *ID-PRE* map and subsequently shows why the already presented solution is still preferable.

Algorithm 2.9: Delayed Distance Updates based on Id-Pre Mapping

```
1:  ProcessBulkUpdateIdPre(S: Structural updates)
2:      toAdjust ← ∅                          ▷ tracks tuples (parentID, childID)
3:      for s ∈ S do
4:          pID ← getParentId(s.fpre)
5:          if s is an atomic insert then                          ▷ Case 'insert'
6:              toAdjust ← toAdjust ∪ (pID, calcFutureId(S,s))
7:          end if
8:          for f ← s.fpre, TABLE.size - 1  do          ▷ iterate following nodes
9:              toAdjust ← toAdjust ∪ (getParentId(f), getId(f))
10:             f ← f + getSize(f)
11:         end for
12:     end for

13:     ApplyStructuralUpdates(S)
14:     for t ∈ toAdjust do
15:         cpre ← getPre(t.childID)                              ▷ ID-PRE map
16:         cdist ← cpre - getPre(t.parentID)                     ▷ ID-PRE map
17:         setDistance(cpre, cdist)
18:     end for

19:     (take care of text node adjacency ...)
20: end
```

ProcessBulkUpdateIdPre, algorithm 2.9
There are some differences between the 'standard' version and delayed distance updates based on the *ID-PRE* map. Algorithm 2.9 sketches the general approach. Value updates are left out of the discussion as they can simply be applied beforehand without taking special care. To leverage the mapping, the basic idea is to store pairs of child and parent ids *(childID,parentID)* prior to the application of updates for each child node that needs to be adjusted. By mapping the ids to pre values and calculating the difference we can afterwards adjust distances directly. Pairs are stored in the structure *toAdjust*.

Implementing it as a map with the childID as a key avoids duplicate entries in *O(1)*. Distances that need to be updated are again determined based on the cache of structural atomic updates. For each of these atomic updates, the id of the first affected tuple and the parent of this tuple is to be found. Calculating the parent id is trivial.

However, in case of an insert operation, an extra step is required to find the value for the child id. As, during an insert, a new tuple is added to the table that could eventually be shifted later on by updates further up the table, it has to be tracked. As its id is not yet readily available, it has to be pre-calculated based on the AUC. More specifically, by traversing the AUC in reverse document order and assigning 'future' ids to all insertion sequences, the actual future id can be determined. This is left to the function *calcFutureId()*. After this step, a new tuple *(childID,parentID)* is added to the record. At this point, the given algorithm spares two details: If a sequence of nodes is inserted, an id tuple has to be added for each of these nodes. Secondly, it is possible to determine whether tracking of an inserted node is needed after all by searching the AUC for eventual changes in the interval $[n.par + 1, n.pre]$. Both tasks can be carried out efficiently.

After taking care of the special situation regarding inserts, the remaining ids that are influenced by the update have to be collected. This is again realized by iteratively calculating the following node and adding the appropriate id tuple to the list. The application of structural updates is then carried out the same way as the actually implemented solution.

Adjusting the actual distances then becomes trivial. The tuples collected in *toAdjust* are traversed and the distance of the node with id childID is updated by calculating the distance of the child and parent pre values. Leveraging the *ID-PRE* map leaves us with the appropriate pre values. Resolving eventual text node adjacency is carried out accordingly.

At the moment, the main disadvantages of delayed distance adjustments based on the *ID-PRE* map are:

- Instead of one integer as reference for each distance that has to be updated, a tuple of parent and child ids has to be determined in advance. This increases the main memory complexity by a constant factor and is therefore negligible.

- In addition, a tuple has to be stored for each node that is inserted during a query, as its distance could be invalidated by structural changes. Naively, the number of additional tuples equals the number of inserted nodes, but this can eventually be reduced by scanning the AUC for structural changes in the interval between the insert location and its parent.

- A future id has to be calculated for each inserted node. Adding this

information to the AUC can be realized by a single linear traversal in reverse document order during preparation.

- Lookup complexity of the *ID-PRE* map is relative to all updates ever applied to a database, which can be a considerable amount. In contrast, mapping old and new pre values based on the AUC yields a complexity relative to the current set of updates. However, this drawback might prove to be not as dramatic, as ids in the table could be reseted periodically to optimize the *ID-PRE* map. Of course, this depends on the size of the database and usage patterns.

It is obvious that, at the current state of affairs, a solution based on the *ID-PRE* map is not preferable. Depending on the future development of BaseX this might be subject to change. In this case, the discussed solution can serve as a stepping stone towards a complete implementation.

2.6.2 Memory Consumption of Distance Tracking

Part of the algorithm to adjust the distances is a set that tracks already updated nodes to avoid repetition. For a use case like the one shown in figure 2.8, this set can almost grow to the size of the database, which is, of course, undesirable. As each entry is realized with an integer, the real-life costs remain manageable ($\approx$ 380MB for $100 * 10^6$ distance updates), especially if one takes into account that pre values are also represented by integers. Yet, it would be nice to find a more compact structure in the future. A solution needs to bring at least some knowledge about the tree structure and it remains to find out whether the time needed to update this knowledge justifies the savings of memory.

2.6.3 Speed-up of $Pre^{old} \leftrightarrow Pre^{new}$

Currently, the mapping between pre values before and after updates is divided into two steps. First, a logarithmic search tries to find the atomic update with the appropriate $fpre$ field. As there could be several of them, a linear search scans the following positions in the cache to find the atomic with the highest index. The pre mapping is leveraged twice for each distance adjustment. Depending on the transaction and document structure, the number of calls could be enormous and a reduction in processing time might pay off. Typical tests with XMark documents showed that the mapping consumes up to 10% of the total processing time. Although the gains are by no means substantial, there are a few simple tweaks adding little overhead.

