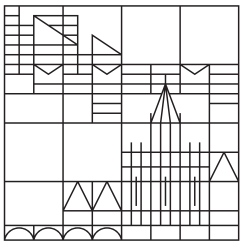
Advanced Storage Structures for Native XML Databases

Dimitar Popov

Master Thesis in fulfillment of the requirements for the degree of Master of Science (M.Sc.)

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

Universität Konstanz



Reviewers:
Prof. Dr. Marc H. Scholl
Prof. Dr. Marcel Waldvogel

Abstract

The XQuery Update Facility has turned native XML databases from static document-oriented database systems to general purpose transactional systems. Consequently, data structures allowing both fast read and write operations have become a necessity. This thesis presents several such structures and focuses on the challenges that emerge by their application in a native XML database. A new algorithm is described, which serves to determine the position of an XML node using its unique identifier. Its formal description is further used as the basis of a proof of the algorithm's correctness. Another central topic is the efficient storage and modification of records with variable length. Their handling in the context of XML databases is discussed in detail as well as optimizations, which improve the performance. Furthermore, mechanisms for updating different kinds of index structures are proposed along with maintaining index statistics.

Zusammenfassung

Native XML Datenbanksysteme sind, dank XQuery Update Facility, nicht mehr statische dokumentorientierte Systeme, sondern universale Transaktionssysteme. Diese Tatsache erfordert den Einsatz von Speicherstrukturen, die außer schnelle Leseoperationen auch schnelle Schreib-Operationen ermöglichen. Diese Arbeit stellt solche Strukturen vor. Ein neuer Mechanismus für die Bestimmung der Position eines XML Knotens mittels seines Identifikators wird dargestellt, gefolgt von einer formalen Beschreibung, die als Basis für einen Beweis der Korrektheit des Algorithmus dient. Ein weiteres Thema ist die effiziente Speicherung und Modifizierung von Datenbankeinträgen mit variabler Länge. Die Behandlung von solchen Einträgen wird im Kontext von XML-Datenbanken eingehend discutiert sowie verschiedene Optimierungen. Schließlich werden Mechanismen zur Aktualisierung von verschiedenen Index-Typen vorgeschlagen und auch Index-Statistiken, die in nativen XML-Datenbanken vorkommen.

Acknowledgments

First, I would like to thank Prof. Dr. Marc H. Scholl and Prof. Dr. Marcel Waldvogel for being referees of this thesis.

I am also sincerely grateful to Dr. Christian Grün for being my advisor and for the valuable guidance and discussions, without which this thesis would not be possible.

I am obliged to my colleagues and friends from the university and from the BaseX team, in particular to Leo Wörteler, Michael Seiferle, and Rositsa Shadura. Thank you all!

Contents

1 Introduction

1.1 Motivation

XML is gaining popularity as a flexible storage format and so are native XML databases. Native XML databases offer significantly better performance in comparison to the so called XML-enabled databases. However, some features which are considered typical and standard for a relational database system, are missing from many of the native XML database implementations. One of the reasons is that while relational databases have been developed since almost 40 years, XML databases are gaining popularity only recently. Another important factor is that implementing an XML database is more complex, because of the greater flexibility which XML offers compared to relational tables and the additional constraints such as document order of nodes. These different requirements are the reason why, although many of the ideas and principles from the RDBMS world can be borrowed, their application is not always straightforward, in some cases a completely new solution needs to be devised and in others, a feature is not needed and can be left out. The main point of the current work is efficient implementation of storage mechanisms of an XML database. At this low level of the architecture of a database system, there are much more similarities between an RDBMS and a native XML database than on higher levels. However, the proposed ideas are specifically tailored for the use in an XML database in order to achieve maximal performance, and at the same time they provide similar features as their counterparts from the relational database world.

1.2 Contribution

The main contribution of the current work is that it provides the necessary algorithms and techniques for implementing efficiently updatable data structures in XML databases. More specifically, this goal is achieved by a new algorithm for mapping id and pre values of XML nodes, the development of an optimized mechanism for storing variable length records, as well as concrete approaches for updating various index types and statistics. The provided sketch of a formal proof of the correctness of the ID-PRE Map is another important result, which guarantees that proper implementations will not return wrong results. The practical value of the work consists in that many of the described approaches are already implemented in the native XML database system BaseX and it is relatively straightforward to implement the rest of them. Thus, BaseX has become a more efficient and flexible system.

1.3 Overview

This thesis is divided into three main sections. The first section defines an algorithm for mapping stable node identifiers to their document-order position (also known as pre values). The second section describes an optimized approach for storing variable-length records. The last section deals with different problems, which occur during index updates and statistics in native XML databases.

2 ID-PRE Mapping

2.1 Introduction

The document order is an important requirement for an XML database – it defines the order in which result nodes should be returned when a query is processed. The order can be also defined by performing depth-first search in the tree represented by an XML document starting from the root of the tree. This traversal is also called pre order traversal. Hence pre values are called the sequentially incremented values generated for each visited node. Obviously, each pre value identifies a node uniquely. However, pre values are not stable node identifiers with respect to update operations. For example, if we consider an insertion of a new node within an XML document, then all nodes with pre values greater than the pre value of the insert position will be assigned new pre values. This may not be a big problem, if the pre values are not explicitly stored but instead the order in which the nodes are stored is in document order. However, stable node identifiers are needed by index structures. Index structures are generally used to speed up queries. The queries themselves need to return the results into document order. This means that there should be mechanism which efficiently finds the pre value of a node with a given stable identifier. One trivial solution would be to store all pairs in a hash map. That guarantees retrieval in constant time but the required memory will grow linearly with the number of nodes. In the following sections another mechanism will be proposed as well as a proof-of-concept implementation, which should show the viability of the proposal. We will start now with formalizing the problem and its goals.

2.2 Preliminaries

Some important notes will be listed in this section. First, it should be clear that the set of id values and the set of pre values have the same cardinality, because they uniquely identify document nodes. Second, in general the problem of mapping elements from one set to another can only be solved by explicitly storing each mapping pair. However, this encoding is not very efficient when the number of elements grows. Therefore, we will make some assumptions about the id and pre values, which allow a more efficient encoding in terms of occupied memory.

Assumption 1. id and pre values are non-negative integers.

Non-negative integers are used in many applications as identifiers, since it is natural for people to work with them.

Assumption 2. id values cannot be reused.

This assumption is also rather natural. It means that when a node with a given id value is deleted, the id value cannot be re-used for newly inserted

nodes. These two assumptions reflect the essence of the sequence objects from the relational world, which generate sequential integer numbers.

Assumption 3. moving of nodes is not required.

Moving of nodes means that one sub-tree of a document is moved to a different position and id values of the nodes of the sub-tree are preserved. This feature has been deliberately left out in XQuery Update Facility 1.0. However, future work may prove that the current proposal can also easily handle moving of nodes.

2.3 Goals

The following notations will be used from now on: N is the set of document nodes P is the set of pre values I is the set of id values.

The goal is to provide a data structure with a function $pre : I \to P$, such that for $\forall p \in P$, $pre(i) = p$ iff i and p identify the same node $n \in N$. Additionally, the structure should require sub-linear memory and all of its functions should work in sub-linear time, i.e. it should be better than the trivial solution of storing all pairs in a hash map.

Similarly to the hash map the data structure needs to be notified about inserted and deleted records. According to assumption 3, it is not required that id values can explicitly change their pre values and all other XQuery Update Facility functions can be expressed with inserting and deleting of nodes. Therefore, the update of the data structure will be performed with only two functions – insert and delete – and we will require that they need sub- linear time.

2.4 Intuitive Description

The idea of the method will be informally described in this section. First, we will consider some particular cases in order to acquire an overview of the problem.

It should be noted that according to 1, when a database is created, the sets of pre and id values are equal. This means that one node has a pre value which is equal to the corresponding id value. In this case, it is not needed to maintain any kind of mapping and finding the pre value of an id value is trivial.

Given a newly created database, let us consider inserting new nodes. If the new nodes are appended at the end of the database, then the pre values of the nodes will be the same as their corresponding newly generated id values. Figure 1 illustrates this.

Obviously, finding the pre value of an id value is again trivial in this case.

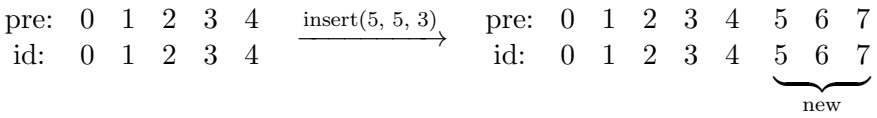


Figure 1: Appending new records to a new database: insert 3 records, the first of which has id 5, at position 5.

What happens if the insertion point is not after the last existing node? In this case all nodes after the insertion point will have pre values incremented with the number of inserted nodes.

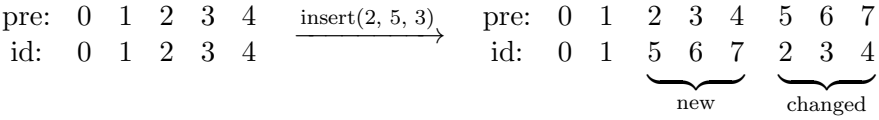


Figure 2: Inserting new records to a new database: insert 3 records, the first of which has id 5, at position 2.

So how can we find the pre value of a given id value after insertion? Obviously, we can split the set of nodes into three intervals:

- when the id value is in the interval $[0, 1]$, then the pre value is the same
- when the id value is in the interval $[2, 4]$, then the pre value is the id value plus 3
- when the id value is in the interval $[5, 7]$, then the pre value is 2 plus offset from the beginning of the interval; e.g. $pre(7) = 2 + 7 - 5 = 4$.

Therefore, we can calculate the pre value of any node given its id value, if we store the following data:

- last id value before starting updates of the database *baseid* (in our case 4)
- the insert-position *pre* (in our case 2)
- the original id value *oid* of the node with pre value *pre* (in our case 2)
- the number of newly inserted nodes *inc* (in our case 3), and
- the id value of the first new node fid (in our case 5).

Having this data we can find the pre value of an id value using the following simple definition:

- all id values which are less than *oid* will have the same pre value
- all id values in $[oid, baseid]$ pre value equal to the id value plus *inc*
- an id value *id* greater than *baseid* will have pre value $pre + id - fid$.

Analogous characteristics has the delete operation. When deleting nodes from the end of the database, then the mapping is trivial. When deleting nodes from the middle, then the nodes after the last deleted node will have pre values decremented with the number of deleted nodes.

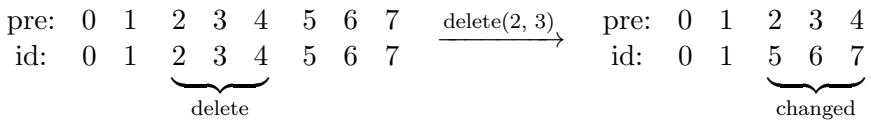


Figure 3: Deleting records from a new database: delete 3 records from position 2.

Again we can split the set of nodes into three parts:

- when the id value is in the interval $[0, 1]$, then the pre value is the same
- when the id value is in the interval $[2, 4]$, then there is no corresponding pre value; this case, however can be ignored, since id values cannot be reused, and we don't require recognizing of non-existing id values.
- when the id value is in the interval $[5, 7]$, then the pre value is 2 plus offset from the beginning of the interval.

If we try to generalize this example, we will see that we have a definition similar to the one in the insert case. The values are however different in the example: the *baseid* is 7 and *inc* is -3.

- all id values which are less than *oid* will have the same pre value
- all id values in $[oid, baseid]$ will have a pre value equal to the id value plus *inc*.

These simple examples show that it might be possible to calculate the pre values, if we store the *baseid* and the following information for each update function: (pre, oid, fid, inc). Therefore, the data structure which we will use will be, in fact, a simple list of (pre, oid, fid, inc) tuples. Given that the function for finding the pre value always relies on finding the correct oid

interval, it would be beneficial to keep the list sorted by *oid*, since we could apply binary search[1].

In addition to the above mentioned values, we also need to know how long an inserted interval is. This especially important when we try to find in which interval an id bigger than *baseid* (i.e. generated by an insert) is located. In this case the *inc* value cannot help us to determine the length of the interval as shown by the next example.

Let us consider a newly created database with 5 nodes and two update operations: insert of 3 new nodes with first id 5 at pre 4 and delete of 2 nodes with id values 1 and 2. The next figure shows the result of the two operations.

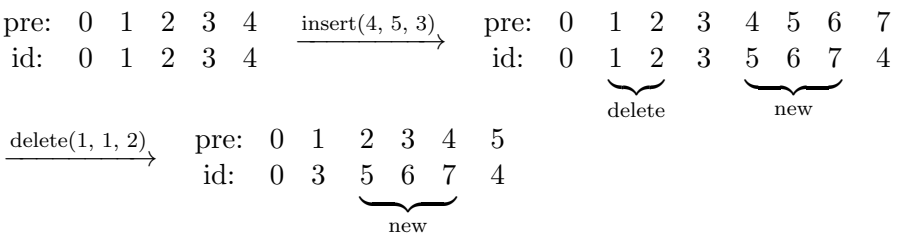


Figure 4: Inserting and deleting records from a new database.

After the delete operation the *inc* value of the interval from pre 2 to pre 4 is 1, which obviously is not its length.

These examples are very simple. In real a case insert and delete operations can occur in random order, at random positions, and affecting random number of subsequent nodes. In the following section we will describe all possible cases, together with the corresponding handling.

2.5 Formal Description

In this section we will define more precisely and formally the data structure and the operations, which can be executed. We will introduce the following functions:

- $pre(id) : ID \rightarrow PRE$
- $insert(pre, id, c) : PRE \times ID \times Z \rightarrow M$
- $delete(pre, id, c) : PRE \times ID \times Z \rightarrow M$

As explained in the previous section, the data structure will be a list of vectors and a natural number value $baseid \in \mathbb{N}$. Each vector represents

[1] Of course, sorted list may not be the optimal data structure here. For example, a tree structure can be used, but this is not part of the current work. The important point is that we can guarantee $\mathrm{O}(log(n))$ when searching for the corresponding interval.

an interval of nodes modified by an update operation. Each vector has the following five components: $(pre, oid, fid, lid, inc)$, where

- pre - pre value
- oid - original id value
- fid - first id value
- lid - last id value

The list is sorted by the first component pre.

