Main Memory XML Update Optimization: algorithms and experiments.
Sahakyan Marina

To cite this version:
Sahakyan Marina. Main Memory XML Update Optimization: algorithms and experiments..
UNIVERSITY PARIS SUD 11 - ORSAY

PHD THESIS

to obtain the title of

Dr. of Science
(informatics)

Defended on November 17, 2011 by
Marina SAHAKYAN

Main Memory XML Update Optimization: algorithms and experiments

Jury:

President:  Pr. Anne VILNAT  University Paris Sud 11 - LIMSI
Reviewers:  Pr. Anne DOUCET  University Paris 6 - LIP6
            Pr. Irene GUESSARIAN  University Paris 6 - LIAFA

Thesis Advisors:  Pr. Nicole BIDOIT-TOLLU
                 University Paris Sud 11 - LRI - INRIA

Assistant Pr. Dario COLAZZO
University Paris Sud 11 - LRI - INRIA

Joint advisors:  Assistant Pr. Hrachya ASTSATRHYAN
                National Academia of Science of Armenia

                Pr. Gevorg MARGAROV
                State Engineering University of Armenia
Acknowledgments
Main Memory XML Update Optimization: algorithms and experiments

Abstract:
XML projection is one of the main adopted optimization techniques for reducing memory consumption in XQuery in-memory engines. The main idea behind this technique is quite simple: given a query $Q$ over an XML document $D$, instead of evaluating $Q$ on $D$, the query $Q$ is evaluated on a smaller document $D'$ obtained from $D$ by pruning out, at loading-time, parts of D that are irrelevant for $Q$. The actual queried document $D'$ is a projection of the original one, and is often much smaller than $D$ due to the fact that queries tend to be quite selective in general.

While projection techniques have been extensively investigated for XML querying, we are not aware of applications to XML updating. This Thesis investigates application of a projection based optimization mechanism for XQuery Update Facility expressions in the presence of a schema. The current work includes study of the method and a formal development of Merge algorithm as well as experiments testifying its effectiveness.

Keywords: XML, XML Updates, XML Projection
## Contents

1 Introduction ......................................... 3  
1.1 Problem statement ................................. 3  
1.2 Main contribution ................................. 4  
1.3 XQuery engines supporting updates ............... 5  
1.4 Related work .................................... 6  

2 Preliminaries ......................................... 9  
2.1 XML ............................................. 9  
2.2 XQuery Update Facility (XUF) .................... 14  
  2.2.1 Simple updates ............................... 14  
  2.2.2 Complex Updates .............................. 16  
  2.2.3 Constraints and semantics .................... 17  
  2.2.4 Snapshot semantics ......................... 18  

3 State of the Art, XQuery Engines .................. 21  
3.1 Introduction .................................... 21  
3.2 MonetDB/XQuery ................................. 22  
  3.2.1 General Data structure ....................... 23  
  3.2.2 Data structure supporting structural updates 26  
  3.2.3 XML Serialization ............................ 32  
  3.2.4 MonetDB/XQuery vs. projection ............... 33  
3.3 Native XML databases ............................. 34  
  3.3.1 BaseX ...................................... 34  
  3.3.2 General Datastructure ....................... 34  
  3.3.3 Data structure supporting structural updates 36  
  3.3.4 XML Serialization ............................ 40  
  3.3.5 BaseX vs. projection ......................... 40  
3.4 eXist ........................................... 42  
  3.4.1 General Data Structure ....................... 43  
  3.4.2 XML serialization ............................ 46  
  3.4.3 eXist vs. projection ......................... 47  
3.5 Saxon Processor .................................. 48  
  3.5.1 General Data Structure ....................... 48  
  3.5.2 XML Serialization ............................ 52  
  3.5.3 Saxon vs. projection ......................... 52  
3.6 Conclusion ...................................... 52
Chapter 1

Introduction

Contents

1.1 Problem statement .................................................. 3
1.2 Main contribution .................................................... 4
1.3 XQuery engines supporting updates ............................... 5
1.4 Related work .......................................................... 6

Recent years have seen the rapidly emerging of XML query and transformation languages, due to the vast class of applications where XML plays a central role. Examples are Web applications, data integration, and P2P distributed database systems.

In these contexts, one of the main emerging needs is the ability to update large XML data sets. There are several proposals of XML update languages, all of them based on extension of XQuery. The most relevant one is that proposed by the W3C in the XQuery Update Facility current draft. This specification states what kind of updates can be applied to XML documents, by formalizing the semantics of the proposed operations, by taking into account only the effects on the data present in main memory. Issues related to the problem of making updates persistent and efficiently executed are not dealt with, and left to the implementation. Addressing these issues is of crucial importance as, very often, the size of XML documents to update can become quite large, and update operations can be quite complex, due to both the intrinsic irregular nature of XML data and to the rich expressiveness of the XQuery update language.

1.1 Problem statement

XML projection is a well-known optimization technique for reducing memory consumption of XQuery in-memory engines. The main idea behind this technique is quite simple: given a query \( q \) over an XML document \( t \), instead of evaluating \( q \) over \( t \), the query \( q \) is evaluated on a smaller document \( t' \) obtained from \( t \) by pruning out, at loading-time, parts of \( t \) that are not relevant for \( q \). The queried document \( t' \), a projection of the original one, is often much smaller than \( t \) due to selectivity of queries.

In order to determine an optimal projection of \( t \) several approaches exist [15, 21, 33, 36]. Most of them are based on query path extraction: all the paths expressing
the data-needs for the query $q$ are first extracted and then used for projecting $t$. In particular, the type based approach [15] assumes that documents are typed by a DTD and combines path extraction with type inference, to determine the type names (labels) of the elements required for the query. This set of type names is dubbed type-projector, and used at loading time to prune out elements whose type labels do not belong to it.

While projection techniques have been extensively investigated for XML querying, we are not aware of any application to XML updating, although several XML querying engines like Galax [3], Saxon [7], QizX [5, 4], BaseX [29, 30] and eXist [2] perform updates in main-memory: the input document is first loaded in main-memory, then updated, and finally stored back on the disk. As a consequence, each one of these systems has some limitations on the maximal size of documents that can be processed. For instance, we checked that for eXist, QizX/open [5] and Saxon it is not possible to update documents whose size is greater than 150 MB (no matter the update query at hand) with standard settings and memory limitations.

XML projection, as described above, cannot be applied directly for updating XML documents. Obviously, updating a projection of a document $t$ is not equivalent to updating the document $t$ itself: the pruned out sub-trees will be missing.

1.2 Main contribution

Our main contribution is that we develop a type based optimization technique for updates. Our update scenario is designed as follows for an update $u$ and a document $t$ typed by a DTD $D$.

- First, we build the projection $t'$ of $t$ using a type-projector $\pi$.
- Then we evaluate the update $u$ over the projection $t'$ and obtain the partial result $u(t')$.
- Finally we process the last step, called Merge, parses in a streaming and synchronized fashion both the original document $t$ and $u(t')$ in order to produce the final result $u(t)$.

For the sake of efficiency, the Merge step is designed so that (a) only child position of nodes and the projector $\pi$ are checked in order to decide whether to output elements of $t$ or of $u(t')$ and (b) no further changes are made on elements after the partial updated document $u(t')$ has been computed: output elements are either elements of the original document $t$ or elements of $u(t')$.

We would like to emphasize that our scenario is totally independent of any particular engine (Saxon, eXist, BaseX and etc.) Our framework lies on the fact that the new technique can be used with any in-memory engine, since it does not require any change in the internal algorithms of the engine itself, nor it requires query rewriting. To make some preliminary tests, we have implemented the proposed projection and merging algorithm in Java.
The main contributions are the following ones:

- i) Design and implementation of a simple and thus efficient algorithm Merge, to make updates persistent. The Merge algorithm uses a buffer whose size is upper bounded by the maximal depth of the input document $t$. This algorithm uses three−level type projector (work of Mohamed-Amine Baazizi).

- ii) Extension of three−level type projector which optimizes memory savings.

- iii) Design and implementation of the extension of the Merge algorithm.

- iv) Extensive experiments whose results validate the efficiency of the proposed approach. We have implemented the projection and merging algorithms in Java and considered several popular systems to perform tests.

1.3 XQuery engines supporting updates

There exist several XQuery engines supporting updates. Well know and most effective once among them are MonetDB/XQuery, BaseX, eXist, Qizx and Saxon. Chapter 3 provides detailed explanation of data structure of these engines. To store XML document all these engines map XML data to certain storage data structure.

To map XML data on the disk MonetDB/XQuery uses relational XML encoding. This encoding aims reduce main-memory consumption and decrease query evaluation time. To update node $n$, MonetDB/XQuery loads pages in the rid table. When MonetDB/XQuery finds a page containing a node matching to the target path, MonetDB/XQuery modifies the found page to make intended updates. Then MonetDB/XQuery writes back modified pages to the disk at actual points. The important point is that MonetDB/XQuery writes back only modified pages (therefore, the second part is also true). In general, disk-write is more time-consuming activity than disk-read. MonetDB/XQuery architecture is carefully designed to minimize disk-write.

BaseX is very efficient for memory savings. Similarly to MonetDB/XQuery BaseX uses relational encoding to map XML Documents. BaseX, to save memory, depending on the kind of the node element, attribute or text stores different references of the node properties in the same column. For instance, in the same column for the element preserves number of attributes and number of children, while for the text and attribute nodes it stores references to the corresponding values.

eXist stores XML documents in hierarchical collection. As a storage unit on a disk it uses $B+Tree$. To insert a node at a target $n$ eXist first retrieves the set of types corresponding to the target path and then evaluates query on them.

To evaluate an update query Saxon maps XML data to DOM like Objects, it is very efficient for the small documents, but quite memory consuming for the big ones.
After analyzing the data structures of these engines we were capable to prove that applying our method helps to optimize the memory limitations issues.

1.4 Related work

The approach here presented introduces substantial novelties wrt the type based approach for queries presented in [15]. As it will be explained in Chapter 4, we adopt a three-level projector, while the projector proposed in [15] is one level. A three level projector allows to optimize (minimize) the size of projection. In particular, it allows to avoid keeping in the projection useless text nodes that would be kept with the technique proposed in [15]: this can result into substantial improvements since in many cases large parts of documents consist of textual content.

Other works propose techniques to optimize update execution time by using static analysis in order to detect independence between several update operations, so that query rewriting techniques can be used for logical optimization [27, 28, 12, 13]. Our work is definitely orthogonal wrt this line of research, and indeed, the two techniques can be combined in order to increase the efficiency in terms of time.

Some recent works [24, 25] addressed the problem of translating an XQuery update expression $u$ into a pure query expression $Q_u$, with the aim of executing the update $u$ via the query $Q_u$. The advantages of these approaches are that updates can be executed even if the XQuery engine only deals with queries, and well established query-optimization techniques can be adopted to optimize update execution. A peculiar characteristic of these approaches [24, 25] is that the query $Q_u$ needs to select and return all nodes that are not updated, while those which are updated are selected and processed to compute new nodes. As a consequence, using standard projection techniques [15, 33] for the query $Q_u$ would lead to no improvement, since the whole document would be projected.

It is worth observing that, although not directly, existing projection techniques [15, 33] could be used for a single update, provided that the projected document is used only to compute the update pending list, so that this last one can be then propagated to the input document in a streaming fashion [22]. Such approach would require some techniques similar to those here developed in order to: opportunely determine the projection, and make node identity persistent in order to propagate, in the second phase, the calculated update pending list. This approach has two drawbacks. Firstly, it does not allow to use XML querying engines in a straight manner as we propose to do: controlling the two phase evaluation of XML updates would become necessary. Secondly, this approach would perform very inefficiently in the quite frequent case where a bunch of $n$ updates has to be executed, according to a given order, because each update would need to be fully processed one after the other entailing the document to be processed/parsed $n$ times.

Our approach is different and allows to evaluate the $n$ updates by processing our method just once: a global projector can be easily inferred (it is sufficient to consider the union of each update projector); the $n$ updates are evaluated on the global
1.4. Related work

projection wrt the specified order; finally, the updates are propagated on the original document in a single pass, using one of the \textit{Merge} functions. As testified by our tests (Chapter 4), this results in a much more efficient processing.

\textbf{Organization} \hspace{0.5cm} The Thesis is organized as follows.
Chapter 2 introduces XML and XQuery Update Facility and provides some basic notifications and definitions.
Chapter 3 examines the Data Structure of well know XQuery engines supporting updates. For each engine the optimizations of using our method while an update query evaluation are reported.
Chapter 4 introduces the main features of our method and \textit{Merge} algorithm through examples. The last section of the Chapter reports the implementation and experiments of the \textit{Merge} algorithm.
Chapter 5 introduces the extension of the method both for the \textit{three-level} type projector and \textit{Merge} algorithm. The implementation and experiments of the extended algorithm are reported in the last section of the Chapter.
Chapter 2

Preliminaries

Contents

2.1 XML ................................................................. 9
2.2 XQuery Update Facility (XUF) ................................. 14
   2.2.1 Simple updates ............................................. 14
   2.2.2 Complex Updates ......................................... 16
   2.2.3 Constraints and semantics ............................... 17
   2.2.4 Snapshot semantics ....................................... 18

The Chapter is organized as follows. In Section 2.1 we introduce some basic notions about XML and provide some basic notations and definitions. In Section 2.2 we introduce XQuery Update Facility: simple and complex updates, snapshot semantics including update primitives and pending update list.

2.1 XML

During the last decade, fast developing and widely used web applications have centered their main functionalities around the management of semistructured data. XML (eXtensible Markup Language) is Semi-Structured Data format (SSD) which is used to manage data whose structure can be highly irregular, can change over time and provides users a high flexibility to exchange different types of data. XML, was developed by an XML Working Group (originally known as the SGML Editorial Review Board) formed under the auspices of the World Wide Web Consortium (W3C) in 1996 [8].

XML is very flexible, which makes it able to easily model the various kind of data format that are present over the Web: HTML data, relational and object database data, structured and unstructured textual data, audio and video data, and etc. Each XML document has both a logical and a physical structure.

According to W3C, the basic component of an XML is the element, which is defined as a piece of text enclosed by open-tag (e.g. <country>) and its corresponding close-tag (<country/>). The following is an example of XML element:

   <country> Singapore </country>.
The content of each XML element takes one of three essential forms: simple text value, a sequence of elements, or a complex sequence which includes the two previous forms: text values and elements.

For the sake of simplicity, we restrict our study to element declarations and omitted the treatment of others such as attributes. Figure 2.1 illustrates the textual representation of a simple XML document. It shows that element nodes are denoted by markup tags. For example, the open-tag `<a>` and the close-tag `</a>` represent an XML element, and the text value "oof" included between both of them refers to the content of this XML element. Elements that do not contain text content are called empty element, such as `<c/>`, `<f/>` and `<g/>`. The elements `<d>` represents a complex element which includes empty elements: `<f/>` and `<g/>`.

Elements can be annotated with attributes that contain meta data about the element and its contents. For simplicity we do not consider attributes in this study (our results can be easily extended to attributes).

```
<docexample>
  <a>
    <b> oof </b>
    <c/>
    <c/>
    <d>
      <f/>
      <g/>
    </d>
  </a>
  <a>
    <d>
      <f/>
      <g/>
    </d>
  </a>
</docexample>
```

Figure 2.1: Textual representation of `docexample.xml`

Figure 2.2 illustrates a graphical tree representation of the XML document given in Figure 2.1. Tree representations are useful for understanding the structure of the XML document, and are also used inside engines to define navigational mechanisms.

In this Thesis we rely on a store-based representation of XML trees. Stores are defined in the following, along the lines of [14].

$I, J, K$ designate sets (id-set) or lists (id-seq) of identifiers denoted by $i, j ...$; ()
2.1. XML

denotes the empty id-seq; \( I \cdot J' \) denotes id-seq composition, and the intersection of \( I \) and \( J \) preserving the order in the id-seq \( I \) is denoted by \( I \cap J \).

A store \( \sigma \) over the id-set \( I \) is a mapping associating each identifier \( i \in I \) with either an element node \( a[J] \) or a text node \( text[st] \) where \( a \) is a label, \( J \) is an id-seq of identifiers in \( I \) (the ordered list of children) and \( st \) is a string. We define:

- \( lab(i) = a \) if \( \sigma(i) = a[J] \), and \( lab(i) = \text{String} \) if \( \sigma(i) = text[st] \),
- \( child(\sigma, K) = \{ j \mid \exists i \in K, \sigma(i) = a[J] \text{ and } j \in J \} \),
- \( roots(\sigma) = \{ i \mid \exists j, i \in child(\sigma, \{ j \}) \} \).

It is worth noticing that we also decorate each node with unique identifier which is calculated by appending to the identifier of the parent node the delimiter "." followed by the numeric value representing the position of the node in the current level. For instance the identifier of \( b \)-node of the tree \( t \) is assigned to \( 1.1 \), since the identifier of its parent \( a \)-node is equal to \( 1 \). The identifier of the root node is set to \( \varepsilon \). A node of a document \( t \) whose identifier is \( i \) is next denoted by \( t @ i \).

The following example defines the store for the document \( t \) of Figure 2.2.

\textbf{Example 2.1.1.} The document \( t \) of Figure 2.2 is a store \( \sigma \).

Its id-set is \( I = \{ \varepsilon, 1, 1.1, 1.2, 1.3, 1.4, 1.4.1, 1.4.2, 2.1, 2.1.1, 2.1.2 \} \).

For example, \( \sigma(\varepsilon) = \text{doc}[1, 2], \) \( \sigma(1) = a[1, 1.2, 1.3, 1.4], \) \( \sigma(1.4) = d[1.4.1, 1.4.2], \) \( \sigma(1.1) = \text{text[oof]}, \) \( \sigma(2) = a[2.1], \) \( \sigma(2.1) = d[2.1.1, 2.1.2]. \)

We have that \( lab(\varepsilon) = \text{docexample}, \) \( lab(1) = a, \) \( lab(2) = a, \) \( lab(1.1) = b, \) \( lab(1.2) = c, \) \( lab(1.3) = c, \) \( lab(1.4) = d \) and etc.

And finally, \( child(\sigma, \{ 1 \}) = \{ 1.1, 1.2, 1.3, 1.4 \}, \) \( child(\sigma, \{ 2 \}) = \{ 2.1 \}, \) \( child(\sigma, \{ 2.1 \}) = \{ 2, 1.1, 2.1.2 \}, \) \( child(\sigma, \{ 1.4 \}) = \{ 1.4.1, 1.4.2 \} \) and \( roots(\sigma) = \varepsilon \).

We only consider stores corresponding to XML forests and trees. A forest \( f \) over \( I \) is given by a pair \( (J, \sigma) \) where \( \sigma \) is as above and \( J = roots(\sigma) \). We write \( \text{dom}(f) \) for \( I \) and \( \sigma_f \) for \( \sigma \) and \( f \sigma_f' \) for the concatenation of two disjoint forests \( f \) and \( f' \).

\textbf{Example 2.1.2.} The document \( t \) of Figure 2.2 is a store \( (roots(t), \sigma) \) where \( roots(t) = \varepsilon \). Obviously, it is a tree. The sub-forest of this store is composed of the two trees of \( \sigma \) rooted respectively at \( t @ 1 \) and \( t @ 2 \).

Similarly, a tree \( t \) over \( I \) is given by \( (roots(t), \sigma_t) \) where \( roots(t) \) is the root identifier of the store \( t \) over \( I \) that is, \( roots(\sigma_t) = \{ roots(t) \} \). The sub-forest of \( t \),

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{docexample.png}
\caption{Tree representation \( t \) of docexample.xml}
\end{figure}
denoted \( subfor(t) \), is defined by \( \Pi_{f \setminus \{\text{roots}(t)\}}(t) \).

For the sake of simplicity we often use \( t \) in place of \( \sigma_t \).

For the sake of the formal presentation, the identifiers used in the definition of a store are sometimes giving the position of the nodes in the XML document (see the motivating example). Such stores are called \( p \)-stores.

**Example 2.1.3.** The identifiers used to define the store of the document in Figure 2.2 are positions of the node. Thus this store is a \( p \)-store. □

A XML document considered as *well formed* if it has correct XML syntax. A *valid* XML document is a *well formed* XML document, which conforms to the rules of a Document Type Definition (DTD).

A DTD defines the structure of XML elements occurring in a document. Each possible tag is declared together with the structure of its content. To this end regular expressions are used.

DTD declarations are of the form:

\[
\text{<!ELEMENT element-name (element-content)>}
\]

element-name is the element tag, while element-content is a regular expression over tags and text-symbols types describing the structure of the element content.

\[
\text{<!DOCTYPE docexample[}
\]

\[
\text{<!ELEMENT docexample (a*)>}
\]

\[
\text{<!ELEMENT a (b*, c*, d?)>}
\]

\[
\text{<!ELEMENT b (#PCDATA)>}
\]

\[
\text{<!ELEMENT c (#PCDATA)>}
\]

\[
\text{<!ELEMENT d ( f | g)*>}
\]

\[
\text{<!ELEMENT f ( EMPTY)>}
\]

\[
\text{<!ELEMENT g ( EMPTY)>}
\]

\[
\text{]}>
\]

Figure 2.3: DTD of *docexample.xml*

Figure 2.3 illustrates the whole declaration for *docexample* document. Each DTD has to begin with the declaration for the root element \( <!DOCTYPE docexample, \) and then it continues with specification for other elements.

The declaration for the root element is given as follows:

\[
\text{<!ELEMENT docexample (a*)>}
\]

It specifies that the element is tagged as *docexample* and that its content must be a sequence of zero or more of elements tagged as *a*. 
2.1. XML

```xml
<!ELEMENT a (b*, c*, d?)>
```

The content of each `a` element consists of an optional `b` element, followed by an optional `c` element, in turn followed by an optional `d` element.

#PCDATA stands for "parsable character data", that is sequences of simple characters, without interleaved XML elements.

<table>
<thead>
<tr>
<th>Doc</th>
<th>a* , e*</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b* , c* , e* , d?</td>
</tr>
<tr>
<td>b</td>
<td>String</td>
</tr>
<tr>
<td>d</td>
<td>(f</td>
</tr>
</tbody>
</table>

Figure 2.4: The DTD $D$

In this Thesis we use a more compact notation for DTDs, coming from [26].

We consider XML trees valid wrt a schema defined by means of the DTD language, which features the core mechanisms of mainstream schema languages.

Given a finite set of labels $\Sigma$, and the reserved symbol `String`, a DTD over $\Sigma$ is a tuple $(D, s_D)$ where $D$ is a total function from $\Sigma$ to the set of regular expressions over $\Sigma \cup \{String\}$, and $s_D \in \Sigma$ is the root symbol. Given a regular expression $r$, the language generated by $r$, respectively the set of symbols in $\Sigma$ occurring in $r$, is denoted by $L(r)$, respectively $S(r)$. We denote $t \in D$ the fact that $t$ is valid wrt $D$.

**Example 2.1.4.** The DTD $D$ given in Figure 2.4 maps the elements of $\Sigma = \{a, e, b, c, d, f, g\} \cup \{String\}$ to regular expressions over $\Sigma$: $doc \rightarrow a* , e*$; $a \rightarrow b* , c* , e* , d?$ etc, where $lab(\epsilon) = s_D$.

Note that for the sake of simplicity, the rules defining `c`, `e`, `f`, and `g` are omitted. These rules are $c \rightarrow \epsilon$, $e \rightarrow \epsilon$, $f \rightarrow \epsilon$ and $g \rightarrow \epsilon$ where $\epsilon$ is an empty regular expression.

Note that $\Sigma$ contains all the labels occurring in the XML document $t$ in Figure 2.2.

![Figure 2.5: The projection $t'$ of $t$](image)

Figure 2.5 illustrates the projection of the tree $t$ of Figure 2.2. As the reader can observe, the projection selects the root node labelled by `docexample` followed
by the $a$-node having the identifier equal to 1. The projection outputs two children nodes of the $a$-node: the $b$-node with identifier is equal to 1.1 and the $d$-node with identifier 1.4. The other children of the $a$-node are pruned out. For the $a$-node whose identifier is equal two 2, the projection selects $d$-node having identifier equal to 2.1.

Given a store $\sigma$ over $I$, the projection on $J \subseteq I$ of $\sigma$, is a store over $J$, denoted $\Pi_J(\sigma)$, defined by: for each $j \in J$, if $\sigma(j) = a[K]$ then $\Pi_J(\sigma)(j) = a[K_{\downarrow J}]$ otherwise $\sigma(j) = text[st]$ and $\Pi_J(\sigma)(j) = \sigma(j)$. The reader should pay attention to the fact that the domain and the "co-domain" of the projection on $J$ of $\sigma$ is $J$.

Example 2.1.5. Let us consider $J = \{\varepsilon, 1, 2, 1.1, 1.4, 2.1\}$. Then $\Pi_J(t)$ (t of fig. 2.2) is the store corresponding to the XML document $t'$ of Figure 2.5. □

2.2 XQuery Update Facility (XUF)

The update language we consider is the one proposed in [14], a large core of XUF. The main features of the language are:

- use of XQuery expressions to compute target node and update content,
- statement-based update executions,
- complex updates,
- constraint checking,
- snapshot semantics.

XQuery uses different types of expressions: path, arithmetics, conditional, logical, comparison and FLWOR expressions.

In XQuery the expression which simply returns the value. The XUF introduces a new category of expression called an updating expression (or statement). Updates are classified into simple and complex updates.

Simple updates are the basic data modification operations like insert, rename or remove. Complex updates can be either conditional or iterative expressions, using simple expressions.