As the list of atomic updates is ordered and minimum and maximum values for $fpre$ over all atomics are readily available, switching binary search for an interpolation search could already shave of some time. Adding an

additional field to each atomic update that already holds the index of the atomic with the same $fpre$ field at the highest position, ultimately renders the linear search superfluous. Assigning the appropriate indexes is easily realized during pre-processing by traversing the AUC in reverse document order. This would especially speed up situations where a large number of atomics affect the same following node.

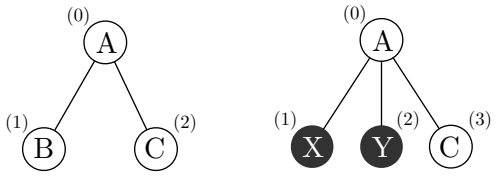
2.6.4 Caching Insertion Sequences

During an updating transaction, main memory is not only consumed by the basic structures of update caching, but especially insertion sequences can consume huge amounts of memory depending on the use case. On one hand, a single insertion can already exceed the main memory limits if the inserted tree is big enough. On the other hand, inserting an identical sequence of nodes into a large number of targets wastes space as well, as each individual instance is cached separately. If the main memory limit is exceeded, the only solution is to split the single transaction into a series of smaller ones. As this is very tedious or potentially impossible, there are better ways to solve this by shifting the effort from the user to the implementation, or to simplify it at least. A first step would be to cache insertion sequences on disk instead of main memory. Secondly, replicating identical insertion sequences should be avoided altogether. Of course, moving caching entirely to disk is as unfeasible as comparing the complete set of insertion sequences for equality. Providing a dedicated interface, either via XQuery or on a lower level, could be a worthwhile compromise and enables the user to activate it on demand.

3 Leveraging Efficient Bulk Updates with the XQuery Update Facility

The XQuery Update Facility [CDF+09] is a part of BaseX since version 6.0 has been released in 2010. Continuous feedback of the user community encouraged us to speed up transactions with the implementation of efficient structural bulk updates. Although efficient bulk updates can be leveraged from all over BaseX, XQuery Update, as the main interface to manipulate XML data, benefits the most. While the implementation is relatively straightforward and realized by the lean AUC layer, it cannot be used directly by the Update Facility. This is due to the AUC order constraint, where atomic updates are applied in reverse document order (section 2.2.5), restrictions by the specification and the fact that the order of statements in the actual query is exchangeable. A basic example visualises the main problem to be solved in order to leverage efficient structural bulk updates with XQuery Update. Given the following query and input document in figure 3.1, how do we fill the AUC in a way that the final result looks as expected?

delete node /A/B, insert node <Y/> after /A/B,
insert node <X/> as first into /A



(a) tree before snapshot

(b) expected result after snapshot

Figure 3.1: XQuery Update example.

3.1 From Update Primitives to Atomic Updates

A more detailed introduction to the XQUF and the concepts of the pending update list (*PUL*) and its primitives is given in [Kir10]. Using XQuery Update statements within a query, each individual statement leads to one or several update primitives. The PUL caches these primitives to apply them all in a bulk operation at the end of the query - or snapshot. Among others, this serves several purposes:

- Changes introduced during a snapshot are only visible in succeeding queries.
- Update primitives can be applied in a specific order defined by the Update Facility to resolve ambiguities.
- Insertion sequences (for insert, replace, etc.) are cached to avoid dirty reads.

Before a PUL can be made effective, the update primitives have to be added as atomics to the AUC. As learned before, the order of structural updates within the AUC is of utmost importance.

3.1.1 Update Primitives and Target Nodes

Each update primitive is aimed at a target node. The target of a delete primitive is the node which is deleted in the end. Remember the *location* field of the atomic updates in the AUC. While the target pre value for a delete primitive equals the location pre value, there are several primitives where this is not the case. Figure 3.2 shows the different types of primitives implemented in BaseX, together with their ranks and the calculation of update locations depending on the target node. Detailed explanations of *location* and *rank* are found below.

Location
The *location* value equals the *target* only for primitives with a *rank* smaller than 7. For these, updates are applied directly to the target node. For primitives with a *rank* greater 6, the location must be re-calculated as it is only 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 3 primitives add insertion sequences directly at the following position of the target node.

Rank
The *rank* is directly derived from the *type* of the primitive and the *location*

primitive	rank	location
insert before	1	target
delete	2	target
replace	3	target
rename	4	target
replace value	5	target
insert attribute	6	target
insert into as first	7	target + targetAttSize
insert into	8	target + targetSize
insert into as last	9	target + targetSize
insert after	10	target + targetSize

Figure 3.2: Order or update primitives and calculation of location relative to target.

of the update. If we want to update a target t, the higher the location value of a primitive is, the higher its rank. If we apply an *insert before* and an *insert after* to the same target, the *insert after* must be applied first due to the the application order of the AUC. All in all, *ranks* are assigned in a way that the result of a query is always consistent with the requirements of the specification. For primitives with a *rank* greater 7 this is especially important as they all access the same *location*, yet the final order of the insertion sequences is of great importance. For primitives with a *rank* smaller than 7 the ranks are chosen accordingly.

3.1.2 Order of Update Primitives

Having determined the rank and location of update primitives, we can finally order them in way that not only satisfies the constraints of the AUC, but also yields the desired result. To impose an order to a list of objects, one only has to be able to compare them pair-wise. To determine the greater one of two given primitives (the one that has to be applied first), comparison is based on *location, target* and *rank*. Unfortunately, it is not enough to solely base comparison on target, location or rank alone. Algorithm 3.1 shows the implementation of a comparator for update primitives based on the Java Comparator [1] interface.