Data Structure

In this section we will define the data structure of the proposed mechanism. First, we will introduce some notations.

Definition 1. $PRE \subset \mathbb{N}_0$ is the set of all pre values.

Definition 2. $ID \subset \mathbb{N}_0$ is the set of all id values.

Definition 3. $baseid \in ID$ is the id value of the last node in a database in which no update operation has been performed.

Definition 4. $OID \subset ID$ is the set of all id values less than or equal to $baseid$.

Definition 5 (ID-PRE Map)**.** An ID-PRE Map M is a $baseid$-value and a set L of tuples $(pre, oid, fid, lid, inc)$, where

- $pre \in PRE$,
- $oid \in OID$ we will refer to these as "original id values",
- $fid, lid \in ID$ and $baseid < fid \leq lid$, and
- $inc \in \mathbb{Z}$.

Although this definition is formally correct, it does not show the semantics of each value. We will briefly explain what is the meaning of each component in order to ease the understanding of the definition of the operations in the following sections. We assume the nodes are ordered by pre values, i.e. in document order.

- $baseid$ is the greatest id value in the database before any update operation has been performed.
- each tuple of the set L represents an update operation

- *pre* is the pre value of the first node inserted or deleted by the corresponding update operation; if subsequent update operations affect nodes with pre values less than *pre*, then the value is adjusted depending on the number of inserted or removed nodes.
- *oid* is the id value of the next node, which has not been added by an insert operation, i.e. which existed in the database before any updates have been performed.
- fid is the id value of the first node inserted or deleted by the corresponding update operation.
- *lid* is the id value of the last node inserted or deleted by the corresponding update operation.
- *inc* is the difference between the pre and id values of all following nodes, which has not been added by an insert operation, with pre value between *pre* of the current tuple and *pre* of the next one; similar to *pre*, if subsequent update operations affect nodes with pre value less than *pre*, then the value is adjusted depending on the number of inserted or deleted nodes.

Note: the adjustments of *pre* and *inc* are not necessary. However, they increase the performance, because they are performed once per update operation and not on each calculation of a pre value.

Operations: pre()

Definition 6. Given an ID-PRE map M with *baseid* and a set of tuples L we define the function $pre : ID \rightarrow PRE$ as follows:

$$pre(id) = \begin{cases} id & \text{if } L = \emptyset \text{ or } id < min\{oid\}; \\ pre_i + id - fid_i & \text{where } t_i = (pre_i, oid_i, fid_i, lid_i, inc_i) \in L; \\ & \text{such that } id \in [fid_i, lid_i], \\ & \text{if } id > baseid, \\ id + inc_i & \text{where } t_i = (pre_i, oid_i, fid_i, lid_i, inc_i) \\ & t_i \in \{t_1, ..., t_n\} \subseteq L, \text{ such that} \\ & pre_i = max\{pre_1, ..., pre_n\} \text{ and} \\ & oid_1 = ... = oid_n \leq id \text{ and} \\ & \nexists t_k \in L, \text{ such that } oid_i < oid_k \leq id, \\ & \text{if } id \leq baseid. \end{cases}$$

The definition can be interpreted as follows:

- if there are no update operations or if the node is not affected by update operations, then its pre value equals its id value.

- if $id > baseid$, then the id value has been inserted after the initial creation of the database. Therefore, there should be a tuple t_i in L, which reflects the insert operation, i.e. $id \in [fid_i, lid_i]$. Since the inserted nodes are at position pre_i, then the pre value of id should be $pre_i + id - fid_i$.
  If there is no such tuple t_i, this means that the node with the given id value has been deleted, and there is no pre value.

- if $id \leq baseid$, then it is an id value of a node which existed before any update operation has been executed. In this case we need to find all update operations which affect the id value. This is why we search the original id values oid of all tuples for the required id value id. As a result we either find a tuple t_i, such that $oid_i = id$, or we take the tuple with the greatest oid less than id. However, since there might be more than one tuple with the same oid, we take the tuple t_i with the maximal pre value. The reason for this is that the tuple will have accumulated in its inc value the effect of all update operations which affect the node with id.

Operations: insert()

First we will define a helper function $increment()$ as follows:

Definition 7. Let $t = (pre, oid, fid, lid, inc) \in L$ and $c \in \mathbb{N}_1$. Then

$$increment(t, c) = (pre + c, oid, fid, lid, inc + c)$$

We will now extend the function to act upon a whole set of tuples.

Definition 8. Given a set of tuples L, we define $increment(L, pre, c)$, as follows:

$$increment(L, pre, c) = (L \setminus L_{\geq}) \cup \{increment(t_i, c) | t_i \in L_{\geq}\}$$

where

$$L_{\geq} = \{t_i | t_i = (pre_i, oid_i, fid_i, lid_i, inc_i) \in L \text{ and } pre_i \geq pre\}.$$

The meaning behind the $increment()$ function is that it increments with c the pre and inc components of all tuples which have pre greater than or equal to the given one.

Definition 9. Given an ID-PRE map M with $baseid$ and a set of tuples L we define the function $insert : PRE \times ID \times \mathbb{N}_1 \rightarrow M'$, $insert(pre, id, c) = M'$

as follows:
Case I: if $L = \emptyset$ and $pre = id$ and $id = baseid + 1$, then

$$M' = \langle baseid + c, L \rangle$$

Case II: if $L = \emptyset$ and either $pre \neq id$ or $id > baseid + 1$, then

$$M' = \langle baseid, L \cup \{(pre, pre, id, id + c - 1, c)\} \rangle$$

Case III: if $L \neq \emptyset$, and $\nexists t' = (pre', oid', fid', lid', inc') \in L$, such that $pre' < pre$, then

$$M' = \langle baseid, increment(L, pre, c) \cup \{(pre, pre, id, id + c - 1, c)\} \rangle$$

Case IV: if $L \neq \emptyset$ and $\exists t' = (pre', oid', fid', lid', inc') \in L$, such that $pre' < pre$, then $M' = \langle baseid, L' \rangle$, where L' is defined as follows:
Case 1: if $\exists t_k = (pre_k, oid_k, fid_k, lid_k, inc_k) \in L$, such that $pre_k = pre$, then

$$L' = increment(L, pre, c) \cup \{(pre, oid_k, id, id + c - 1, inc' + c)\}$$

Case 2: if $\exists t_k = (pre_k, oid_k, fid_k, lid_k, inc_k) \in L$, such that $pre_k < pre < pre_k + lid_k - fid_k$, then

$$\begin{aligned} L' = increment(L \setminus \{t_k\}, pre, c) \cup \{ & \\ t_k^1 = & (pre_k, \quad oid_k, \quad fid_k, \quad lid_k^*, \quad inc_k^*), \\ t = & (pre, \quad oid_k, \quad id, \quad id + c - 1, \quad inc_k^* + c), \\ t_k^2 = & (pre + c, \quad oid_k, \quad lid_k^* + 1, \quad lid_k, \quad inc_k + c)\} \end{aligned}$$

where

$$\begin{aligned} lid_k^* &= fid_k + pre - pre_k - 1 \text{ and} \\ inc_k^* &= inc_k - (lid_k - lid_k^*) \end{aligned}$$

Case 3: if none of the cases 1 or 2, then

$$L' = increment(L, pre, c) \cup \{(pre, pre - inc', id, id + c - 1, inc' + c)\}.$$

Intuitively the definition of the *insert*() function can be explained as follows: Cases I and II are trivial since there are no recorded update operations. Case I refers to the situation when new entries are appended at the end of the database. Case II, on the other hand, describes the insert of nodes somewhere in the middle of a database in which either no updates have been executed, or if any, they have been trivial ones such as in Case I. Therefore, we need simply to add the new tuple to the empty set L.

The rest of the cases show how new tuples interact with existing ones. If we represent the original records of database and the inserted or deleted

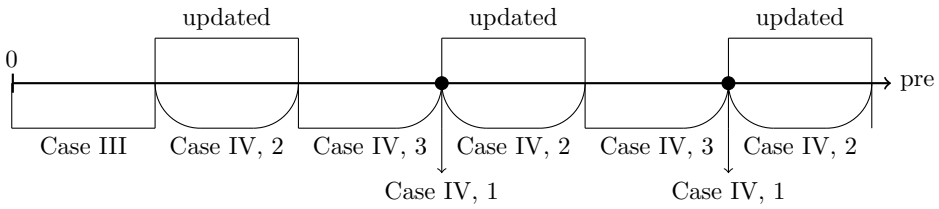


Figure 5: Insert intervals

ones on a straight line, we could split the set of pre values into regions as shown on figure 5.

When new records are inserted in one of the region types, the corresponding case is applied. Let us start with Case III: we insert new records in front of all other updated records. Consequently, we need to simply add the new tuple and increment all following tuples with the number of inserted records. We move on with Case IV in which we assume that there is at least one tuple before the insertion point. 3 sub-cases can be derived:

- Sub-case 1: the insertion point is at the same pre value at which there is already a tuple t_k. In this case we need to increment all following tuples (including t_k) and add the new tuple. The new tuple must have the same *oid* as t_k, since it is inserted at the same position as t_k and therefore, affects the same database records. Further, the *inc* value of the new tuple should reflect all update operations before the insertion point, and this is why we add the inc' value from the previous tuple to the number of newly inserted nodes.

- Sub-case 2: the insertion point is in the middle of an interval represented by an existing tuple t_k. In this case we need to split the tuple into two new tuples. The length of the first new interval is $pre - pre_k - 1$ and therefore its corresponding tuple has last id value equal to $lid_k^* = fid_k + pre - pre_k - 1$. Further, inc_k should be decreased with the length of the second new interval, namely $inc_k^* = lid_k - lid_k^*$. The second new tuple must represent an interval starting at the split point. This means that the pre value of t_k^2 should be equal to pre and the first id should be $lid_k^* + 1$. Finally, we need then to insert the new tuple. However, since we insert at position pre and we already have a tuple at that position (t_k^2), we apply the same operations as in sub-case 1: the *inc* value of t should reflect all operations before pre, i.e. $inc = inc_k^* + c$, and all following tuples, including t_k^2 should be incremented with c.

- Sub-case 3: the insertion point is in an interval in which no previous updates have been executed. This case is similar with sub-case 1. The only difference is that $oid = pre - inc'$. This expression comes

from the definition of the $pre()$ function: $pre(id) = id + inc \Leftrightarrow id = pre(id) - inc$.

The definition is complete since it defines the function for any possible insertion point. We will now proceed with the delete operation.

Operations: delete()

In the previous section we saw the different cases which can occur when inserting new records at a given position. The main difference between the delete and the insert operation is that by the latter one there are more possibilities how the deleted interval overlaps with the existing intervals, which were modified by other update operations. We will now give a detailed description of all the possible cases, but first, we will define some helper functions.

Definition 10. Given a set of tuples L, a pre value $pre \in PRE$ and a non-negative integer $c \in \mathbb{N}_1$, we define $range(L, pre, c)$ as follows:

$$range(L, pre, c) = \{t_i | t_i = (pre_i, oid_i, fid_i, lid_i, inc_i) \in L \text{ and } pre \leq pre_i \leq pre_i + lid_i - fid_i \leq pre + c - 1\}.$$

Definition 11. Given a set of tuples L, we define $max(L)$ as follows:

$$max(L) = t = (pre, oid, fid, lid, inc) \in L, \text{ such that}$$
$$\nexists t' = (pre', oid', fid', lid', inc') \in L, \text{ for which } pre < pre'.$$

Given a tuple $t = (pre, oid, fid, lid, inc) \in L$, we introduce the following convenience designations:

$$pre_{min} = pre$$
$$pre_{max} = pre + fid - lid.$$

Definition 12. Given an ID-PRE map M with $baseid$ and a set of tuples L we define the function $delete : PRE \times ID \times \mathbb{N}_1 \rightarrow M'$, $delete(pre, id, c) = M'$ as follows:
Case I: if $L = \emptyset$ and $pre = id$ and $id + c = baseid + 1$, then

$$M' = \langle baseid - c, L \rangle$$

Case II: if $L = \emptyset$ and either $pre \neq id$ or $id \neq baseid + 1$, then

$$M' = \langle baseid, L \cup \{(pre, id, -1, -1, -c)\} \rangle$$

Case III: if $L \neq \emptyset$, then $M' = \langle baseid, L' \rangle$, where L' is defined as:

$$L' = increment((L \setminus \{t', t''\}) \setminus range(L, pre, c), pre + c - 1, -c) \cup K$$

where $\{t', t''\} \cap range(L, pre, c) = \emptyset$, $pre \leq pre'_{max} \leq pre + c - 1$ and $pre \leq pre''_{min} \leq pre + c - 1$, and K is defined for the different cases as follows:
Case 1 (add new): if $\nexists t'$ and $\nexists t''$, then

$$K = \{(pre_{start}, oid, -1, -1, inc - c)\}, \text{ where}$$
$$oid = \begin{cases} id & \text{if } range(L, pre, c) = \emptyset \\ oid^* & \text{else} \end{cases}$$
$$inc = \begin{cases} 0 & \text{if } range(L, pre, c) = \emptyset \text{ and } \nexists t_k \in L, \text{ such that } pre_k \leq pre \\ inc_k & \text{if } range(L, pre, c) = \emptyset \text{ and } \exists t_k \in L, \text{ such that } pre_k \leq pre \\ inc^* & \text{if } range(L, pre, c) \neq \emptyset \end{cases}$$

Case 2 (shrink from beginning): if $\nexists t'$ and $\exists t''$, then

$$K = \{(pre_{start}, oid'', fid'' + pre_{end} - pre''_{min} + 1, lid'', inc'' - c)\}$$

Case 3 (shrink from end): if $\exists t'$ and $\nexists t''$, then

$$K = \{(pre'_{min}, oid', fid', fid' + pre_{start} - pre'_{min} - 1, inc - c)\}, \text{ where}$$
$$inc = \begin{cases} inc' & \text{if } range(L, pre, c) = \emptyset \\ inc^* & \text{if } range(L, pre, c) \neq \emptyset \end{cases}$$

Case 4 (shrink both): if $\exists t'$ and $\exists t''$, then

$$K = \{(pre'_{min}, oid', fid', fid' + pre_{start} - pre'_{min} - 1, inc - c),$$
$$(pre_{start}, oid'', fid'' + pre_{end} - pre''_{min} + 1, lid'', inc'' - c)\}, \text{ where}$$
$$inc = \begin{cases} inc' & \text{if } range(L, pre, c) = \emptyset \\ inc^* & \text{if } range(L, pre, c) \neq \emptyset \end{cases}$$

Case 5 (split): if $\exists t'$, such that $pre'_{min} < pre_{start} < pre_{end} < pre'_{max}$, then

$$K = \{(pre'_{min}, oid', fid', fid' + pre_{start} - pre'_{min} - 1, inc - c),$$
$$(pre_{start}, oid', fid' + pre_{start} - pre'_{min} + c, lid', inc' - c)\}$$

where

$$pre_{start} = pre,\ pre_{end} = pre + c - 1 \text{ and}$$
$$t^* = (pre^*, oid^*, fid^*, lid^*, inc^*) = max(range(L, pre, c)).$$

We will now explain the formal definitions in more detail. We will start with definition 10 of the $range()$ function. What this function does is simply to select all tuples of a given set L, which lie completely in the interval $[pre, pre + c - 1]$.