2.2.1 Simple updates

Simple updates support the following operations:

- insertion of new XML fragments,
2.2. XQuery Update Facility (XUF)

SimpleUpdate ::= InsertExpr | DeleteExpr | ReplaceExpr

InsertExpr ::= "insert" ("node" | "nodes")

SourceExpr InsertExpr TargetChoice TargetExpr

InsertExpr TargetChoice ::= "into" | "as first into" | "as last into"

"before" | "after"

DeleteExpr ::= "delete" ("node" | "nodes") TargetExpr

ReplaceExpr ::= "replace" ("value" "of")? "node" TargetExpr

"with" SourceExpr

Figure 2.6: The syntax of simple updates

• deletion of existing fragments,
• replacement of an existing fragment by a new one.

For each case XQuery computes the location where the update occurs and the content of the update.

The syntax of simple updates expressions is given in Figure 2.6. In this syntax TargetExpr is an XPath expression which computes the target location where the update is taking place. SourceExpr is an XQuery expression, that returns a new document fragment which is to be inserted or replaced at the target location.

The following example given in Figure 2.7 illustrates the result of the evaluation of the simple update expressions "insert", "replace" and "delete" on the document doc.xml.
ComplexUpdate ::= FLWUpdate | ConditionalUpdate

FLWUpdate ::= "update" (ForClause | LetClause)\textsuperscript{+}
               WhereClause? SimpleUpdate\textsuperscript{+}

ConditionalUpdate ::= "update" "if" (XQueryExpr | "then")\textsuperscript{+}
                    SimpleUpdate "else"? SimpleUpdate

Figure 2.8: The syntax of complex updates

Figures 2.7-(a),(b) illustrate XML document $doc$ and its tree representation. Figure 2.7-(c) illustrates the document after the execution of the simple update expression $su_1$ specified by:

\begin{verbatim}
insert  
  <f><z/></f>
as first into $doc(doc.xml)/a$.
\end{verbatim}

XQuery element constructor constructs a new inserted "as first" subtree rooted at node labelled by $f$ whose identifier is 5.

Figure 2.7-(d) illustrates the updated $doc$ after the execution of the simple update expression $su_2$ which replaces the $d$-node and specified by:

\begin{verbatim}
replace  
  $doc(doc.xml)/d$
with $doc(doc.xml)/p$.
\end{verbatim}

Figure 2.7-(f) illustrates the updated $doc$ after the execution of the simple update expression $su_3$ specified by:

\begin{verbatim}
delete  
  $doc(doc.xml)/k$
\end{verbatim}
This expression deletes the last element of the $a$-node.

\section{Complex Updates}

Complex updates are built from simple updates using either conditional or FLWOR expressions having syntax as illustrated in Figure 2.8.

Complex updates can be either conditional or iterative expressions, using simple expressions. Conditional updates relies on if-then-else query expressions.

Let us consider the conditional update statement $cu_1$ specified by:

\begin{verbatim}
update  
  if empty $doc(doc.xml)/p/f$
then
\end{verbatim}
2.2. XQuery Update Facility (XUF)

(a) XML document doc
(b) Tree of doc
(c) Result of the conditional update
(d) XML document doc'
(e) Result of FLWUpdate

Figure 2.9: Complex updates execution

```
delete /doc(doc.xml)/p
else
    replace node /doc(doc.xml)/p with <b/>
```

This update deletes p-node if it does not contain a child element labelled by f, otherwise it replaces it with a new b-node.

The result of the execution of cu₁ over the document doc is illustrated in Figure 2.9-(c).

FLWUpdate expression used to apply simple updates throughout iterations. For instance, let us consider the FLWUpdate statement cu₂ specified by:

```
update for $x in doc(doc.xml)/a/p
where $x/f
insert node <g/> as last into $x
```

This update checks for each p-node whether it contains a child f-node and, in this case, it inserts a new element as the last child of that p-node. The result of the execution of cu₂ over the document doc' is illustrated in Figure 2.9-(e).

2.2.3 Constraints and semantics

While executing updates a set of basic semantic constraints must be respected to preserve the logical structure of the data model instance. For each update the following constraints are preserved:
Insert - The TargetExpr of a simple update must be a single node. If it contains an empty value or more than one node a static error is raised and the insertion is not performed. When the into is specified, the result must be evaluated to a single element or document node; any other non-empty result raises a type error.

If before or after is specified, the result of TargetExpr must be a single element, text, comment, or processing instruction node; any other non-empty result raises a type error.

Delete - The TargetExpr result must be a simple expression; otherwise a static error is raised and be a sequence of zero or more nodes; otherwise a type error is raised.

Replace - The SourceExpr must be a content sequence, which is any sequence of zero or more element nodes, atomic values, processing instructions and comment nodes.

2.2.4 Snapshot semantics

A snapshot semantics is used in XML update languages to avoid the inconsistent results and ensure the semantics integrity. For instance, when the consecutive updates in a single FLWUpdate expression impact the same XML nodes, the execution of this expression can lead to inconsistent result. According to [9] snapshot semantics: all the variables in the for/let clauses of FLWUpdate must be bound with respect to the initial snapshot before the simple updates in the body of the FLWUpdate are executed. Based on the initial snapshot the simple update expressions are executed sequentially and evaluated independently of each other.

The XUF 1.0 defines an entire query as one snapshot, within which the updating expression is evaluated, resulting in a pending update list. A pending update list is an unordered collection of update primitives, which represent node state changes that have not yet been applied.

Update primitives The main points of the semantics of these primitives are below described (see [9] for more details).

Given a tree $t = (\text{roots}(t), \sigma)$, where the store $\sigma$ is over $I$, the $\$target$ variable is bound to a node having identifier $i \in I$, while the variable $\$content$ is bound to sequence of nodes having an id-seq of identifiers in $I$.

insertBefore( $\$target$, $\$content$) - This primitive inserts $\$content$ into the tree $t$ immediately before $\$target$. Note that the order of these nodes is preserved.

insertAfter( $\$target$, $\$content$) - This primitive inserts $\$content$ into $t$ immediately after $\$target$.

insertInto( $\$target$, $\$content$) - The insert primitive inserts $\$content$ which are inserted as children of $\$target$. The choice of position is implementation-dependent.
2.2. XQuery Update Facility (XUF) 19

insertIntoAsLast( $target, $content) - The insert primitive inserts $content into $t which are inserted as the last children of $target.

insertIntoAsFirst( $target, $content) - The insert primitive inserts $content into $t which are inserted as the first children of $target.

delete( $target) - This primitive removes given $target node from the data model.

replaceNode( $target, $content) - This primitive replaces a given $target node with one or more new nodes bound to $content.

rename( $target $newName) - Changes the node-name of $target to new name.

replaceValue( $target as node(), $string-value as xs:string) - This primitive replaces the string value of $target with $string-value.

The semantics of an update primitive do not become effective until their pending update list is processed by the applyUpdates routine.

Update primitives in appending lists are applied in the following order:

First, all insertInto, replaceValue, and rename primitives are applied. Next, all insertBefore, insertAfter, insertIntoAsFirst, and insertIntoAsLast primitives are applied. Next, all replaceNode primitives are applied. Next, all delete primitives are applied. It is worth noticing that pending update list can not have more then one rename(replaceNode or replaceValue) primitives have the same $target node.
State of the Art, XQuery Engines

Contents

3.1 Introduction ........................................... 21
3.2 MonetDB/XQuery ........................................ 22
  3.2.1 General Data structure .............................. 23
  3.2.2 Data structure supporting structural updates ........ 26
  3.2.3 XML Serialization ................................. 32
  3.2.4 MonetDB/XQuery vs. projection .................... 33
3.3 Native XML databases ................................. 34
  3.3.1 BaseX ............................................. 34
  3.3.2 General Datastructure ............................ 34
  3.3.3 Data structure supporting structural updates ....... 36
  3.3.4 XML Serialization .................................. 40
  3.3.5 BaseX vs. projection .............................. 40
3.4 eXist .................................................. 42
  3.4.1 General Data Structure ............................ 43
  3.4.2 XML serialization ................................... 46
  3.4.3 eXist vs. projection .............................. 47
3.5 Saxon Processor ........................................ 48
  3.5.1 General Data Structure ............................ 48
  3.5.2 XML Serialization ................................... 52
  3.5.3 Saxon vs. projection .............................. 52
3.6 Conclusion ............................................. 52

In this Chapter we present main strategies adopted by XML query engines to represent, store and manipulate XML document. Besides illustrating how projection can improve query processing in each of these systems, this Chapter constitutes a contribution in its own, providing a detailed overview of several existing systems.

3.1 Introduction

Currently existing XML database management systems can be classified into three categories: XML-enabled, Native XML and main-memory XQuery processors.
**XML-enabled** - These systems map XML data to traditional relational databases, by encoding XML data into *tables of tuples*. They accept XML as input and redirect XML as output. This entails that the database does the conversion itself. An example of this system is MonetDB/XQuery [19, 20].

**Native XML** - The internal model of such databases depends on XML and defines a logical model for XML documents, according to which the documents are stored and retrieved. It is worth noticing that the XML files are not necessarily stored in the form of text files. The model includes elements, attributes and PCDATA. Main database engines that belong to this category are: BaseX [29, 30], Qizx [4, 5] and eXist [2, 34].

**Main-memory XQuery processors** - They are very efficient on small XML files. On the contrary, while querying larger files the behavior of these systems is less efficient because the temporary XML representations occupy 6 to 8 times the size of the original file in main memory. Examples of this processors are Saxon [7] and Galax [3].

It is worth noticing that we can consider MonetDB/XQuery both as XML-enabled DBMS and as native XML database, since the XML documents are mapped into a relational representation. From a technical viewpoint, MonetDB/XQuery is an XML database "implemented on the top of a relational storage". As described in the [20], MonetDB/XQuery uses a relational table to represent the structure of an XML document. From a user's viewpoint, MonetDB/XQuery can store only XML documents and accept only XQuery. In this context, MonetDB/XQuery is a "native XML DBMS".

The following sections provide detailed explanation of the data structures used in MonetDB/XQuery, BaseX, eXist and Saxon. Each section covers XML encoding (if used), axis relations, general data structure and data structure supporting updates. At the end of each section we compare the differences between evaluating queries on XML document using the engines with and without projection (see Chapter 4).

The QizX, Zorba and Galax systems are not covered in this chapter, since there are not enough documentations explaining their internal data structure.

### 3.2 MonetDB/XQuery

MonetDB/XQuery is an open source column-oriented database management system, which stores data table into files on the disk. The important point is that these data table files are "memory mapped files".

In ModetDB/XQuery, a data table is represented as a set of arrays in the memory. Each array is directly mapped to a file on the disk. In MonetDB/XQuery XML data
3.2. MonetDB/XQuery

manipulations (queries, updates, etc.) have to be mapped into SQL expressions. To support this mapping several compilation techniques have been designed and developed by some research teams [23, 31, 37]. MonetDB uses the one proposed by [31]. It is worth noticing that compilation used for query evaluation does not involve interaction with the back-end, once delivered to DBMS, the emitted SQL code evaluates the input XQuery expression by means of a single SQL query [31].

3.2.1 General Data structure

Relational XML encoding in MonetDB/XQuery The relational encoding of XML documents is described in [19, 20]. This encoding is based on preorder and postorder traversal ranks, is used to encode the XML tree structure. In pre/post encoding the pre value describes the preorder traversal rank of the tree starting from the root, while the post value describes the postorder traversal rank, which visits the root last. The pre and post values are mapped into a two-dimensional plane, where each node partitions the plane into four regions, is used to calculate a step’s axis. We will illustrate it by means of an example.

Example 3.2.1. Relational storage used in MonetDB/XQuery.

Figure 3.1 illustrates actual relational XML representation used in MonetDB/XQuery, which instead of pre/post encoding uses pre/size/level encoding. MonetDB/XQuery instead of the post column stores two columns holding a tree level and a subtree size. This pre/size/level encoding is equivalent to pre/post
since \( post = pre + size - level \).

Figure 3.1-(a) shows the example document \( doc \). In Figure 3.1-(b) \( pre \) and \( post \) ranks are assigned to the nodes of the XML tree. For instance, for the root node labelled by \( a \) the \( pre \) and \( post \) ranks are equal to 0 and 8 respectively. Figure 3.1-(c) illustrates the nodes in a \( pre/post \) plane. As the reader can observe, for each node the quadrants of the \( pre/post \) plane correspond to the major XPath axes: \( descendant \), \( following \), \( ancestor \) and \( preceding \). For our example, the corresponding axes for the target node labelled by \( p \) for a given XPath expression \( a/p \) are following:

- \( ancestor \) - The \( a \)-node having coordinates \( pre=0 \) and \( post=8 \).
- \( preceding \) - The \( b \)-node with \( (pre=1, post=0) \), the \( d \)-node with \( (pre=2, post=3) \) followed by the \( c \)-node having \( (pre=3, post=2) \) and the node of type text \( s \) with \( (pre=4, post=1) \);
- \( following \) - The nodes labelled by \( k \), \( f \) and \( e \).
- \( descendant \) - The \( p \)-node contains no descendants.

Finally, Figure 3.1-(d) shows the actual relational XML representation used in MonetDB/XQuery, which instead of the \( post \) column stores two columns holding a tree level and subtree size. This encoding represented in our example is generated while traversing the XML tree and stored in the \( doc \) table. The encoding contains the three attributes \( pre \), \( size \) and \( level \), below described.

\( pre \) - Is a unique value associated to a node \( n \). When \( n \) is traversed a \( pre \) value is assigned to that node and is incremented throughout the traversal. It is denoted by \( n.pre \).

For our example, while parsing the document \( doc \), the first node found is the root labelled by \( a \); thus its \( n_a.pre \) property is set to 0. When the next child \( n_b \) labelled by \( b \) is parsed, the \( pre \) value is incremented and thus \( n_b.pre \) is set to 1. The same is true for the remaining nodes. It is worth noticing that the texts are considered as nodes and are encoded. For example, for the text node "s" it is true that: \( n_s.pre=4 \).

\( size \) - Is the number of nodes in the subtree below a node \( n \) and is denoted by \( n.size \).
For our example, because the root labelled by \( a \) has 8 descendants, the value \( n_a.size \) is equal to 8. For the first child of the \( a \)-root we have that \( n_b.size=0 \): it has no children; while for the \( k \)-node \( size \) is equal to 2. For the text node \( n_s \) the value of the \( size \) property is set to 0.

\( level \) - Is the distance from the root to a node \( n \) and it is denoted by \( n.level \). For instance, \( n_a.level=0 \), \( n_b.level=1 \) and \( n_s.level=3 \).

Some additional properties of a node are stored in \( doc \), like:
3.2. MonetDB/XQuery

kind - The kind of a node $n$ is either an element, a text or an attribute. For the $a$-root we have that $n_\alpha.kind=elem$, while for the text node $n_\gamma.kind=text$.

prop - This property stores tag names for element nodes, or a text value for text nodes.

This relational representation is used to express the semantics of XPath axes.

<table>
<thead>
<tr>
<th>Axis</th>
<th>Relational characterization</th>
</tr>
</thead>
<tbody>
<tr>
<td>ancestor($n,n'$)</td>
<td>$n.pre &lt; n'.pre \text{ AND } n'.pre \leq n.pre + n.size$</td>
</tr>
<tr>
<td>descendant($n,n'$)</td>
<td>$axis(n', n, \text{ descendant}) \text{ AND } n.level = n'.level + 1$</td>
</tr>
<tr>
<td>child($n,n'$)</td>
<td>$n.pre &lt; n'.pre \text{ AND } n'.pre \leq n.pre + n.size$</td>
</tr>
<tr>
<td>following($n,n'$)</td>
<td>$n.pre &gt; n'.pre + n'.size$</td>
</tr>
<tr>
<td>preceding($n,n'$)</td>
<td>$n.pre + n.size &lt; n'.pre$</td>
</tr>
</tbody>
</table>

Table 3.1: Relational characterization

Axes Relationships For instance, given a tree $t$ and two nodes $n_1$ and $n_2$ it is true that:

$$n_1 \in n_2/\text{ancestor} \iff n_1.pre < n_2.pre$$

$$\text{AND}$$

$$n_2.pre \leq n_1.pre + n_1.size$$

Table 3.1 represents XPath semantics for some of the axes. The full list is given in [32].

For our example illustrated in 3.1, the children of the node labelled by $k$ in the document $doc$ are the nodes labelled by $f$ and by $e$.

These children are calculated using the rule for finding the child relationship child($n,n'$) from the table 3.1, where we have:

$$k.pre=6, k.size=2, f.pre=7$$

$$k.pre < f.pre \text{ AND } f.pre \leq k.pre + k.size$$

(3.1)

$$k.pre=6, k.size=2, e.pre=8$$

$$k.pre < e.pre \text{ AND } e.pre \leq k.pre + k.size$$

(3.2)

One of the reasons why MonetDB/XQuery uses size/level instead of post, is related to the node skipping property: node skipping allows to find out that certain regions of pre values do not contain any result nodes for XPath step. Thus, it avoids any data access or computation and skips over these tuples. For example, finding all children of the $a$-node ($n_\alpha.pre$) works by
checking the first child: \( n_b.pre = n_a.pre + 1 = 1 \); then skipping to its siblings: \( n_d = n_b + \text{size}[n_b.pre] + 1 = 1 + 0 + 1 = 2 \) (d-node) until the last child a-node is reached: \( (n_a.pre + \text{size}[n_a.pre]) \).

The encoding described above is not able to support XML structural updates. There are two important issues: the first one arises as a result of a subtree insertion and requires renumbering all \( \text{pre} \) values starting from the inserted point. The second issue is that the \( \text{size} \) values of the ancestors of the inserted node must be recalculated.

Figure 3.2 illustrates an example how the \( \text{pre/size/level} \) document encoding is affected by an insertion of new nodes.

**Example 3.2.2. New nodes insertion.**

Let us assume that new subtree (containing nodes labelled by \( n \) and by \( m \)) has been inserted at the target node labelled by \( p \). For our example 3.2.1, the \( \text{pre} \) values of all following nodes after the insert point must be changed, as well as, the \( \text{size} \) values of all ancestor nodes.

Figure 3.2-(c) exhibits the changes applied on the relational storage after the update which inserts nodes labelled \( n \) and \( m \) as the children of the node labelled by \( p \). As a result of this insertion the \( \text{pre} \) values of the following nodes of p-node must be re-calculated: \( \text{pre} \) value of the \( k \)-node is changed to 8, of \( f \)-node to 9 and for \( e \)-node to 10. All \( \text{pre} \) values of the followings are augmented by two (two elements have been inserted). The next change must be applied on the \( \text{size} \) value of the ancestor of the \( p \)-node the \( a \)-node. This value is augmented by the number of the inserted elements, for our example by two. The new value of the \( n_a.size \) is set to 10. In Figure 3.2-(c) all the changes are colored gray.

The recalculation of these property values can be expensive and complex, therefore the storage scheme illustrated in Figure 3.2 is used as a read-only representation of the XML encoding. Next we present the encoding which supports the structural updates.

### 3.2.2 Data structure supporting structural updates

To support the updatable representation of the XML Encoding in [20] the following changes on the table \( \text{pre/size/level} \) have been proposed:

1. The table is called \( \text{rid/size/level} \).
2. It is divided into logical pages, where the size is defined in terms of the number of tuples.
3. Each logical page contains unused tuples.
4. New logical pages are appended at the end.
Figure 3.2: The impact of structural updates on pre/size/level XML storage
Figure 3.3: Insert within a logical page

5 - The pre/size/level table is a view on rid/size/level with all pages in logical order. This is implemented by mapping the underlying table into a new virtual memory region.

The XML updating algorithm deals with a pre-view table that is a virtual table comprising pre, size and level column. This pre-view table is implemented by a rid(row-id) table and a page offset table (pg_Off table, which is explained in the example given in Figure 3.5).

To look up a tuple with a specific pre-value in a pre-view table, we need to calculate the corresponding rid-value from given pre-value. The "Swizzling" technique is used to efficiently perform this computation. It is worth noticing that the rid column is non-materialized integer column that is mapped to the row-id of the table.

Two possible update scenarios are possible for the updatable representations:

- **an insert which is handled within a logical page,**
- **an insert when a new logical page is inserted.**

To illustrate the differences we provide examples for each of scenario. The following examples illustrate the updatable representation of the document doc from Figure 3.1-(a) for each case. First example exhibits the case: insertion within a logical page.

**Example 3.2.3. Insert within a logical page.**

As it is illustrated in Figure 3.3-(a), the document doc is stored in two logical pages, each page containing 8 tuples filled in by the properties of nodes. For example, the size properties of the a-node and p-node are set to 10 and 2 respectively. Each
3.2. MonetDB/XQuery

<table>
<thead>
<tr>
<th>rid</th>
<th>size</th>
<th>level</th>
<th>prop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>d</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>3</td>
<td>s</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>p</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>null</td>
<td>k</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>null</td>
<td>i</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>1</td>
<td>k</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>2</td>
<td>i</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2</td>
<td>e</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>null</td>
<td>e</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>null</td>
<td>e</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>null</td>
<td>e</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>null</td>
<td>e</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>null</td>
<td>e</td>
</tr>
</tbody>
</table>

(a) before the update

<table>
<thead>
<tr>
<th>rid</th>
<th>size</th>
<th>level</th>
<th>prop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>16</td>
<td>0</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>d</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>3</td>
<td>s</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>1</td>
<td>p</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>2</td>
<td>m</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>1</td>
<td>k</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>2</td>
<td>f</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2</td>
<td>e</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>null</td>
<td>e</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>null</td>
<td>e</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>null</td>
<td>e</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>null</td>
<td>e</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>null</td>
<td>e</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>2</td>
<td>l</td>
</tr>
<tr>
<td>17</td>
<td>6</td>
<td>null</td>
<td>e</td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>null</td>
<td>e</td>
</tr>
<tr>
<td>19</td>
<td>4</td>
<td>null</td>
<td>e</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>null</td>
<td>e</td>
</tr>
<tr>
<td>21</td>
<td>2</td>
<td>null</td>
<td>e</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>null</td>
<td>e</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td>null</td>
<td>e</td>
</tr>
</tbody>
</table>

(b) after the update

Figure 3.4: Insert with a new logical page insertion

A page has certain percentage of tuples stored as "unused". For our example we keep at least two unused tuples (see the tuples colored gray). The `n.level` values for these tuples are set to `null`. The `n.size` values are set equal to the number of directly following consecutive unused tuples, which allows to skip unused tuples. The unused tuples have an important role for the inserts that do not cause insertion of new logical pages.

Let us suppose that an update specified by `insert nodes {<n/><m/>} into a/p` is applied on the document `doc`. When the update is executed the new nodes are added to the table. Figure 3.3-(b) reflects the modifications applied on the `rid/size/level` table. The new nodes: `n` and `m` (illustrated in bold) are inserted in the logical page 0; `size` values are set to 0 and `level` to 2. Because this insertion fits into the page 0, there is no necessity to recalculate the `size` values of `a`- and `p`-nodes. It is important to note, that in the `rid/size/level` table we added the `prop` column, which is done to make the example more easy to follow.

The next example illustrates an update where the insert triggers a new logical page insertion in the `rid/size/level` table.

**Example 3.2.4.** New logical page insertion.

Let us assume that an update specified by `insert nodes {<n/><m/><l/>} into a/p` is applied on the document `doc`. As it is illustrated in Figure 3.4-(b) the newly
inserted nodes, labelled by \( n, m \) and \( l \), do not fit into the free space of the page 0, which has only two unused tuples having \( \text{rid}=6 \) and \( \text{rid}=7 \). Therefore a new page 2 is inserted. The first tuple of this page is filled in by the properties of the \( l \)-node: \( n_l.\text{size}=0, n_l.\text{level}=2 \). As the reader can observe, the insertion of the page triggers recalculation of the size properties of \( a \)- and \( p \)-nodes, which are set to et to 16 and 10 respectively. It is worth noticing that there is no need to recalculate the rid values of the preceding and following of the \( p \)-node.

To support the structural updates the storage schema is enriched by the following tables \( pg_{\text{Off}} \) and \( node/rid \) tables.

- **pg_{\text{Off}}** - used to maintains a logical page order under updates and used to construct the pre/size/level view.

- **node/rid** - used to translate unique node numbers into pre. It is worth noticing that node/rid used in the updatable representation to deal with the issue related to the attribute table. The problem is that the read-only schema uses the pre value (now changed to rid) to find the attribute of the node. Because in updatable schema all pre columns are replaced by rid, in rid/size/level table a unique property: node, is assigned to each node, which is the number that never changes.

The following two examples illustrate the new update schema enriched by \( pg_{\text{Off}} \) and \( node/rid \) tables, for two update scenarios.

**Example 3.2.5. Insert within the page: \( pg_{\text{Off}} \) and \( node/rid \).**

Let us consider that new nodes \( n \) and \( m \) are inserted as last children of the \( c \)-node (see fig. 3.5-(b)). First, the tuple preserving the properties of the \( p \)-node is
moved further in the page: new \( p.\text{rid}=7 \). Second, the new \( \text{rid} \) value of the \( p \)-node is modified in the \( \text{node/rid} \) table (see fig. 3.5-(d)). Finally, newly inserted nodes labelled \( n \) and \( m \) are stored into the \( \text{rid/size/level} \) (see the italic lines in the page 0 and in the \( \text{attributes} \) table (see fig. 3.5-(c))). It is worth noticing that a unique node number must be assigned to each of them. This number can be looked up in the \( \text{node/rid} \) table by searching for the \( \text{node} \) values of the entries where \( \text{rid}=\text{null} \). In the case that we do not find any node number, new tuples are appended to the \( \text{node/rid} \) table and new \( \text{node} \) values are assigned to them in \( \text{rid/size/level} \).

**Example 3.2.6. Insert with new logical page insertion:** \( \text{pg\_Off} \) and \( \text{node/rid} \).

Let us assume that new nodes \( n \), \( m \) and \( l \) are inserted as the children of the \( p \)-node (see fig. 3.6-(b)). First, because the insert does not fit into the page 0, a new logical page 2 (colored pink) is appended to \( \text{rid/size/level} \). Next, a new entry for the page 2 is appended to the \( \text{pg\_Off} \) table, and the offsets of all pages after the insert point are incremented (see fig. 3.6-(f)). The \textit{size} values of the ancestor \( a \)-node is set to 16. Finally the \textit{node} and the \textit{rid} values of newly inserted elements are added to the \( \text{node/rid} \) table.