Calling *Compare(a,b)*, the function compares the second argument b to the first argument a. If a is greater it returns 1, if a is smaller it returns -1, if no clear decision can be made, the return value is 0.

[1] http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html, *01.10.2013*

Algorithm 3.1: Comparing two Update Primitives

```
1:  COMPARE(a: Update Primitive, b: Update Primitive)
2:    determine a.location, b.location..

3:    if a.location > b.location then                          ▷ Case 1
4:      return 1
5:    end if
6:    if b.location > a.location then
7:      return -1
8:    end if

9:    if a.location > b.target and a.target < b.target then    ▷ Case 2
10:     return 1
11:   end if
12:   if b.location > a.target and b.target < a.target then
13:     return -1
14:   end if

15:   if a.target > b.target then                              ▷ Case 3
16:     return 1
17:   end if
18:   if b.target > a.target then
19:     return -1
20:   end if

21:   if a.rank > b.rank then                                  ▷ Case 4
22:     return 1
23:   end if
24:   if b.rank > a.rank then
25:     return -1
26:   end if
27: end
```

First, the *location* fields of the two given update primitives have to be determined, as this is the most important property for ordering. Locations are calculated based on type and target as shown in figure 3.2. Afterwards, there are four distinct cases that take care of all possible combinations of primitives:

- *Case 1* makes sure that primitives, that operate at different locations of the table, are ordered correctly.
- *Case 2* orders primitives correctly that operate at the same table location, with one of them operating within the subtree of the other.

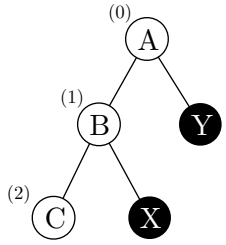
- *Case 3* orders primitives at the same location but with different target nodes.
- *Case 4* finally orders primitives of a different type that operate both on the same target node.

Case 1
If location values of a and b already differ, 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 and 'wins' the comparison.

Case 2
If location values are equal, it is determined whether one of the two update primitives 'happens' in the subtree of the other. An example with two *insert into* statements visualises the case:

insert node <Y/> into /A,
insert node <X/> into /A/B

(a) query



(b) tree

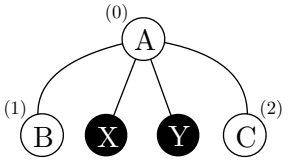
Figure 3.3: Primitive ordering: Case 2.

Let's consider primitive a inserts X into B and primitive b inserts Y into A. Both primitives want to add the new nodes at the same location, hence $a.location = b.location = 3$. Consequently, *Case 1* is skipped. For each insertion sequence, to end up at the right location under the appropriate parent, primitive b must be applied first. In the given case, the update of primitive a takes place in the subtree of $b.target$ as it meets the conditions $b.location > a.target$ and $b.target < a.target$. Node Y has to end up after X and is therefore inserted first, which is achieved by the ordering $(b,\ a)$. The example evolves around statements of type *insert into*, yet the same applies to the types *insert into as last* and *insert after*.

Case 3
The target value can potentially settle the comparison if *Case 1* and *Case 2* do not apply, which is the case for the next example:

insert node <X/> after /A/B,
insert node <Y/> before /A/C

(a) query



(b) tree

Figure 3.4: Primitive ordering: Case 3.

Primitive a inserts X after B and b inserts Y before C. (a, b) represents the final order as $b.target > a.target$ b.

Case 4
If finally even the targets of both primitives are the same, a final decision is made based on the rank. For example, this is the case if the two given primitives operate on the same target node with one of them being a *rename* and the other one a *delete* primitive. To not violate the specification, the delete is applied after the rename which is directly reflected in the assignment of ranks.

Comparator Properties
There are three important properties to guarantee for a valid comparator. The function $sgn()$ is an abbreviation of the mathematical *signum* function and returns 1 if the argument is greater 0, 0 if the argument is 0 and -1 if the argument is smaller 0. The three properties are as follows:

$$sgn(compare(x, y)) = -sgn(compare(y, x)) \tag{3.1}$$

$$compare(x, y) \wedge compare(y, z) \implies compare(x, z) \tag{3.2}$$

$$compare(x, y) = 0 \implies sgn(compare(x, z)) = sgn(compare(y, z)), \forall z \tag{3.3}$$

We will not provide any proofs, yet we are sure that all the constraints are met. If we switch arguments when calling the compare function we get the opposite result, hence an identical ordering. The second property, transitivity, is fulfilled as it is directly reflected in the hierarchical nature of the decision process. The last property is only met in theory, as there will never be a case where two identical primitives are compared. Comparisons based on location, target and rank yield a clear result without exception, as the XQuery Update Facility module already sorts out the issue of duplicates on a higher level.

4 Performance of Efficient Bulk Updates

Up to now, chapter 2 introduced us to the general performance drawback of structural bulk updates. It has been shown how the problem can be solved in theory and in practice by implementation of the AUC. We then talked about the necessary adjustments to the XQuery Update Facility module in chapter 3 to finally leverage efficient structural bulk updates. Looking at the theoretical part of this work it is already clear that explicit distance adjustment might reduce processing time by magnitude. However, as new time and memory overhead occurs naturally, it is important to find out how much of the potential advantage is conserved after all.

This chapter is divided into four parts. The first part shows the general setup of performance tests. The second part compares a version of BaseX without efficient structural bulk updates with an up-to-date version. A third part takes a closer look at the perfomance of the three replace strategies basic, lazy & rapid and their overhead. We conclude with a discussion on the memory consumption of our optimization.