The next definition 11 defines the function $max()$, which given a set of tuples L returns the tuple with the highest pre.

We now turn to the actual definition of the $delete()$ function. Similar to the $insert()$ function, we first have two trivial cases I and II.

Case I describes the generated mapping when deleting records from a database in which no updates have been executed and the deleted records are located at the end. In this case we need simply to decrease the value of $baseid$ with the number of deleted records c.

Case II handles again the case when no updates have been executed in the database, but this time the deleted records are not located at the end. This means we need to insert a new tuple to denote the delete operation. The pre value and the original id value are directly provided as arguments of the function and we use them. Since we delete c records, we set the inc component to $-c$, which means that the pre values of all records after the deleted ones are decreased by c. The fid and lid components are set to -1. There are several reasons for this. First, according to assumption 1 the id values are non-negative integers, which means that -1 is an invalid id value. This is also valid for the id values of the deleted records, because according to assumption 2 id values cannot be reused. Second, since both fid and lid values are equal, the length of the interval represented by the tuple is 0, which naturally corresponds to the effect of a delete operation.

We now continue to Case III. It describes how the delete operation changes the mapping when there are already other tuples representing updates. In such case we need to remove all tuples for records with pre values in the deleted interval. This is exactly the range defined by the function $range(L, pre, c)$. Further, we need to modify the tuples where deleted interval boundaries lie, if such exist. Finally, similar to the $insert()$ function, all subsequent records need to be decremented by c (or equivalently incremented by $-c$).

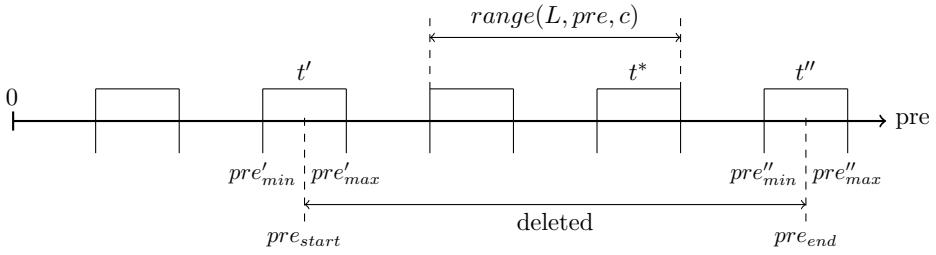


Figure 6: Delete intervals

Figure 6 depicts graphically the main terms used in the definition. The tuples t' and t'' does not belong to $range(L, pre, c)$ and are such that the start and end of the delete interval lie within them. Of course, this is not always the case: it is possible that either the start or the end, or both of

them does not land in an existing tuple. Each sub-case of Case III describes one such possibility.

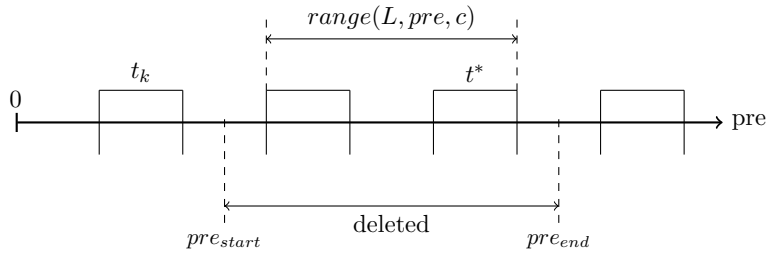


Figure 7: Delete intervals: case III, sub-case 1

Sub-case 1 handles the case when both start and end points do not land in existing tuples. In this case, all tuples from $range(L, pre, c)$ are removed and a new one is inserted. Figure 7 depicts this case.

An important question is how the *oid* and *inc* components of the new tuple are defined. If there are no tuples which are removed, i.e. the set returned by $range(L, pre, c)$ is empty, then we take the *id* argument of the *delete*() function as *oid*. In this sub-case it is safe to do so, because we are sure that the given *id* argument is not generated by an insert (otherwise, the starting point would land into a tuple). The *inc* component is the *inc* of the previous tuple t_k minus c or, if t_k does not exist, simply $-c$. The reason for this is that the *inc* component of the new tuple must reflect not only the delete but also all previous updates (if any) of records with smaller pre values.

On the other hand, when there are tuples to be removed, then it is important to set the *oid* component to the *oid* component of the last tuple t^* from the removed set. This is because it is not sure, if the *id* argument is generated by an insert operation. This can well be the case, if, for example, pre_{start} is exactly the same as the *pre* value of the first tuple to be removed. The *inc* component must be set to the *inc* component of t^* minus c, since, similar to the previous case, it must reflect the updates represented by the removed tuples.

We move on to sub-case 2. Here the start of the deleted interval again does not lie in an existing tuple, but the end does, as shown by figure 8.

In this situation we need to "cut" the first half of the tuple t'', decrement its *inc* value with c, so that it reflects the delete operation and set its pre value to pre_{start}, since its first record will move to the left after the delete.

Sub-case 3, shown in figure 9, is very similar to sub-case 2.

Now, we need to "cut" the second half of the tuple t' and set the *inc* component to $inc - c$. Similar to sub-case 1, the reason behind this is, that we need to reflect somehow the effect of the updates represented by the removed tuples.

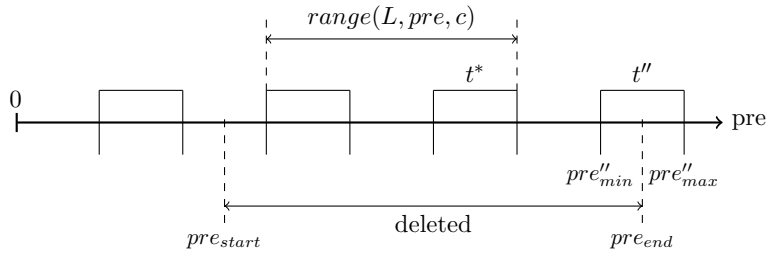


Figure 8: Delete intervals: case III, sub-case 2

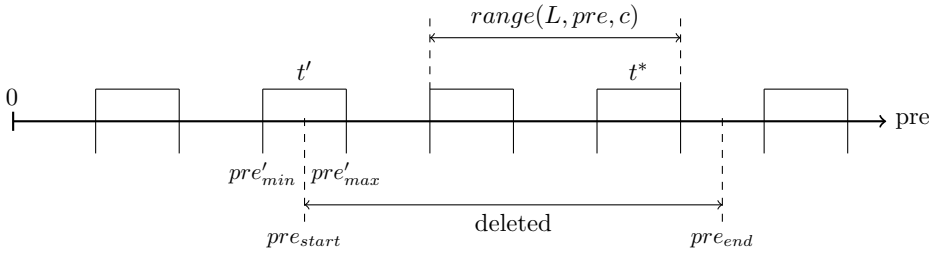


Figure 9: Delete intervals: case III, sub-case 3

Sub-case 4 is depicted in figure 6. It is handled by combining sub-case 2 and 3: the tuple t' is shrunk from the end, t'' from the beginning, and the *inc* components are adjusted.

The sub-cases described so far are well-defined when $t' \neq t''$ and $t' = t''$. Sub-case 5, on the other hand, can occur only if $t' = t''$, namely when the delete operation removes only records, which have been inserted by another update. In this case, the delete interval is completely contained in a tuple (see figure 10).

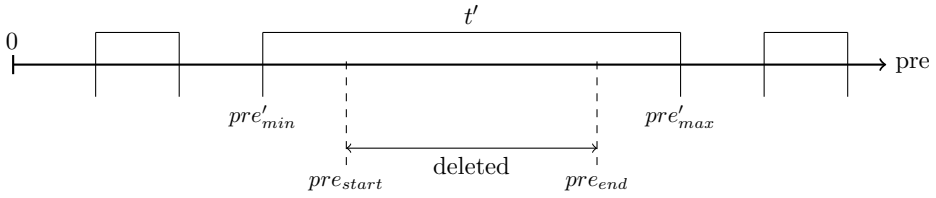


Figure 10: Delete intervals: case III, sub-case 5

Consequently we need to split the tuple t' into two adjacent tuples. The first tuple will end at $pre_{start} - 1$, and the second one will start at pre_{start}. The fid, nid, and *inc* components of the new tuples are adjusted correspondingly to reflect their lengths.

2.6 Correctness

Formal definitions are difficult to read and understand and this is why we have provided detailed explanations after each function definition. On the other hand, they allow easier identification of inconsistencies and they are necessary when one wants to prove correctness of an algorithm. A complete correctness proof of the algorithm will not be presented. Instead we will describe the main idea of one possible proof.

First, we need to define what we would consider as a correct behavior of our algorithm. The simplest algorithm for ID-PRE mapping is storing all id values sorted by their corresponding pre value. If we show that for each id value our algorithm gives the same pre value as this naive implementation, then it can be considered correct. However, before we start with the proof, we need to formally define this trivial algorithm.

Definition 13. A trivial ID-PRE Map M_0 is a finite sequence of id values $(id_0, id_1, ..., id_n)$ and the following operations:

- $pre_0 : ID \to PRE$, defined as

$$pre_0(id) = i, \text{ where } id_i \in M_0 \text{ such that } id_i = id$$

- $insert_0(pre, id, c) = M_0'$, defined as

$$M_0' = (id_0, id_1, ..., id_{pre-1}, id, id + 1, ..., id + c - 1, id_{pre}, ..., id_n)$$

- $delete_0(pre, id, c) = M_0'$, defined as

$$M_0' = (id_0, id_1, ..., id_{pre-1}, id_{pre+c}, ..., id_n)$$

Now, we are ready to prove the following statement:

Theorem 1. Given an ID-PRE Map M and a trivial ID-PRE Map M_0, then

$$\forall id \in ID,\ pre(id) = pre_0(id).$$

Proof. We will not present the complete proof of the theorem. Instead, we will only sketch the proof and the details will be left out of the scope of this work.

This proof will use mathematical induction over the number of executed update operations.

Base Case. The base case is when no update operation has been executed. In this case the id values will be equal to the pre values. This also means that the set L of the ID-PRE map M will be empty and therefore, according to definition 6, $pre(id) = id$.

Inductive Step. Let us assume that the statement holds when $n > 0$ update operations have been executed in the database, respectively. the ID-PRE Map. We will show that the statements holds for $n + 1$, too.

The new update operation can be either an insert or a delete. For each update operation type we need to consider all possible cases as defined in the corresponding function definition. After that we need to show that in this case the statement holds. For the sake of brevity we will prove the statement only for one case; the rest of the cases can be proved analogously.

Let us assume that the new update operation is a delete, such that we fall in Case III, sub-case 2 of definition 12, i.e. $L \neq \emptyset$ and $\nexists t'$, such that $t' \notin range(L, pre, c)$ and $pre \leq pre'_{max} \leq pre + c - 1$, but $\exists t''$, such that $t'' \notin range(L, pre, c)$ and $pre \leq pre''_{min} \leq pre + c - 1$. According to the definition, the resulting ID-PRE Map is $M' = \langle baseid, L' \rangle$, where L' is

$$L' = increment((L \setminus \{t''\}) \setminus range(L, pre, c), pre + c - 1, -c) \cup$$
$$\{(pre_{start}, oid'', fid'' + pre_{end} - pre''_{min} + 1, lid'', inc'' - c)\}$$

This case is depicted by figure 8.

Now let us consider the result of the $pre()$ function for a given id_k as defined by definition 6.

Case 1: if $id_k < min\{oid\}$, then $pre(id_k) = id_k$. In this case the pre value of the record has not been affected by updates. This means that $pre_0(id_k) = id_k$, too.

Case 2: if $id_k > baseid$, then $pre(id_k) = pre_i + id_k - fid_i$, where $t_i = (pre_i, oid_i, fid_i, lid_i, inc_i) \in L'$, such that $id_k \in [fid_i, lid_i]$. We need to consider two cases here:

- if $pre_i < pre$, then the record is not affected by the update and therefore according to the induction assumption $pre(id_k) = pre_0(id_k)$.
- if $pre_i \geq pre$, then the value pre_i should have been decremented with c. This means, that before the delete operation $pre'(id_k) = pre_i + c + id_k - fid_i$. However, according to the induction assumption, before the delete $pre'_0(id_k) = pre'(id_k)$, too. The effect of the delete operation on the trivial ID-PRE Map is that the pre values of all records after the deleted range are decreased by c. Therefore,

$$pre_0(id_k) = pre'_0(id_k) - c = pre'(id_k) - c =$$
$$pre_i + c + id_k - fid_i - c =$$
$$pre_i + id_k - fid_i = pre(id_k)$$

Case 3: $id_k \leq baseid$ can be proved similarly to case 2. □

2.7 Implementation and Performance

Besides providing the means to prove correctness of the algorithm, the formal definition can be used for a straightforward implementation of the algorithm. The only difficulty which needs to be solved is how to find the tuples we need. In this section we will present a simple implementation in order to show the practical applicability of the approach. It uses binary search and linear scan, so it has a lot of room for optimizations, some of which are proposed in the next section "Future Work".

Data Structure

The formal definition from the previous section uses tuples of the form $(pre, oid, fid, lid, inc)$. In this implementation we will use an array of integers for each component. Elements from the different arrays with the same index correspond to the components of one tuple. The tuples will be sorted according to their pre values. This will ensure a logarithmic complexity for searching by pre and oid.