It is worth noticing that when we said that a new logical page is appended to \textit{rid/size/level} it is appended to the end, while for the \textit{pre/size/level} view it is inserted in between (see the difference in fig. 3.6(a) and (b)). Therefore to look up
a tuple with a specific pre-value in a pre/size/level-view, we need to calculate the corresponding rid-value from given pre-value. This process is called swizzling and helps to efficiently perform this computation. Example in Figure 3.6 illustrates the update schema together with the rid-pre swizzling.

By using $2^n$ as the page size, we can calculate rid by using "bit operations" (this is the most important point) including bitwise AND and bit shift operations. This approach is very efficient because the bit operations are not expensive.

The page size used in our example is 8 ($n=3$). If $n=3$, we can calculate pre from rid by using the following formula:

$$pre = pg_{Off}[rid >> 3] << 3 + rid & 00000111; \quad (3.3)$$

As it is illustrated in Figure 3.6, the pre value of the l-node is calculated as follows:

The swizzle procedure takes as an input two parameters: the rid of the node and the pg_{Off} table. We have that l.\text{rid}=16 and $n=3$ hence:

$$l.\text{pre}=\text{swizzle}(l.\text{rid}, pg_{Off})=pg_{Off}[rid >> 3] << 3 + rid \& 7.$$  

It is worth noticing that 7 is a binary representation of 0000 0111 which is a "bit masks". 0000 0111 is a mask for taking lower 3 bits of data by using bitwise AND operation. "x AND 0000 0111 is equivalent for the modulo operation such that "x mod 8 (00001000)". Please note, that in our exploration the size of pre and rid is 8 bit. We used 8-bit for the convenience of presentation. Because $l.\text{rid}=16$ we have that:

$$l.\text{pre}=pg_{Off}[16 >> 3] << 3 + 16 \& 7 \text{ where } 16 >> 3 = 2 \text{ and } 16 \& 7 = 0$$

Thus $l.\text{pre}=pg_{Off}[2] << 3 + 0$.

As the reader can see in Figure 3.6-(f) the offset entry for the page 2 in pg_{Off} is set to 2 and $l.\text{pre}=1 << 3 + 0 = 8 + 0 = 8$ hence the $l.\text{pre}=8$.

To calculate rid from pre we use the following formula:

$$\text{rid} = pre \& 00000111 + pg_{Off}[(pre \&11110000) >> 3] << 3; \quad (3.4)$$

The $l.\text{rid}=16$.

### 3.2.3 XML Serialization

An encoded XML document stored in a table can be serialized as an XML document. The serialization processes by scanning the nodes in the table in ascending
pre column order and by outputting them to the output console. The problem arises while closing the tags of the nodes having descendants. Thus, each node n is pushed onto a stack S to remember to print the closing tag of n. The post (post = pre + size − level) rank of n encodes the relative order of closing tags in the serialized XML text.

3.2.4 MonetDB/XQuery vs. projection

Figure 3.7 illustrates the relational storage for MonetDB/XQuery using projection technique (see Chapter 4). Using the projection we can benefit from two kinds of possible optimizations for some of the cases. The first one skips new logical page insertion and the second one skips to move the nodes after the insertion point.

As it is illustrated in Figures 3.7-(b), applying the projection on doc, in order to perform an update (insert new nodes n, m and l to the target p-node) we need to store only a- and p-nodes. Thus, differently from the example in Figure 3.6 there is no necessity to insert a new logical page and (see fig. 3.7-(c) ) deal with sifts and recalculations of the size property.

The second optimization achieved with the projection is that for the example illustrated in Figure 3.5 (insert nodes {<n/><m/>} into a/d/c ) it is not necessary to perform the move of the p-node within the page. Hence there is no need to change the p.rid value in the node/rid table. As the reader can see using projection helps to perform less expensive updates.

As it has been stated in the Introduction and the Preliminaries Chapters to make updates persistent the Merge algorithm is used. The issue here is that execution of Merge, as the reader can see in Chapter 5, increases the total execution time for the XML document whose size is less then 150MB. One of the reasons of this increase is due to the facts that, the updated document is first serialized and after is used as
an input to the Merge algorithm. This time could be reduced by integrating the Merge algorithm with the serialization. To achieve this, several changes must be applied to the implementation. Mainly to add the steps of the algorithm while scanning the table.

3.3 Native XML databases

The following two sections cover the data storage architecture of the Native XML database systems. As described in the introduction of this Chapter, the main difference between NXD and enabled databases is that the first one maps the XML documents into logical models, for example like DOM [1]. This storage proceeds in the following way: first the XML document is parsed (for example by using SAX Parser [6]). While parsing the document, the NXD database system translates each element of the document tree to its logical representation, which is then stored at the backend of the system. Then the XQuery expression will be evaluated using that stored data.

3.3.1 BaseX

BaseX was developed as native XML database. BaseX is a database prototype, which maps XML documents to a table-based tree encoding [29, 30]. It is derived from the XPath accelerator encoding used in the MonetDB/XQuery.

The following example given in Figure 3.8 illustrates the relational mapping of an XML document in BaseX. First we provide an example explaining the general data structure and then give an example for the updatable structure.

3.3.2 General Datastructure

Example 3.3.1. Document encoding in BaseX (pre/par encoding).

Figure 3.8-(a) shows the XML document doc. (We choose the doc from the example 3.2.1, to make the BaseX storage comparable to the MonetDB/XQuery). Figure 3.8-(b) shows the tree of doc assigned with pre values. Important to note that, similarly to the MonetDB/XQuery mapping, the pre values are assigned to the text nodes. Figure 3.8-(c) exhibits the node table, storing the pre/par references of all nodes. These references point to the disk blocks. The references of the node table are the followings:

- **pre** - Similarly to the pre property of the MonetDB/XQuery encoding, this property is assigned to each node and is incremented throughout the tree traversal. For our example the pre value of the root node labelled a is set to 0.

- **par** - This value represents a direct mapping between children and their parents. For instance, the a-node is the parent of b-node since \( n_a.pre = n_b.par \)
3.3. Native XML databases

(a) XML document
(b) Tree with pre/par values
(c) Relational storage

<table>
<thead>
<tr>
<th>id</th>
<th>tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>.0000</td>
<td>a</td>
</tr>
<tr>
<td>.0001</td>
<td>b</td>
</tr>
<tr>
<td>.0010</td>
<td>d</td>
</tr>
<tr>
<td>.0011</td>
<td>c</td>
</tr>
<tr>
<td>.0100</td>
<td>p</td>
</tr>
</tbody>
</table>

TagIndex

<table>
<thead>
<tr>
<th>id</th>
<th>tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>.0000</td>
<td>s</td>
</tr>
</tbody>
</table>

TextIndex

<table>
<thead>
<tr>
<th>pre</th>
<th>par</th>
<th>token</th>
<th>kind</th>
<th>att</th>
<th>attVal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>a</td>
<td>elem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>b</td>
<td>elem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>d</td>
<td>elem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>c</td>
<td>elem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>s</td>
<td>text</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>p</td>
<td>elem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>k</td>
<td>elem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>i</td>
<td>elem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>s</td>
<td>elem</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(d) Internal representation

Figure 3.8: XML encoding in BaseX (pre/par)
**token** - This value represents a tag name or a text content. Token for the $a$-node is $a$, for the text $s$-node is $s$.

**kind** - This value represents the kind of a node: can be either an element or a text.

**att** - This value represents the attribute names of a node. *null* reference is assigned if no attributes are given, which is a case for our example.

**attVal** - This value represents the attribute value of a node. □

All textual tokens like tags, texts, attribute names and values are uniformly stored in a hash structure and referenced by integers. To optimize CPU processing, the table data is encoded with integer values (see fig. 3.8-(d)). Important to note that, the integer values are stored as integer arrays. For instance, for the $b$-node having $n_b.par=.....0000(0\text{ in decimal})$ kind/token properties are stored together where the first bit set to 0 defines the element kind. The remaining bits $...0001$ is the id value, which points to the second entry in the *TagIndex* table, where the *tag*=$b$. For the text node $s$ with $n_s.par=.....0011(3)$, we have that *kind*=1 and the text value is searched in the *TextIndex* table at the entry *id*=0, where *text*=$s$.

The weak point of this encoding is that the update operations based on *pre*/par can get expensive. Because in case of the insertion or deletion the change of *pre* value triggers the change in *par* after the update point.

To support the updates in less expensive way the *pre*/dist/size encoding is used which is depicted in Figure 3.9-(d).

### 3.3.3 Data structure supporting structural updates

The following example illustrated in Figure 3.9 encodes the *doc* document in *pre*/dist/size. Figure 3.9-(d) illustrates the internal representation of the table on the disk. It is worth noticing that default storage reserves 16 bytes for a single table row.

**Example 3.3.2.** Document encoding in BaseX (*pre*/dist/size encoding).

The *doc* is mapped into *pre*/dist/size where:

- **pre** - Note, that the *pre* value is not stored in the internal representation of the table on the disk, since it is implicitly given by the table position. As the reader can see, for our example in Figure 3.9-(d), $n_a.pre=0$, $n_b.pre=1$ of *pre*/dist/size are implicitly given at positions 0000 and 0010.

- **size** - This value contains the number of descendants of a node. The *size* value for the $a$-node is set to 00 00 00 08. Note, that for the text node $n_s$ we do not store the *size* value.
3.3. Native XML databases

(a) XML document

(b) Tree with pre/dist values

(c) Relational storage

(d) Internal representation

Figure 3.9: XML encoding in BaseX (pre/dist/size)
• **kind** - This value preserves six different node kinds. For instance, \( n_b.kind = 01 \) and \( n_s.kind = 2 \). All six kinds are stored in the *Kind* table.

• **dist** - This value stores the distance to the parent node, allowing access to parents and ancestors. The *dist* value for the \( a \)-node is 0 for the \( f \)-node is 2. For our example, the internal representation preserves 8 bits for the *dist* value. For instance, for the \( p \)-node we have: \( n_p.dist = 00 \ 00 \ 00 \ 05 \). It is worth noticing that the *dist* value can get large.

• **tag** - This value stores references to the *TagIndex* table where the corresponding tag names are indexed and referenced by integer keys. For instance, \( n_a.tag = 0001 \) and in the *TagIndex* table the entry corresponding to 0001 preserves the tag name \( a \).

• **txt** - This text value of a text node is stored in text containers and table entries reference test offsets. For our example we have that \( n_s.txt = 00 \ 00 \ 00 \ 00 \ 00 \) (see the column *attS/Size/txt* in fig. 3.9-(d)) and the corresponding text value in the *Texts* container is stored at the offset 00 and contains the text value "s". Note that, the text value of the next text node in the tree will be stored at the offset 01.

• **attS** - This value stores an attribute size, which denotes the number of attributes of an element. In the internal representation of the *pre/dist/size/* the first two bits preserve the *attS* value. For our example, it is set to 00 for all nodes (see first to bits of the column *attS/Size/txt* in fig. 3.9-(d)).

• **attN** - This value represents an attribute name. *null* reference is assigned if no attributes are given, which is a case for our example.

• **attVal** - This value represents an attribute value.

The following paragraph analyzes the evaluation of XPath axes in BaseX.

**Axes Relationships**

**parent** - \( n' \) is a parent of \( n \) if \( n'.pre = n.pre - n.dist \). For our example we have that \( d \)-node is a parent of \( c \)-node since \( n_d.pre = n_c.pre - n_c.size = 3 - 1 \)

**ancestors** - The *ancestors* step is evaluated using \( n'.pre = n.pre - n.dist \) to access next parent \( n' \) and traversal is completed if \( n' = \text{null} \).

The evaluation procedure:

```plaintext
for each c in context do
    n ← node(c.pre - c.dist)
    while n! = null do
        add n to results
        n ← node(n.pre - n.dist)
return ordered results without duplicates.
```
next sibling - \( n' \) is a sibling of \( n \) if \( \text{pre}(n)+\text{size}(n)+1 \). For our example, \( h \)-node is a next sibling of \( d \)-node since \( \text{pre}(n_p)+\text{size}(n_p)=5+0+1=6 \).

child - For a given context \( c \) the size property simplifies the evaluation of child axis. The child step is evaluated using \( \text{pre}(n)+\text{size}(n)+1 \) to access next sibling \( n' \) and traversal is completed if \( \text{pre}(n) = \text{pre}(c)+\text{size}(c) \) where \( c \) is a context node. The evaluation procedure:

\[
\text{foreach } c \text{ in context do} \\
\text{\hspace{1cm} } n \leftarrow \text{node}(c.\text{pre}+1) \\
\text{\hspace{1cm} } \text{while } n.\text{pre}! = c.\text{pre}+c.\text{size} \text{ do} \\
\text{\hspace{2cm} } \text{add } n \text{ to result} \\
\text{\hspace{2cm} } n \leftarrow \text{node}(n.\text{pre}+n.\text{size}+1) \\
\text{\hspace{1cm} } \text{return } \text{ordered results without duplicates.}
\]

descendant-or-self - For each context \( c \) the descendant-or-self is searched between \( c.\text{pre} \) and \( (c.\text{pre}+c.\text{size}-1) \). The evaluation procedure:

\[
\text{foreach } c \text{ in context do} \\
\text{\hspace{1cm} } \text{foreach } n \text{ in nodes}(n.\text{pre}=c.\text{pre};n.\text{pre}=c.\text{pre}+c.\text{size}-1) \text{ do} \\
\text{\hspace{2cm} } \text{add } n \text{ to results} \\
\text{\hspace{1cm} } \text{return } \text{ordered results without duplicates.}
\]

It is worth noticing that to facilitate updates, the table structure is organized in disk blocks. Similar to MonetDB/XQuery, the table is divided into pages holding a fixed number of tuples. On the contrary to MonetDB/XQuery, these pages may not contain gaps in between tuples. A block directory references the first \( \text{pre} \) value.
of each block. The dist and size values have to be recalculated if deletions and insertions are performed. The size values are updated for all ancestor of that node and the dist values are updated for the following siblings and the following siblings of the ancestor nodes.

Note, that on the contrary to the pre/par in the pre/dist/size encoding the subtrees preserve their original distance when moved or inserted.

For instance Figure 3.10-(b) illustrates the changes applied on the tree given in Figure 3.10-(a) after the insertion of a new text node "os" as the first child of the c-node. As the reader can observe, the dist values of the subtree rooted at the k-node are not affected by this insertion. The recalculation of dist value is performed on the s, p and k-nodes: \( n_s.dist = 2 \), \( n_p.dist = 6 \) and \( n_k.dist = 7 \). The number of the total updates of the dist values for this insertion is equal to 3.

The same update applied on the pre/par encoding is more expensive. As it is illustrated in Figure 3.10-(d) the insertion result in the recalculation of all nodes of the subtree rooted at the k-node. The number of updates on par values is equal to 5, thus the updates on pre/par gets more expensive compared to pre/dist/size.

### 3.3.4 XML Serialization

In BaseX the encoded XML document stored in the current database can be "exported" to the specified path. Similarly to MonetDB/XQuery, the implementation (Java classes: Export, XMLSerialazer and Serializer) scans the arrays where the nodes are stored and for each node it outputs the tags, the attributes and text. The implementations uses the stack to close the tags.

### 3.3.5 BaseX vs. projection

Figure 3.11 illustrates the difference between the update insertion using "pure" BaseX vs. using projection. An update specified by an insertion of a new text node "os" as the first child of the c-node. As it has been explained in Figure 3.10-(b) the insertion of the new node results in the recalculation of the dist value for the c, p and h-nodes. On the contrary, if the projection is used, only the a, d and c-nodes of the doc are mapped into pre/dist/size (see fig. 3.11-(b),(c)). Thus, there is no need to recalculate the dist values of the p and h-nodes, as it was the case in Figure 3.10-(b). As the reader can observe, we are obliged to update the dist value of the s-node: \( n_s.dist = 2 \).

Another optimization, which can be achieved, by using the projection is the following. Recall that, in BaseX the gaps between the tuples in the pages are not allowed, therefore, for instance, if a structural update performs a deletion, first, nodes are deleted and possible gaps on the page are filled by shifting following tuples, which is time consuming. On the contrary, using projection may optimize the execution time for some of the cases as it is illustrated in Figures 3.11-(d), (f), (g) and (k). Let
3.3. Native XML databases

(a) XML document

(b) projection with pre/dist/size values

(c) after insertion

(d) disk block before the deletion of d-node without projection

(f) disk block after the deletion of d-node without projection

(g) disk block before the deletion of d-node with projection

(k) disk block after the deletion of d-node with projection

Figure 3.11: Optimizations in BaseX using projection
us assume that an update specified by a deletion of the $d$-node is performed on the document. If we execute the update without projection, as it is shown in Figures 3.11-(d),(f) tuples following the target $d$-node must be shifted up. While, as it is illustrated in Figures 3.11-(g),(k), using the projection requires no shifting.

It is worth noticing that, after the projection the projected document is merged with the original one, thus for small documents the execution of the update query is less time consuming without the projection. The time execution can be reduced by integrating the Merge algorithm with the XML serialization process of BaseX.

### 3.4 eXist

eXist is a Native XML Database which stores XML document in hierarchical collection. To map the XML document eXist use dynamic level numbering [18]. Important to note, that the general data structural used in eXist supports the structural updates. First we start by an example illustrating the DLN encoding.

**Example 3.4.1. DLN encoding used in eXist.**

To illustrate DLN we use the document $doc$ (see fig. 3.12-(a)). As the reader can observe, while traversing $doc$ a unique $ID$ is assigned to each node. For instance, the node labelled by $c$ has $ID$ equal to 1.2.1. This $ID$ is calculated in the following way: $ID$ of the root labelled by $a$ is assigned to value 1 (see fig. 3.12-(b)). $ID$ of the child is calculated by appending to $ID$ of the parent node the delimiter "." and the numeric value representing the position of the node in the current level, denoted $level$ value. As it is illustrated in Figure 3.12-(b), the node labelled by $d$ (child of the $a$-node) has $level$ value equal to 2, thus $ID$ of the $d$-node is equal to 1.2. Finally, $ID$ of the $c$-node is calculated as follows: $n_d.parentID.n_c.level$=1.2.1. It is worth noticing that the $level$ value of a left sibling of a given node must be less than the last one. For our example we have that $ID$ of the $k$-node is 1.4 which is less than $ID$ of the $p$-node 1.3.

This encoding makes possible to avoid the renumbering of $ID$ values after a new node insertion. For example, as it is illustrated in Figure 3.12-(c), new nodes have been inserted before the node labelled by $b$ and after the node labelled by $f$. New
3.4. eXist

IDs are calculated using the idea of sub-value. For example, ID of the new node labelled n is equal to 1.0/1. This sub-values can be used recursively. For instance, to insert a node between nodes having IDs 1.1/1 and 1.1/2 we can add a further sub-value level and assign 1.1/1/1 to the new node.

Based on this numeric scheme we can easily identify structural relationships between nodes, such as parent/child, ancestor/descendant or previous-/next-sibling.

Axes Relationships Before giving the rules identifying these relationships, it is important to note that, all IDs are encoded in bits. In binary encoding, the level separator ‘.’ is represented by a 0-bit while ‘/’ is written as a 1-bit. For example, the id 1.1/4 is encoded as follows:

0001 0 0001 1 0100

All path relationships are calculated on the bits. Based on this encoding we have the following properties of DLN:

1. It supports the computation of ancestor-descendant relationships between two nodes using the length l of the ID with smaller value. If the first l bits of a node n are identical then it is an ancestor of a node n'. At least one 1-bit is appended to ID of the parent node leading to a greater ID value,

2. n, n', are the following sibling nodes with ids ID_1, ID_2, respectively, if ID_2 was created using ID_1 and a sub-value, then ID_1<ID_2, and the prefix of ID_1 is equal to the one of ID_2. Finally, ID_2 has at least one 1-bit at the next position.

Next, we cover the general data structure used in eXist.

3.4.1 General Data Structure

The following example illustrates the data storage architecture used in eXist.

Example 3.4.2. Data storage architecture.

First of all it is important to note that, in eXist documents are managed in hierarchical collections, similar to storing files in a file system. This collection hierarchy is managed by the collections.dbx. Each collection has unique identifier ID. The next important feature of this architecture is that data is stored on B+ trees and paged files. The file dom.dbx collects nodes in a paged file and associates unique node identifiers to the actual nodes.

Figure 3.13-(c) illustrates the structure of dom.dbx for XML documents given in Figure 3.13-(a,b). Each document in the collection has its own unique ID - docID. The stored data is backed by multiroot B+ trees. B+ tree keys are pairs of <docId, nodeID>. Using these keys we could search for the address of the actual node object corresponding to the given nodeID. For example, if we need to search for the
Figure 3.13: Data storage in eXist
3.4. eXist

address of the node with ID=1.2.3 in the first document (docID is 1), the engine searches in the $B+$ tree starting from the root doc1, then terminates after finding the leaf in tree, containing the searched key. After that reads the storage address from DataPage, where the node object nodeIDs is preserved and accesses it. After that, all properties of the node object could be retrieved. The same scenario will take place if we search for any node objects in the second document, with only difference that the processing will start from, the root docID=2.

Because the access to the persistent DOM is always expensive, eXist has been implemented in a way that XPath or XQuery expressions are processed mostly without accessing dom.dbx (an example of this processing is given in the next paragraph). On the contrary, the query is executed in such a way that node relationships could be identified using the node sets. This can be achieved via the combination of structural joins and numbering scheme. All this is managed by the third indexed file in the architecture, called elements.dbx. It maps an element and an attribute QNames to a list of tuples $<$docID,nodeID$>$, where:

$docID$ – unique identifier for the document

$nodeID$ – ID assigned by a level-order traversal of the document tree (DLN)

Similarly to the dom.dbx, elements.dbx, depicted in Figure 3.13-(d), is backed by $B+$ trees, with the difference that the keys are a pair of $<$collection-id, name-id$>$. The addresses in the leaves of $B+$ tree are pointing to the array values containing an ordered list of nodeIDs separated by docIds. For example, to find, all elements labelled p, the query engine will need a single index lookup to retrieve the complete set of node identifiers pointing to that elements. Suppose we are searching for the nodes IDs of an element labelled p in the tree Figure 3.13. The key for this node is $<1,ns:p>$. Once the leaf, where the searched key is stored, is found we can retrieved the address of the array value containing all NodeIDs of the tag, which is in our example equal to 1.3. Next, depending on the query expression, if during the evaluation process the engine needs to retrieve some property of the node object, like attribute or text, it will be first looked up in elements.dbx to retrieve the nodeID of the searched tag. Once nodeID of the tag is found, the engine will search for the address of the node object corresponding to that ID in the dom.dbx

The last index file is words.dbx. By default, eXist indexes all text nodes and attribute values by tokenizing text into keywords. In words.dbx, which has a similar structure as elements.dbx, the extracted keywords are mapped to an ordered list of documents and unique node identifiers. The $B+$ tree key for this file consist of a pair $<$collectionID, keyword$. Each entry in the value list points to a text or attribute node where the keyword occurred.
XQuery Processing  Based on the numbering scheme features, eXist uses structural joins to evaluate path expressions.

Example 3.4.3. XPath expression evaluation.

For example, consider the XPath expression specified by \( a//b [s="so"] \)  which is executed on the tree \( \text{doc}_1 \) 3.13-(a). The expression is decomposed into three sub parts:

- **a** - The engine retrieves the set of nodes labelled by \( a \).
- **b** - The engine retrieves the set of nodes labelled by \( b \).
- **b[s]** - The engine retrieves the set of nodes labelled by \( b \) and having a child text node \( s \).
- **s="so"** - The engine retrieves all nodes having keyword "so".

It is important to note, that the exact positions of all elements for the first two expressions are retrieved from \( \text{element.dbx} \) in a way explained in the previous section. Therefore, each node in the set is described by the \( <\text{docID}, \text{nodeID}> \) tuple ordered by document order. While for the third one it is looked up at \( \text{words.dbx} \).

Then the engine executes structural join on the first two node sets, to find all nodes from the \( b \) set being descendants nodes in the \( a \) node set. As a result of this join a new set is created which serves as the context node set for the following expression. Therefore, this new node set for expression \( a//b \) becomes ancestor for the node set for expression \( b[s] \), while the descendant node set is generated from the evaluation of the expression \( s="so" \). Figure 3.14 exhibits two tables containing the sets and the joins results for the \( a//b \) and \( b[s] \), receptively.

### 3.4.2 XML serialization

To serialize a document stored in the database eXist uses several Java classes. The Serializer base class, used to serialize a document or document fragment back to XML. This class offers two overloaded methods: serialize() and toSAX().serialize(). The first one returns the XML as a string, the second one generates a stream of SAX events.
Serializer accepts NodeValue, NodeProxy, and DocumentImpl. NodeValue class represents a node value and may either be an in-memory node or a persistent node. DocumentImpl class represents a persistent document object stored in the database. NodeProxy is an internal proxy class which stores the node’s unique id and the document it belongs to. NodeProxy is acting as a placeholder for all types of persistent XML nodes during query processing.

### 3.4.3 eXist vs. projection

Figure 3.15 illustrates the difference between the execution of XPath expression from the example 3.4.3, with and without projection. Figure 3.15-(b) shows doc3 (see fig. 3.15-(a)) stored in element.dbx, while Figure 3.15-(c) stores the projected doc3. As the reader can observe, the second file contains only one node labelled by b having the nodeID=1.2, while the first one stores three nodes: nodeID=1.2, nodeID=1.3.1 and nodeID=1.3.2. The reason is that while the projection the b-nodes having nodeIDs 1.3.1 and 1.3.2 have been pruned out (see Chapter 4). As a consequence, to evaluate the path a//b selected setB (see fig. 3.15-(f)) of the projected doc3 contains only one element vs. the three ones of the original one (see fig. 3.15-(d)). Therefore, we can state that using the projection with eXist optimize the memory usage.
Similarly to MonetDB/XQuery and BaseX to reduce the total execution time with the projection it is necessary to integrate the Merge algorithm with the implementation of the serialization.