4.1 Setup

All performance tests are based on the XMark XML benchmark project [SWK+02]. Using XMark documents helps to get an idea for performance in real-world scenarios and is therefore chosen over a synthetic test of the individual low-level layers. Although queries have already been defined and are widely used, we define our own, as XQuery Update is not covered by the test suite. Yet, the documents are created with the XMark document generator. Scaling factors, sizes, number of nodes in the table, number of date elements and the size of the people element subtree are shown in figure 4.1. Commodity hardware used for testing is shown in figure 4.2.

In general, the bulk-query and replace scenarios are run in the same way. Each combination of query and document is tested several times, until no significant decline of processing time can be noted. The fastest recorded processing time wins. A fresh, thus defragmented database is created for each

factor	size	nodes	date elements	people size
0.01	1 MB	$3.3 * 10^4$	$1.0 * 10^3$	$5.2 * 10^3$
0.10	11 MB	$3.2 * 10^5$	$9.2 * 10^3$	$5.0 * 10^4$
1.00	116 MB	$3.2 * 10^6$	$9.0 * 10^4$	$5.1 * 10^5$
10.00	1167 MB	$3.2 * 10^7$	$9.0 * 10^5$	$5.1 * 10^6$
100.00	11670 MB	$3.2 * 10^8$	$9.0 * 10^6$	$5.1 * 10^7$

Figure 4.1: XMark test documents statistics.

cpu	memory	disk	os	java
Core i3 3.2GHz, 64bit	8GB	999 GB	OS X 10.8.4	Oracle v7u12

Figure 4.2: Hardware used for testing (2010 Apple iMac).

query. The number of executed runs depends on the size of the document. For the smaller documents (1MB, 11MB) this equals 20 runs, for the 116MB document 10 runs and for the two biggest documents 5 and 3 runs. Although the answering time for the two tested versions of BaseX varies greatly, the number of runs remains the same to ensure equity.

We test the corresponding BaseX standalone versions and increase the heap size to 6GB (*-Xmx6G*). For the 11.7GB document the text and attribute indexes of BaseX are deactivated to not exceed the memory limits. No options are changed apart from this.

4.2 Bulk Queries

We use the *<date>* element as the target for bulk queries, as it is well spread over the complete XMark document and can be found at the beginning, in the middle and at the end. It consists of the element node itself and contains a single text node as a child. Three queries compare the performance between BaseX 7.3 and BaseX 7.7, from now on referred to as v73 and v77. The individual queries and results are listed and discussed in figures 4.3, 4.4 and the following paragraph. Value updates are not shown in the diagram as we focus on the efficiency of structural bulk updates.

Q1: Value updates
for $d in //date/text() return
replace value of node $d with 99.99.9999

Although value updates are not part of the optimization, the comparison shows at least that there is no significant difference between the two tested versions. Overhead for v77 with efficient bulk updates seems to be kept at

update	1MB	11MB	116MB	1.1GB	11.7GB
Q1: value	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: deletes	80 ms	7577 ms	810.58 s	-	-
	10 ms	82 ms	0.96 s	22.4 s	1803 s
Q3: inserts	133 ms	9644 ms	1019.10 s	-	-
	17 ms	143 ms	1.96 s	145.9 s	17735 s

Figure 4.3: Bulk update processing times of BaseX 7.3 (top) and BaseX 7.7 (bottom).

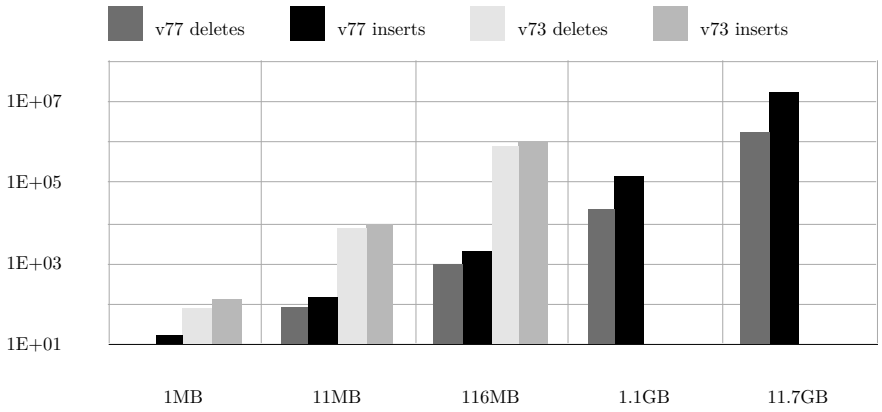


Figure 4.4: Bulk updates processing time trends, in milliseconds for BaseX 7.3 and 7.7.

bay. Throughout the documents 1MB to 116MB, the processing time scales more or less linearly without noticeable difference between v73 and v77. Yet, for the last two bigger documents there is a sudden super-linear step-up in processing time. While the document sizes increase only by a factor of 10, the actual processing time grows by a factor between 50 and 100. As stated in section 2.5, actual text values are not directly stored in the table but referenced via an offset. Alternating between the location of the currently accessed page of the table on disk and the end of the file that holds text values might add up to finally explain this increase in processing time.