Function: pre()

The $pre()$ function is relatively simple.

Algorithm 1 Function $pre(id)$

```
1:  function PRE(id)
2:      if size = 0 or id < pre_0 then
3:          return id                                  ▷ Not affected by updates
4:      else if id > baseid then
5:          for i ← 0, size − 1 do                     ▷ Find the tuple, which added id
6:              if fid_i ≤ id ≤ lid_i then
7:                  return pre_i + id − fid_i
8:              end if
9:          end for
10:         return −1                  ▷ Record has been deleted: should not occur
11:     else
12:         i ← BSEARCHLAST(oid_list, id)
13:         return id + inc_i
14:     end if
15: end function
```

As already said, the implementation follows closely the different cases from the formal definition of $pre()$. We perform a linear scan, in which we search for an id value, that has been inserted by an update operation and we should also note that the function $bsearchLast(array, value) \rightarrow index$

performs a binary search for the *value* in the given *array* and returns the greatest index of the greatest element which is less than or equal to *value*.

Function: insert()

The *insert*() implementation follows closely the formal definition, too. The function $bsearch(array, value) \rightarrow index$ performs binary search for the *value* in *array* and returns the *index* where the *value* was found, or if the *value* is not found, $-index - 1$ is returned, where *index* is the position, where the value should be inserted. The procedure *add*() inserts a new tuple at the position specified by its first argument and $increment(index, value)$ increments each element in the arrays *pre* and *inc* with *value* starting with the element at position *index*.

Function: delete()

The *delete*() function is also to a great extend based on its formal definition but due to its complexity it is distributed across several helper functions. The trivial case when there are no other tuples is handled simply and corresponds directly to the definition. However, when there are other tuples, we need to find which tuples are affected by the delete. Therefore, we use the function $find(pre) \rightarrow index$, which performs binary search for *pre* in the array with pre values. The binary search procedure considers a tuple a positive match not only when the pre component is equal to *pre*, but also when the *pre* is within the interval defined by the tuple. Therefore, the result of the $find$ function is the index of either the tuple which contains *pre*, or the tuple with the least pre component bigger than *pre*.

After locating the first affected tuple, we need to find the last one. We also need to identify the tuples which are completely "contained" in the delete interval so that we can remove them. This is performed by the *range*() function which, starting from the first affected tuple at i_{start}, checks if a tuple is completely within the interval $[pre, pre_{end}]$. The result of the function are the indexes of the first and the last tuple to remove. If the last index is smaller than the first one, then no tuples can be removed, i.e. no tuples are completely contained in the delete interval, which further means that at most one tuple is affected (and hence, $i_{end} = i_{start}$). On the other hand, we need to remove the tuples, which is the task of the function $remove(index, index)$.

What remains to be done is to adjust the tuples which are not completely contained in the delete interval. The function *adjustTuples*() does exactly this using the sub-cases of case III of definition 12. The function $copy(index, index)$ adds a new tuple using the values of another one. We should also notice, that in all cases the tuple at index i_{end} is processed and therefore, the *increment*() function starts from $i_{end} + 1$. The only case in which this not true, is in sub-case 3. This is why i_{end} is decremented by -1.

Algorithm 2 Function $insert(pre, id, c)$

```
1: procedure INSERT(pre, id, c)
2:     if size = 0 then
3:         if pre = id and id = baseid + 1 then
4:             baseid ← baseid + c                                  ▷ Case I
5:         else
6:             ADD(0, pre, pre, id, id + c − 1, c)                  ▷ Case II
7:         end if
8:     else
9:         i ← BSEARCH(pre_list, pre)
10:        if i = 0 or i = −1 then
11:            i ← 0
12:            oid = pre                                            ▷ Case III
13:            inc = c
14:        else if i > 0 then
15:            oid = oid_i                                          ▷ Case IV, 1
16:            inc = inc_{i−1} + c
17:        else
18:            i ← −i − 1
19:            n ← lid_{i−1} − fid_{i−1} + 1
20:            if pre < pre_{i−1} + n then                          ▷ Case IV, 2
21:                ADD(i, pre, oid_{i−1}, fid_{i−1} + pre − pre_{i−1}, lid_{i−1}, inc_{i−1})
22:                lid_{i−1} ← fid_i − 1                            ▷ Shrink the old tuple
23:                inc_{i−1} ← inc_{i−1} − n + pre − pre_{i−1}
24:                oid ← oid_{i−1}
25:                inc ← inc_{i−1} + c
26:            else
27:                oid ← pre − inc_{i−1}                            ▷ Case IV, 3
28:                inc ← inc_{i−1} + c
29:            end if
30:        end if
31:        INCREMENT(i, c)
32:        ADD(i, pre, oid, id, id + c − 1, inc)
33:    end if
34: end procedure
```

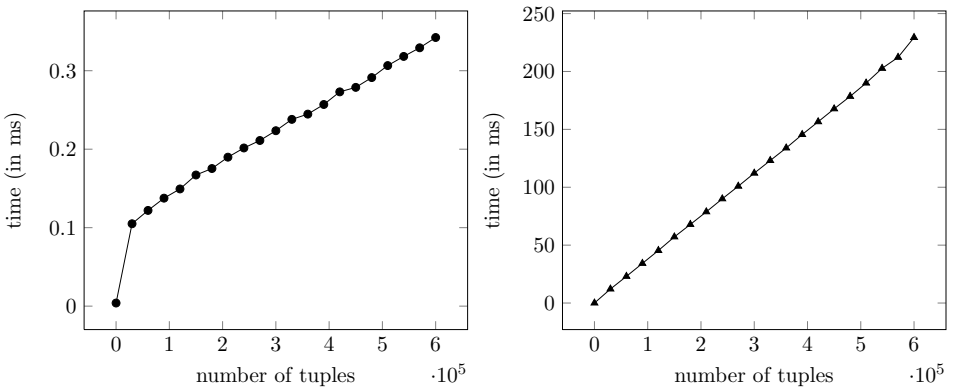
Algorithm 3 Function $delete(pre, id, c)$

```
1: procedure DELETE(pre, id, c)
2:     if size = 0 then
3:         if pre = id and id + c = baseid + 1 then
4:             baseid ← baseid − c                                  ▷ Case I
5:         else
6:             ADD(0, pre, id, −1, −1, −c)                          ▷ Case II
7:         end if
8:     else                                                         ▷ Case III
9:         pre_end ← pre + c − 1
10:        i_start ←FIND(pre)
11:        r_start, r_end ←RANGE(i_start, pre, pre_end)
12:        if r_start ≤ r_end then
13:            inc ← inc[r_end]
14:            oid ← oid[r_end]
15:            i_end ← r_start
16:            REMOVE(r_start, r_end)
17:        else
18:            inc ← inc[i_start − 1] or 0, if i_start ≤ 0
19:            oid ← id
20:            i_end ← i_start
21:        end if
22:        ADJUSTTUPLES(i_start, i_end, pre, oid, inc, c)
23:    end if
24: end procedure
```

Algorithm 4 Function $adjustTuples(i_{start}, i_{end}, pre, oid, inc, c)$

```
1:  procedure ADJUSTTUPLES(i_start, i_end, pre, oid, inc, c)
2:      if size ≤ i_start then
3:          ADD(i_start, pre, oid, −1, −1, inc − c);                ▷ Sub-case 1
4:      else if i_start < i_end then
5:          if i_end < size and pre[i_end] ≤ pre_end then
6:              SHRINKFROMSTART(i_end, pre, c)                      ▷ Sub-case 4
7:              SHRINKFROMEND(i_start, pre, inc − c)
8:          else
9:              i_end ← i_end − 1                                   ▷ Sub-case 3
10:             SHRINKFROMEND(i_start, pre, inc − c)
11:         end if
12:     else if pre < pre_end < pre[i_start] then
13:         ADD(i_end, pre, oid, −1, −1, inc − c);                  ▷ Sub-case 1
14:     else if pre ≤ pre[i_start] < pre_end then
15:         SHRINKFROMSTART(i_end, pre, c)                          ▷ Sub-case 2
16:     else
17:         i_end ← i_end + 1                                       ▷ Sub-case 5
18:         COPY(i_start, i_end)
19:         SHRINKFROMSTART(i_end, pre, c)
20:         SHRINKFROMEND(i_start, pre, inc − c)
21:     end if
22:     INCREMENT(i_end + 1, −c)
23: end procedure
```

Performance: pre()



(a) with the first 1000 id values (b) with the second 1000 id values

Figure 11: Performance of *pre*().

Figure 11 shows how the performance of the *pre*() function changes with the number of tuples in the ID-PRE map. The ID-PRE map used in this test assumes a database with 1000 initial records (i.e. $baseid = 1000$). Subsequently, inserts are executed until 30000 new tuples are generated in the ID-PRE map and then the performance of *pre*() is measured for the first 1000 id values (from 0 to 999) and the second id values (from 1000 to 1999). This test is repeated 7 times with the last test using 600000 tuples.

The values of the first graph show the time needed by the function to calculate the pre values of the first 1000 id values and the second graph – of the second 1000 id values. The reason for the huge performance difference is due to the fact that for id values smaller that *baseid* the algorithm 1 uses binary search, i.e. the complexity is $\mathrm{O}\,(log(N))$, and for id values greater than *baseid* – linear search ($\mathrm{O}\,(N)$).

Performance: insert()

Figure 12 depicts the performance of the *insert*() function. Similar to the previous test, the ID-PRE map has $baseid = 1000$. Then inserts are executed until 30000 new tuples are generated and the performance of single *insert*() call is measured. This is repeated 7 times, and each *insert*() call has the same arguments.

If we look at the implementation of the *insert*() function, we will notice that the operations which determine the complexity are the binary search and the incrementing of the *pre* and *inc* components of subsequent tuples (we assume that adding a new element needs constant time). Thus, the complexity of the function is $\mathrm{O}\,(log(N) + N)$. While the increment operation substantially increases the execution time, it is acceptable, since otherwise

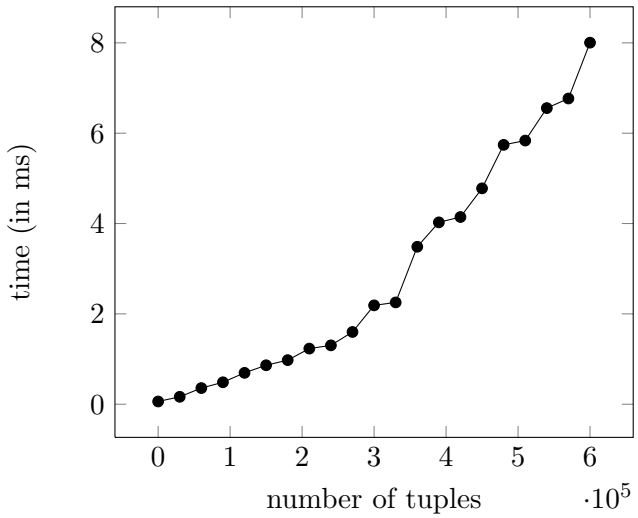


Figure 12: Performance of *insert*() for a single operation.

it should have been implemented in *pre*(), which is expected to be called much more often and therefore, would have worse overall performance.

Performance: delete()

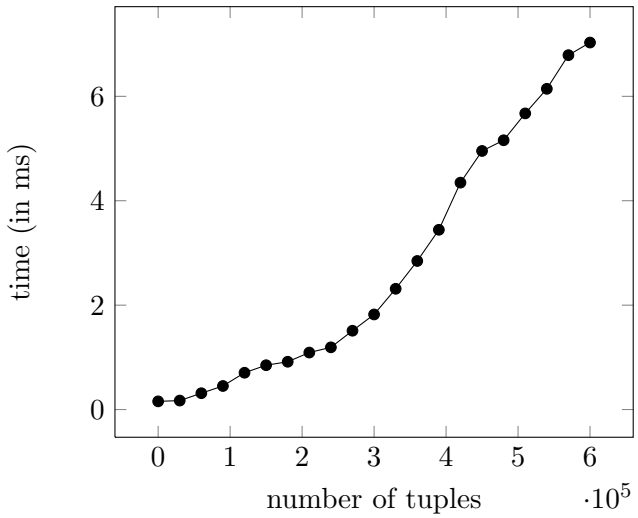


Figure 13: Performance of *delete*() for a single operation.

The performance of the *delete*() function is depicted on figure 13. The tests are performed under the same conditions as for the *insert*() function. In each test a single delete operation is executed with the same arguments.

The complexity of the *delete*() function is the same as this of the *insert*().

First we use binary search to find the start of the deleted interval. After that a linear search is applied to find its end and finally to increment all subsequent tuples. This gives complexity $\mathrm{O}\left(log(N) + N\right)$.

2.8 Related Work

Different approach, than the one presented here, has been used by the authors of MonetDB/XQuery [BGvK+06]. In their implementation, they divide the set of document nodes into pages, each capable of storing k nodes. The nodes in each page are sorted by their pre values, but the pages themselves are physically ordered by the id value of their first nodes. Additionally, a separate page-mapping table is maintained, which defines the logical, pre order, of the pages.

Compared to our ID-PRE Map, this solution differs in that although the mapping contains only one record per page, it is still linear with respect to the number of stored nodes. In contrast the algorithm proposed here scales linearly with the number of update operations, which in general is much smaller than the actual number of nodes. Further, using the page-map table approach means that not all pages will be fully utilized, even though later insert operations may add new nodes to them.

2.9 Future Work

Bulk Operations

The functions, which we defined in the previous sections, represent only a single operation. However, this is seldom the case in real scenarios – it is much more probable that we need the pre values of a set of id values than of a single one. Additionally, a transaction may consists of multiple insert or delete operations each modifying a different range of database records. Thus, depending on the implementation of the XML database, the overall performance can be increased if some assumptions are made about the input data. For example, if we need to find the pre values of a list of id values and we assume that the list is sorted, we could optimize the binary search in the $pre()$ function to traverse a smaller range of tuples each time. Of course, before making such optimizations, one should analyze usage patterns in the corresponding database system.