3.5 Saxon Processor

In this paragraph we will provide some details and explanations about the architecture of the Saxon Processor. It is important to note, that Saxon must not be considered as a database system. In general the query compilation is done without any information about the content of the input document in advance, which means that it does not maintain persistent indexes, like NXDs or enabled databases.

3.5.1 General Data Structure

Differently from implementations that wraps external object models e.g. DOM and etc., Saxon has two native models: linked tree and tinytree, each one implementing their own builder and navigation classes. The first model is an "object-per-node" tree structure in which parent nodes contain a list of their children. The second one represents a document using six arrays of integers. We will explain the structure of the tinytree using the following example:

Example 3.5.1. TinyTree array.

To illustrate this example we chose the document doc used in Example 3.2.1. Data corresponding to each node is stored in arrays. These arrays are preserved in document order and contain one entry for each node and are indexed by node number. It is important to note, that the attribute and namespace nodes are stored in attributes and namespace tables, respectively. Each array contains:

- **node code** - This is an integer value that references to the NamePool object. It is used to determine the prefix, local name, or namespace URI of an element or an attribute name. As the reader can see, Figure 3.16-(c) exhibits the NamePool object, where the local names are stored. For instance, the node code value for the root labelled by a is equal to n1. If we search for the corresponding name code in the NamePool, we will find that the local name value of that element is a.

- **depth** - It preserves the depth of the node in the tree. For instance, the depth value stored in the array for the root 1 is equal to 0. For the node labelled by e set to 2.

- **node kind** - Stores the type of the nodes (e.g. element, text or comment). Each node type is represented as an integer value from 1 to 12. For our example, we have two types of nodes: element node and one text node. As it is illustrated in Figure 3.16-(a) the kind value for a-node is set to 1, while for the text node s having index 4 in the array, it is set to 3.
3.5. Saxon Processor

XML document *doc*

<table>
<thead>
<tr>
<th>Array Name</th>
<th>Array Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>name kind</td>
<td>1 1 1 1 3 1 1 1 1</td>
</tr>
<tr>
<td>next sibling</td>
<td>2 5 6 8</td>
</tr>
<tr>
<td>alpha</td>
<td>-1 -1 -1 0 -1 -1 -1 -1</td>
</tr>
<tr>
<td>beta</td>
<td>-1 -1 -1 -1 1 -1 -1 -1</td>
</tr>
<tr>
<td>depth</td>
<td>0 1 1 2 1 1 2 2 2</td>
</tr>
<tr>
<td>node code</td>
<td>n₁ n₂ n₃ n₄ n₅ n₆ n₇ n₈</td>
</tr>
<tr>
<td>array Indexes</td>
<td>0 1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

(a) TinyTree Arrays

![Diagram](image)

(c) NamePool

<table>
<thead>
<tr>
<th>name code</th>
<th>URI</th>
<th>Local Name</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>n₁</td>
<td>URI</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>n₂</td>
<td>URI</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>n₃</td>
<td>URI</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>n₄</td>
<td>URI</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>n₅</td>
<td>URI</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>n₆</td>
<td>URI</td>
<td>k</td>
<td></td>
</tr>
<tr>
<td>n₇</td>
<td>URI</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>n₈</td>
<td>URI</td>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.16: TinyTree structure
next sibling - It contains the indexes of the next siblings. For instance, in our example the next sibling of the node labelled by \( p \) having index 5, is the node labelled by \( k \), whose index in the array is equal to 6. Therefore, the corresponding value in the next sibling is set to 6.

alpha/beta - The meaning of "alpha" and "beta" depends on the node type. For text nodes, comment nodes, and processing instructions the alpha value of this property points to StringBuffer holding the text. For element nodes, "alpha" is an index into the attributes table, and "beta" is an offset into the namespaces table. If an element does not have any attributes or namespaces the value is set to \(-1\). For instance, in our example, there is no any element that has an attribute or namespace, therefore the values in the arrays are set to \(-1\). As the reader can observe, we have one text node the value of which is stored in the StringBuffer (see fig. 3.16-(b)). This buffer contains only "s" string. It is worth noticing, that alpha for the c node is set to 0. The beta for the same node stores the offset of the text value in the buffer, which is the length of the string. For our example, the length of the text is equal to 1, thus beta value is set to 1.

The TinyTree model is designed to minimize the memory usage. It avoids the overhead of instantiating one Java object for each node in the tree, which is the case for DOM. The only drawback of this model is that it can not support XQuery Updates. Updates are supported by the mutable linked tree model, which will be explained latter.

3.5.1.1 Updatable Data Structure

As it has been stated in the previous section the TinyTree model is efficient for memory savings, but it is not updatable, since it is based on static allocation of space in fixed arrays. In order to support XQuery updates in Saxon, some changes have been applied on the linked tree model. It has been changed in a way to become similar to a mutable tree. Similar to a mutable tree, the linked tree, which preserves the references to the children, now additionally, stores references to the parent Objects. Figure 3.17 illustrates the structure of the mutable linked tree for doc. In this tree, for instance, the root node labelled a is represented as an object that stores the local name a, and the list of references to the children objects: \( n_b, n_d, n_p \) and \( n_k \). The value and the parent properties are set to null. The first child of the a-node, b-node preserves following properties: name=b, parent=n_a, which is a reference to the parent object. The children and the value properties are set to null. As the reader can see, the remaining nodes are mapped into the objects in the same way, except the text node n_s, where value="s", while name=null.

Let us suppose that an update query specified by insert nodes \{<n/>,<m/>\} into a/p is applied.
3.5. Saxon Processor

Figure 3.17: Mutable linked tree before update

Figure 3.18: Updated mutable linked tree
The result of this update is illustrated in Figure 3.18. As the reader can observe, new Objects: Object \(_n\), \(n\) and Object \(_n\), \(n\) are created with parent variables containing the reference to Object \(_n\), \(n\).

### 3.5.2 XML Serialization

Saxon has two ways of serialization: raw and wrapped outputs. The first one works only if the result consists of a single document or element node. It outputs the subtree rooted at that element node in the form of a serialized XML document. The second one works for any result sequence (ex. a sequence of integers, a sequence of attributes). Each item is wrapped as an XML element, with details of its type and value.

To produce the wrapped output, first the result sequence is wrapped as an XML tree, next this tree is serialized. To produce the unwrapped output, we skip the wrapping stage and just call the serializer directly. The QueryResult class is used for both cases. The QueryResult.wrap method is used during the wrapped output. It takes as an input the iterator produced by evaluating the query and produces as an output a DocumentInfo object representing the results wrapped as an XML tree. The QueryResult.serialize method takes any document or element node as an input, and writes it to a specified destination, using specified output properties. The destination is supplied as an object of the Result class.

### 3.5.3 Saxon vs. projection

As the reader can observe, the updatable data structure used in Saxon is not very effective for memory savings. The weak point is that it occupies lots of space and operates in main-memory.

Applying the projection helps to optimize the memory usage: executing the update specified in the example given in Figure 3.18 on doc allocates eleven Objects vs. four using projection. It is worth noticing that using the projection requires several changes to be applied to integrate the Merge algorithm with the serialization.

### 3.6 Conclusion

In this Chapter we have examined internal data representation and evaluation strategies of main XQuery engines, namely: MonetDB/XQuery, BaseX, eXist and Saxon.

Both MonetDB/XQuery and BaseX use relational XML encoding to store data. These systems compared to the others are most efficient for memory savings. Nevertheless, executing structural updates (performing insertion or deletion) using these systems maybe be more expensive from the execution time point of view.

For instance, executing structural updates performing an insertion of new nodes using MonetDB/XQuery can require a new logical page insertion. In this case
several shifts of nodes within the existing page and recalculations of some properties are performed.
For BaseX new nodes insertion may, as well, result in a new page insertion and the recalculation of some properties values. It worth noticing that in BaseX no gaps are allowed, thus if after executing an update the page contains an empty tuple, the tuples following it are shifted up.

To perform an update eXist first retrieves the set of types corresponding to the target path and then evaluates query on them.
Saxon maps each tree node to a DOM Object, thus projection optimizes the memory usage by pruning the nodes which are not used by an update.
As the reader can observe all these systems have memory limitations, next Chapter introduces the method to optimize memory usage while updating documents using these systems.
Chapter 4

Enabling XML Update Optimization based on Type Projection

Contents

4.1 Motivations ................................................. 55
4.2 The three level type-projector .......................... 58
4.3 Merge for enabling XML Update Optimization... 69
   4.3.1 The procedure \textit{NoMerge} ....................... 71
   4.3.2 Procedure \textit{OlbMerge} .......................... 75
4.4 Implementation and Experiments .................... 80
   4.4.1 Implementation issues ............................ 80
   4.4.2 Experiments ........................................ 87
4.5 Conclusion .................................................. 104

This chapter is devoted to introducing and illustrating, through examples, the main features of our method of XML Update Optimization and especially of the \textit{Merge} Algorithm.

In Section 4.2 we introduce some basic notions of the \textit{three-level} type projector.

In Section 4.3 we introduce the Merge algorithm together with its two procedures \textit{NoMerge} and \textit{OlbMerge}.

4.1 Motivations

The choices and assumptions made in the formal presentation are motivated. As it has been stated in Introduction, XML projection is technique to reduce memory consumption in XQuery in-memory engines. The main idea behind this technique is quite simple: given a query $q$ over an XML document $t$, instead of evaluating $q$ over $t$, the query $q$ is evaluated on a smaller document $t'$ obtained from $t$ by pruning out, at loading-time, parts of $t$ that are irrelevant for $q$. The queried document $t'$, a projection of the original one, is often much smaller than $t$ due to selectivity of queries.

Applying, the Projection Technique is not sufficient, since updating a projection of a document $t$ is not equivalent to updating the document $t$ itself: the pruned out
sub-trees will be missing. Thus a method has to be found in order to make updates persistent.

We propose to investigate a type based technique for optimizing updates. The update scenario is designed as follows for an update $u$ and a document $t$ typed by a DTD $D$. First, the projection $t'$ of $t$ is built using a type-projector $\pi$. Second, the update $u$ is performed over the projection $t'$, yielding the partial result $u(t')$. We would like to emphasize that no rewriting of the update $u$ is required. The last step, called Merge, parses in a streaming and synchronized fashion both the original document $t$ and $u(t')$ in order to produce the final result $u(t)$. For the sake of efficiency, the Merge step is designed so that (a) only child position of nodes and the projector $\pi$ are checked in order to decide whether to output elements of $t$ or of $u(t')$ and (b) no further changes are made on elements after the partial updated document $u(t')$ has been computed: output elements are either elements of the original document $t$ or elements of $u(t')$. It should be noted that the revalidation issue is not considered.

To sum up, our technique processes following three steps:

**Step 1** - an update type projector $\pi$ for $u$ is inferred and $t$ is projected wrt $\pi$. The notion of update type projector has been defined as well as the inference of the type projector. This part is M. A. Baazizi’s contribution [11].

**Step 2** - the update $u$ is evaluated over the projected document $\pi(t)$ producing an updated partial document $u(\pi(t))$;

**Step 3** - the fully updated document $u(t)$ is built by merging the initial document $t$ and $u(\pi(t))$; this step, called Merge, is detailed below in section 4.3. This part is my contribution [16, 17].

**Example 4.1.1. Motivating example**
Let us consider the update $u$ specified by $\text{for } x \text{ in } /doc/a \text{ where } x/d \text{ return delete node } x/b$ with DTD $D$ and document $t$ of Figures 4.1-1 and 4.1-4. Note that for the sake of simplicity, the two rules defining $f$ and $g$ are omitted. These rules are $f \rightarrow \epsilon$ and $g \rightarrow \epsilon$ where $\epsilon$ is an empty regular expression. In order to produce the final result $u(t)$, we extract the type-projector, then project the document $t$, execute the update and merge the initial document $t$ and the partial updated document $u(t')$.

**projection extraction** - Intuitively, the paths corresponding to data relevant for the update $u$ are $/doc/a/b$ and $/doc/a/d$. The labels of nodes traversed by these paths are $\{doc, a, b, d\}$. The projector inferred for the update $u$ is given by this set of labels (see fig. 4.1-3), with the intention to keep the nodes of the document that are typed by labels in $\pi$.

**projection** - First, we apply projection on the document $t$. Notice that each node of the initial document $t$ is adorned with its label $(a, b, ...)$ and with an identifier $i$ inside square brackets $(1, 1.1, 1.2...)$. A node of a document $t$
4.1. Motivations

Figure 4.1: A motivating example of the update scenario
whose identifier is \( i \) is denoted by \( t@i \). We make the choice that the identifier of a node in \( t \) gives its position in \( t \) according to document order. However, it should be reminded that this only holds for the initial document \( t \). During the projection we pruned out all nodes having labels that do not belong to \( \pi \) (see fig. 4.1-5). The projection first outputs node \( t@\varepsilon \) labelled by \( \text{doc}\in\pi_{\text{no}} \). Next, it selects node \( t@1 \) labelled by \( a\in\pi_{\text{no}} \) and its children \( t@1.1 \) and \( t@1.4 \) labelled by \( b\in\pi_{\text{no}} \) and \( d\in\pi_{\text{no}} \). Projecting the second sub-tree of the root proceeds in a similar manner. Note that the position of the node \( t'@1.4 \) in \( t' \) is 1.2 and not 1.4 like it is in \( t \).

**update** - The partial updated document \( u(t') \) (see fig. 4.1-6) reflects the changes applied by the update \( u \): node \( t'@1.1 \) labelled by \( b\in\pi_{\text{no}} \) of the projection \( t' \) has been deleted.

**merge** - In this example we illustrate the basic steps of the *Merge* algorithm. The goal is to build the final result starting from \( t \) and the partially updated document \( u(\pi(t)) \). The idea is to parse both documents in a synchronized manner. For example, *Merge* proceeds as follows: first while merging \( t \) and \( u(t') \), nothing special happens until the nodes \( t@1 \) (see fig. 4.1-4) and \( u(t')@1 \) (see fig. 4.1-6), both labelled \( a \), have been parsed. At this point, the two nodes examined by *Merge* are: the first child node \( t@1.1 \) labelled \( b \) of \( t@1 \), and the first child node \( u(t')@1.4 \) labelled \( d \) of \( u(t')@1 \). Because the child rank 4 of \( u(t')@1.4 \) is strictly greater than the child rank 1 of \( t@1.1 \) and because the label \( b \) belongs to the projector \( \pi \) indicating that the node \( t@1.1 \) has been projected in \( t' \), the node \( t@1.1 \) is not output (it has been deleted by the update \( u \)), the original document \( t \) is further parsed. The next two nodes examined are: \( t@1.2 \) labelled \( c \) and \( u(t')@1.4 \) labelled \( d \). Once again, the child rank 4 of \( u(t')@1.4 \) is strictly greater than the child rank 2 of \( t@1.2 \), however this time, the label \( c \) does not belong to the projector \( \pi \) (the node \( t@1.2 \) was not needed for the partial update and thus not projected in \( t' \)) and thus the node \( t@1.2 \) is output in the final result (see fig. 4.1-7), the original document \( t \) is further parsed. The process will continue parsing \( t \) and \( u(t') \) until both documents are fully scanned. Note that, positions of nodes (more precisely child rank) in the initial document play a crucial role in the *Merge* process. □

The following sections cover the Projection Technique (4.2) and The Merge Algorithm (4.3) respectively.

### 4.2 The three level type-projector

The content of this section is the contribution of M. A. Baazizi, who developed the formalization of the projector.

The *type-projector* designed for the purpose of update optimization has features
related to

-the update expressions and
-the Merge algorithm.

The type-projector \( \pi \), used to prune out the tree \( t \) is a 3-level projector which consists of components \( \pi_{\text{no}} \), \( \pi_{\text{olb}} \) and \( \pi_{\text{eb}} \), where \( \text{no} \) stands for "node only", \( \text{olb} \) stands for "one level below" and \( \text{eb} \) stands for "everything below". Each component is a set of labels (node types). In the following paragraphs, we provide examples motivating each component of the type-projector \( \pi \).

The behavior of the projector is different for each kind of components. For instance, during projection:

- if a node is labelled by a type in \( \pi_{\text{no}} \), it is projected and its children are visited to check if they need to be projected,
- if a node is labelled by a type in \( \pi_{\text{olb}} \), it is projected as well as its children. Each child will be visited and if its label belongs to \( \pi \) its children will be examined for projection \( \text{wrt} \) to the semantics of the projector,
- if a node is labelled by a type in \( \pi_{\text{eb}} \), it is projected together with all its descendants.

Note that:

- if a node label does not belong to any of the projector components it is not projected (exceptions are side effects of \( \pi_{\text{olb}} \) and \( \pi_{\text{eb}} \) components, see above) and its descendants are pruned out,
- if a node is not projected, then its children are not projected either,

Note that, if a node is projected as a side effect of \( \pi_{\text{olb}} \) and \( \pi_{\text{eb}} \) and its label does not belong to \( \pi \), then its children are not projected.

In the following paragraph, we provide examples motivating each component of \( \pi \). First we provide example explaining the use of the \( \pi_{\text{no}} \) component. To make it clear to the reader we explain the four steps of the projection technique applied on the document being updated: projector extraction, projection, update and merge.

Example 4.2.1. "node only" label and delete operation.

To explain the application of the 3-level projection which contains only component \( \pi_{\text{no}} \), we will consider the example of Figure 4.2.

projection extraction - The update \( u_1 \) (see fig. 4.2-1) involves a "delete" operation. Intuitively, the path corresponding to data relevant for the update \( u_1 \) is \( \text{doc}/a/d \) and the types of nodes traversed by this path are \( \text{doc}, a, d \). The type
involves a projection. The projection outputs the following nodes:

\[ \pi_{no} = \{doc, a, d\} \]

(2) The three level type projector \( \pi_1 \)

(3) The XML document \( t \)

(5) The partial result \( u_1(t_1) \)

(4) The projection \( t_1 \) of \( t \) wrt \( \pi_1 \)

Figure 4.2: The projector component \( \pi_{no} \), illustrated for "delete"

projector for this update will contain only one component \( \pi_{no} = \{doc, a, d\} \) (see fig. 4.2-2).

projection - During projection only the nodes labelled by types contained in \( \pi_{no} \) are projected. In our example projection first outputs node \( t@\varepsilon \) (labelled by \( doc \in \pi_{no} \)) of the tree \( t \) (see fig. 4.2-3,4). Next, it selects node \( t@1 \) labelled by \( a \in \pi_{no} \) and node \( t@1.1 \) labelled by \( d \in \pi_{no} \). After that it outputs nodes \( t@2 \) and \( t@2.1 \) for the same reason. Note that, the projection does not output node \( t@3.1 \) labelled by \( d \in \pi_{no} \), since the parent node \( t@3 \) is labelled by \( e \in \pi_{no} \) and has not been projected.

update - The partially updated document \( u_1(t_1) \) (see fig. 4.2-5) reflects the result of the execution of query \( u_1 \) over the projection \( t_1 \). The new partially updated tree \( u(t_1) \) contains only nodes \( t_1@1 \) and \( t_1@2 \) because the children nodes \( t_1@1.1 \) and \( t_1@1.2 \) have been deleted.

merge - Merge is processed on the trees \( t \) and \( u_1(t_1) \) to obtain a final result \( u_1(t) \) (equivalent to the update performed on \( t \)). We cover the behavior of Merge in details in the next section. 

Example 4.2.2. "node only" component and "rename" operation.

The update \( u_2 \), given in Figure 4.3 involves a "rename" operation.

projection extraction - As for the previous example the path corresponding to relevant data wrt to the update is \( doc/a/c \) and the type-projector contains only a \( \pi_{no} \) component (4.3-2).

projection - The projection outputs the following nodes: \( t@\varepsilon \) labelled by \( doc \in \pi_{no} \), \( t@1 \) labelled by \( a \in \pi_{no} \) followed by \( t@1.3 \) and \( t@1.4 \) labelled by \( c \in \pi_{no} \), finally node \( t@2 \) labelled by \( a \in \pi_{no} \) (see fig. 4.3-4). Once again, note that the node \( t@3.2 \) is not projected because its parent node \( t@3 \) is labelled by \( e \in \pi_{no} \) thus not projected.
4.2. The three level type-projector

for $x$ in /doc/a
return
rename node $x/c$ as "b"

(1) The update $u_2$

(2) The three level type projector $\pi_2$

(3) The XML document $t$

(4) The projection $t_1$ of $t$ wrt $\pi_2$

(5) The partial result $u_2(t_1)$

Figure 4.3: The projector component $\pi_{no}$, illustrated for "rename"

update - The partially updated tree $u_2(t_1)$ (see fig. 4.3-5) illustrates the changes applied during the evaluation of the update $u_2$ on the projected tree $t_1$. Mainly the labels of two nodes $t_1@1.3$ and $t_1@1.4$ labelled by $c \in \pi_{no}$ have been renamed to $b$. 

merge - Merge synchronizes the trees $t$ and $u_2(t_1)$ to obtain the final result $u_2(t)$.

The $\pi_{olb}$ component is introduced for queries involving "insert as first/last", "insert before/after" or "replace" operations. Replace updates have to be treated like insert wrt to the target path: replace is a delete followed by an insert.

Example 4.2.3. The example given in Figure 4.4 motivates the need of the "one level below" component of the projector. Let us show that the "node only" projection is not adequate here by showing the whole scenario. We start with by treating the example using only $\pi_{no}$ component, to show its deficiency.

Using the $\pi_{no}$ projection

projection extraction - The update query $u_3$ involves an "insert as first" operation (see fig. 4.4-1). Intuitively, the path corresponding to data relevant for the update $u_3$ is $doc/a$ and the types of nodes traversed by this path are $doc, a$. Thus, let us consider the projector containing one component $\pi_{no} = \{doc, a\}$ (see fig. 4.4-2).

projection - The projection $t_1$ (see fig. 4.4-4) for the given projector applied on the document $t$ proceeds as follows: first the root node $t@\epsilon$ labelled by $doc$ is selected, followed by the nodes $t@1$ and $t@2$ labelled by $a$.

update - The result of the evaluation of the query $u_3$ on the projected tree $t_1$ is illustrated in Figure 4.4-5. The subtrees $t_3@1$ and $t_3@2$ of the partially updated tree $u_3(t_1)$ (denoted $t_3$) contain two children nodes $t_3@i$ and $t_3@i1$. Note that, $i$ and $i1$ are new identifiers and that they convey no information about the child rank of the new nodes.
Figure 4.4: The projector component $\pi_{\text{olb}}$, illustrated for "insert as first"
merge - While parsing $t$ and $t_3$ and examining node $t@1.1$ and $t_3@i$ there is no way to decide whether $t_3@i$ has to be output as first or in another order. Recall here our assumption: no rewriting performed on the update and Merge has no access to the update. □

The projector of Figure 4.4-2 is not appropriate, because it does not keep enough information for the last step of the evaluation. The proposed solution is to introduce another component $\pi_{\text{olb}}$ (see fig. 4.4-6).

Introducing the $\pi_{\text{olb}}$ projection

projection extraction - The new projector for the update $u_3$ takes into account that the path /doc/a is the target of an insertion. As such, the projector will have 2 components: the type doc of category "node only" and the type a of category "one level below". The label a belongs to $\pi_{\text{olb}}$ because the update $u_3$ is suppose to perform an insert "below" nodes of type a.

projection - Applying this projector to the document $t$ proceeds as follows: for our example it first outputs node $t@\varepsilon$ followed by node $t@1$ labelled by $a\in\pi_{\text{olb}}$ together with its children $t@1.1$, $t@1.2$, $t@1.3$ and $t@1.4$ (see fig. 4.4-7). We have the same for $t@2$.

update - The partially updated tree $u_3(t_2)$ (denoted as $t_3'$) contains all children of $t_2@1$ and $t_2@2$ plus newly inserted ones having identifiers i and i1 (see fig. 4.4-8).

merge - During the Merge phase the synchronization of the trees $t$ and $t_3'$ leads to the correct result. While synchronizing nodes $t@1.1$ and $t_3'@i$, $t_3'@i$ is output as the first child (see fig. 4.4-9). Merge uses the fact that $t@1$ is of type $a\in\pi_{\text{olb}}$ to enter a mode where $t_3'$ guides the synchronization: it is known that every child of $t@1$ have been projected and thus every child of $t_3'@1$ (the old and the new one) are in the right order. □

Example 4.2.4. "one level below" component for "insert before" operation.

Now let us consider the example given in Figure 4.5 with the update $u_4$ which involves an "insert before" operation.

projection extraction - This update intends to insert a new node before the target path /doc/a/d (see fig. 4.5-1). Thus the projector $\pi_4$ has two components: the "node only" component $\pi_{\text{no}}=\{\text{doc}\}$ and the "one level below" component $\pi_{\text{olb}}=\{a\}$ (see fig. 4.5-2).

projection - Applying this projector to the document $t$ (see 4.5-3) proceeds as follows: first it outputs node $t@\varepsilon$ labelled by $\text{doc}\in\pi_{\text{no}}$ followed by node $t@1$ labelled by $a\in\pi_{\text{olb}}$ (see fig. 4.5-4); after that, it outputs all children of $t@1$. It proceeds in similar way on $t@2$.

update - Figure 4.5-5 illustrates the changes applied by the update $u_4$: nodes $u_4(t_1)@i$ and $u_4(t_1)@i1$ has been inserted before the nodes $u_4(t_1)@i$ and $u_4(t_1)@i1$ respectively.
for $x$ in /doc/a
return
insert node <e/>
before $x/d$

(1) The update $u_4$

$\pi_{no} = \{doc\}$
$\pi_{olb} = \{a\}$

(2) The three level type projector $\pi_4$

(4) The projection $t_4$ of $t$ wrt $\pi_4$

Figure 4.5: The projector component $\pi_{olb}$, illustrated for "insert before"

**Example 4.2.5.** This example illustrates the "one level below" and mixed-content.

This example shows that the "node only" projection is not appropriate when dealing with mixed content.