Q2: Deletes
delete node //date

Taking a look at figure 4.4, the advantage of v77 is clearly visible. As expected, we can remark a reduction of processing time by magnitude, throughout all queries and document sizes. For documents 1MB to 116MB, the pro-

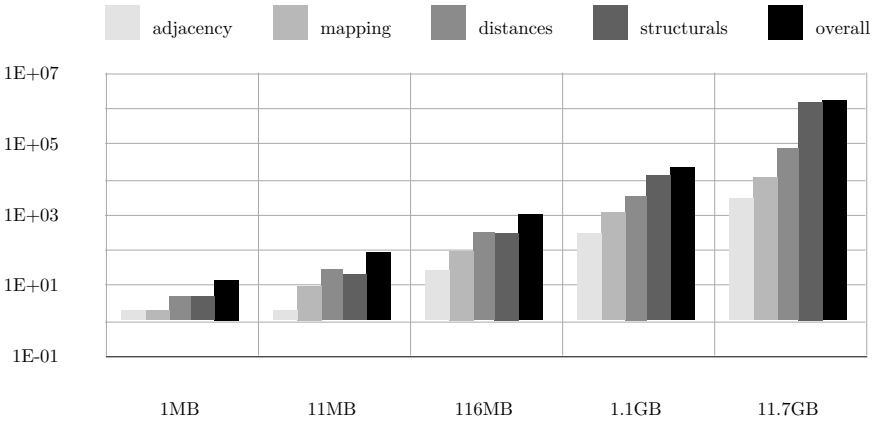


Figure 4.5: Processing time trends for different parts of the delete bulk update (*Q2*, in milliseconds).

cessing time grows linearly for v77 with the document size, whereas for v73 the processing time 100-folds with each step-up. The trend explains well why we aborted the tests for the two biggest documents with v73, as they take at least several days to complete. V77 still performs very admirably for the bigger documents, deleting around $1.8 * 10^6$ nodes in only 22.4 seconds and about $18 * 10^6$ nodes in 30 minutes, respectively. Yet, as seen with the value bulk updates, we can also remark super-linear and even quadratical increase for the last step-up in document size. To gain more insight, we disassembled the complete processing time of *Q2* into several key-parts (fig. 4.5).

- *Overall* This describes the complete processing time of *Q2*. The *overall* value is always higher than the sum of the four other parts, as a few processes, i.e. query parsing, are not investigated separately.
- *Structurals* Accumulates the time for node deletion, updating of size values and shifting of following tuples on disk.
- *Distances* Times the complete process of distance adjustment. This includes the calculation of the appropriate set of nodes, calculation of the new distances and writing distances to disk. It also includes the time for pre value mapping (see *Mapping*).
- *Mapping* Accumulates time that is spent to map old pre values to new ones and v.v. Mapping is a pure main-memory exercise.
- *Adjacency* Accumulates the time for all text node adjacency-related operations, which includes checks and the actual resolution.

We can see that the time spent for pre value mapping and resolution of text node adjacency increase linearly with the document size. Also, the trend for

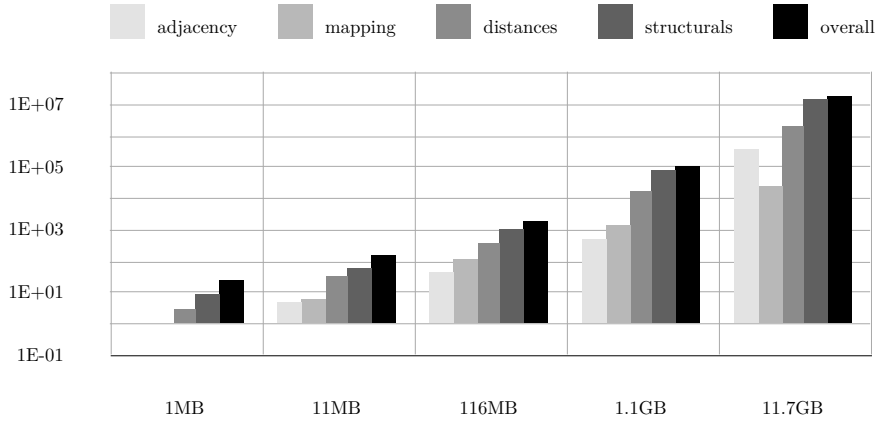


Figure 4.6: Processing time trends for different parts of the insert bulk update (*Q3*, in milliseconds).

distance adjustment does not explain the sudden deterioration, as it nicely follows a linear pattern, exhibiting a slight dent only during the last step-up. Yet, we also see that deleting the actual nodes on disk gets increasingly expensive and, towards the end, eats up almost the complete processing time. With expensive distance adjustments now eliminated from the equation, the weakest part of a structural bulk update is the actual deletion of nodes on disk and the required tuple shifts. Remembering the rules for logical paging in BaseX, *Q2* only leaves some blank space at the end of the pages where nodes are deleted. Consequently, the database does not get any smaller with the deletion of nodes. Kircher [Kir10] provides a possible explanation for the performance drop in his thesis. To sum it up, several instances of the deleted *date* elements reside on the same logical page. As each of them is deleted individually, the same pages are repeatedly read-from and written-to disk. Buffer strategies might catch some of the overhead, but clearly not all of it. Experiments showed that doubling the page size already pays off. Investing more time into finding a good balance between page size and overall performance might proof worthwhile.

Q3: Inserts
for $d in //date return
insert node <ndate>99.99.9999</ndate> after $d

The overall trend follows more or less what we have seen with the delete bulk update in *Q2*. V77 leaves v73 behind starting already at the smallest document size of 1MB. Up to 116MB, the processing time increases linearly with the document size. In contrast to v77 and *Q2* we can already observe a severe performance drop, stepping-up to document 1.1GB. Yet, inserting

about $1.8 * 10^6$ nodes at approximately $9 * 10^5$ locations in less than two and a half minutes remains impressive. We also dismantled processing time for *Q3* in figure 4.6. As observed with *Q2* already, the insertion of nodes is by far the most expensive part. While the pre value mapping holds up well, each task that is I/O related grows a lot faster. For the last two step-ups, even distance updates slow down quadratically. The cause for this is easily found. In contrast to the bulk delete, nodes are added to the table. Upon database creation, BaseX fills logical pages to the rim to keep database size down. 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:

- A lot of partially filled pages are created, as there is no redistribution of existing tuples between existing pages.
- 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.