Optimization

In the previous section we showed how the number of tuples in the ID-PRE map affects the performance of the different operations. We can consider the performance of the update functions acceptable mainly because they are executed only once per database update regardless of the number of updated database records. Much more critical is the performance of the

$pre()$ function, which needs to be called for every id value. In the best case, if an id value is not affected by updates, the $pre()$ function needs constant time – it just returns the id value itself. The worst case is when each single database record is inserted by a separate insert operation, and the order of execution of these inserts does not follow the pre order. In this case, each database record would correspond to a tuple in the ID-PRE map and the map will occupy 5 times more main memory as the trivial map defined in 13 [2]. Further, since all id values will be greater than *baseid* (except the id value of the first insert, which will be an append operation), we will fall in the conditional branch of the function where we need to execute a linear search to find the corresponding tuple. Therefore, in this situation it is not appropriate to use ID-PRE mapping.

While we cannot control the order in which the update operations are executed, we could optimize the $pre()$ function. The case which needs to be optimized is obviously the one when the id value is greater than *baseid* and we perform binary search. The reason for this is that the tuples are ordered by their pre values.

One possible optimization is to maintain an additional index, which can be used for fast search by id values. On the other hand, this will bring additional overhead in the update operations.

Further improvement brings the support of merging neighboring records which represent consequent id-pre intervals. Figure 14 depicts such scenario. This can decrease the number of records in the ID-PRE map and hence, increase the performance. This extension can be also very easy implemented: after executing each insert or delete operation, we need to check if the two new neighboring records can be merged.

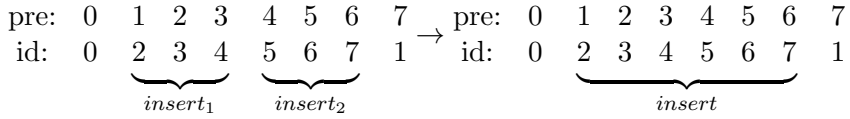


Figure 14: Merging neighboring records.

Another simple and straightforward optimization is to leverage the multiprocessing capabilities of modern computer systems. There are two possibilities to apply parallelization. First, we could parallelize the execution of the linear search in the $pre()$ function by splitting the set of tuples into k sections and starting k threads, each of which searches for the given id value in one of the section. While this does not solve the problem in general, it may bring the performance to an acceptable level.

A second opportunity is to parallelize the calls to the $pre()$ function. In many cases a query in an XML database will return not a single database node but several ones. In this case we could again split the set of id values

[2] 5 times, because each tuple consists of 5 components

into k sections and start k threads, each of which executes the $pre()$ function of each id value from a section.

There are also cases in which parallelization actually decreases the overall performance. Therefore, it should be applied after careful performance testing and tuning.

Concurrent Access

Similar to other database resources, such as indexes, the ID-PRE map participates in transactions. This is why it is important to ensure safe concurrent access to it. There are two main questions which need to be solved: first, what concurrency control should be used when more than one transaction calls a method of the ID-PRE map and second, how the calls the the ID-PRE map functions should be integrated in the database operations. The answers of these questions depend mainly on the concurrency control protocol used by the concrete database implementation.

Persistent Storage

Another open question related to transactions is how and when the ID-PRE map should be saved to a persistent storage. This a very important requirement to any database which claims to support ACID transactions. Currently, we have considered the ID-PRE map as an in-memory data structure and the proposed implementation vastly relies on very fast random access. On the other hand, a significant advantage is that it requires relatively small amount of main memory, which also suggests that a complete flush of the data structure to persistent storage will not be expensive.

2.10 Summary

The purpose of this section is not to provide a complete optimized implementation, but rather to analyze the different cases which when implementing the algorithm and provide formal means of proving its correctness. The formal definitions can serve as the base for various implementations with different performance characteristics. On the other hand, the feasibility and the general applicability of the approach can be proved only by real use-cases.

3 Variable-Length Record Storage

3.1 Introduction

Storing records with variable length is very common and well-studied problem in the relational databases. Variable-length records occur even more often in native XML databases than in relation databases. They appear mainly in the form of texts: text nodes, attribute values, tag names, or namespace names. Therefore, it is important to have an efficient and robust mechanism to store such records.

The standard solution for storing variable-length records is the slotted page layout, a description of which can be found in the book "Database Management Systems" from Ramakrishnan[RG00]. This approach can as well be used in native XML databases. It is already used in the native XML databases Sedna[FGK06] and Natix[FHK+02].

The purpose of this section is not to describe the slot page approach, but rather to reveal some implementation details, which may be difficult to figure out.

3.2 Requirements

Our main goal is to efficiently store and retrieve records with variable length. This means that if a record is represented as an array of bytes, we would like to store it and get a record identifier *rid*, which we can later use to retrieve or delete the record. Thus, the interface of our implementation should provide at least the following methods:

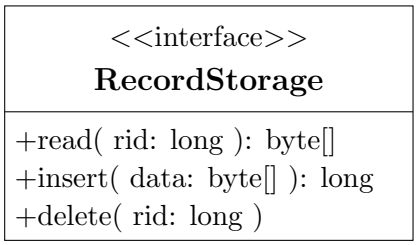


Figure 15: Record storage: interface definition.

Further, we require that the space occupied by deleted records should be re-used. Therefore, when inserting a new record with a given length, we must be able to quickly find a place where it can be inserted.

An additional requirement which we pose on the implementation is to perform block-wise the input and output operations from and to the underlying storage. This requirement is determined by the way how modern database systems work, rather than a real need. It is important because the storage mechanism needs to be integrated into the rest of the system.

3.3 Block Management

The requirement to perform block-wise I/O operations means that we need to manage somehow the blocks in a file. A simple approach is to use the first block of a file to store a bit map, in which every bit corresponds to a block and shows if it is used or not. For example, if we assume that the size of a block in a file is 4096 bytes, then we have 32768 bits with which we can track the next 32768 blocks of the file. If all these blocks are occupied, then we need another block which we use to track the next 32768 blocks, and so on. Basically, this approach divides a file into segments containing fixed number of blocks. Each segment has a header block with a bit map with the occupied and empty blocks in the segment. Figure 16 illustrates this file layout.

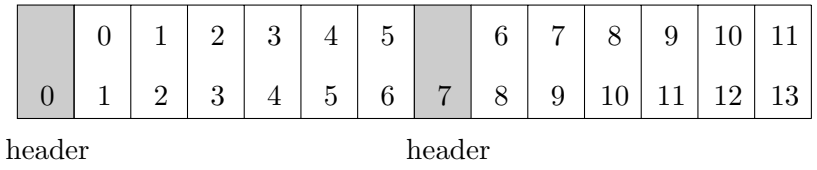


Figure 16: Empty block management: header blocks with bit maps.

This file organization should be transparent for the higher levels of the system. Therefore, header blocks cannot be read and higher levels need to use logical block identifiers. Let us assume that all blocks in a file, including the header blocks, are enumerated starting from 0. Now, if we assume that the logical identifiers start from 0, too, then we can calculate the actual block identifier using the following formula:

$$translate(logicalId) = \left\lfloor \frac{logicalId}{SegmentSize} \right\rfloor + logicalId + 1$$

where $SegmentSize$ is number of blocks in each segment. Figure 16 exemplifies this with $SegmentSize = 6$.

The implementation itself must have at least the methods defined in figure 17.

It is relatively easy to implement these four methods. The methods *readBlock* and *writeBlock* should first translate the logical block identifier, then position at the actual block, and finally read, respectively write, the block content.

Deleting a block is very simple, too: given the block number, we need to set the corresponding bit in the corresponding segment header block to 0. The corresponding header block number can be calculated using the following formula:

$$header(logicalId) = \left(\left\lfloor \frac{logicalId}{SegmentSize} \right\rfloor + 1 \right) SegmentSize$$

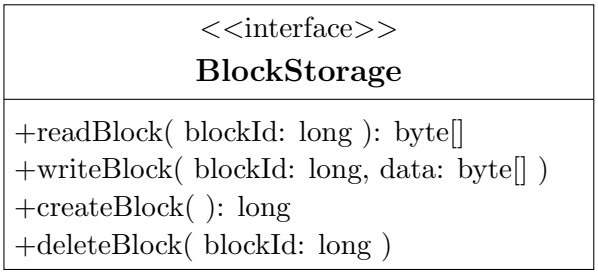


Figure 17: Block management: interface definition.

Optionally, if the deleted block is the last block in the file, we may wish to shrink the file.

Creating a new block is a little bit more involved, because we need to search the header blocks for a free block. If we find a 0 bit in one of the header blocks, then we set it to 1 and return the logical block number of the corresponding block. If all blocks are occupied, then we need to allocate a new header block and set its first bit to 1.

While executing performance tests, it was found that scanning all header blocks when a new data block needs to be created is not very efficient. A simple optimization is to maintain an in-memory bit-map where every bit represents a header block and has value of 1, if the corresponding segment does not have free blocks (i.e. the corresponding header block contains only 1 bits) and 0, otherwise. Since the number of header blocks, respectively segments, is relatively small compared to the number of blocks in the file, this bit-map will not pose a significant memory overhead. Further, it needs to be updated only when a header block becomes full. The biggest advantage, as we will see later in the section with the performance measurements, is that we avoid the linear complexity when we need to allocate a new block.

3.4 Record Storage

Besides keeping track of empty blocks, another important requirement is the ability to reuse the space freed after a record has been deleted. In order to do this we need to keep the number of free bytes in each block and we need to do it efficiently. For example, the naive solution of checking each block for free space would be very inefficient, because in the worse case we need to check every block just to insert an entry.

A common solution to this problem is the so called directory of blocks which is described in [RG00]. In essence, the approach consists of maintaining of a so called directory which contains references to the actual data blocks and, additionally, the number of free bytes in each data block (in our implementation we will use number of used bytes, for reasons explained further below). The directory itself is a linked list of blocks. This means,

that it is only needed to check the directory blocks for a block with enough empty space and since one directory blocks contains typically many references to data blocks, the search is much more efficient. Figure 18 depicts this data structure.

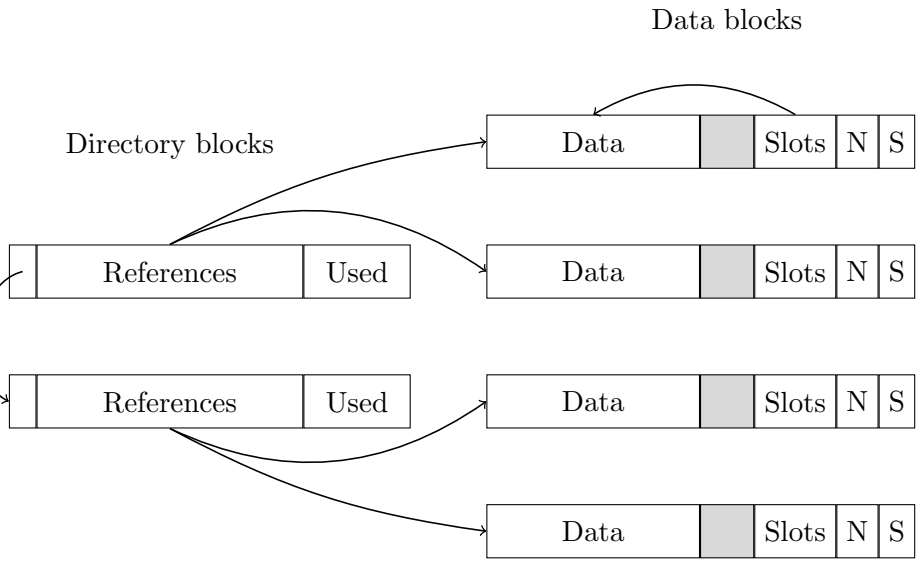


Figure 18: Directory of blocks.

We will now give more details about the concrete implementation of this approach.

Directory

The directory data structure is a linked list of blocks. Each block has a reference to the following one, or if it is the last one – NIL. The first block must be always stored at a fixed position (usually at the first address of the underlying storage).

As already said, each directory block contains references to $BLOCKS$ data blocks. Thus, besides using it as a means to quickly find a data block with enough free space, we can also use the directory to abstract the physical data block addresses. This will give us later the possibility to change the actual block locations without affecting other components. So, how can we implement this mapping between physical and logical block addresses? The directory is a linked list of blocks, each containing $BLOCKS$ references. Therefore, we can enumerate each reference starting from 0. We can then use this enumeration as a logical identifier of each data block.

We will now define the concrete methods which the directory data structure needs to implement (see figure 19).

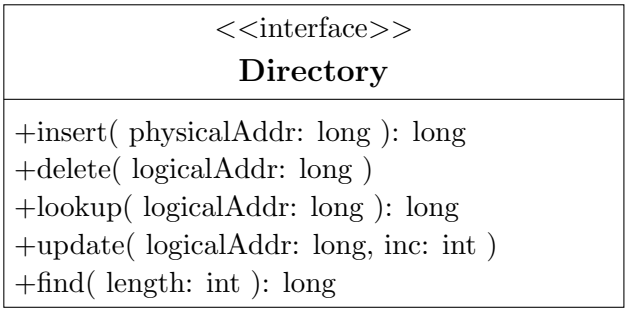


Figure 19: Directory interface definition.

The *insert* function registers the physical address of a newly created data block and returns its logical address. In order to do this, the function needs to find a free place in the arrays with references of each directory block. Let us assume that the i^{th} cell of the n^{th} directory block is free. Then the function writes in that cell the data block physical address and returns $i + n \cdot BLOCKS$ as logical block address. If no free cell is found, then we need to create a new directory block.

The *delete* function does the opposite: it removes the data block from the directory. It goes to the directory block with index $\lfloor \frac{logicalId}{BLOCKS} \rfloor$ and marks the reference with index ($logicalId \mod BLOCKS$) as empty. The functions *lookup* and *update* function use the same logic, however, instead of marking the reference as empty, the *lookup* function returns the reference value, and the *udpate* function increments the corresponding *used* entry with *inc*.

The last function is the $find$ function which searches the directory for a block with enough free space to store a record with length *length*. This function needs to be called each time when we want to insert a new record. Therefore, its performance determines the overall performance.

The simplest approach would be to traverse the whole directory from the beginning each time the function is called. However, in many cases this may be very slow. For example, let us consider the case when we create a new database and we continuously add new records. In this case, it does not make sense to start searching the directory from the beginning, because we are sure that no deletes have been performed, and hence there are no empty regions in the previous blocks. In order to be able to handle this case the directory implementation should be able to store the last used data block and start the $find$ function from there. Further, we need to extend the *RecordStorage* interface with a new method *append*, which should call the $find$ function without instructing it to start the search from the beginning of the directory.