**Using the $\pi_{no}$ projection**

Consider the update $u_5$ specified by

\[
\text{for } x \text{ in } /doc/a \text{ where } x/b/text() = 'foot' \text{ return delete node } x/d
\]

(see fig. 4.6-1). Let us consider the document $t$ given in Figure 4.6-3 and its projection $\pi_5(t)$.

**projection extraction** - Intuitively, /doc/a/d and /doc/a/b/text() are the paths corresponding to data relevant for the update $u_3$. The associated types are $\pi_5 = \{doc, a, b, String, d\}$ (see fig. 4.6-2).

**projection** - Let us consider the document $t$ given in Figure. 4.6-3 and its projection $\pi_5(t)$. Notice that projecting $t$ wrt $\pi_5$ has the side effect to concatenate the two Strings 'fo' and 'ot' (see fig. 4.6-4).
4.2. The three level type-projector

(1) The update $u_5$

$$\pi_{\text{no}} = \{\text{doc}, a, b, \text{String}, d\}$$

(2) The type projector $\pi_5$

$$\pi_5 \equiv \{\text{doc}, a, b, \text{String}, d\}$$

(3) The XML document $t$

$$\pi_{\text{olb}} = \{\text{doc}, a, b, \text{String}, d\}$$

(4) The projection $t_1$ of $\pi_5$

(5) The partial result $u_5(t_1)$

(6) The three level type projector $\pi'_5$

(7) The projection $t_2$ of $\pi'_5$

(8) The partial result $u_5(t_2)$

Figure 4.6: The projector component $\pi_{\text{olb}}$, for String and mixed-content
**update** - The node $t@1.4$ labelled by $d$ is deleted when the update $u_5$ is applied to the projected document $t_1$ (see fig. 4.6-5).

**merge** - Recall the assumption that *Merge* is not supposed to change the elements parsed in $t$ and $u_5(t_1)$ and has only access to the projector. The problem here is due to mixed-content nodes: when merging the initial document $t$ and the partial updated result $u_5(t_1)$, there is no way to be able to recover the right descendant for $t@1.1$.

The projector of Figure 4.4-2 is not appropriate, because of mixed-content nodes and their behavior. We now present how to solve this problem using the $\pi_{olb}$ component (see fig. 4.4-6).

**Using the $\pi_{olb}$ projection**

**projection extraction** - The new projector $\pi'_5$ generated for the example will have two components: $\pi_{no}=$\{$doc, a, d$\} and $\pi_{olb}=$\{$b$\} (see fig. 4.6-6).

Indeed, we could have solved the problem, in a syntactic manner, by extending the extracted path $/doc/a/b/text()$ to $/doc/a/b/text()/parent :: node()/child :: node()$ leading (by type inference) to a simple projector $\{doc, a, b, c, d, String\}$ which in fact projects the whole document $t$. On the other hand, the projector $\pi'_5$ allows us to restrict the projection of text nodes to children of $b$ nodes. To better illustrate this, let us assume that $doc$ is now defined by $doc(a | String)*$, then applying the simple projector $\{doc, a, b, c, d, String\}$ would lead to project all text children of $a$-nodes although not useful for the update.

**projection** - Applying projector $\pi'_5$ on the document $t$ does not concatenate Strings 'fo' and 'ot', since it projects all children of $t@1.1$ (see fig. 4.6-7)

**update** - Figure 4.6-8 illustrates the changes applied by the update $u_5$ on the tree $t_2$: node $t_2@1.4$ is deleted.

**merge** - The synchronization of the children of $t@1$ and $u_5(t_2)@1$ is guided by the nodes of $u_5(t_2)$.

**Example 4.2.6.** This example illustrates the "everything below" component for extracting element.

For example in Figure 4.7 the update $u_6$ involves a "replace" operation (see fig. 4.7-1). Recall that "replace" operation is a delete followed by an insert, therefore, the type-projector must contain $\pi_{olb}$ component.

**projection extraction** - The path $/doc/a/d$ is meant to return the element copied at the target node computed by $/doc/a/b$, thus the complete subtrees rooted at nodes of type $d$ have to be completely projected. Thus, for this update, the projector $\pi'_6$ is composed of three sets of types (see fig. 4.7-2): $\pi_{no}=$\{$doc$\} of category "node only", $\pi_{olb}=$\{$a$\} of category "one level below", and $\pi_{eb}=$\{$d$\} of category "everything below".
projection - Applying the projector $\pi_6$ to the document $t$ (see fig. 4.7-3) proceeds as follows: first root $t@e$ is projected, followed by node $t_1@1$. Because $t@1$ is labelled by $a \in \pi_{olb}$ all its children are projected. Note that, the complete subtrees rooted at node $t@1.1$ is projected (see fig. 4.7-4).

update - After executing the update $u_6$ on the projected tree $t_1$ the tree having the root $t_1@1.2$ is replaced by a new one (see fig. 4.7-5).

merge - While processing nodes $t@2.1$ and $u_6(t_2)@2.1$ Merge outputs the tree having root $u_6(t_2)@2.1$. □

We now proceed to a formal presentation of the three level projector. Once again, this part of the work is the contribution of A. Baazizi.

Update type projector  First of all, we formally define three-level type projectors:

Définition 1 (Type Projector). Given a dtd $(D, s_D)$ over the alphabet $\Sigma$, a type projector $\pi$ is a triple $($$\pi_{no}, \pi_{olb}, \pi_{eb}$$)$ such that ($\pi$ also denotes $\pi_{no} \cup \pi_{olb} \cup \pi_{eb}$):

i) $\pi \subseteq \Sigma$,

ii) $\pi_{no}$, $\pi_{olb}$ and $\pi_{eb}$ are pairwise disjoint, and

iii) $s_D \in \pi$ and for each $b \in \pi$ there exists $a \in \pi$ such that $D(a) = r$ and $b$ occurs in $r$.

The $\pi_{no}$ (resp. $\pi_{olb}$ and $\pi_{eb}$) component of $\pi$ contains "node only" types (resp. "one level below" and "$\forall$ below" types). Notice that condition iii) ensures some closure property wrt to the dtd $D$: label $a \in \pi$ cannot be deconnected from the root label $s_D$, although it does not need to be connected in all possible manners (see projector $\pi_4$ below). Notice that the String type itself never belongs to a type projector $\pi$: as explained in the example 4.2.5, a string is projected "indirectly"
when its parent node type is of category 'olb' or 'eb'.

The next definition formalizes the effect of executing a type projector on a document.

**Définition 2 (Type Projection).** Let us consider the DTD \((D, s_D)\), the type projector \(\pi=(\pi_{no}, \pi_{olb}, \pi_{eb})\) and the document \(t \in D\) with \(\text{roots}(t) = \{r_1\}\) and \(\text{subfor}(t)=F\). The projection of \(t\) wrt \(\pi\), denoted \(\pi(t)\), is the tree \(\Pi_{K(t, \pi)}(t)\) where \(K(t, \pi)\) is recursively defined by:

- if \(\text{lab}(r_1) \notin \pi\) then \(K(t, \pi) = \emptyset\),
- if \(\text{lab}(r_1) \in \pi\) then \(K(t, \pi) = \{r_1\} \cup K_\alpha(F)\) for \(\alpha \in \{\text{no}, \text{olb}, \text{eb}\}\) with:
  
  \[
  K_\alpha(F) = \emptyset \text{ if } F = () \text{ and otherwise, assuming } F = t' \circ F',
  \]

  \[
  K_{\text{no}}(F) = K(t', \pi) \cup K_{\text{no}}(F'),
  \]

  \[
  K_{\text{olb}}(F) = K(t', \pi) \cup K_{\text{olb}}(F') \text{ if } \text{lab}(r_U) \in \pi
  \]

  \[
  K_{\text{eb}}(F) = \text{dom}(F).
  \]

**Example 4.2.7. Example of the projection.**
For our example illustrated in Figure 4.7 the projector \(\pi\) is well-defined: it consists of three components \(\pi_{\text{no}}, \pi_{\text{olb}}\) and \(\pi_{\text{eb}}\). For the document \(t\), the set \(K(t, \pi)\) is \(\{\varepsilon, 1, 1.1, 1.2, 1.3, 1.4, 1.1.1, 1.1.2, 2, 2.1, 2.1.1\}\). This set has been obtained as follows:

\[
K(t, \pi) = \{\varepsilon\} \cup K_{\text{no}}(F) \quad \text{where } F \text{ is the sub-forest of } t \text{ and } \alpha = \text{no} \text{ because } \text{lab}(\varepsilon) \in \pi_{\text{no}}
\]

Let assume that \(F = t_1 \circ F'\) where \(t_1\) is the first tree of the forest \(F\) then:

\[
K_{\text{no}}(F) = K(t_1, \pi) \cup K_{\text{no}}(F')
\]

\[
K(t_1, \pi) = \{1\} \cup K_{\text{olb}}(F_1) \quad \text{where } F_1 \text{ is the sub-forest of } t_1 \text{ and } F_1 = t_{11} \circ F'_{11}
\]

\[
K_{\text{olb}}(F_1) = K(t_{11}, \pi) \cup K_{\text{olb}}(F'_{11})
\]

\[
K(t_{11}, \pi) = \{1.1\} \cup K_{\text{eb}}(F_{11}) \quad \text{where } F_{11} \text{ is the sub-forest of } t_{11}
\]

\[
K_{\text{eb}}(F_{11}) = \{1.1.1, 1.1.2\}
\]

Let assume that

\[
F'_{1} = t_{12} \circ F'_{2} \quad \text{where } t_{12} \text{ is the first tree of the forest } F'_{1}.
\]

\[
F'_{2} = t_{13} \circ F'_{3} \quad \text{where } t_{13} \text{ is the first tree of the forest } F'_{2}.
\]

\[
F'_{3} = t_{14} \circ F'_{4} \quad \text{where } t_{14} \text{ is the first tree of the forest } F'_{3}.
\]

\[
K_{\text{olb}}(F'_{1}) = \{1.2\} \cup K_{\text{olb}}(F'_{2})
\]

\[
K_{\text{olb}}(F'_{2}) = \{1.3\} \cup K_{\text{olb}}(F'_{3})
\]

\[
K_{\text{olb}}(F'_{3}) = \{1.4\} \cup K_{\text{olb}}(F'_{4})
\]

etc.
The closure property iii) of definition 1 entails that the result of a type projection is a well-formed tree although it may not conform to the DTD $D$.

### 4.3 Merge for enabling XML Update Optimization based on type projection

This section formalizes the Merge algorithm and provides the detailed explanations and examples for each step. Recall that the task of Merge is to build the result $u(t)$ of the update $u$ over $t$ starting from the initial p-tree $t$ and the updated partial tree $u(\pi(t))$.

The following assumptions are important for the definition of Merge.

1. The input XML document $t$ is valid with respect to the DTD $D$. For the purpose of the formal presentation, we assume that the tree $t$ is a p-store: the identifiers are the node positions (in document order).

2. The execution of the update $u$ has possibly produced new identifiers for the purpose of node creation induced by replace and insert operations.

The goal of merging the input document $t$ and the partial update $t'$ is to construct the update $u(t)$. Merging processes by parsing both trees $t$ and $t'$. The merge algorithm is decomposed as follows:

- The procedure $TreeMerge$ takes as input two subtrees $\tau$ and $\tau'$. The first one, $\tau$, is a subtree of the initial tree $t$. The second one, $\tau'$ is a subtree of the partially updated tree $t'$.

Let us assume that:

$$
\text{lab}(\text{roots}(\tau))=a_i \\
\text{lab}(\text{roots}(\tau'))=b_i \\
\text{subfor}(\tau)=F_i \\
\text{subfor}(\tau')=F_u 
$$

$Merge$ takes care of synchronization of parsing the trees $t$ and $t'$. Here we assume that the trees $\tau$ and $\tau'$ have identical root identifier: $\text{roots}(\tau)=\text{roots}(\tau')$. 

![Figure 4.8: TreeMerge processing](image-url)
They may have different labels if the update $u$ has renamed the label of the node $\text{roots}(\tau)$. The procedure $\text{TreeMerge}$ is quite simple: it builds a tree whose root is $\tau'$ root (see fig. 4.8) and whose sub-forest $F_r$ is generated as follows: the label $a_i$ of $\text{roots}(\tau)$ is checked with respect to $\pi$ components in order to decide how to merge the sub-forests $F_i$ and $F_u$. The procedure $\text{TreeMerge}$ is presented formally by:

$$F_r = \begin{cases} 
\text{NoMerge}(F_i \mid F_u) & \text{if } \text{lab}(\text{roots}(\tau)) \in \pi_{\text{no}} \\
\text{OlbMerge}(F_i \mid F_u) & \text{if } \text{lab}(\text{roots}(\tau)) \in \pi_{\text{olb}} \\
\text{subfor}(t_u) & \text{if } \text{lab}(\text{roots}(\tau)) \in \pi_{\text{eb}}
\end{cases}$$

Note that, in case of $\text{lab}(\text{roots}(\tau)) \in \pi_{\text{eb}}$ we have $\text{TreeMerge}(\tau \mid \tau') = \tau'$.

Next, the parent node of $F_i$, resp. of $F_u$ is denoted by $n$, resp. by $m$.

Now we are going to explain the functions $\text{NoMerge}$ and $\text{OlbMerge}$ which are formalized in Figures 4.9 and 4.15. For the sake of simplicity, the update projector $\pi$ is kept implicit in the specification.

The functions $\text{NoMerge}$ and $\text{OlbMerge}$ have to be thought of as mechanisms parsing in parallel two forests: $F_i$ belonging to the initial p-tree $t$ and $F_u$ belonging to the updated partial tree $u(\pi(t))$; synchronization is captured by the fact that the parent nodes of $F_i$ and $F_u$ are assumed to share the same identifier; because of projection and update, $F_u$ contains identifiers belonging to $t$, besides the new ones due to insert and replace operation.

The two functions differ on the following pre-conditions: (see the definition of $\text{TreeMerge}$)

$\text{NoMerge}$ assumes that (†) the parent node $n$ of the forest $F_i$ is of category "node only" which implies that, because of synchronization, i) none of the top level trees in $F_u$ is of type $\text{String}$, ii) root identifiers of top level trees in $F_u$ belong to $F_i$ that is $\text{roots}(F_u) \subseteq \text{roots}(F_i)$.

$\text{OlbMerge}$ considers that (††) the node $n$ is of category "one level below" which implies that each node in $\text{roots}(F_i)$ has been projected and that $\text{roots}(F_u)$ are exactly the top level nodes of $F_u$ that have to be output by $\text{OlbMerge}$.

We provide explanations and examples for each line of the formalization. First we start with the procedure $\text{NoMerge}$, next we explain the procedure $\text{OlbMerge}$.

The reader should pay attention to the fact that next we use $t_i$ and $t_u$ to designate the first tree of the forest $F_i$ (resp. $F_u$). In the following examples, we will explain
4.3. Merge for enabling XML Update Optimization

The function \( \text{NoMerge} \) (see fig. 4.9) proceeds as follows:

**Line 2:** Line 2 takes care of the case where the current parsed tree \( t_i \) of \( F_i \) is of type \( \text{String} \). The assumption \( \dagger \) entails that it has been pruned out by \( \pi \). Thus, \( t_i \) is simply output.

**Example 4.3.1.** Consider the update \( u_1 \) specified by \( \text{for } x \text{ in } /a \text{ where } x/a \text{ return rename node } x/c \text{ with } "b" \) illustrated in Figure 4.10.

**projector** - The type projector \( \pi_1 \) derived from the update \( u_1 \) has one component \( \pi_{\text{no}} = \{a,c\} \).

**update** - Figure 4.10-2 illustrates the changes applied by the update \( u_1 \): node \( F_i@1.1 \) labelled by \( c \) is renamed to \( b \) (see fig. 4.10-2).

**merge** - Because the parent node of \( F_i \) (see fig. 4.10-1) is labelled by \( a \in \pi_{\text{no}} \), the forests \( F_i \) and \( F_u \) are going to be processed by \( \text{NoMerge} \). Here, because the parsed tree \( t_i \) is of type \( \text{String} \) and the condition \( \sigma_{t_i}(\text{roots}(t_i))=\text{text}[st] \) is satisfied, \( \text{NoMerge} \) executes line 2 and outputs tree \( t_i \) as the first tree of \( F_r \) (see fig. 4.10-3). □
Figure 4.11: Example for the procedure $\text{NoMerge}$ line 3

**Line 3:** Line 3 deals with the case where the label $a$ of the root $\text{roots}(t_i)$ of $t_i$ belongs to $\pi$ (thus a subtree of $t_i$ has been projected) and $\text{roots}(t_i)$ does not occur in $F_u$ (the projection of $t_i$ has been deleted by the update). When $F_u$ is not empty, this latter fact is identified by comparing the identifiers of the currently parsed nodes (which are positions in $F$). $\text{roots}(t_i)>\text{roots}(t_i)$ indicates that the tree $t_i$ comes before the tree $t_u$ in the forest $F_i$. Thus $t_i$ is not output.

**Example 4.3.2.** Consider the update $u_2$ specified by $\text{for } x \text{ in } /a/b \text{ where not } x/c \text{ return delete node } x$ illustrated in Figure 4.11.

**projector** - The type projector $\pi_2$ derived from the update $u_2$ has one component $\pi_2 = \{a, b, c\}$.

**update** - Figure 4.11-2 illustrates the changes applied by the update $u_2$. The parent node of $F_u$ labelled by $a$ contains only two descendants $F_u@1.3$ and $F_u@1.3.1$ labelled by $b$ and $c$ respectively (trees rooted at $F_i@1.1$ and $F_i@1.2$ of the original tree have been deleted). Note that, the node $F_u@1.3$ has not been deleted by the update $u_2$, since it contains a child node labelled by $c$.

**merge** - Because the parent node of $F_i$ is labelled by $a \in \pi_2$ (see fig. 4.11-1), the forests $F_i$ and $F_u$ are going to be processed by $\text{NoMerge}$. First $\text{NoMerge}$ examines nodes $F_i@1.1$ and $F_u@1.3$, where $F_i@1.1$ is labelled by $b \in \pi_2$. Because the rank 3 of $F_u@1.3$ is strictly greater than the rank 1 of $F_i@1.1$ (the tree with the root $F_i@1.1$ has been deleted by the update $u_2$), $\text{NoMerge}$ applies line 3. Mainly, $\text{NoMerge}$ skips the tree with the root node $F_i@1.1$ (see fig. 4.11-3) and moves only on $F_i$. After that, $\text{NoMerge}$ processes nodes $F_i@1.2$ and $F_u@1.3$. Once again, we have that node $F_i@1.2$ is labelled by $b \in \pi_2$ and the rank 3 of $F_u@1.3$ is strictly greater than the one of $F_i@1.2$. Therefore, $\text{NoMerge}$ skips tree $F_i@1.2$, according to line 3, and parses $F_i$. Finally it examines nodes $F_i@1.3$ labelled $b \in \pi_2$ and $F_u@1.3$. This time we have that the ranks of the two nodes are equal, thus $\text{NoMerge}$ applies line 4, which is explained in the next paragraph. □

**Line 4:** Line 4 takes care of synchronization on the nodes $\text{roots}(t_u)$ and $\text{roots}(t_i)$: these nodes can only differ by their labels because of some potential renaming. In that case, the tree $\text{TreeMerge}(t_i \mid t_u)$ is output.
4.3. Merge for enabling XML Update Optimization...

Example 4.3.3. Consider the update $u_3$ specified by

\begin{verbatim}
for $x$ in /a/d return insert node <e/> as last into $x$
\end{verbatim}

as illustrated in Figure 4.12.

- **projector** - Because the update $u_3$ involves an "insert" operation, the type projector $\pi$ derives from it has two components $\pi_{no}=\{a\}$ and $\pi_{olb}=\{d\}$.

- **update** - Figure 4.12-2 illustrates the changes applied by the update $u_3$. Nodes $F_i@1.2$ and $F_u@1.3$ contains newly inserted nodes $F_i@i$, $F_u@i1$ and $F_u@i2$ respectively, where $i1$ and $i2$ are new identifiers.

- **merge** - Because the parent node of $F_i$ is labelled by $a\in\pi_{no}$ (see fig. 4.12-1), the forests $F_i$ and $F_u$ are going to be processed by $NoMerge$. First, $NoMerge$ processes nodes $F_i@1.1$ and $F_u@1.1$ and since they have equal ranks (the condition $\text{roots}(F_i@1.1)=\text{roots}(F_u@1.1)$ is true), $NoMerge$ synchronizes them according to line 4. It proceeds as follows: builds a tree $t_r$ having root node $F_u@1.1$ (see fig. 4.12-3). Because $F_i@1.1$ is labelled by $d\in\pi_{olb}$ the sub-forest of $t_r$ is defined by the procedure $OlbMerge$. Section 4.3.2 provides detailed explanation of the $OlbMerge$ behavior. For our example, we assume that the synchronization has been done and $NoMerge$, according to line 4, processes nodes $F_i@1.2$ and $F_u@1.2$. Here we have that $F_i@1.2$ is labelled by $d\in\pi_{olb}$ hence, once again it outputs $F_u@1.2$ and the synchronization of the first level nodes is specified by $OlbMerge$. Finally $NoMerge$ processes nodes $F_i@1.3$ and $F_u@1.3$ specified by line 4. \square

**Line5:** Finally, line 5 deals with the case where the label $a$ of $t_i$ root does not belong to the projector $\pi$ implying that $t_i$ has been pruned out. Hence $t_i$ is output.

Example 4.3.4. Let us slightly change the previous example by changing the input and adding a tree rooted at $t_i@1.1$ labelled by $b\notin\pi$ as the first child of the parent of $F_i$ (see fig. 4.13-1) and a tree rooted at $t_i@1.5$ labelled by $c\notin\pi$.

- **projector** - Projector applied to the document $t$ does not changed.

- **merge** - This time, $NoMerge$ first processes nodes $F_i@1.1$ and $F_u@1.2$. Because $F_i@1.1$ is labelled by $b\notin\pi$ it executes line 5 and outputs the tree rooted at $F_i@1.1$.
Example 4.3.5. The following example illustrated in Figure 4.14 explains the behavior of Merge while mixing cases. The example assumes that the update is composed of several elementary changes given in the previous examples. Let us consider the update $u_5$ specified by

\begin{verbatim}
for $x$ in /a
    return
    
    { rename node $x/c$ with "k",
      insert node <e/> as last into $x/d,$
      delete node $x/b$
    }
\end{verbatim}

As it has been explained in Chapter 2, the update primitives are held in the pending update list and are applied in restricted order. For our example we have that: first rename, next insert as last operations and finally delete operation are applied.

**projector** - The type projector $\pi_5$ derived from the update $u$ has two components $\pi_{no} = \{a, b, c\}$ and $\pi_{olb} = \{d\}$.

**update** - Figure 4.14-2 reflects all changes applied on the document $t$. Mainly, the first tree rooted at $F_i@1.1$ has been deleted, the new nodes have been inserted as the last children to nodes $F_i@1.2$ and $F_i@1.3$ resp., and the label of the node $F_i@1.4$ has been renamed (see fig. 4.14-2).

**merge** - Because the parent node of $F_i$, is labelled by $a \in \pi_{no}$ the forests $F_i$ and $F_u$ are going to be processed by NoMerge. Because the root $F_i@1.1$ is labelled by $b \in \pi_{no}$ and $\text{roots}(F_i@1.1) < \text{roots}(F_u@1.2)$ (the rank 2 of $F_u@1.2$ is strictly greater than the rank 1 of $F_i@1.1$), NoMerge executes line 3. Thus NoMerge skips the tree rooted at $F_i@1.1$ and executes the procedure NoMerge on the trees rooted at $F_i@1.2$ and
4.3. Merge for enabling XML Update Optimization...

\( F_1 \) @ 1.2, where \( F_1 \) @ 1.2 is labelled by \( d \in \pi_{olb} \). This time, since the ranks are equal, \( \text{NoMerge} \) applies line 4. Mainly it builds a tree \( t_r \) having the root \( F_1 @ 1.2 \) (see fig. 4.14-3). Because \( \text{lab}(\text{roots}(F_1 @ 1.2)) \in \pi_{olb} \) the sub-forest of \( t_r \) is defined by \( \text{OlbMerge} \). For our example, we assume that the synchronization has been done and \( \text{Merge} \) process nodes \( F_1 @ 1.3 \) and \( F_u @ 1.3 \), once again, specified by line 4. After that, it examines nodes \( F_1 @ 1.4 \) and \( F_u @ 1.4 \) specified by line 4, thus it selects \( F_1 @ 1.4 \).

4.3.2 Procedure \text{OlbMerge}

Recall that the function \( \text{OlbMerge} \), specified in Figure 4.15 is built assuming that (\( \top \top \)) the node \( n \) is of category "one level below" which implies that each node in \( \text{roots}(F_1) \) has been projected and that \( \text{roots}(F_u) \) are exactly the top level nodes of \( F_u \) that have to be output by \( \text{OlbMerge} \). Parsing \( F_1 \) and \( F_u \) in parallel is essentially guided by \( F_u \), as opposed to \( \text{NoMerge} \).

Similarly to the procedure \( \text{NoMerge} \), we provide examples in order to illustrate each line of the formalization of \( \text{OlbMerge} \).