The sum of these issues greatly increases the number of I/O operations and gets increasingly severe with growing document size. Adding a text node contributes in the same manner, as the inserted text has to be added to the appropriate file. Again, investing more time in paging strategies and the avoidance of fragmentation could really pay off. We talk a little bit more about the implications of the document order on disk access strategies in chapter 5.

4.3 Basic, Lazy & Rapid Replace

A few different approaches to replace operations have been introduced in section 2.5.1. The *rapid replace* clearly targets a reduction of processing time, whereas the *lazy replace* is capable of that as well, but mainly tries to reduce fragmentation. This comes at a cost, namely when the replaced and replacing tree are topologically different. We now try to shed some light on the dynamics taking a closer look at a few distinct setups including the worst-case. Figure 4.7 summarises the absolute processing times, whereas figure 4.8 tries to put the different performances into context, with the *basic replace* at the centre. Furthermore, there are two scenarios:

- *Even* The replacing sequence has the exact same size or node count as the replaced target. Theoretically, following tuples do not have to be shifted. We also use XQuery and the following expression for the comparison:

  replace node //people with //people

	1MB	11MB	116MB	1.1GB	11.7GB
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 (uneven)	17 ms	121 ms	1302 ms	14.62 s	267.9 s
rapid (uneven)	13 ms	105 ms	1103 ms	11.87 s	169.9 s

Figure 4.7: Performance of different replace types and scenarios in BaseX 7.7.

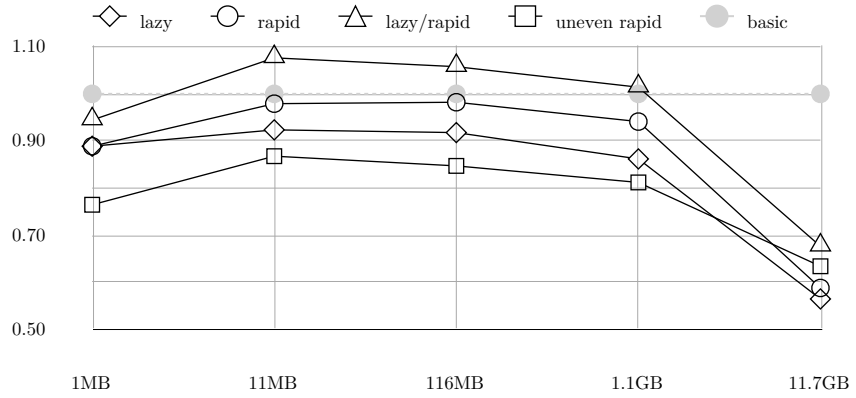


Figure 4.8: Relative comparison of different replace scenarios.

The size of the *people* subtree for different documents is shown in figure 4.1.

- *Uneven* We replace the target node *people* with the *europe* subtree, which is about half the size:

  replace node //people with //europe

  As the two trees do not feature the same structure we cannot apply a *lazy replace*. Yet, *uneven* comparison gives a better feel for the superiority of the *rapid replace*.

To get comparable results, the individual replace approaches are explicitly activated in-code. A modified version of BaseX 7.7 is tested that includes the not yet officially released *lazy replace*.

Basic

The *basic replace* serves as a starting point for the comparison. The target

node is first deleted, followed by the insertion of the replacing tree at the same location. For the two biggest documents, we witness a super-linear drop of performance. Yet, replacing around $50 * 10^6$ nodes in under 6 minutes is impressive. Explaining the deterioration, the same rules apply as for bulk inserts and deletes. In figure 4.8, the *basic replace* serves as the relative centre of comparison for all the other replace techniques.

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 (approx. 10%) than the *basic replace*, the difference grows with the document size, peaking at around 43% time savings for the 11.7GB document. But what we really gain here is a reduced rate of fragmentation.

Rapid
The *rapid replace* generally follows the results of the *lazy replace*. Whereas the difference at the beginning is even smaller ($< 5\%$), it clearly outperforms the naive approach for document 11.7GB 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 interesting part. In this specific setup, we provoked the worst-case by making sure the *lazy replace* sequentially scans the complete replaced tree before aborting, due to structural differences with the replacing tree. Hence, the *rapid replace* steps in. For documents 11MB to 1.1GB we can indeed see that this combination is outperformed, even by the *basic* approach. However, residue stays well below 10% most of the time. For the 11.7GB document, the advantage of the *basic replace* is already far gone ($> 30\%$). In case the *lazy replace* fails while causing considerable overhead, a *rapid replace* can almost make up for the time lost and even outperforms the naive approach in some cases.

Rapid (uneven)
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, as an initial comparison of tree sizes already terminates the scan. We can also see, that such a scenario puts the *rapid replace* further ahead. This once again comes down to logical paging. Whereas the *basic* approach first deletes some tuples

and afterwards re-inserts them at the physical end of the table, the optimized replace overwrites tuples in-place, thus avoids switching locations on disk repeatedly.

4.4 Memory Consumption of Efficient Bulk Updates

Naturally, one might ask the question how much memory overhead we add with the method of efficient structural bulk updates. After all, we add quite some information in form of the AUC to the already memory-snatching implementation of the PUL.