Another important optimization, which should be applied, is to maintain a cache with the addresses of all directory blocks in an array. This

is necessary, because, if we need to access the n^{th} directory block in the linked list, we need to read all previous $n-1$ directory blocks. Caching the block addresses will allow finding the address of the needed directory block directly.

Data Blocks

The data blocks contain the actual records. Each data block can be regarded as if split into two areas: one with the actual data and one with meta-data. The meta-data area is usually stored at the end of the block and the data area in the beginning of the block. Thus, both sections can grow until no free space remains in the block.

The meta-data area contains the so called directory of slots – an array of offsets from the beginning of the block, showing the start positions of the records. The length of each record is stored in the first bytes of the record. Figure 18 shows the structure of a data block. The field S shows the size of the data area and N is the length of the slot array. A more detailed description of the approach can be found in [RG00]. Here, we will only present the methods which we need for a data block in order to implement our record storage.

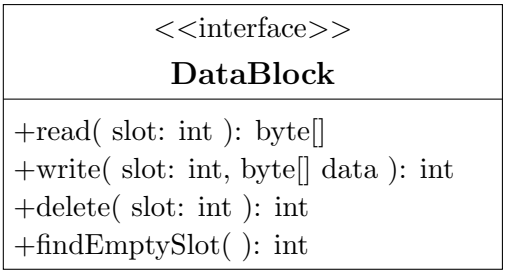


Figure 20: Data block interface definition.

Figure 20 shows what methods we need. Most of them do not need explanation. We should only note, that the methods *write* and *delete* return the number of bytes which are needed to store, respectively freed, after the operation. This is needed, since the number of used or freed bytes, is not the same as the length of the record. First of all, the length of the record should be stored, too. Second, it is not known if a new slot will be allocated/freed or an existing one will be reused, i.e. if the slot directory will grow, respectively shrink. Third, the slots may not be byte-aligned; for example if we assume that a slot is 12 bits, then allocating a slot with odd index will grow the array with only 1 byte, while allocating a slot with even index, with 2 bytes. Figure 21 illustrates the example.

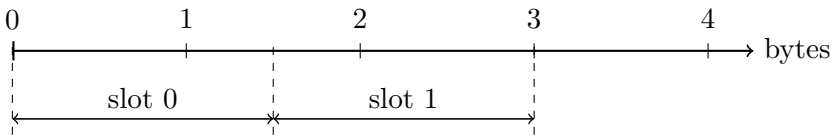


Figure 21: Slots which are not byte aligned (12 bits).

Putting It Together

Having defined the interfaces and their implementation, we are almost ready to sketch our implementation of variable-length record storage. There two more questions which remain to be answered: first we need to define the size of each field in the data structures and second, we need to define the structure of a record identifier.

To answer the first question we need to know the size of a block B and the size of a block reference R. Then we can calculate the number of data block references in a directory block n using the following formula:

$$B = R + n \cdot R + n \cdot \frac{log_2 B}{8}, \text{ hence}$$
$$n = \left\lfloor \frac{8 \cdot (B - R)}{8 \cdot R + log_2 B} \right\rfloor.$$

The meaning behind the first formula comes directly from the structure of a directory block: a directory block with length B contains a reference with length R to the next directory block, further n references to data blocks, and finally, n values, each with length $log_2 B/8$, representing the number of used bytes in each data block. For instance, if we assume that the $B = 4096$ and $R = 5$, then $n = 629$.

Similar reasoning can be used to determine the size of the slots in the data blocks, as well as the size of S (the size of the data area) and N (the number of slots). An important consideration for the size of a slot is to have enough bits, so that every possible offset in the data block can be stored, i.e. in most practical cases a slot would be at least $log_2 B$ bits long. The same is valid for the size of S.

The second problem which we should solve is what values should we use as record identifiers. Our main consideration in this case is to efficiently locate a record. Given how our proposed implementation works, it should be sufficient to use the logical data block number for the high bits and the slot number as the low bits of the record identifier.

Now, the methods of the *RecordStorage* interface can be readily implemented. The *read*() method is simple: we need to extract the data block logical address and the slot number from the record identifier. Then we need to look up the logical address and find the actual physical address from the

directory. Finally, we need to read the data block and decode the record data at the given slot.

Algorithm 5 Function *read*(*rid*)

```
1: function READ(rid)
2:     logicalAddr ← BLOCK(rid)
3:     slot ← SLOT(rid)
4:     physicalAddr ← directory.LOOKUP(blockNum)
5:     dataBlock ← GOTODATABLOCK(physicalAddr)
6:     record ← dataBlock.READ(slot)
7:     return record
8: end function
```

The function *append* can be easily implemented, too. The only peculiarity to note is that when searching for a block with enough space, we also need to consider the number of bytes which we need for storing the length of the record, as well as the the number of bytes which we need for storing the record offset in the block.

Algorithm 6 Function *append*(*record*)

```
1: function APPEND(record)
2:     logicalAddr ← directory.FIND(rlen + |rlen| + |slot|)
3:     if logicalAddr = NIL then
4:         physicalAddr ← NEWDATABLOCK( )
5:         logicalAddr ← directory.INSERT(physicalAddr)
6:     else
7:         physicalAddr ← directory.LOOKUP(logicalAddr)
8:     end if
9:     dataBlock ← GOTODATABLOCK(physicalAddr)
10:    slot ← dataBlock.FINDEMPTYSLOT( )
11:    used ← dataBlock.WRITE(slot, record)
12:    directory.UPDATE(logicalAddr, used)
13:    return RID(logicalAddr, slot)
14: end function
```

The implementation of the *insert*() function is the same as that of *append*(). The only difference is that we need to instruct the directory to start the search for data block with enough space from the beginning.

The implementation of the *delete*() method is very similar to the implementation of the *read*() method. The difference is that instead of reading the record, we delete it from the data block. Additionally, if the block remains empty after the deletion of the record, we need to delete it from the directory. Otherwise, we need to decrease the number of used bytes for the data block in the directory.

Algorithm 7 Function *delete*(*rid*)

```
1: function APPEND(record)
2:     logicalAddr ← BLOCK(rid)
3:     slot ← SLOT(rid)
4:     physicalAddr ← directory.LOOKUP(blockNum)
5:     dataBlock ← GOTODATABLOCK(physicalAddr)
6:     freed ← dataBlock.DELETE(slot)
7:     if freed = BLOCKSIZE then
8:         directory.DELETE(logicalAddr)
9:     else
10:        directory.UPDATE(logicalAddr, −freed)
11:    end if
12: end function
```

3.5 Storing Records Which Span Several Blocks

An important question, which have not been discussed yet, is how to store records which are larger than a block. It is proposed in [RG00] to split a record into chunks and store each chunk as a separate record together with the *rid* of the next chunk (effectively storing a record a linked list of chunks). While the idea is relatively simple, there are several practical issues which will discuss.

The scheme described in [RG00] can be well implemented without changing the previously described implementation: we just need to add one more level of abstraction which should take care to split large records into chunks. However, when reading a large record, the implementation should be able to differentiate between a normal record and a chunk of a record. Further, our implementation automatically stores the length of a record, but this is not necessary, since we know the length of each chunk. Finally, we don't need a slotted organization of the data blocks, since we know that only a single chunk is stored in the block. Thus, it would be simpler to extend the implementation to support larger-than-block records instead of adding a new component on top of it.

First, we will introduce a new type of data block – a chunk block. The difference to the normal data block is that it does not have a slot directory, the size of the data area is always the size of the block, and when we store a chunk in such a block we do not store its length.

Second, we add new methods for reading, writing, and deleting a chunked record. A peculiarity of the *write* method is that it must store the chunks in reverse order so that when we store a chunk we already have the *rid* of the next one. The last chunk (which, thus, will be stored first) should be stored as normal record, because it's length is less than the length of a block and in this way we also mark the end of the linked list. The *rid* of the whole

record is *rid* of the first chunk (which is stored last), which allows reading the chunks in the correct order.

Aside from these special points, extending the record storage to support records larger than a block is relatively straightforward – the *DataBlock* interface should be extended to support chunks and the *Directory* interface remains unchanged.

3.6 Performance Measurements

We will now present some performance measurements in order to test the viability of the proposed implementation. We will compare the results to another implementation, which is also able to store variable-length records. This implementation is however very simple: it does not re-use space which remains empty after a record is deleted. Instead, it simply appends each new record at the end of the used file. However, this implementation provides the highest possible performance for all operations since it does not have any overhead from meta-data structures.

Both implementations use the same clock-based buffer manager with 16 pages each with size 4096 bytes.

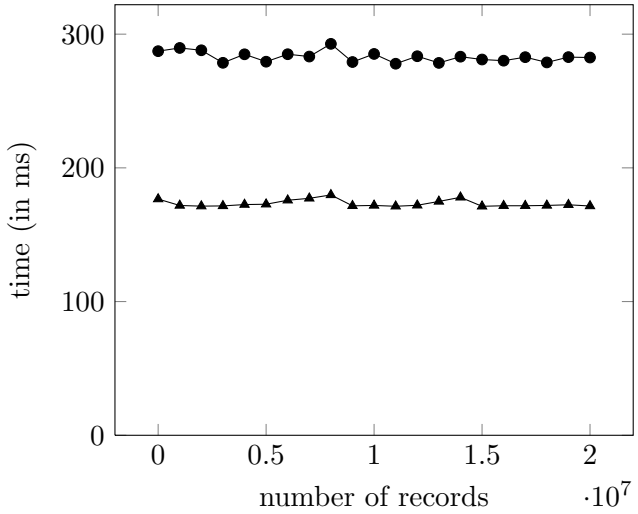


Figure 22: Performance of *append*() —●— compared to a trivial implementation —▲—: appending 1000000 records.

In figure 22 each point represents the time needed by each implementation to append 1000000 records with length 20 bytes starting with an empty file. The linear complexity of both algorithms can be seen well. In addition, although our implementation has the overhead of managing meta-data, it is roughly 1.5 times slower than the trivial implementation.

Another case in which the trivial implementation is expected to be extremely fast is accessing a record by *rid*. It uses as a *rid* the offset from the

beginning of the file where the record is stored, so there is no overhead. In contrast, the proposed implementation needs to first locate the data block address in the directory, read the data block, and only then decode the record from the block.

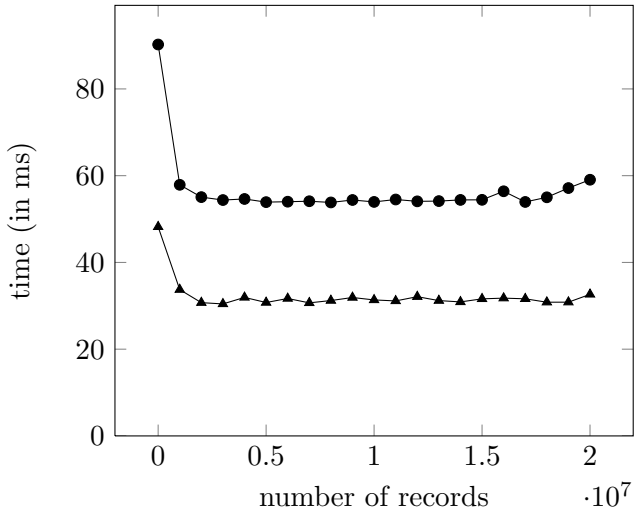


Figure 23: Performance of *read*() —●— compared to a trivial implementation —▲—: reading 10000 random records.

Figure 23 shows that both implementations use constant time regardless of the size of the file. However, the implementation which employs the directory structure, is about 2 times slower than the trivial implementation. This is attributed mainly to the fact that in order to read a record it needs to read 2 blocks, while the trivial implementation needs only 1.

3.7 Future Work

Optimizing the Directory

As already mentioned, the most performance critical function of the variable-record storage mechanism is the search for a data block where a record with a given length can be saved.

The data structure which we use for the directory is very simple – a linked list and we need to iterate sequentially through the directory blocks when we search for a free data block. The problem with this approach is that when we insert a sequence of records, we need to use the *append* function in order to not always start the search from the beginning of the directory. However, since the records are not sorted by their length, this approach does not guarantee us that if a space with the necessary length exists, it will be used.

One possible solution is to provide additional insert method which inserts a set of records. This method could sort the records according to their length and only then start the insert.

Another possible optimization provides the delete method. When we delete a lot of records, it well can be that they are located in different data blocks which are referenced by different directory blocks. We can again provide a new delete method which has as input a set of *rid*'s. These *rid*'s can be grouped by the logical block address they have and only then perform the actual delete. Further, the *DataBlock* interface can provide a "bulk" delete method, too. The advantage, besides only one function call, is that we need for the update of the block only one I/O operation.

Further optimizations can be applied by changing the data structure, which is used by the directory. For example, we could maintain an in-memory index by the free space of the first n data blocks. When we want to insert a record with length l, we search the index and it returns the block which has enough space to store the record, and all other blocks either don't have enough space, or the space they have is larger than the space in the returned block.

It should be noted, that the goal of these optimizations is not to fill the empty spaces in the data blocks with new records in an optimal way. Finding an optimal distribution of the records across the data blocks is equivalent to the bin packing problem, which is a combinatorial NP-hard problem [JDU$^+$74] and therefore, either requires an excessive amount of time to find an optimal solution, or use an approximation.

Whatever data structure we use, we must always take care that accessing by a *rid* should be as fast as possible, and only then try to optimize the update operations. This is due to the fact, that in general, read operations in a database are much more frequent than updates.

Concurrent Access

Concurrent access to the data is another important feature, which has not been considered. If we bear in mind that the described mechanism works with a single file, then we can allow the execution of many parallel *read*() executions, but only a single write operation (*insert*(), *append*(), or *delete*()). This corresponds to implementing the solution of the well-known reader-writer problem.

On the other hand, if the storage mechanism can distribute records across several files, then it may be possible to run updating operations, which affect different files, in parallel.