**Line c.2** Line c.2 deals with the case where the current parsed tree \( t_u \) of \( F_u \) is either of type \text{String} or a newly inserted element. This latter case is identified by checking whether the identifier \( \text{roots}(t_u) \) is new (\( \notin \text{dom}(t) \)). Hence, the tree \( t_u \) is output. The reader may notice that no move on \( F_1 \) is performed.
Chapter 4. Enabling XML Update Optimization ...

Example 4.3.6. Let us consider the update $u_6$ illustrated in Figure 4.16 specified by 
\[
\text{for } \$x \text{ in } /a \text{ return insert nodes ("uz"<e/>) as first into } \$x.
\]

- **projector** - The update $u_6$ involves an "insert" operation, hence the type projector $\pi$ derived from it contains the component $\pi_{olb}=\{a\}$.

- **update** - Figure 4.16-2 illustrates the changes made by the update. The node $F_u@1$ has a newly inserted child of type String and a new node $F_u@i$. It is worth noticing that $F_u$ contains as well nodes $F_u@1.1$ and $F_u@1.2$ which have been projected based on the assumption $f \notin \pi$ and $g \notin \pi$.

- **merge** - We have that the parent node $F_i@1$ of $F_i$ (see fig. 4.16-1) is labelled by $a \in \pi_{olb}$, thus the first level nodes of $F_i$ and $F_u$ must be synchronized following the procedure $OlbMerge$. First $OlbMerge$ processes nodes $F_i@1.1$ and $t_u$ of type String and since the condition $\sigma_i(\text{roots}(t_i))=\text{text}[st]$ is true $OlbMerge$ executes line c.2. Hence, $OlbMerge$ selects tree $t_u$ as the first tree of the forest $F_r$ and recalls $OlbMerge$ parsing only $F_u$. Next $OlbMerge$ examines nodes $F_i@1.1$ and $F_u@i$. Because the condition $\text{new}(\text{roots}(F_u@i))=\text{true}$ is satisfied, once again, the output is specified by line c.2: $OlbMerge$ outputs a tree with root $F_u@i$ and moves on $F_u$. Finally, $OlbMerge$ examines nodes $F_i@1.1$ and $F_u@1.1$, then $F_i@1.2$ and $F_u@1.2$ specified by line c.5, the behavior of which is explained in the paragraph devoted to that line. The rest of merging $F_i$ and $F_u$ is explained later, because it uses other cases. □

**Line c.3** Line c.3 is similar to line 3, although it should be paid attention to the sub-case where the root of $t_i$ is of type String: $t_i$ is then ignored because the corresponding String element in $F_u$ (updated or not by $u$) has, eventually, already been output by a previous application of line c.2.

Example 4.3.7. Let us consider the update $u_7$ specified by 
\[
\text{for } \$x \text{ in } /a \text{ return replace node } \$x/f \text{ with } <e/> \text{ illustrated in Figure 4.17.}
\]

- **projector** - Recall that the update operation "replace" is considered as "delete" followed by an "insert". Thus the projector contains the component $\pi_{olb}=\{a\}$. Note that, the update $u_7$ replaces only the nodes having the child labelled by $f$, thus the second component of the projector is $\pi_{no}=\{f\}$.

- **update** - Figure 4.17-2 illustrates the result of the evaluation of the update $u_7$: the tree rooted at $F_i@1.1$ of the forest $F_i$ has been replaced by a new tree with root $F_u@i$. 

---

Figure 4.16: Example for the procedure $OlbMerge$, line c.2
4.3. Merge for enabling XML Update Optimization...

merge - The parent node of $F_i$ is labelled by $a \in \pi_{\text{olb}}$ (see fig. 4.17-1), thus merging its children is specified by $\text{OlbMerge}$. First $\text{OlbMerge}$ parses the trees $t_i$ and $t_u$ of type $\text{String}$. The action is specified by line c.2, since $t_u$ is a type of $\text{String}$. Therefore, $\text{OlbMerge}$ outputs the tree $t_u$ as a first tree of the forest $F_r$ and moves on $F_u$ only. After that $\text{OlbMerge}$ is executed with input: $t_i$ of type $\text{String}$ and $F_u@i$. Because we have that $\text{new}(\text{roots}(F_u@i))=\text{true}$, $\text{OlbMerge}$, once again, executes line c.2 and outputs the tree rooted at $F_u@i$. After that, $\text{OlbMerge}$ examines $t_i$ this time with node $F_u@1.2$. Because $t_i$ is of a type $\text{String}$ the action is line c.3. Mainly, $\text{OlbMerge}$ skips $t_i$ and moves on $F_i$ examining this time nodes $F_i@1.1$ and $F_u@1.2$. The rank 2 of $F_i@1.2$ is strictly greater than the rank 1 of $F_i@1.1$ and $F_i@1.1$ is labelled by $f \in \pi_{\text{no}}$, thus the execution specified by line c.3. According to it, $\text{OlbMerge}$ skips the tree with the root $F_i@1.1$ (this tree has been replaced) and moves on $F_i$. Finally it synchronizes $F_i@1.2$ and $F_u@1.2$ (line c.4), which is dual to line 4 of the procedure $\text{NoMerge}$ and $\text{OlbMerge}$ outputs the tree with the root $F_u@1.2$ into $F_r$. □

Lines c.4, c.5 are the dual of lines 4,5 of the $\text{NoMerge}$ definition. The reader should pay attention to line c.5 where, although implicit, the equality $\text{roots}(t_i)=\text{roots}(t_u)$ holds (as opposed to the case line 5 of $\text{NoMerge}$): even if $a \notin \pi$, because of $(\dagger)$, the node identified by $\text{roots}(t_i)=\text{roots}(t_u)$ is in both forests $F_i$ and $F_u$.

Example 4.3.8. For example, let us consider the example illustrated in Figure 4.18. The update $u_8$ is specified by for $\$x$ in /a where $\$x/d$ return insert node $<e/>$ as first into $\$x$.

projector - The update $u_8$ involves an "insert" operation thus the type projector $\pi_8$ derived from it contains the components $\pi_{\text{olb}}=\{a\}$ and $\pi_{\text{no}}=\{d\}$.

update - Because the parent node of the forest $F_i$ has child labelled by $d$, a new node $F_u@i$ labelled by $e$ is inserted as its first child, after the update query evaluation.

merge - The parent node of $F_i$ is labelled by $a \in \pi_{\text{olb}}$ (see fig. 4.18-1) thus merging top level nodes is specified by $\text{OlbMerge}$. While processing nodes $F_i@1.1$ and $F_u@i$ $\text{Merge}$ executes line c.2. Next, The nodes $F_i@1.1$ and $F_i@1.1$ are processed according to line c.4, since the ranks are equal and $F_i@1.1$ is labelled by $d \in \pi_{\text{no}}$, while nodes $F_i@1.2$ and $F_u@1.2$ are merged according to line c.5, because $F_i@1.2$...
The following paragraph explains the behavior of \textit{OlbMerge} for the case where \( \text{lab}(\text{roots}(t_i)) \in \pi_{eb} \):

\textbf{Example 4.3.9.} Let us consider the update \( u_7 \) specified by \texttt{for $x$ in /a return replace node $x/b$ with $x/c$} illustrated in Figure 4.19.

\textbf{projector} - The update involves the "replace" operation and it replace a sub-tree having root node labelled by \( b \) with another sub-tree having root labelled by \( d \). Recall that replace has the same behaviour as "delete" followed by "insert" hence the first component of the projector \( \pi_{olb} = \{a\} \). The second one: \( \pi_{eb} = \{c\} \) is necessary to extract the sub-tree whose root is labelled by \( c \).

\textbf{update} - Figure 4.19-2 illustrates the partially updated forest \( F_u \), where node \( F_u@i \) is the result of the deletion \( F_i@1.2 \) and the insertion of a new tree being the copy of the tree rooted at \( F_i@1.1 \). Note that, the identifier of \( F_u@i \) is not the same as \( F_i@1.1 \).

\textbf{merge} - Because the parent node of the \( F_i \) is labelled by \( a \in \pi_{olb} \), its children are processed by \textit{OlbMerge}. While parsing them \textit{OlbMerge} proceeds in the following way. First it examines nodes \( F_i@1.1 \) and \( F_u@1.1 \). Because the ranks are equal and \( F_i@1.1 \) is labelled by \( c \in \pi_{eb} \) (see fig. 4.19-1,2) the action is specified by line c.4. Thus \textit{OlbMerge} selects \( F_u@1.1 \) as the first tree of \( F_u \) and calls \textit{TreeMerge} to determine which procedure to apply on its sub-forests. Because we have that \( F_i@1.1 \) is labelled by \( c \in \pi_{eb} \) all its descendants are output into into \( F_r \) (see fig. 4.19-3) and \textit{OlbMerge} moves both on \( F_i \) and \( F_u \). This time nodes \( F_i@1.2 \) and \( F_u@i \) are processed according to line c.2: it outputs the tree rooted at \( t_u@i \) and moves on \( F_u \). Finally, according to line c.1', \textit{OlbMerge} skips \( F_i \).

The next example illustrated in Figure 4.20 explains the behavior of \textit{OlbMerge} for an update mixing all previous cases.

\textbf{Example 4.3.10.} Let us consider the update \( u_8 \) specified by

\texttt{for $x$ in /a return}

\{
4.3. Merge for enabling XML Update Optimization...

Figure 4.19: Example for the case \( \text{lab}(\text{roots}(t_i)) \in \pi_{eb} \)

- insert nodes \( \text{"uz"<e/>} \) as first into \$x\$,
- insert node \<$p/>\) as last into \$x\$,
- rename node \$x/d\) with "h",
- replace node \$x/f\) with \<$e/>\),
- replace node \$x/b\) with \$x/c\),

As it has been explained in Chapter 2, the update primitives are held in the pending update list and are applied in restricted order. For our example, first rename, next insert as first, then insert as last operations and finally, replace operations are applied.

**projector** - The projector \( \pi_9 \) derived from this update contains three components \( \pi_{olb} = \{a, f, b\} \), \( \pi_{no} = \{d\} \) and \( \pi_{eb} = \{c\} \).

**update** - Figure 4.20-2 illustrates the result of the evaluation of the update \( u_8\). The first tree of the forest \( F_u \) is a tree of type String, followed by newly inserted node \( F_u[@1] \). Next, the label \( d \) of the node \( F_u[@1.1] \) is renamed to \( h \). The node \( F_u[@1.2] \) labelled \( f \) is replaced by a new node \( F_u[@11] \) labelled by \( e \). The node \( F_u[@1.3] \) labelled \( b \) is replaced by the subtree having root node \( F_u[@1.5] \). Note that, the identifiers are not the same. The nodes \( F_u[@1.4] \) and \( F_u[@1.5] \) are not changed. Finally a new node labelled by \( p \) is inserted as the last child of the parent node \( F_u[@1] \).

**merge** - The parent node of the forest \( F_i \) is labelled by \( a \in \pi_{olb} \), thus the action is specified by \( OlbMerge \). First, \( OlbMerge \) processes nodes \( F_i[@1.1] \) and the first tree \( t_u \) of type String of \( F_u \) as specified by line c.2, since the condition \( \sigma_{t_u}(\text{roots}(t_u)) = \text{text}[st] \) is true. \( OlbMerge \) outputs the tree \( t_u \) as the first tree of the forest \( F_i \) (see fig. 4.20-3) and moves on \( F_u \) only: processing the nodes \( t_u[@1.1] \) and \( t_u[@1] \). Once again, the action is specified by line c.2, since this time we have that the condition \( \text{new}(\text{roots}(t_u)) = \text{true} \) is satisfied. Therefore, \( OlbMerge \) selects the tree with the root \( F_u[@1] \) and continues with parsing \( F_u \). This time, \( OlbMerge \) examines nodes \( F_i[@1.1] \) and \( F_u[@1.1] \) and because their ranks are equal and \( F_i[@1.1] \) is labelled by \( d \in \pi_{no} \) the action is specified by line c.4. Thus, it outputs \( F_u[@1.1] \) and calls \( TreeMerge \) to determine which procedure among \( NoMerge \) and \( OlbMerge \) has to be applied on the sub-forest. Because \( F_i[@1.1] \) is labelled by \( d \in \pi_{no} \), \( NoMerge \) must be applied. \( OlbMerge \) moves on both \( F_i \) and \( F_u \). This time, nodes \( F_i[@1.2] \) and \( F_u[@11] \) are processed as specified by line c.2 (tree rooted at \( F_i[@1.2] \) has been replaced by \( F_u[@11] \)). \( OlbMerge \) selects tree rooted at \( F_u[@11] \) and
moves on $F_u$, parsing nodes $F_i \oplus 1.2$ and $F_u \oplus 1.2$. Once again, the action is that of line c.2, hence $OlbMerge$ outputs the tree rooted at $F_u \oplus 1.2$ and moves on $F_u$. Now the nodes $F_i \oplus 1.2$ and $F_u \oplus 1.4$ are processed specified by line c.3, since the rank 4 of $F_u \oplus 1.4$ is strictly greater then the rank 2 of $F_i \oplus 1.2$ labelled by $f \in \pi_{no}$. According to line c.3, $OlbMerge$ skips the tree rooted at $F_i \oplus 1.2$ and processes nodes $F_i \oplus 1.3$ and $F_u \oplus 1.4$ as specified by line c.3. After skipping the tree rooted at $F_i \oplus 1.3$, $OlbMerge$ is applied on nodes $F_i \oplus 1.4$ and $F_u \oplus 1.4$, this time processing them according to line c.4. It executes $TreeMerge$ to determine procedure used to build the sub-forest of the tree rooted at $F_u \oplus 1.4$. None of the trees rooted at $F_i \oplus 1.4$ and $F_u \oplus 1.4$ has children, hence $OlbMerge$ examines nodes $F_i \oplus 1.5$ and $F_u \oplus 1.5$. They have equal ranks, but because node $F_i \oplus 1.5$ is labelled by $c \in \pi_{eb}$ $Merge$ outputs the tree rooted at $F_u \oplus 1.5$ into $F_r$, and moves both on $F_i$ and $F_u$. Finally, $OlbMerge$ outputs node $F_u \oplus 1.5$, according to line c.1.

**Theorem 4.3.11.** Let $u$ be an update over $D$ and $\pi$ be the inferred type projector for $u$. Then for each $p$-tree $t \in D$, we have: $\text{Merge}(t | u(\pi(t))) \sim u(t)$.

Above, value equivalence $\sim$ captures the idea that the two processes return the same document up to node identifiers.

### 4.4 Implementation and Experiments

This Section is not complete for the moment.

#### 4.4.1 Implementation issues

In order to validate the effectiveness of our method, we have implemented both projection and merge algorithms in Java. The only technical gap between the formal method and its implementation concerns node identifiers or positions. Although made explicit in the formal scenario, the implementation does not materialize positions in the input document $t$: it is not necessary. Positions are generated on the fly while parsing $t$, during projection and during $Merge$. Indeed, for each node, the implementation generates its rank among its siblings: full node position is not necessary. In $\pi(t)$, this rank is stored by means of a special new attribute for node only/one level below nodes and by means of another new attribute for $\forall$ below node.
The potential overhead due to these special attributes is mitigated by the size reduction ensured by projection. The use of two distinct attributes is required for technical reason related to insertion and replace updates and also to the way source elements are copied during their execution.

The algorithm Merge is implemented by means of two threads, parsing resp. $t$ and $\pi(t)$. These threads are defined in terms of classes obtained by extending existing SAX parser classes \[6\]. While processing the XML document the SAXParser calls methods in the DefaultHandler subclass instance corresponding to what the parser finds in the XML file. To react to these method calls we override the corresponding methods in the DefaultHandler subclass. The two threads interact with each other according to the Producer-Consumer pattern.

According to this pattern the Producer generates a piece of data, puts it into the buffer and starts again. At the same time the Consumer thread is consuming the data, removing it from the buffer one piece at a time. The issue here is to make sure that the producer will not try to add data into the buffer if it’s full and that the consumer will not try to remove data from an empty buffer. We use this pattern to send the horizontal position (the rank values) of each node being parsed to the next thread, besides this we pass as well the current Merging mode (i.e., NoMerge, OlbMerge, etc.).

The solution for Producer used in our implementation is the following: Producer waits if the buffer is full, while Consumer removes an item from the buffer and notifies Producer who starts to fill the buffer again.

**Class Diagram** Figure 4.21 illustrates the Class Diagram which "encapsulates" our implementation. As the reader can observe, we have the following classes:

**OriginalDocHadler** - This class is a subclass of the DefaultHandler class and overrides certain inherited methods, like `startDocument()`, `startElement()`, `characters()` and etc. This class parses the Original XML document, which has not been updated. OriginalDocHadler uses the methods of SmartQueue class.

**UpdatedDocHadler** - This class, similarly to the previous one, extends the DefaultHandler class, but parses the Updated XML Document.

**Producer** - This class extends the Thread class and overrides the `run()` method.

In this method we create an instance of the OriginalDocHadler class and call its `parse()` method to parse the Original XML document.

**Consumer** - This class extends the Thread class and overrides the `run()` method.

In this method we create an instance of the UpdatedDocSaxHadler class and call its `parse()` method to parse the Updated XML document.
SmartQueue - this class implements the Producer-Consumer pattern. To support this pattern, this class has three methods: put(Object) and take() and notifyToOtherThread().
SmartQueue, deals with the output result of Merge. For example, the `appendElement()` method appends Open and Close tags, together with attributes and string values of a parsed nodes. `delString()` deletes the content of `lastElementChild` variable. `outputBuffer()` writes the content of the `StringBuffer` to the resulting XML document. It is worth noticing, that the variable `mergeMode` preserves the name of the `Merge` procedure according to which the current two nodes must be examined. For example, `NoMerge` or `OlbMerge`. We have the third `mergeMode` case `EvbMerge` which is explained in the next paragraph.

**Implementation issue for "everything below" component**  As it has been explained in Chapter 4 during projection phase we assign unique identifiers to each node, which are used while Merging two documents. Therefore, in the implementation of the projection, when we parse the XML document, to each node we assign a `label` attribute (e.g. we have `<name label="2">`, for the node having identifier 1.2.1.2) to store the horizontal position of the identifier. During the `Merge` process we use this attribute to compare the child ranks. The same is true for the case where the projector contains "everything below" component.

Let us recall the example 4.2.6 from Section 4.2. Figure 4.22-4 illustrates the projected tree \( t_1 \) with `label` attributes, which preserve the horizontal positions. Figure 4.22-5 illustrates the partial result \( t' \), where the "in place of" inserted element labelled by \( d \) has new 1 identifier. The issue here is that this node contains, as well, the `label` attribute, which has been assigned during projection phase. According to the formalization given in Section 4.3, when the parent node of a forest \( F_i \) is labeled by \( a_{\in \pi_{olb}} \), `Merge` executes the `OlbMerge` procedure on the first level children of this forest. Recall that the comparisons of the identifiers is essential during the `Merge` process. Therefore, in the implementation, for each node in \( t' \) the `label` values are retrieved and compared with the calculated on fly horizontal positions of nodes from \( t \). An issue arises while processing the children \( t@1.2 \) and \( t'@i2 \), because \( t'@i2 \) contains the attribute `label`=1, which belongs to the "in place of" inserted node \( t@2.1 \). As a consequence, while comparing the `labels` 2 and 1 we fall into a case out of the formalazation.

An other, special case, not illustrated here, arises when the `label` value of a node from \( t' \) is greater than the one of \( t \). For example, if the "in place of" inserted node \( t'@i \) has `label`=3, and we compare the nodes \( t@1.2 \) and \( t'@i \). In this case, according to `OlbMerge`, because the label value of \( t'@i \) is greater than the one of \( t@1.2 \), we should skip the tree rooted at \( t@1.2 \), although this is incorrect in this case.

To deal with this problem we proposed the following solution illustrated in Figure 4.23. As the reader can observe, in Figure 4.23-4, during projection we do not assign the `label` attribute to the node labelled by \( d_{\in \pi_{eb}} \), instead we assign the `evb` attribute. For our example we have that the nodes \( t@2.1 \) is assigned to a `evb`=1. It is worth
(1) The update $u_6$

$$
\pi_{na} = \{\text{doc}\}
\pi_{ab} = \{a\}
\pi_{eb} = \{d\}$$

(2) The three level type projector $\pi_6$

(3) The XML document $t$

(4) The projection $t_1$ of $t$ wrt $\pi_6$

(5) The partial result $t'$

Figure 4.22: The projector component $\pi_{eb}$ and label attribute

(1) The update $u_6$

$$
\pi_{na} = \{\text{doc}, b\}
\pi_{ab} = \{a\}
\pi_{eb} = \{d\}$$

(2) The three level type projector $\pi_6$

(3) The XML document $t$

(4) The projection $t_1$ of $t$ wrt $\pi_6$

(5) The partial result $t'$

Figure 4.23: The projector component $\pi_{eb}$ and evb attribute
noticing that we do not assign evb attributes to descendants. This time, during "Merge" process, we do the following steps. First we process the nodes t@1.1 and t'@i and because its label value is null we output the tree rooted at t@i. Next for the same reason we output the tree rooted at t'@2. After that, we skip the trees rooted at t@1.1 and t@1.2, since they have been replaced. Note, that for the nodes t@2.1 and t'@2.1 we compare the values of evb attribute, to skip the deleted trees if it is the case.

**Sequence Diagram** Figure 4.37 illustrates the sequence diagram of the Merge algorithm implementation.

First, the Producer thread starts parsing the Original document. Producer calls the startElement() method (see 1) of OriginalDocHandler. Note that, in this diagram we do not consider startDocument() and endDocument() methods. Here we have two possible scenarios: 1.1 and 1.2. If the type of the node does not belong to one of the projector components, then the appendElement() (see 1.1) method of SmartQueue is executed to append to StringBuffer the Open tag of the parsed element, with its attributes (e.g., <item id="item0">).

Otherwise, the horizontal position of the parsed node and the current merging mode curTau are stored in vector_Producer and the put(vector_Producer) method (see 1.2) of SmartQueue is executed. This method calls notify() to start the Consumer thread, while Producer is set to waiting state. It is worth noticing, that the curTau variable can be equal to one of the followings: "NoMerge", "OlbMerge", "EvbMerge". This value depends on the type of the parsed node. For instance, if it belongs to the π_no component, then it is set to "NoMerge".

First, we follow the scenario of 1.1, which corresponds to the lines 2 and 5 of the NoMerge formalization given in Section 4.3. In this case, in the next step, Producer calls the characters() method (see 1.2.5.1.2) and outputs the string value of the parsed node. After that, the endElement() method (see 1.2.5.1.4) is executed and the Close tag is output (e.g., </item>). It is worth noticing that, if a node has descendants, startElement() is called instead of characters().

According to the scenario 1.2, Consumer starts parsing the Updated document and calls the startElement() method (see 1.2.2). In this method we first, retrieve the curTau and horizontal position values from the buffer. To achieve this we call the take() method (see 1.2.3) of SmartQueue. After consuming these values, first we output the Open tag (see 1.2.4), then we set the mergeMode, which corresponds to the procedure of Merge, according to which the processing continues. If curTau="NoMerge", then it means that the parsed node in Producer thread was labelled by type that belongs to the π_no component. Thus, we set the mergeMode="NoMerge" and the further processing is specified by NoMerge procedure. Here we have the following two possible cases, either the retrieved horizontal
position of the identifiers is equal to the one of the parsed node, or the current horizontal position is greater than the one retrieved from the buffer.

The first case corresponds to line 4 of the NoMerge formalization given in Section 4.3. In this case, we simply set the value of `what_to_Do_Str_Orig="write"` and call the `notifyToOtherThread()` (see 1.2.5.0) method, without putting any data to the buffer. After that, Producer calls the `characters()` method (see 1.2.5.1.2) to output the string value of the parsed node. If `what_to_Do_Str_Orig="write"` the `appendElement()` method (see 1.2.5.1.3) is executed. Finally, Producer calls the `endElement()` method (see 1.2.5.1.3) and because the `mergeMode="NoMerge"` the `notifyToOtherThread()` (see 1.2.5.1.4.2) method is called to output the Close tag from Consumer.

For the second case (which corresponds to line 3 of the NoMerge formalization) we add to the buffer the horizontal position of the parsed node together with the local stack size (stack.size), which we need to re-calculate the vertical position of a node. After that, we call the `put(vector_Producer)` method (see 1.2.5.1) of SmartQueue, which in its turn notifies the Producer thread (see 1.2.5.1.1). Note, that before notifying the other thread we set the `mustDelete` variable of SmartQueue to true. This aims at skipping outputting the descendants of a deleted node. Once the parsed node in Producer thread has the horizontal and vertical positions equal to the ones of the node kept in the buffer, the `mustDelete` is set to false.

If `curTau="OlhMerge"`, similarly to the above case, we have two possible cases: either the horizontal positions are equal, or the position of the node parsed by the Consumer thread is greater. First case corresponds to line c.4 of the OlhMerge formalization, while the second one maps line c.3. It is worth noticing that in this case we set the `what_to_Do_Str_Orig="skip"` and `what_to_Do_Str_Up="write"` to output the string value of a node, which corresponds to line c.2 of OlhMerge formalization.

If `curTau="EvbMerge"`, then the `mergeMode` variable is set to "EvbMerge" and instead of calling the `put()` or `notifyToOtherThread()` methods, Consumer calls first, the `characters()` (see 1.2.5.2) then `endElement()` methods to output the string value and the close tag of the parsed node. At the end of the `endElement()` method the synchronization is passed back to Producer (see 1.2.5.2.4). If `mergeMode="OlhMerge"` and the parsed node in the Consumer thread contain neither label nor evb attributes the merge mode is set to `mergeMode="New"`.

This case corresponds to line c.2 of OlhMerge procedure formalization. In this case, the Producer thread stays in waiting state, while Consumer continues parsing descendants until it outputs the close tag of the parent node.

The last value of `mergeMode` is "OlhChild" and is set while parsing the first level children of nodes typed by π_olh, if the types of these nodes are not in any projector component. The behavior in this case is the same as "NoMerge". Note, that this case corresponds to line c.5 of the OlhMerge formalization.