First, we think the test results speak for themselves. During the bulk insert tests, we added up to $18 * 10^6$ nodes to the database. All these nodes are cached in main memory, in addition to the information of the PUL and AUC. Yet, the 6GB of memory, assigned to the Java virtual machine, have not been exceeded. The same applies to the replace scenario. Assigning 2GB of memory is already enough to deal with the second largest document.

Indeed, stepping up from version 7.3 to 7.7 of BaseX, there is some gross increase in memory consumption - but only if we limit our investigation to the AUC. A great deal of space requirements could be shaved off by optimizing the container that holds insertion sequences (which is beyond the scope of this work). Of course, this only applies to the cases with actual inserts or replaces. For all the others, we can safely say that the AUC enables us to execute queries of a former unthinkable magnitude on, by todays standards, already out-dated commodity hardware. If memory is still sparse, sections 2.6.2 and 2.6.4 could provide some relief.

5 Related Work

In the course of this work, we developed a method to reduce the immense costs of distance adjustments for structural bulk updates on the *Pre/Dist/Size* encoding. While this is a necessary step for us, the work of others focusses on the development of an encoding scheme that is easily updatable in the first place. The goal is to find a numbering scheme that requires little re-labeling of existing nodes. Of course, both approaches have their advantages and disadvantages that we now try to put into perspective. Besides updatability, locking-capability is another factor that has to be taken into account and is tightly coupled with the encoding. Erat provides a deeper look into the matter of fine granular mapping in XML databases in his bachelor thesis [Era13].

Regarding BaseX, reducing evaluation time of reading transactions, as well as lowering memory footprint and disk space requirements have always been the top priorities. As a consequence, the *Pre/Dist/Size* mapping embodies these qualities to the full extent. Representing each node by a fixed-length tuple not only speeds-up processing, but also lowers complexity and memory consumption in general:

- As tuples are stored contiguously and require the same space, each one of them can be accessed directly via the pre value offset. An additional index to locate tuples is rendered superfluous.
- Document structure and number of levels have no influence on the length of *Pre/Dist/Size* triples. Consequently, there is no need to compress labels or to apply heuristics based on the nature of a document.
- XPath axes are efficiently evaluated by basic algorithms, as discussed in section 2.2.2.

Nothing good comes without a price tag, hence it was decided to compromise on updatability. With such a mapping, structural changes of the tree are expensive. While costs for size updates and pre value shifts remain manageable, costs for distance adjustments do not.

Another family of numbering schemes is able to handle structural changes naturally. ORDPATH [OOP+04], as the most popular representative, uses

prefix-labeling to model relationships within the tree. The children of the node with label *1* are labeled *1.1, 1.3, etc.*. Relying on odd numbers exclusively, new nodes can be inserted between existing ones by the use of even numbers. Even numbers serve as a caret and do not increase the depth of a node. Consequently, there is no need to re-label existing nodes. DeweyIDs [HHMW05] refine this concept, regarding the implementation. Fine-grained locking is discussed as well. Yet, as updatability increases, several problem fields are created in exchange, as the length of a label is no longer fixed:

- Accessing tuples on disk requires an additional index.
- For documents with a bigger depth or in case of frequent insertions, labels get bulky. Different solutions can be found in the literature. In case of DeweyIDs, the authors utilise Huffman codes to reduce memory consumption directly. In [AS09], Alkhatib and Scholl propose a method called *CXDLS* that exploits the redundant and verbose nature of XML. By separating the tree structure from the content, topologically equivalent subtrees are not labeled repeatedly. Assigning ORDPATH labels to the compressed tree structure reduces space requirements and increases query performance.

A number of NXDs exist that make use of dynamic labeling schemes. Sedna[1] is a highly regarded representative. Taking a look at its documentation reveals that the internal representation is based on a Dewey encoding. The same holds true for eXist-db[2]. Both of them use B^+ trees to index nodes on disk. Comparing BaseX to either one of them with regards to update performance, it is hard to stay unbiased. Neither Sedna nor exist-db cache bulk updates and are consequently not able to offer XQUF support. Updates are applied immediately, no caching takes place. On the one hand, this may lead to side-effects, on the other, no resources are occupied with caching. For a bulk delete of approximately 1.8 million nodes involving the 1.1GB XMark document, BaseX 7.7 performed about twice as fast, compared to Sedna. While this is not enough to derive a final conclusion, it shows that BaseX and Sedna are at least competitive.

In the end, the document order is a real delimiter of performance. In contrast to relational databases, where sets and not sequences are managed, XML databases store content in a way that the result can be efficiently assembled. Insertion consequently leads to a kind of fragmentation, where nodes, added within the tree, are physically appended after all other pages on disk which affects query performance in multiple areas. MonetDB is another NXD based on a similar encoding as BaseX. In [BMR05], the authors limit this effect by leaving empty space on logical pages upon the creation of

[1] http://www.sedna.org, *Native XML Database System, Institute for System Programming, Russian Academy of Sciences*

[2] http://exist-db.org,

a database. As a heuristic approach, the benefits correlate strongly with each individual use case. In [TBS02], Tatarinov, Viglas, Beyer et al. already came to the conclusion, that the document order indeed summons a number of new challenges if ordered XML is to be managed with the unordered relational model. The authors divide XML numbering schemes into three sub-categories:

- *Global Order.* For example, the Pre value stores the absolute location of a node. Upon insertion and deletion, this requires a large amount of re-labeling.
- *Local Order.* Only stores the position of a node relative to its siblings. On one hand, this effectively reduces the amount of re-labeling. On the other, evaluating location steps gets more expensive and is carried out in a recursive manner.
- *Dewey Order.* Forms the middle ground by combining the absolute and relative approach. The ORDPATH encoding is a direct offspring of *Dewey Numbers*.