Storing Index Id-Lists

An important application of the variable-record storage is persisting the id-lists of an inverted index. An index id-list is a list containing identifiers of records, which have the same index key. Naturally, these lists can be with different length, can be modified, deleted, or new ones can be inserted. Thus, they can be treated as records with variable length. A detailed description of such storage mechanism is described in [ZMSD93]. It is very similar to the approach proposed in the current work. However, this solution is not optimal with respect to update operations, given the following characteristics of the id-lists:

- First, in contrast to a document retrieval system, in an XML database not only the documents are indexed but also the separate attribute values and text nodes. This increases significantly the average size of an id-list, especially when a whole collection of XML documents is indexed, and even when compressed, the probability that an id-list spans several blocks increases.
- Second, as described in [WMB99], the id-lists contain the id-values in a compressed form, usually by first sorting the id-values, and then calculating the differences between each of them, as shown in figure 24.

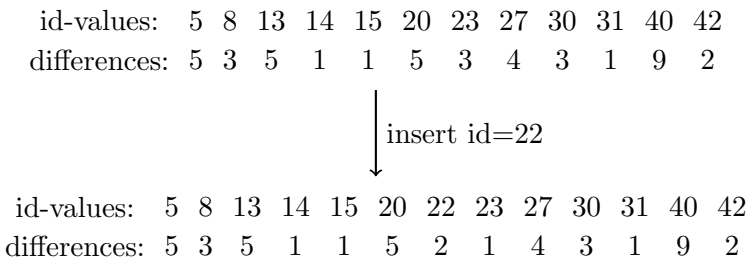


Figure 24: Storing id-lists in compressed form.

So, how do these two factors influence the performance of the update operation? In order to delete or insert a new id-value, we need to first find the position where the corresponding id-value is located or should be inserted. However, since we store only the differences, we cannot apply binary search, even though the values represent a sorted array. Thus, we need to scan the list from the beginning until we find the correct place. For long id-lists, which span several blocks, this might be very expensive, especially given the fact, that most insert operations will actually need to append the id-value to the list, since it is newly generated, and hence bigger than any other. An exception is `replace value of node`, where an existing id-value may move from one id-list to another.

In order to optimize the update of long id-lists, we may store some additional data about the list. For example, if the list spans several blocks, it means that it is split into chunks. Therefore, if we store the first actual id-value in each chunk together with the address of the chunk, we no longer need to scan the id-list from its beginning to locate a given value. Further, if we need to append a new id-value, we always know the address of the block which contains the end of the list. Figure 25 illustrates the idea using the example from figure 24. This approach, however, needs to be further specified and prototyped.

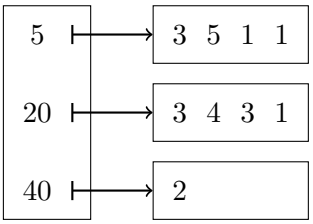


Figure 25: Storing id-lists which span several blocks.

3.8 Summary

In this section we described some problems concerning variable-length record storage and their possible solutions. Since they concern the lower levels of a database management system, they are common to both native XML and relational databases. We showed also that most of the principles for storing variable-length records can be applied to native XML databases, too. However, there are some difficulties and specifics in the implementation, which need to be dealt with. Additionally, the data structures and caching application need to be carefully designed in order to achieve adequate performance.

4 Index Updates

4.1 Introduction

In the previous two sections we described algorithms and data structures, which allow efficient execution in native XML databases not only of read but also of update operations. In the current section we will concentrate our attention on implementing incremental updates in various index structures used in native XML databases.

Similar to relational databases, native XML databases use indexes in order to optimize queries. For example, indexes with all text nodes and all attribute values can be created and then XPath predicates can be evaluated using the corresponding index. These match closely the standard column indexes found in most relation databases. Further, XQuery and XPath support full-text queries, which can be optimized by using special full-text indexes. These indexes are well-known from the relational databases, too.

In contrast to data in relational databases, however, native XML databases deal with semi-structured data, i.e. besides containing data, XML documents define themselves the structure of the data. This means that XML documents do not need to have a specific, predefined structure in order to be processed. However, this flexibility can be also a disadvantage: first, it is difficult for users to create useful queries without knowing the structure of the data, and second, it is difficult for the DBMS to efficiently execute queries. This is why a special type of indexes have been devised, which represent the structure of the data. These indexes are called *path summaries*[BCM05] or *DataGuides*[GW97]. We will use the term *path summary*, since it is more wide-spread in the XML community.

In the following sections will present some specifics regarding index updates in native XML databases, as well as their practical application in the open source native XML database system BaseX.

4.2 Text and Attribute Values Index

Introduction

We will begin with the most well-known type of index – the value index. XML documents contain their data in the text nodes and attribute values (processing instructions and comments can be processed, if needed, as text nodes). These values usually participate in query predicates with comparison operators such as `=`, `<`, `>`, etc. Such operators can be evaluated very efficiently using a standard inverted index, which uses a tree structure for the keys and id lists.

Maintaining such index during database updates is also relatively standard. Updating the id lists has been discussed in the previous section and there are numerous tree data structures which support search and update

operations with logarithmic complexity. The most widely used data structure is the B+-tree and its variants [Com79].

These standard data structures can be used for indexing text nodes and attribute values. However, executing update operations causes some additional effects, which need to be handled by implementations. As already mentioned in the Section 2 ID-PRE Mapping, indexes need to use stable node identifiers. Therefore, in order to return nodes in document order when searching for a key, we need to translate the id values to pre values first, i.e. we need to use an ID-PRE Map.

Further, when performing an insert operation, many new records are inserted – some represent already existing keys in the index, some are completely new. In both cases delaying the insert of new entries in the index and caching them can bring performance improvements, because we will be able to add several new id values with a single operation to an id list corresponding to a given key.

Similar reasoning can be applied when deleting nodes – caching the id values to be deleted means less update operations in the index and its id lists.

Value Indexes in BaseX

Further problems can be identified when a concrete database implementation is considered. The native XML database system BaseX uses as an index structure a simple array sorted by the index keys. Besides being simple to implement, this approach provides a very good performance for read-only databases. However, updating the index is not as efficient as it would be with, for example, a B+-tree. Nonetheless, support for incremental updates exists and reasonable speed is achieved thanks to the optimization techniques presented below.

The fact that we have to deal with a sorted list of keys means that when inserting new keys, the list need to be resized. Therefore, we need to shift all bigger keys to the right. Here we can again make use of buffering the inserted records. Additionally, we sort the inserted keys and we search the index in order to identify which keys are new. The advantage we gain from these pre-processing steps is that we need to perform the shifting of existing keys only once. Let us consider the example in figure 26.

We need to insert 7 new records with keys a, c, f, and g. We have sorted and grouped the records in the pre-processing step and we have identified that the keys c, f, and g do not exist in the index. This means that we insert 3 new keys. Then starting from the end of both key lists, we compare the keys pair-wise – one key from the index and one key from the new ones. Since h is bigger than g, we move it 3 positions to the right. Then we insert keys f and g with their corresponding id lists. Then we move d and e 1 position to the right, and so on until all new records have been inserted. If

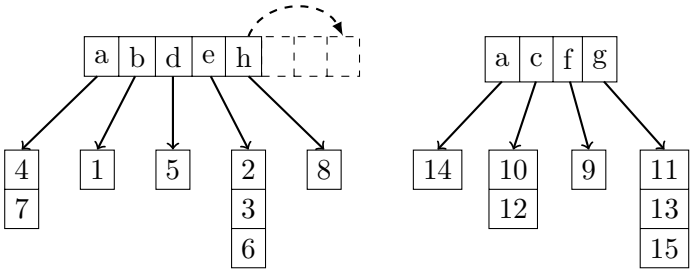


Figure 26: Inserting new records in BaseX value index.

the two keys are equal, such as the case with a, then we insert the new id values in the existing id list. The result is shown on figure 27.

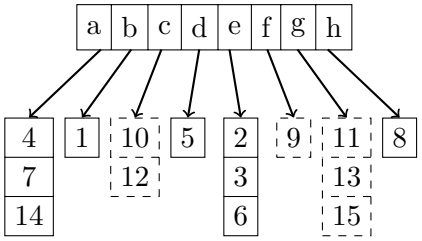


Figure 27: Result of inserting new records in BaseX value index.

If there are no new keys, sorting the input keys can nonetheless bring advantage, since we can more efficiently perform the binary search for each subsequent key, by limiting the search to the position of the previously found match.

Similar approach can be used when implementing the delete operation. We need to collect all records which are being deleted and sort their keys. This time we start traversing the list with deleted keys from the beginning. We find the first key from the input list in the index and delete the corresponding id values from the id list. If the id list remains empty, then we need to delete the key. This means that all subsequent records, which are smaller than the next deleted key, should be shifted to the left. If not, then we perform again binary search for the next input key and so on until all input keys are processed.

Performance of Value Indexes in BaseX

An interesting question is what the performance impact on database updates is when using these data structures as indexes. Figure 28 shows the results of a performance test conducted with a relatively big database in which the value index contains around 1.6 million records and around 508 thousand keys.

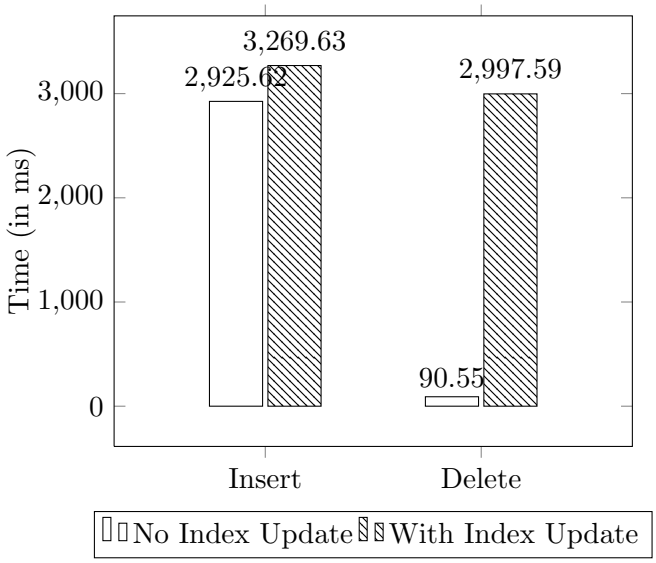


Figure 28: Value index update performance in BaseX.

The measurement results in the case of insert show clearly that, compared to the overall performance, updating the index is not significant. However, the difference for delete is huge. The reason for this is that BaseX executes very efficiently delete operations by first removing records only from its main data structure (which is very fast since all records are stored sequentially in document order) and then updating the distance values of all ancestors and following siblings of the deleted node. This advantage is lost when the indexes need to be updated, because we need to analyze all deleted nodes and collect the values of all text and attribute nodes. After that we need to update the index itself. This difference in the performance, although not that huge, should be expected regardless of the used data structure. However, the performance impact may be worth it, when we strive to optimize read-only queries, which profit from value indexes.

4.3 Path Summary

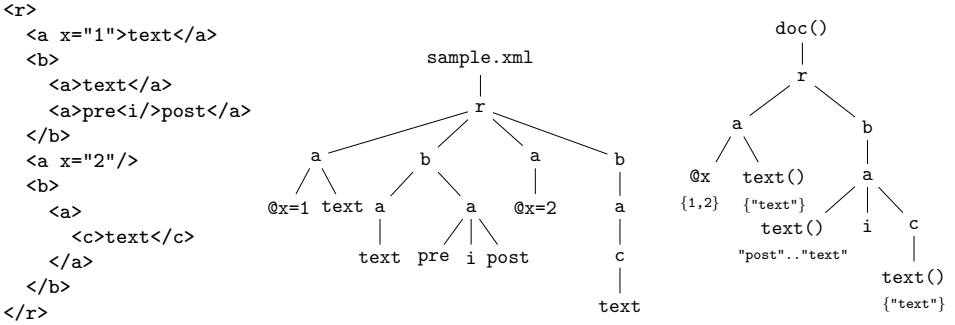
Introduction

The next index type, which we will analyze, is the path summary. As already mentioned, a path summary represents the structure of an XML document. Intuitively it can be defined as the tree of the XML document, in which, however, all nodes from a given level are represented by a single node, if all of the following conditions are true:

- they have a common parent in the path summary
- they are from the same type (i.e. element, attribute, text node, etc.)

- they have the same name (in the case of element and attribute).

An example is given by figure 29.



(a) XML document (b) XML document tree (c) Path summary

Figure 29: Path summary of an XML document.

This definition corresponds to the definition of *strong DataGuide* from [GW97]. However, DataGuides are more general than the path summary we present here, because they are designed for graph-structured databases. While XML documents are trees, references between nodes can be created using ID/IDREFs, which makes essentially an XML document a graph. We will, however, ignore this possibility, which will simplify significantly the used algorithms.

Statistics

Besides defining the structure of an XML document, a path summary may also contain additional information about the nodes, which may be useful to the query optimizer. A typical example is the number of nodes, which are reachable with a given path. This is analogous to the number of rows in a table of a relational database. If we continue further with this analogy, we can also store data about the values (text nodes and attribute values). For instance, the attribute `/r/a/@x` from figure 29 has only integer values and they are only 1 and 2. On the other hand, the node `/r/b/a/text()` contains text values the smallest one of which is `"post"` and the biggest `"text"`.

In contrast to relational databases, XML documents may not have a defined schema. This means that the `@x` attribute may as well have a text value. Therefore, as a meta-information, we can also store the type of data deduced from the existing values in the database.

Storing further meta-information is of course possible, but, beside the node count, we will limit our analysis to the following categories for each value (i.e. text node or attribute value):

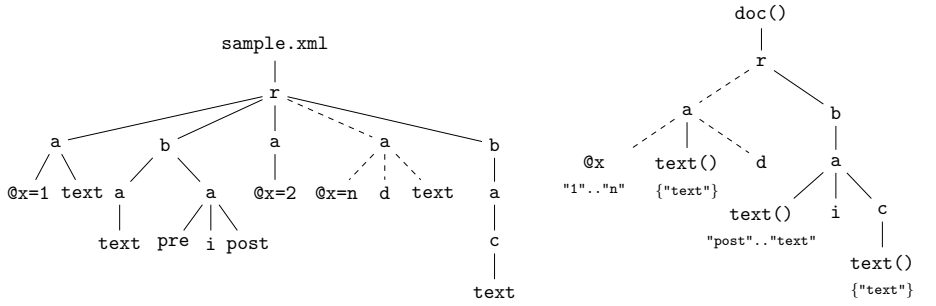
- data type: specific type such as numeric, date, etc, or general text

- enumerating all distinct values together with their distribution (if the number of distinct values is smaller than a given boundary), or storing only the minimal and maximal value, if there too many distinct values.