It is worth noticing that, when any node has been deleted during the execution of an update query, it can happen that the `startElement()` method of Producer calls
4.4. Implementation and Experiments

Figure 4.24: Structure of XMark documents

the `endElement()` method of `Consumer` (line c.1’ of the `OlbMerge` formalization). This can happen when the child node has been deleted and `Consumer` must close the tag of the parent node. In this case, our solution is to set the `mustDelete` to true, add the horizontal position value -1 to the buffer and send it to `Producer`. When `Producer` reads the Close tag of the parent the value of `mustDelete` is set back to false.

4.4.2 Experiments

Several tests have been performed using our Java implementation and 20 updates on XMark documents [35] of growing size. Figure 4.24 illustrates a part of the structure of all XMark documents. As reader can observe all child nodes of the site element contain date elements on their descendant axis, except of the category, people and catgraph elements.

These updates, together with their associated projectors, are reported in the The updates and the corresponding projectors paragraph, and cover the main update operations made available by XQuery Update Facility (insert, rename, replace and delete). All experiments were performed on a 2.53 Ghz Intel Core 2 Duo machine (2 GB main memory) running Mac OSX 10.6.4.

The updates used are classified into five categories: insert (U1, U6, U7, U11, U12, U13, U17), delete (U4, U8, U10, U14, U16), replace (U2, U9, U15, U18, U19, U20), replace value of (U3) and rename (U5).

Three-level type projectors extracted from these updates are illustrated in Table 4.4. The first and the third categories aim at testing the `OlbMerge` procedure, while the second and the fourth ones aim at testing the `NoMerge` procedure. The updates U15, U19 and U20 aim at testing the correctness of our technique using
the documents that contain recursive nodes. The updates U5, U9, U15, U19 and U20 test the correctness of our technique when the projector contains "everything below" component. We use the updates U8 and U16 to show the effectiveness of our method while applying a very selective projector.

To perform our tests, documents having sizes 128MB, 1GB, 1-5GB and 2GB have been generated using XMark generator.

The first kind of tests aims at detecting memory limitations of four popular query processors implemented in Java: Saxon EE 9.2.0.2 [7], Qizx Free-Engine-3.2.0 [4] and eXist 1.2.5 [2]. We set to 512 MB the Java virtual machine memory, while the size of XMark documents considered goes from 50 MB to 2 GB. The sizes of largest documents these processors could update without projection are reported in fig. 4.25. For this test, we used the less memory consuming update U4. Three out of four systems cannot deal with documents whose size is greater than 150 MB, while Qizx is able to process documents whose size is slightly higher than the Java virtual memory size (this is due to some efficient techniques adopted by Qizx for compacting internal document representation).

The second kind of tests evaluates our projection based technique. We focused on two systems Saxon, Qizx and BaseX, and used the whole set of 20 updates. In both cases, tests show that our technique can ensure great improvements.

As it has been explained at the beginning of in this Chapter, the projection technique is divided into three steps: Projection, Update and Merge. Each of this steps is divided into intermediate phases, and it is important to consider the execution time of each phase. Figures 4.26, 4.28 and 4.30 illustrate the phases which are processed while updating the projected document using Qizx, Saxon and BaseX, respectively.

First, we execute the projection. The total execution time of the projection is divided into three phases: \( t_{\text{proj1}} \), reading the input document, \( t_{\text{proj2}} \), projecting and storing the projected data in a buffer and \( t_{\text{proj3}} \) (writing the stored data to a file once the buffer is full).

The intermediate phases of updating the projected documents and storing the update result are different for Saxon compared to Qizx and BaseX. The update execution in Qizx (or BaseX) process as follows. First, a document
Figure 4.26: Execution with Projection using Qizx

Figure 4.27: Execution with Projection using Qizx Optimization
Figure 4.28: Execution with Projection using Saxon

Figure 4.29: Execution using Projection using Saxon, Optimization
is imported: \( t_{\text{qizz1}} \). While importing the document Qizx creates indexes and stores nodes, attributes and text values on the disk. After that, the update query is executed on the stored document, where \( t_{\text{qizz2}} \) is the execution time of the update. Note that, Qizx applies the resulting changes directly on the stored data, thus in order to get the updated document we need to export it (to serialize): \( t_{\text{qizz3}} \).

It is worth noticing, that for Qizx the exporting time is very small for the documents with small sizes and we do not consider this time during our test. Therefore as it is reported in Table 4.6 the total update execution time for the projected document using Qizx is the following: importing + query execution \((t_{\text{qizz1}} + t_{\text{qizz2}})\).

On the contrary, for BaseX the exporting time is considerable, therefore as it is reported in Table 4.10 the total update execution time for the projected document using BaseX is the following: importing + query execution + exporting \((t_{\text{qizz1}} + t_{\text{qizz2}} + t_{\text{qizz3}})\).

The update execution using Saxon process as follows. First, the update query is analyzed: \( t_{\text{saxon1}} \). Next, the tree mapping XML document is built: \( t_{\text{saxon2}} \). Finally, the update is executed on that tree: \( t_{\text{saxon3}} \). Note that, the time spent on writing (serializing) the updated document is included in the query execution time. Therefore, as it is illustrated in Table 4.7 the total execution time on the projected document is equal to: analysis + tree built + execution \((t_{\text{saxon1}} + t_{\text{saxon2}} + t_{\text{saxon3}})\).

The Merge step is the same for the both systems. First, we read the original
and the updated documents: \texttt{t\_merge1}. Next, we process Merge: \texttt{t\_merge2}, storing the intermediate results in a buffer and finally writing the result of Merge to a file: \texttt{t\_merge3}.

It is worth noticing that the intermediate steps like writing the pruned document then importing it to execute an update, or writing the updated document then reading it to execute Merge, can be optimized, as it is illustrated in Figures 4.27 and 4.29. To achieve this, during the projection, we need to store the projected nodes directly on the disk for Qizx (without writing to a file) and to create a node Object for each projected node for Saxon. This is considered as one of the future optimizations of our technique.

Figure 4.31: Documents size reduction after pruning
4.4. Implementation and Experiments

4.4.2.1 Projection

The document size reduction of projected documents are reported in Figure 4.31 and Table 4.1. The execution time of projection is illustrated in Figure 4.32.

As the reader can observe, the best results, 4KB for 2GB, are reported for the updates U8 and U16. The worst result is reported for the update U9, 535.6 MB for 2GB, due to the low selectivity of the "one level" component (941 850 nodes) and the considerable number of descendants (3 200 564 nodes) of the "everything"
The similar results are reported for the execution time of projection time. The best results are reported for the updates U8 (42.792 sec.) and U16 (44.121 sec. for 2GB), while the worst one is for the update U9 (115.86 sec. for 2GB). The execution time is long for this update, because there is more I/O calls between a buffer and a result file.

### 4.4.2.2 Merge

The execution time of the Merge process is reported in Figure 4.33. As the reader can observe, once again, the best results are reported for the updates U8 and U16 (122.516 and 129.893 sec. for 2GB respectively). On the contrary to the projection, the worst result is reported for the update U5 (359.423 sec. for 1-5GB). While for the update U9 the execution time is 189.66 sec. for 1-5GB. This kind of result is due to the number of thread notifications while the Merge process.

### 4.4.2.3 Update

Figures 4.34, 4.35 and 4.36 illustrate results of the tests performed on Saxon, Qizx and BaseX, respectively. In all figures, missing value for time means memory failure. Tables 4.6 and 4.7 reports both the Qizx and Saxon update execution times for our 20 updates without projection and update execution times on the projection. It is worth noticing that for the case without projection we are illustrating the execution time only for 128MB, since Saxon is not able to execute the updates on bigger documents due to memory limitations. Table 4.9 reports the total update execution times on the projection using Saxon and Qizx respectively.

<table>
<thead>
<tr>
<th>Or. Size MB</th>
<th>U1</th>
<th>U2</th>
<th>U3</th>
<th>U4</th>
<th>U5</th>
<th>U6</th>
<th>U7</th>
<th>U8</th>
<th>U9</th>
<th>U10</th>
</tr>
</thead>
<tbody>
<tr>
<td>128MB</td>
<td>2.1</td>
<td>5</td>
<td>1.2</td>
<td>1.5</td>
<td>7.5</td>
<td>3.9</td>
<td>4.6</td>
<td>4.8</td>
<td>0.00391</td>
<td>43.5</td>
</tr>
<tr>
<td>1GB</td>
<td>19.1</td>
<td>46.6</td>
<td>11.1</td>
<td>14</td>
<td>69.6</td>
<td>36.6</td>
<td>43.1</td>
<td>0.00391</td>
<td>311</td>
<td>5.2</td>
</tr>
<tr>
<td>1.5GB</td>
<td>25.8</td>
<td>63</td>
<td>15</td>
<td>18.9</td>
<td>93.9</td>
<td>49.4</td>
<td>58.2</td>
<td>0.00391</td>
<td>418.2</td>
<td>7.1</td>
</tr>
<tr>
<td>2GB</td>
<td>33</td>
<td>90.5</td>
<td>19.2</td>
<td>24.2</td>
<td>120.2</td>
<td>64.2</td>
<td>74.4</td>
<td>0.00391</td>
<td>535.6</td>
<td>9.1</td>
</tr>
<tr>
<td>nodes for 128MB</td>
<td>59 031</td>
<td>170 952</td>
<td>37 634</td>
<td>75 787</td>
<td>132 585</td>
<td>159 783</td>
<td>206 794</td>
<td>15 606</td>
<td></td>
<td></td>
</tr>
<tr>
<td>πolb, descendants</td>
<td>59 031</td>
<td>99 535</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>77 300</td>
<td>55 318</td>
<td>0</td>
<td>59 031</td>
<td>0</td>
</tr>
<tr>
<td>πeb, descendants</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>49 358</td>
<td>0</td>
<td>200 134</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>nodes for 2GB</td>
<td>1 210 952</td>
<td>2 726 240</td>
<td>600 308</td>
<td>887 180</td>
<td>4 169 353</td>
<td>2 112 414</td>
<td>2 459 102</td>
<td>3</td>
<td>4 742 717</td>
<td></td>
</tr>
<tr>
<td>πolb, descendants</td>
<td>941 850</td>
<td>1 585 084</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1 231 928</td>
<td>880 458</td>
<td>0</td>
<td>941 850</td>
<td>0</td>
</tr>
<tr>
<td>πeb, descendants</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>789 334</td>
<td>0</td>
<td>3 200 564</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Size reduction by projection
Concerning Saxon, tests results are synthesized in Figures 4.34.1 and 4.34.2, reporting, respectively, total execution time by not using and by using projection. They clearly show that our technique succeeds in its primarily purpose: making possible to update very large documents with in-memory systems, in the presence of memory limitations. Note that, the total time in the case of projected documents (see fig. 4.34.2) includes time for i) projecting the input, ii) storing the projection, iii) updating the projection and storing it, and iv) performing the final merge.

As it is reported in Figure 4.34.1 our technique succeeds, as well, to optimize the update execution time for several updates. Mainly, for the 128 MB document, we have the following reductions of execution times, expressed in percentages: U3 (5.20%),
(1) Updating without projection  
(2) Updating on the projection

![Bar chart showing results of tests performed on Saxon](image)

Figure 4.34: Results of the tests performed on Saxon
U4(7.40%), U6(2.90%), U8(38.50%), U10(32.90%), U14(7.20%), U15(31.40%), U16(38.40%), U17(30.70%), U18(26.10%), U19(31.10%) and U20(13.40%). This is because the time spent for projection, merging and reading/writing documents is recovered by a faster update process thanks to a significantly smaller size of the projected document (fig. 4.31). Nevertheless, for the following updates execution without projection is lower: U1(-28.50%), U2(-2.80%), U5(-70.80%), U7(-34.40%), U9(-54.60%), U11(-27.60%), U12(-58.20%) and U13(-10.40%).

As the reader can observe the updates U5, U12 and U9 report more than 50% of penalization while executing updates using projection. There are several reasons why projection is more time consuming for these updates.

The first reason is that the *Merge* processing is very expensive for these updates, which is due to the number of nodes being synchronized (many thread notifications are called during the processing). For instance, as it is reported in Table 4.1 for the update U9 the number of the projected nodes for the document having size 128MB is the greatest: 296 794. Then in the second place we have the update U5: 261 233. Finally the update U12: 257 939.

Here it is worth noticing that, even the number of nodes being projected for U9 is greater than the number of nodes for U5, the execution time for *Merge* is less expensive for U9. The reason is that for U9 we have 200 134 nodes which have been projected because of being the decedents of the *annotation*- node ($\pi_{eb} = \{\text{annotation}\}$), thus are not synchronized.

Also observe that in Figure 4.34-(2) for U5, U9, U11, U12, U14 and U20 Saxon was not able to update documents having size greater than 1 GB (due to memory failure). The projector of this update reveals that this is due to its low selectivity. It is worth noticing that for the update U2 the memory failure reports the memory limitations related to the attributes storing, which proves the effectiveness of the projection technique, since we do not project the attributes which are not used by an update.

<table>
<thead>
<tr>
<th>Or. Size MB</th>
<th>U1</th>
<th>U2</th>
<th>U3</th>
<th>U4</th>
<th>U5</th>
<th>U6</th>
<th>U7</th>
<th>U8</th>
<th>U9</th>
</tr>
</thead>
<tbody>
<tr>
<td>128MB</td>
<td>28.65</td>
<td>31.56</td>
<td>36.49</td>
<td>27.74</td>
<td>-49.25</td>
<td>83.87</td>
<td>-64.74</td>
<td>55.25</td>
<td>17.48</td>
</tr>
<tr>
<td>1GB</td>
<td>46.14</td>
<td>-</td>
<td>48.48</td>
<td>-</td>
<td>-17.83</td>
<td>-</td>
<td>33.85</td>
<td>65.16</td>
<td>1.93</td>
</tr>
<tr>
<td>1.5GB</td>
<td>47.11</td>
<td>-</td>
<td>50.14</td>
<td>-</td>
<td>-11.78</td>
<td>-</td>
<td>29.06</td>
<td>63.73</td>
<td>0.31</td>
</tr>
<tr>
<td>2GB</td>
<td>49.53</td>
<td>-</td>
<td>50.14</td>
<td>-</td>
<td>-10.73</td>
<td>-</td>
<td>34.97</td>
<td>64.34</td>
<td>4.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Or. Size MB</th>
<th>U10</th>
<th>U11</th>
<th>U12</th>
<th>U13</th>
<th>U14</th>
<th>U16</th>
<th>U17</th>
<th>U18</th>
<th>U20</th>
</tr>
</thead>
<tbody>
<tr>
<td>128MB</td>
<td>80.85</td>
<td>21.12</td>
<td>-10.17</td>
<td>23.22</td>
<td>82.66</td>
<td>55.84</td>
<td>52.23</td>
<td>49.05</td>
<td>30.02</td>
</tr>
<tr>
<td>1GB</td>
<td>92.5</td>
<td>-</td>
<td>2.19</td>
<td>30.46</td>
<td>-</td>
<td>59.1</td>
<td>58.28</td>
<td>57.85</td>
<td>-</td>
</tr>
<tr>
<td>1.5GB</td>
<td>-</td>
<td>-</td>
<td>9.82</td>
<td>40.26</td>
<td>-</td>
<td>61.17</td>
<td>60.95</td>
<td>59.7</td>
<td>-</td>
</tr>
<tr>
<td>2GB</td>
<td>-</td>
<td>-</td>
<td>15</td>
<td>35.51</td>
<td>-</td>
<td>62.58</td>
<td>63.4</td>
<td>78.37</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.2: Gain in terms of the execution time with projection using BaseX
Chapter 4. Enabling XML Update Optimization

Figure 4.35: Results of the tests performed on Qizx

(1) Updating without projection  (2) Updating on the projection

- U20: 70% 90.527
- U19: 83.30% 89.101
- U18: 82.70% 88.812
- U17: 83.30% 87.967
- U16: 84.80% 82.714
- U15: 84.10% 89.223
- U14: 53.40% 87.52
- U13: 73.40% 89.295
- U12: 56.40% 88.624
- U11: 51.70% 91.426
- U10: 80.40% 85.65
- U9: 41.70% 88.912
- U8: 85.30% 85.831
- U7: 98.80% 44.769
- U6: 53.10% 89.818
- U5: 46.80% 89.552
- U4: 75.70% 82.11
- U3: 71.30% 88.952
- U2: 68.10% 89.485
- U1: 54.10% 44.015
4.4. Implementation and Experiments

Figure 4.36: Results of the tests performed on BaseX
Qizx  Qizx shows less severe memory limitations. Total execution times are reported in fig. 4.35-1 and 4.35-2. We still have great improvements in terms of memory: with projection, we can update up to 2GB for all updates except U9, while without projection the limit is 520 MB. However, for Qizx, projection also ensures sensible total execution time reduction. This is in part due to the fact that Qizx needs a significant time to build auxiliary indexes at loading time. This improvement in terms of execution time also testifies the effectiveness of our design choices at the projector, and Merge function level. For the 128MB document, we have the following reductions of execution times, expressed in percentages: U1(54.10%), U2(68.10%), U3(73.30%), U4(75.70%), U5(46.80%), U6(53.10%), U7(39.80%), U8(85.30%), U9(41.70%), U10(80.40%), U11(51.70%), U12(56.40%), U13(72.40%), U14(53.40%), U15(84.10%), U16(84.80%), U17(83.30%), U18(82.70%), U19(83.30%) and U20(70%).

BaseX  Update execution times for updates performed on BaseX, without and with using the projection, are reported in Table 4.8. Total execution times, including the projection, update evaluation and Merge process are reported in Table 4.10. In the tables the empty columns indicate that the execution time was more than 25 minutes. (Note that we did not report any execution time for the updates U15 and U19, because a "duplicate attributes" error was raised during the updates evaluation.) As the reader can observe BaseX shows the best results for the update execution without projection from the point of view of memory usage, nevertheless the update execution time is very long. For instance, the execution time of the update U10 performed on the document having size of 1-5GB is 1.9 hour. On the contrary, using projection optimizes the execution time for the same update. As it is reported in Table 4.2 for the update U10 we have 80.85% and 92.5% of gain for documents having sizes 128MB and 1GB respectively. This considerable gain of the execution time while using projection is due to a significant decrease of shifts to be performed on the pages after an update execution, as it has been explained in Chapter 3.

A last kind of tests we made concerns the computation of a unique projection (using a global projector $\pi_{gb}$) for the following updates executed in the following order: U5, U3, U6, U18 and U8. The document has been projected once, then all the updates have been evaluated on the projection, and finally Merge has been executed once to obtain the final document. The obtained results are reported in Table 4.3. As the reader can observe with Saxon, Qizx and BaseX this took, respectively, 87.869, 93.581 and 326.476 seconds on the 128MB document. For this document, the sum of total times needed to projecting, updating and merging for each single update was much higher, respectively 115.055, 442.965 and 442.507 seconds for Saxon, Qizx and BaseX.
4.4. Implementation and Experiments

The updates and the corresponding projectors

U1. for $x$ in $\text{doc/site/closed_auctions/closed_auction}$
where not ($x/\text{annotation}$) return
insert node $<\text{annotation}>\text{Empty Annotation}</\text{annotation}>$
as last into $x$

U2. for $x$ in $\text{doc/site/people/person/address}$
where $x/\text{country/text()}=\text{"United States"}$ return
(replace node $x$ with
$<\text{address}>
\textit{<street>{x/\text{street/text()}}</street>}
\textit{<city>"NewYork"</city>}
\textit{<country>"USA"</country>}
\textit{<province>{x/\text{province/text()}}</province>}
\textit{<zipcode>{x/\text{zipcode/text()}}</zipcode>}
</\text{address}>$)

U3. for $x$ in $\text{doc/site/regions/item/location}$
where $x/\text{text()}=\text{"United States"}$
return (replace value of node $x$ with $\text{"USA"}$)

U4. delete nodes $\text{doc/site/regions/item/mailbox/mail}$

U5. for $x$ in $\text{doc/site/text/bold}$ return
rename node $x$ as $\text{"emph"}$

U6. for $x$ in $\text{doc/site/people/person}$
where not($x/\text{homepage}$)
return insert node
$<\text{homepage}>\text{www.}{x/\text{name/text()}}\text{Page.com}</\text{homepage>}
after $x/\text{emailaddress}$

U7. for $x$ in $\text{doc/site/people/person}$,
for $y$ in $\text{doc/site/people/person}$
where $x/\text{name} = y/\text{name}$
and not ($y/\text{address}$) and $x/\text{address/country} = \text{"Malaysia"}$
return insert node $x/\text{address}$
after $y/\text{emailaddress}$

<table>
<thead>
<tr>
<th></th>
<th>Workload evaluation using a global projector</th>
<th>Update execution without the projection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Saxon</td>
<td>Qizx</td>
</tr>
<tr>
<td>$\pi_{gb}$ projection</td>
<td>9.903</td>
<td>9.903</td>
</tr>
<tr>
<td>update</td>
<td>33.858</td>
<td>39.57</td>
</tr>
<tr>
<td>merge</td>
<td>44.108</td>
<td>44.108</td>
</tr>
<tr>
<td>total</td>
<td>87.869</td>
<td>93.581</td>
</tr>
</tbody>
</table>

Table 4.3: Workload evaluation
<table>
<thead>
<tr>
<th>$\pi_{no}$</th>
<th>$\pi_{olb}$</th>
<th>$\pi_{eb}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1: site, closed_auctions, annotation</td>
<td>closed_auction</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>U2: site, people, address</td>
<td>person, country, street, province, zipcode</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>U3: site, regions, africa, asia, australia, europe, namerica, samerica, item</td>
<td>location</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>U4: site, regions, africa, asia, australia, europe, namerica, samerica, item, mailbox, mail</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>U5: site, regions, africa, asia, australia, europe, namerica, samerica, listitem, bold, mailbox, mail, item, description, text, open_auctions, open_auction, closed_auctions, closed_auction, annotation, parlist</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>U6: site, people, homepage, emailaddress</td>
<td>person, name</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>U7: site, people, emailaddress</td>
<td>person, name, country address</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>U8: site, regions, australia</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>U9: site, open_auctions, open_auction, closed_auctions</td>
<td>closed_auction annotation</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>U10: site, open_auctions, open_auction</td>
<td>privacy</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>U11: site, open_auctions, bidder, initial</td>
<td>open_auction, increase</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>U12: site, regions, africa, asia, australia, europe, namerica, samerica, mailbox, mail</td>
<td>item, date</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>U13: site, open_auctions, open_auction, annotation, description, keyword, bold</td>
<td>text, emph</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>U14: site, regions, africa, asia, australia, europe, namerica, samerica, item, description, parlist, listitem, mailbox, mail, closed_auctions, closed_auction, annotation, open_auctions, open_auction, text, emph</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>U15: site, categories, category, listitem</td>
<td>description</td>
<td>parlist</td>
</tr>
<tr>
<td>U16: site, closed_auctions</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>U17: site</td>
<td>closed_auctions</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>U18: site, categories, category, description, parlist</td>
<td>listitem</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>U19: site, categories, category, description</td>
<td>parlist</td>
<td>listitem</td>
</tr>
<tr>
<td>U20: site, open_auctions, increase</td>
<td>open_auction</td>
<td>bidder</td>
</tr>
</tbody>
</table>

Table 4.4: Three-level type projector for updates
4.4. Implementation and Experiments

<table>
<thead>
<tr>
<th>$\pi_{no}$</th>
<th>$\pi_{ob}$</th>
<th>$\pi_{eb}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>site, regions, africa, asia, australia, europe, namerica, samerica, bold, mailbox, mail, item, description, text, open_auctions, open_auction, closed_auctions, closed_auction, annotation, parlist, people, homepage, categories, category, description</td>
<td>location, person, name, parlist, listitem</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

Table 4.5: Three-level type projector for workload

U8. delete nodes $\text{doc/site/regions/australia}$

U9. let $\kappa := \text{doc/site/closed_auctions/closed_auction[last()]}$
    for $\beta$ in $\text{doc/site/open_auctions/open_auction[last()]}$
        return replace node $\kappa/annotation$ with $\beta/annotation$

U10. for $\chi$ in $\text{doc/site/open_auctions/open_auction}$
    where ($\chi/privacy$="Yes")
    return delete node $\chi$

U11. for $\chi$ in $\text{doc/site/open_auctions/open_auction}$
    where $\chi/bidder/increase < 20$
    return insert node
    
    <bidder>
    <date>08/17/2000</date>
    <time>15:15:15</time>
    <personref/>
    <increase>1.50</increase>
    </bidder>
    after $\chi/initial$

U12. for $\chi$ in $\text{doc/site/regions/item}$
    where ($\chi/mailbox/mail/date/text()="07/04/1998"$)
    return insert node <incategory/> before $\chi/mailbox$

U13. for $\chi$ in $\text{doc/site/open_auctions/open_auction/annotation/description/text}$
    where ($\chi/keyword/emph/text()="unique"$) and ($\chi/bold$)
    return insert node <emph>newText</emph> before $\chi/bold$

U14. for $\chi$ in $\text{doc/site/text/emph}$
    return delete node $\chi$

U15. for $\chi$ in $\text{doc/site/categories/category/description/parlist}$
    where ($\chi/listitem/parlist$) return
    replace node $\chi$ with $\chi/listitem/parlist[1]$
U16. for $x$ in $doc/site/closed_auctions$
    return delete node $x$

U17. for $x$ in $doc/site/closed_auctions$
    return insert node
    <closed_auction>
        <seller/>
        <buyer/>
        <itemref/>
        <price>39.58</price>
        <date>02/15/1998</date>
        <quantity>1</quantity>
        <type>Regular_new</type>
        <annotation/>
    </closed_auction> as last into $x$

U18. for $x$ in $doc/site/categories/category/description/parlist/listitem$
    where ($x/parlist) return
    replace node $x/parlist$ with <text>newText</text>

U19. for $x$ in $doc/site/categories/category/description/parlist/listitem$
    return replace node $x$ with $x/parlist/listitem[1]$

U20. for $x$ in $doc/site/categories/category/description/parlist/listitem$
    return replace node $x$ with $x/parlist/listitem$

4.5 Conclusion

In this Chapter we have presented the experiments performed with the propose to prove the effectiveness of our method. Our goal was to illustrate that our technique optimizes the memory limitations of the existing native XML update engines. We have tested XML documents having sizes of 128MB, 1GB, 1-5GB and 2GB. The results of these experiments demonstrate that the updates execution with using projection can process documents having sizes up to 2GB. While executing the same updates without projection fails to evaluate updates on the documents with sizes staring from 1GB for Saxon and Qizx. On the contrary, the experiments report the effectiveness of BaseX for the memory usage, some of the updates have been executed up to 2GB. Nevertheless, that executing updates using BaseX is more expensive from the execution time point of
view. The experiments demonstrate that using projection results in time improvements for the most of the cases. These improvements are explained by the fact that in BaseX for some of the updates performing an insertions or deletions of nodes results in new page insertions or tuple shifts. Therefore, the time improvements while executing for some of the updates using projection is because no shifts are required.