While Tatarinov et al. state that only the *Dewey Order* performs reasonable for queries *and* updates, we arrive at a different conclusion.

Implementing support for the XQUF on top of *Pre/Dist/Size* is mostly a burden. Yet, to improve the updatability of our global encoding scheme, it helps us to make a virtue of necessity. As the pending update list relies on copious caching anyway, adding little information already solves the problem for us. As the bulk insert and delete tests showed, the prevailing costs now shift to the actual handling of tuples on disk - a problem that persists regardless of the encoding. We conclude for now that if updates are induced with XQuery Update, choosing a numbering scheme like ORDPATH is not naturally the best choice, as it simply shifts overhead to another level. Adding the fact that the XQUF is by far the most important interface to induce document changes finally places the *Pre/Dist/Size* encoding among the very best, with regards to overall performance.

6 Conclusion

In chapter 2 we developed a general feel for the costs of structural bulk updates. We followed this with a concept to carry out previously expensive distance adjustments efficiently. The implementation of the *atomic update cache (AUC)* adds reasonable overhead to the obligatory *pending update list (PUL)*. On top of that, we could add optimizations regarding replaces and merging of atomic updates. We also provided meaningful direction for future undertakings. Some of them evolve around improvement of the AUC itself by speeding up the mapping of pre values or reducing memory consumption. Enabling the user to decide how insertion sequences are dealt with could further reduce memory consumption of update transactions where necessary.

It has never been easy to force the XQuery Update Facility into cooperation with the *Pre/Dist/Size* mapping. Once again, some adaptions had to be made to leverage efficient structural bulk updates. A non-trivial mapping between pending update list and AUC was introduced in chapter 3, that translates a set of update primitives into a ready-to-apply sequence of atomic updates.

Chapter 4 took a closer look at the dynamics of different bulk update scenarios. A direct comparison of BaseX 7.3 and a modified version of BaseX 7.7, featuring the discussed optimizations, highlighted the superiority of the new approach. We also pointed out how structural updates could be further accelerated by adjusting paging strategies. The *lazy replace* is a worthwhile addition to the toolbox and puts a lid on fragmentation. Producing considerable overhead in the worst case of failure, it could still be shown that a *rapid replace* makes up for the time lost. A general discussion on memory consumption of the AUC completes this chapter.

Finding an encoding scheme that performs outstanding, for both reading and writing queries - unlike others, we did not tackle this problem on a theoretical level (chapter 5). Instead, we built upon the already successful *Pre/Dist/Size* mapping in BaseX and greatly extended its updating capabilities with an additional layer. The final implementation is fully consistent with the XQuery Update Facility specification and has already proven its efficiency and reliability in production use.

List of Figures

List of Algorithms

Acknowledgements

First of all, I want to thank Prof. Dr. Marc H. Scholl and Prof. Dr. Marcel Waldvogel for being my referees.

I am especially thankful to Dr. Christian Grün and Dr. Alexander Holupirek who encouraged me to continue my studies, and also Marc Scholl who readily included me into the group, providing me with the basis to do so. Even in this special case where a reliable solution seemed far out, Christian happily let me dig around in the dicey internals of BaseX. He has been a constant advisor throughout my studies, which is greatly appreciated on my part.

Bibliography

[AS09] Ramez Alkhatib and Marc H. Scholl. *Compacting XML Structures Using a Dynamic Labeling Scheme.* In *Proceedings of the 26th British National Conference on Databases: Dataspace: The Final Frontier*, BNCOD 26, pages 158–170, Berlin, Heidelberg, 2009. Springer-Verlag.

[BMR05] Peter A. Boncz, Stefan Manegold, and Jan Rittinger. *Updating the Pre/Post Plane in MonetDB/XQuery.* In *XIME-P*, 2005.

[CD99] James Clark and Steve DeRose. *XML Path Language (XPath) Version 1.0, W3C Recommendation.* http://www.w3.org/TR/1999/REC-xpath-19991116, November 1999.

[CDF+09] Don Chamberlin, Michael Dyck, Daniela Florescu, Jim Melton, Jonathan Robie, and Jérôme Siméon. *XQuery Update Facility 1.0.* http://www.w3.org/TR/xquery-update-10, Jun 2009.

[Era13] Jens Erat. *Fine Granular Locking in XML Databases.* Bachelor thesis, University of Konstanz, 2013.

[FMM+07] Mary Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. *XQuery 1.0 and XPath 2.0 Data Model (XDM).* http://www.w3.org/TR/xpath-datamodel, Jan 2007.

[Gr0] Christian Grün. *Storing and Querying Large XML Instances.* PhD thesis, University of Konstanz, Sep 2010.

[HHMW05] Michael P. Haustein, Theo Härder, Christian Mathis, and Markus Wagner. *DeweyIDs - The Key to Fine-Grained Management of XML Documents.* In *IN PROC. 20TH BRASILIAN SYMPOSIUM ON DATABASES*, 2005.

[Kir10] Lukas Kircher. *BaseX: Extending a Native XML Database with XQuery Update.* Bachelor thesis, University of Konstanz, 2010.

[OOP+04] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. *ORDPATHs: Insert-Friendly XML Node Labels.* In *SIGMOD*, pages 903–908, 2004.

[Pop12] Dimitar Popov. *Advanced Storage Structures for Native XML Databases.* Master thesis, University of Konstanz, 2012.

[SWK+02] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. *XMark: A Benchmark for XML Data Management.* In *VLDB*, 2002.

[TBS02] Igor Tatarinov, Kevin Beyer, and Jayavel Shanmugasundaram. *Storing and querying ordered xml using a relational database system.* In *In SIGMOD*, pages 204–215, 2002.