Updating Path Summary

Updating strong DataGuides has been described in detail in [GW97]. However, as already said, the algorithms needed in the case of path summaries are simpler. More specifically, we need to consider only the cases, in which a new sub-tree is inserted or an existing one is deleted. The only operation from the XQuery Update Facility, which may cause complications, is `rename node`. However, it can be implemented by first deleting a the node and then re-inserting it with the new name.

We will now describe how a given path summary can be updated. We will begin with inserting a new tree at a given position, using figure 30 as an example.



(a) XML document tree (b) Path summary

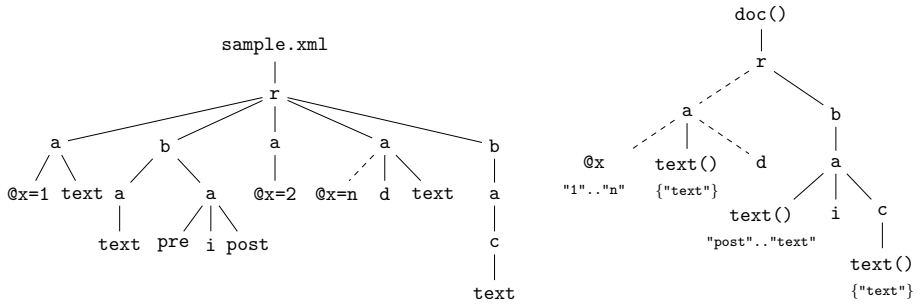
Figure 30: `insert node <a x="n"><d/>text</a> after /r/a[@x=2].`

The path summary insert procedure in this case runs as follows. First, we identify the parent of the sub-tree in the path summary using the target node from the insert query. In our case this the node with label `r`. Then we check if there is a child node of `r` with label `a`. Since this is the case, we increment the number of XML nodes represented by this path summary node. We continue further with the attribute `@x="n"`. In the path summary, a node `@x` exists which in addition shows that all corresponding attribute values contain only one of the numbers 1 and 2. The new attribute value is, however, `"n"`, which means that we have to change the type from numeric to text. Further, if we assume that the minimal number of distinct values which we can store is only 2, then we are not able to store all 3 distinct texts. Thus, we need to store only the minimal and maximal values for these node.

The rest of the new XML nodes can be recorded in the path summary analogously. This procedure is almost the same as the procedure used by

building the path summary from scratch. The only difference is that we need to locate the parent node first.

The important conclusion which we can make, concerning statistics about values, is that when a new node is being added, the constraints described by the statistic values will either remain the same, or will be relaxed (which is natural since the set of described elements is larger).



(a) XML document tree (b) Path summary

Figure 31: `delete node /r/a/@x[.="n"]`.

Let us now consider the delete operation `delete node /r/a/@x[.="n"]` applied to the XML tree in figure 31. After executing this operation, the nodes `/r/a/@x` will again have only the numeric values 1 and 2. However, in order to determine that, we need to check all nodes to assert that this is really the case – a price that we may not be willing to pay, because it can make the update operation very slow. Thus, even though the delete operation decreases the set of nodes, we cannot narrow the constraints and the path summary remains unchanged.

Such inaccuracies need to be reflected in the path summary with a flag, which shows whether the statistics represent precisely the data, or not. This flag can be global, i.e. for the whole path summary. However, an improvement would be to store such flag in each node in the path summary. This will allow us to take full advantage of statistics, which remain exact after an update operation.

4.4 Full Text Index

One type of indexes, which we have not considered, are full-text indexes. Full-text indexes are used in the field of information retrieval. Usually they are inverted indexes, which, in contrast to the above discussed text and attribute value index, use as keys not the whole text value, but rather the single tokens of which it consists. Full-text indexes can be rather complex, first because they contain a lot of data, and second because there are numerous query options, which they may be able to accelerate. Examples of such options are stemming, scoring, filtering by language, boolean expressions,

etc. This is why implementing an efficient full-text index is hard and even harder is implementing efficient updates.

Full text indexes can be implemented as inverted indexes storing the tokens in a balanced tree structure and the id lists as variable-length records. However, in addition to the id values, the positions of the tokens in each text node can be saved, too. This means, that for each indexed token, we need to store the id values of each text node, where the token occurs, as well as the positions of the token in each text node. Thus, an id list for a key is actually a list of records, where each record has a text node id and a list of positions. Figure 32 describes the structure of such a list.

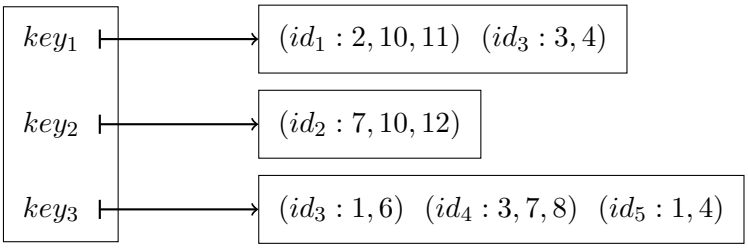


Figure 32: Storing id values and positions in a full-text index.

Besides complicating the storage layout, storing the position increases significantly the overall size of the index. Therefore, it is needed to compress position values similar to the approach described in section 3.7. Further, as also described in that same section, in order to improve the performance when updating large lists, we need to again store some of the id values uncompressed form together with references to the chunks from which the list consists. Thus, we can quickly identify the chunk we need and update only it. Figure 33 shows that using the example from figure 32.

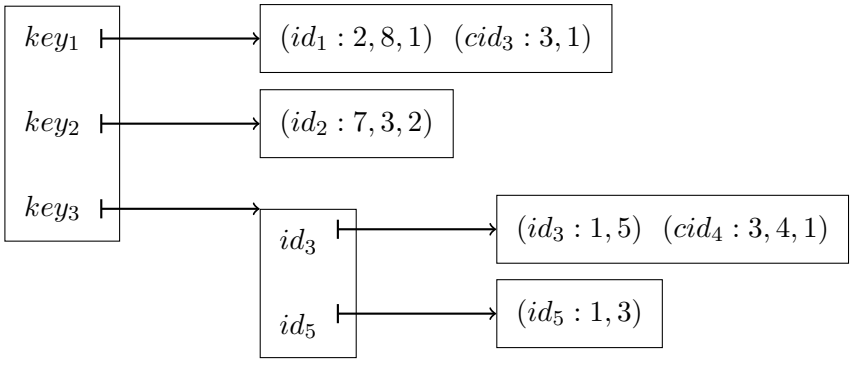


Figure 33: Storing id values and positions in a full-text index, so that updates are more efficient.

We will now consider the concrete full-text index implementation of BaseX. Its full-text index first groups all tokens by their length and then creates

a separate sub-index for each group (figure 34 depicts the index structure).

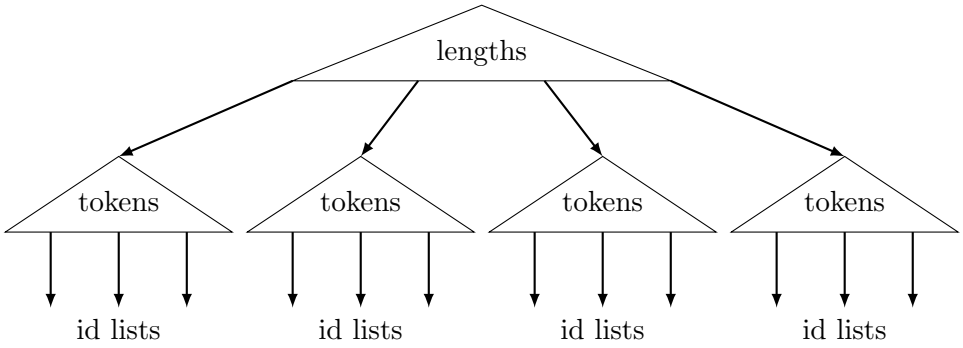


Figure 34: Full-text index structure in BaseX.

This index structure allows efficient evaluation of wildcard searches as well as queries with approximate string matching based on edit distance, such as the Levenshtein distance. However it is tailored to perform efficiently only in read-only databases, because it uses sorted arrays with binary search. This can be improved by using a B+-tree, or applying techniques similar to the ones demonstrated in section 4.2, where we have described how the BaseX value index can be updated. Implementing a prototype of such an index is out of scope of this thesis.

4.5 Index Updates and the Database System

Implementing index structures that allow efficient updates is not the only factor that determines the overall performance of the system. An important question is how to integrate such components with the rest of the system. We saw that each index type uses different types of nodes from the XML document. This means that the best alternative is to use an event based mechanism, which processes inserted or deleted nodes and notifies registered listeners. This approach allows asynchronous index updates, if sufficient system resources are available.

When inserting new nodes, BaseX uses a mechanism similar to the event-based one (the delete operation does not traverse the deleted nodes, when index updates are not needed). The current index update mechanism is not executed asynchronously by the main update process, because the index update relies on the fact that all inserted or deleted nodes are first buffered. When inserting or deleting huge number of nodes, the buffer may consume a considerable amount of the system memory. This is why, its size needs to be limited and several index update operations need to be executed per one database operation.

4.6 Future Work

Partial Indexes

Partial indexes are indexes which contain only part of the whole database defined using an expression (they may be considered analogous to materialized views from relational databases). Updating a partial index is difficult, because inserted or deleted nodes need to be checked for nodes, which correspond to the partial index condition. Efficiently performing such updates is subject to future research.

Typed Indexes

As explained earlier, if not constrained to a given schema, XML documents are not typed. However, in many cases some nodes always contain only values of a certain type. In this case a partial index can be defined on these nodes, however, comparing them not as texts, but as typed values, either provided by the user or guessed from the path summary. On the other hand, when a new node is added to the index, but has different type than the rest of the index keys, then the index becomes invalid. The simplest solution is to discard the index. Another more useful solution would be to maintain a second index, ordered by the newly introduced type. Thus, the overhead of indexes will be twice as much, but we can still benefit from the old typed index. These and other possible solutions need to be further developed.

Coupling Value Indexes with Path Summary

Another interesting topic is coupling the value index with the path summary. This means that for each id value stored in the value index representing a text node or an attribute value, additionally a reference to the corresponding path summary node is stored. The motivation behind this proposal is that queries such as

```
/descendant-or-self::*/*:x[text() = "y"]
```

can be re-written to

```
db:text("y")/parent::*:x[parent::*],
```

where the function `db:text()` performs a look up in the value index and returns all matching text nodes. The rest of the path expression can be evaluated using the path summary, if the nodes returned from the value index contain references to their corresponding nodes in the path summary. Thus, such queries, which occur quite often, can be evaluated by using only index access and without accessing the main data structure.

This optimization comes, however, to a price. First, the size of value index increases, because the path summary references need to be stored next

to each id value. Second, when inserting a new record in the value index, the corresponding path summary reference needs to be already known. This means that the path summary should be always updated before the value index, and this further means that both updates cannot be executed in parallel.

4.7 Summary

In this section we have presented several approaches for updating various kinds of indexes in native XML databases. First, we have described an index on all text nodes and attribute values, which uses a traditional approach, and we have shown how an update mechanism can be implemented, even though the used data structures are not optimal with respect to updates. Second, we have presented how structural and meta-data can be updated, when stored in a path summary. Finally, we have discussed some specifics related to updating full-text indexes and how index updates are integrated into the XML database management system.

5 Conclusion

Several storage data structures have been presented in this thesis. Each of them targets a specific problem related to data updates in native XML databases.

The first such structure is the ID-PRE Map, which allows efficiently finding the pre position of a node given its unique id value in an XML database affected by updates. The algorithm is described both intuitively and formally, so that it can be easily comprehended and provide the necessary basis for formal proof of its correctness. The performance of the approach is good even though there are many optimization possibilities in the implementation. This implementation is integrated into the open source database system BaseX and although still experimental, the first real case results are promising.

Another topic discussed in this thesis is storing variable-length records. Although there are numerous publications on this problem, implementing such mechanism is not easy and straightforward. Special attention has been paid to optimizations, which allow performance of the most important operations comparable to the performance of a straightforward but very fast implementation.

The first two mechanisms are an important prerequisite for implementing index updates. In the last part of this thesis, approaches for updating three kinds of indexes have been presented. For traditional inverted indexes, such as the text and attribute value indexes and the full-text indexes, an important component, beside the key data structure, is the mechanism for storing the record references (the so-called id lists). We have described how the mechanism for storing variable-length records can be used to store such records and additionally, improvements to the standard solutions have been proposed.

The main goal in each case has been to achieve maximum performance when executing read operations and at the same time provide reasonably fast updates. While there is still a lot of room for optimizations, most of the approaches have been implemented or will be implemented in the XML database system BaseX, since their usefulness can be fully proved only by their practical application.

References

[BCM05] Attila Barta, Mariano P. Consens, and Alberto O. Mendelzon. Benefits of Path Summaries in an XML Query Optimizer Supporting Multiple Access Methods. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *VLDB*, pages 133–144. ACM, 2005.

[BGvK+06] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered By A Relational Engine. In *Proceedings of ACM SIGMOD International Conference on Management of Data 2006*. ACM, 2006.

[Com79] Douglas Comer. The Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[FGK06] Andrey Fomichev, Maxim Grinev, and Sergei D. Kuznetsov. Sedna: A Native XML DBMS. In Jirí Wiedermann, Gerard Tel, Jaroslav Pokorný, Mária Bieliková, and Julius Stuller, editors, *SOFSEM*, volume 3831 of *Lecture Notes in Computer Science*, pages 272–281. Springer, 2006.

[FHK+02] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele, and Till Westmann. Anatomy of a native XML base management system. *VLDB J.*, 11(4):292–314, 2002.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*, pages 436–445, 1997.

[JDU+74] David S. Johnson, Alan J. Demers, Jeffrey D. Ullman, M. R. Garey, and Ronald L. Graham. Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms. *SIAM J. Comput.*, 3(4):299–325, 1974.

[RG00] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, second edition, 2000.

[WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, CA, second edition, 1999.

[ZMSD93] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. Storage Management for Files of Dynamic Records. In *Australian Database Conference*, pages 26–38, 1993.