As the reader can observe the execution of the updates U5, U9, U11, U12, U14 and U20 on Saxon reports not very satisfactory results. This is due to low selectivity of the three-level type projector for these updates. In the next Chapter we present the extension of our method, mainly the extension of three-level type projector which optimizes memory savings.
### Qizx update execution

#### without projection

<table>
<thead>
<tr>
<th>Update query</th>
<th>128MB</th>
<th>1GB</th>
<th>1.50GB</th>
<th>2GB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>importing time</strong></td>
<td>43.969</td>
<td>5.947</td>
<td>8.557</td>
<td>10.3</td>
</tr>
<tr>
<td><strong>query exec. time</strong></td>
<td>0.046</td>
<td>0.041</td>
<td>0.048</td>
<td>0.054</td>
</tr>
<tr>
<td><strong>total exec. time</strong></td>
<td>44.015</td>
<td>5.988</td>
<td>8.605</td>
<td>10.354</td>
</tr>
</tbody>
</table>

**U2**

| **importing time** | 43.888 | | 20.455 | 27.36 | 38.263 |
| **query exec. time** | 45.597 | | 24.901 | 34.825 | 45.887 |
| **total exec. time** | 89.485 | | 45.356 | 62.185 | 84.15 |

**U3**

| **importing time** | 43.854 | | 5.065 | 6.634 | 8.825 |
| **query exec. time** | 45.098 | | 7.081 | 9.532 | 11.597 |
| **total exec. time** | 88.952 | | 12.146 | 16.166 | 20.422 |

**U4**

| **importing time** | 43.859 | | 6.485 | 9.029 | 11.186 |
| **query exec. time** | 38.251 | | 7.296 | 10.035 | 9.558 |
| **total exec. time** | 82.11 | | 13.781 | 19.064 | 20.744 |

**U5**

| **importing time** | 43.317 | | 32.858 | 44.801 | 57.091 |
| **query exec. time** | 46.235 | | 35.505 | 46.132 | 61.142 |
| **total exec. time** | 89.552 | | 68.363 | 90.933 | 118.233 |

**U6**

| **importing time** | 43.817 | | 20.81 | 27.539 | 33.574 |
| **query exec. time** | 46.001 | | 24.958 | 33.157 | 42.062 |
| **total exec. time** | 89.818 | | 45.768 | 60.696 | 75.636 |

**U7**

| **importing time** | 43.811 | | 23.25 | 30.242 | 38.335 |
| **query exec. time** | 0.958 | | 62.834 | 107.09 | 159.086 |
| **total exec. time** | 44.769 | | 86.084 | 137.332 | 197.421 |

**U8**

| **importing time** | 43.82 | | 0.92 | 0.114 | 0.111 |
| **query exec. time** | 42.011 | | 0.043 | 0.037 | 0.044 |
| **total exec. time** | 85.831 | | 0.963 | 0.151 | 0.155 |

**U9**

| **importing time** | 43.725 | | 104.839 | 151.426 | 193.123 |
| **query exec. time** | 45.187 | | 121.378 | 170.81 | - |
| **total exec. time** | 88.912 | | 226.217 | 322.236 | - |

**U10**

| **importing time** | 44.368 | | 2.582 | 3.302 | 4.224 |
| **query exec. time** | 41.282 | | 1.541 | 2.067 | 2.931 |
| **total exec. time** | 85.65 | | 4.123 | 5.369 | 7.155 |


<table>
<thead>
<tr>
<th>Qizx update execution</th>
<th>Qizx update execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>without projection</td>
<td>on the projection</td>
</tr>
<tr>
<td>Update query 128MB</td>
<td>128MB 1GB 1.50GB 2GB</td>
</tr>
<tr>
<td>U11</td>
<td></td>
</tr>
<tr>
<td>importing time</td>
<td>44.116</td>
</tr>
<tr>
<td>query exec. time</td>
<td>47.31</td>
</tr>
<tr>
<td>total exec. time</td>
<td>91.426</td>
</tr>
<tr>
<td>importing time</td>
<td>2.945</td>
</tr>
<tr>
<td>query exec. time</td>
<td>3.551</td>
</tr>
<tr>
<td>total exec. time</td>
<td>6.496</td>
</tr>
<tr>
<td>U12</td>
<td></td>
</tr>
<tr>
<td>importing time</td>
<td>43.899</td>
</tr>
<tr>
<td>query exec. time</td>
<td>44.725</td>
</tr>
<tr>
<td>total exec. time</td>
<td>88.624</td>
</tr>
<tr>
<td>importing time</td>
<td>3.71</td>
</tr>
<tr>
<td>query exec. time</td>
<td>3.226</td>
</tr>
<tr>
<td>total exec. time</td>
<td>6.936</td>
</tr>
<tr>
<td>U13</td>
<td></td>
</tr>
<tr>
<td>importing time</td>
<td>43.707</td>
</tr>
<tr>
<td>query exec. time</td>
<td>45.588</td>
</tr>
<tr>
<td>total exec. time</td>
<td>89.295</td>
</tr>
<tr>
<td>importing time</td>
<td>1.77</td>
</tr>
<tr>
<td>query exec. time</td>
<td>1.586</td>
</tr>
<tr>
<td>total exec. time</td>
<td>3.356</td>
</tr>
<tr>
<td>U14</td>
<td></td>
</tr>
<tr>
<td>importing time</td>
<td>43.865</td>
</tr>
<tr>
<td>query exec. time</td>
<td>43.655</td>
</tr>
<tr>
<td>total exec. time</td>
<td>87.52</td>
</tr>
<tr>
<td>importing time</td>
<td>3.331</td>
</tr>
<tr>
<td>query exec. time</td>
<td>3.221</td>
</tr>
<tr>
<td>total exec. time</td>
<td>6.552</td>
</tr>
<tr>
<td>U15</td>
<td></td>
</tr>
<tr>
<td>importing time</td>
<td>43.982</td>
</tr>
<tr>
<td>query exec. time</td>
<td>45.241</td>
</tr>
<tr>
<td>total exec. time</td>
<td>89.223</td>
</tr>
<tr>
<td>importing time</td>
<td>0.895</td>
</tr>
<tr>
<td>query exec. time</td>
<td>0.494</td>
</tr>
<tr>
<td>total exec. time</td>
<td>1.389</td>
</tr>
<tr>
<td>U16</td>
<td></td>
</tr>
<tr>
<td>importing time</td>
<td>43.974</td>
</tr>
<tr>
<td>query exec. time</td>
<td>38.74</td>
</tr>
<tr>
<td>total exec. time</td>
<td>82.714</td>
</tr>
<tr>
<td>importing time</td>
<td>0.127</td>
</tr>
<tr>
<td>query exec. time</td>
<td>0.04</td>
</tr>
<tr>
<td>total exec. time</td>
<td>0.167</td>
</tr>
<tr>
<td>U17</td>
<td></td>
</tr>
<tr>
<td>importing time</td>
<td>43.872</td>
</tr>
<tr>
<td>query exec. time</td>
<td>44.095</td>
</tr>
<tr>
<td>total exec. time</td>
<td>87.967</td>
</tr>
<tr>
<td>importing time</td>
<td>0.655</td>
</tr>
<tr>
<td>query exec. time</td>
<td>0.138</td>
</tr>
<tr>
<td>total exec. time</td>
<td>0.793</td>
</tr>
<tr>
<td>U18</td>
<td></td>
</tr>
<tr>
<td>importing time</td>
<td>44.001</td>
</tr>
<tr>
<td>query exec. time</td>
<td>44.811</td>
</tr>
<tr>
<td>total exec. time</td>
<td>88.812</td>
</tr>
<tr>
<td>importing time</td>
<td>0.372</td>
</tr>
<tr>
<td>query exec. time</td>
<td>0.12</td>
</tr>
<tr>
<td>total exec. time</td>
<td>0.492</td>
</tr>
<tr>
<td>U19</td>
<td></td>
</tr>
<tr>
<td>importing time</td>
<td>43.994</td>
</tr>
<tr>
<td>query exec. time</td>
<td>45.107</td>
</tr>
<tr>
<td>total exec. time</td>
<td>90.101</td>
</tr>
<tr>
<td>importing time</td>
<td>0.876</td>
</tr>
<tr>
<td>query exec. time</td>
<td>0.347</td>
</tr>
<tr>
<td>total exec. time</td>
<td>1.223</td>
</tr>
<tr>
<td>U20</td>
<td></td>
</tr>
<tr>
<td>importing time</td>
<td>43.7</td>
</tr>
<tr>
<td>query exec. time</td>
<td>46.827</td>
</tr>
<tr>
<td>total exec. time</td>
<td>90.527</td>
</tr>
<tr>
<td>importing time</td>
<td>3.568</td>
</tr>
<tr>
<td>query exec. time</td>
<td>4.099</td>
</tr>
<tr>
<td>total exec. time</td>
<td>7.667</td>
</tr>
</tbody>
</table>

Table 4.6: Qizx update execution without projection and on the projection
<table>
<thead>
<tr>
<th>Update query</th>
<th>Saxon update execution without projection</th>
<th>Saxon update execution on the projection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128MB</td>
<td>1GB</td>
</tr>
<tr>
<td>U1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>analysis time</td>
<td>0.169</td>
<td>0.173</td>
</tr>
<tr>
<td>tree built time</td>
<td>7.933</td>
<td>0.432</td>
</tr>
<tr>
<td>execution time</td>
<td>8.176</td>
<td>0.546</td>
</tr>
<tr>
<td>total</td>
<td>16.278</td>
<td>1.151</td>
</tr>
<tr>
<td>memory used</td>
<td>498281944</td>
<td>23075200</td>
</tr>
<tr>
<td>U2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>analysis time</td>
<td>0.181</td>
<td>0.178</td>
</tr>
<tr>
<td>tree built time</td>
<td>7.933</td>
<td>0.906</td>
</tr>
<tr>
<td>execution time</td>
<td>17.576</td>
<td>1.693</td>
</tr>
<tr>
<td>total</td>
<td>25.69</td>
<td>2.777</td>
</tr>
<tr>
<td>memory used</td>
<td>512195048</td>
<td>53457896</td>
</tr>
<tr>
<td>U3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>analysis time</td>
<td>0.172</td>
<td>0.193</td>
</tr>
<tr>
<td>tree built time</td>
<td>8.004</td>
<td>0.223</td>
</tr>
<tr>
<td>execution time</td>
<td>12.847</td>
<td>0.633</td>
</tr>
<tr>
<td>total</td>
<td>21.023</td>
<td>1.049</td>
</tr>
<tr>
<td>memory used</td>
<td>504621904</td>
<td>21211328</td>
</tr>
<tr>
<td>U4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>analysis time</td>
<td>0.156</td>
<td>0.152</td>
</tr>
<tr>
<td>tree built time</td>
<td>8.092</td>
<td>0.314</td>
</tr>
<tr>
<td>execution time</td>
<td>12.363</td>
<td>0.631</td>
</tr>
<tr>
<td>total</td>
<td>20.611</td>
<td>1.097</td>
</tr>
<tr>
<td>memory used</td>
<td>501430264</td>
<td>23681936</td>
</tr>
<tr>
<td>U5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>analysis time</td>
<td>0.16</td>
<td>0.157</td>
</tr>
<tr>
<td>tree built time</td>
<td>8.078</td>
<td>1.358</td>
</tr>
<tr>
<td>execution time</td>
<td>18.398</td>
<td>2.228</td>
</tr>
<tr>
<td>total</td>
<td>26.636</td>
<td>3.743</td>
</tr>
<tr>
<td>memory used</td>
<td>530171776</td>
<td>101053248</td>
</tr>
<tr>
<td>U6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>analysis time</td>
<td>0.175</td>
<td>0.199</td>
</tr>
<tr>
<td>tree built time</td>
<td>8.26</td>
<td>0.711</td>
</tr>
<tr>
<td>execution time</td>
<td>17.522</td>
<td>1.457</td>
</tr>
<tr>
<td>total</td>
<td>25.957</td>
<td>2.367</td>
</tr>
<tr>
<td>memory used</td>
<td>516794712</td>
<td>484040608</td>
</tr>
<tr>
<td>U7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>analysis time</td>
<td>0.178</td>
<td>0.172</td>
</tr>
<tr>
<td>tree built time</td>
<td>8.035</td>
<td>0.753</td>
</tr>
<tr>
<td>execution time</td>
<td>11.002</td>
<td>1.411</td>
</tr>
<tr>
<td>total</td>
<td>19.215</td>
<td>2.336</td>
</tr>
<tr>
<td>memory used</td>
<td>516562888</td>
<td>47116112</td>
</tr>
<tr>
<td>U8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>analysis time</td>
<td>0.153</td>
<td>0.151</td>
</tr>
<tr>
<td>tree built time</td>
<td>8.157</td>
<td>0.008</td>
</tr>
<tr>
<td>execution time</td>
<td>12.398</td>
<td>0.102</td>
</tr>
<tr>
<td>total</td>
<td>20.708</td>
<td>0.261</td>
</tr>
<tr>
<td>memory used</td>
<td>497630640</td>
<td>11396000</td>
</tr>
<tr>
<td>U9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>analysis time</td>
<td>0.166</td>
<td>0.162</td>
</tr>
<tr>
<td>tree built time</td>
<td>7.982</td>
<td>1.589</td>
</tr>
<tr>
<td>execution time</td>
<td>12.427</td>
<td>2.757</td>
</tr>
<tr>
<td>total</td>
<td>20.575</td>
<td>4.508</td>
</tr>
<tr>
<td>memory used</td>
<td>498240800</td>
<td>115656128</td>
</tr>
<tr>
<td>U10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>analysis time</td>
<td>0.166</td>
<td>0.164</td>
</tr>
<tr>
<td>tree built time</td>
<td>7.968</td>
<td>0.152</td>
</tr>
<tr>
<td>execution time</td>
<td>18.778</td>
<td>1.779</td>
</tr>
<tr>
<td>total</td>
<td>26.912</td>
<td>2.095</td>
</tr>
<tr>
<td>memory used</td>
<td>511625264</td>
<td>18784608</td>
</tr>
</tbody>
</table>
### 4.5. Conclusion

<p>| Table 4.7: Saxon update execution without projection and on the projection |
|-----------------------------|-----------------------------|-----------------------------|-----------------------------|-----------------------------|-----------------------------|
| analysis time               | tree built time             | execution time              | total                        | memory used                |
| 0.167                       | 0.165                       | 0.163                       | 0.165                       | 0.164                      |
| 7.976                       | 0.129                       | 0.561                       | 0.723                       | 0.899                      |
| 12.657                      | 0.3                        | 1.261                       | 1.71                        | 1.975                      |
| 20.8                        | 0.594                       | 1.985                       | 2.598                       | 3.038                      |
| 498525408                   | 89266161                   | 53135120                    | 64903808                    | 85213224                   |
| analysis time               | tree built time             | execution time              | total                        | memory used                |
| 0.158                       | 0.158                       | -                           | -                           | -                          |
| 8.495                       | 1.36                        | -                           | -                           | -                          |
| 32.182                      | 2.196                       | -                           | -                           | -                          |
| 40.835                      | 3.714                       | -                           | -                           | -                          |
| 512383104                   | 85633200                   | -                           | -                           | -                          |
| analysis time               | tree built time             | execution time              | total                        | memory used                |
| 0.166                       | 0.163                       | 0.163                       | 0.164                       | 0.164                      |
| 7.975                       | 0.102                       | 0.443                       | 0.581                       | 0.738                      |
| 12.552                      | 0.256                       | 0.64                        | 0.878                       | 1.018                      |
| 20.693                      | 0.521                       | 1.246                       | 1.623                       | 1.92                       |
| 498075423                   | 15679328                   | 24466160                    | 32162144                    | 39067860                   |
| analysis time               | tree built time             | execution time              | total                        | memory used                |
| 0.175                       | 0.174                       | 0.174                       | 0.172                       | 0.174                      |
| 8.014                       | 0.081                       | 0.275                       | 0.316                       | 0.412                      |
| 12.542                      | 0.234                       | 0.522                       | 0.644                       | 0.751                      |
| 20.731                      | 0.489                       | 0.971                       | 1.132                       | 1.337                      |
| 499037944                   | 15035272                   | 23899600                    | 24514856                    | 26556376                   |
| analysis time               | tree built time             | execution time              | total                        | memory used                |
| 0.159                       | 0.157                       | 0.158                       | 0.157                       | 0.158                      |
| 8.079                       | 0.126                       | 0.578                       | 0.734                       | 0.897                      |
| 12.516                      | 0.334                       | 1.016                       | 1.416                       | 1.67                       |
| 20.754                      | 0.617                       | 1.752                       | 2.307                       | 2.725                      |
| 499026008                   | 9247736                    | 47510304                    | 68573672                    | 79026760                   |
| analysis time               | tree built time             | execution time              | total                        | memory used                |
| 0.18                        | 0.177                       | -                           | -                           | -                          |
| 8.122                       | 1.147                       | -                           | -                           | -                          |
| 44.695                      | 25.076                      | -                           | -                           | -                          |
| 52.997                      | 26.4                        | -                           | -                           | -                          |
| 511534016                   | 66656024                   | Cannot build tree           | -                           | -                          |</p>
<table>
<thead>
<tr>
<th></th>
<th>BaseX update execution without the projection</th>
<th>BaseX update execution on the projection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128MB</td>
<td>1GB</td>
</tr>
<tr>
<td>Importing time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U1</td>
<td>18.353</td>
<td>156.771</td>
</tr>
<tr>
<td>Update</td>
<td>0.131</td>
<td>0.259</td>
</tr>
<tr>
<td>Exporting time</td>
<td>11.353</td>
<td>106.841</td>
</tr>
<tr>
<td>Total</td>
<td>29.837</td>
<td>272.414</td>
</tr>
<tr>
<td>U2</td>
<td>18.858</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>354.673</td>
<td></td>
</tr>
<tr>
<td>Exporting time</td>
<td>11.95</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>384.881</td>
<td></td>
</tr>
<tr>
<td>U3</td>
<td>18.361</td>
<td>165.684</td>
</tr>
<tr>
<td>Update</td>
<td>1.027</td>
<td>7.609</td>
</tr>
<tr>
<td>Exporting time</td>
<td>11.63</td>
<td>117.677</td>
</tr>
<tr>
<td>Total</td>
<td>31.018</td>
<td>290.97</td>
</tr>
<tr>
<td>U4</td>
<td>17.489</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>185.246</td>
<td></td>
</tr>
<tr>
<td>Exporting time</td>
<td>9.532</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>212.267</td>
<td></td>
</tr>
<tr>
<td>U5</td>
<td>17.084</td>
<td>165.661</td>
</tr>
<tr>
<td>Update</td>
<td>1.343</td>
<td>12.136</td>
</tr>
<tr>
<td>Exporting time</td>
<td>11.281</td>
<td>112.277</td>
</tr>
<tr>
<td>Total</td>
<td>32.536</td>
<td>299.074</td>
</tr>
<tr>
<td>U6</td>
<td>18.882</td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td>291.624</td>
<td></td>
</tr>
<tr>
<td>Exporting time</td>
<td>11.356</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>321.862</td>
<td></td>
</tr>
<tr>
<td>U7</td>
<td>18.072</td>
<td>171.486</td>
</tr>
<tr>
<td>Update</td>
<td>2.33</td>
<td>219.07</td>
</tr>
<tr>
<td>Exporting time</td>
<td>11.762</td>
<td>149.056</td>
</tr>
<tr>
<td>Total</td>
<td>32.164</td>
<td>539.612</td>
</tr>
<tr>
<td>U8</td>
<td>18.932</td>
<td>170.856</td>
</tr>
<tr>
<td>Update</td>
<td>0.028</td>
<td>0.45</td>
</tr>
<tr>
<td>Exporting time</td>
<td>10.831</td>
<td>106.988</td>
</tr>
<tr>
<td>Total</td>
<td>29.791</td>
<td>277.889</td>
</tr>
<tr>
<td>U9</td>
<td>18.008</td>
<td>172.562</td>
</tr>
<tr>
<td>Update</td>
<td>0.693</td>
<td>0.747</td>
</tr>
<tr>
<td>Exporting time</td>
<td>11.817</td>
<td>111.739</td>
</tr>
<tr>
<td>Total</td>
<td>29.918</td>
<td>284.781</td>
</tr>
<tr>
<td>U10</td>
<td>18.55</td>
<td>159.103</td>
</tr>
<tr>
<td>Update</td>
<td>75.543</td>
<td>6900.808</td>
</tr>
<tr>
<td>Exporting time</td>
<td>11.411</td>
<td>108.157</td>
</tr>
<tr>
<td>Total</td>
<td>105.504</td>
<td>7168.068</td>
</tr>
</tbody>
</table>
### 4.5. Conclusion

Table 4.8: BaseX update execution without projection and on the projection

<table>
<thead>
<tr>
<th>Update query</th>
<th>128MB</th>
<th>1GB</th>
<th>1.50GB</th>
<th>2GB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>without the projection</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>BaseX update execution</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>U11</strong></td>
<td>18.59</td>
<td>323.069</td>
<td>12.169</td>
<td>353.768</td>
</tr>
<tr>
<td><strong>importing time</strong></td>
<td>18.565</td>
<td>159.556</td>
<td>222.67</td>
<td>293.134</td>
</tr>
<tr>
<td><strong>update</strong></td>
<td>17.814</td>
<td>160.613</td>
<td>229.253</td>
<td>294.838</td>
</tr>
<tr>
<td><strong>exporting time</strong></td>
<td>11.807</td>
<td>106.304</td>
<td>157.704</td>
<td>212.259</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td>30.952</td>
<td>283.974</td>
<td>415.866</td>
<td>565.703</td>
</tr>
<tr>
<td><strong>U12</strong></td>
<td>18.114</td>
<td>0.58</td>
<td>17.066</td>
<td>11.801</td>
</tr>
<tr>
<td><strong>importing time</strong></td>
<td>19.697</td>
<td>0.814</td>
<td>17.066</td>
<td>11.801</td>
</tr>
<tr>
<td><strong>update</strong></td>
<td>0.384</td>
<td>0.326</td>
<td>6.396</td>
<td>8.002</td>
</tr>
<tr>
<td><strong>exporting time</strong></td>
<td>0.331</td>
<td>10.627</td>
<td>18.298</td>
<td>30.4</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td>24.308</td>
<td>27.119</td>
<td>41.75</td>
<td>60.074</td>
</tr>
<tr>
<td><strong>U13</strong></td>
<td>17.066</td>
<td>1.693</td>
<td>10.432</td>
<td>13.388</td>
</tr>
<tr>
<td><strong>importing time</strong></td>
<td>17.814</td>
<td>160.613</td>
<td>229.253</td>
<td>294.838</td>
</tr>
<tr>
<td><strong>update</strong></td>
<td>0.206</td>
<td>8.663</td>
<td>2.578</td>
<td>6.396</td>
</tr>
<tr>
<td><strong>exporting time</strong></td>
<td>12.262</td>
<td>110.049</td>
<td>159.793</td>
<td>211.907</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td>30.282</td>
<td>279.325</td>
<td>391.624</td>
<td>415.866</td>
</tr>
<tr>
<td><strong>U14</strong></td>
<td>18.804</td>
<td>1441.761</td>
<td>11.801</td>
<td>220.921</td>
</tr>
<tr>
<td><strong>importing time</strong></td>
<td>18.661</td>
<td>160.249</td>
<td>225.684</td>
<td>299.406</td>
</tr>
<tr>
<td><strong>update</strong></td>
<td>0.026</td>
<td>0.106</td>
<td>0.196</td>
<td>0.185</td>
</tr>
<tr>
<td><strong>exporting time</strong></td>
<td>10.077</td>
<td>92.492</td>
<td>125.768</td>
<td>166.269</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td>28.764</td>
<td>252.847</td>
<td>351.648</td>
<td>465.86</td>
</tr>
<tr>
<td><strong>U16</strong></td>
<td>18.697</td>
<td>0.275</td>
<td>0.275</td>
<td>0.3</td>
</tr>
<tr>
<td><strong>importing time</strong></td>
<td>18.335</td>
<td>159.541</td>
<td>224.741</td>
<td>298.024</td>
</tr>
<tr>
<td><strong>update</strong></td>
<td>0.002</td>
<td>0.032</td>
<td>0.041</td>
<td>0.041</td>
</tr>
<tr>
<td><strong>exporting time</strong></td>
<td>11.417</td>
<td>110.229</td>
<td>153.81</td>
<td>234.228</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td>29.874</td>
<td>269.002</td>
<td>351.648</td>
<td>532.762</td>
</tr>
<tr>
<td><strong>U17</strong></td>
<td>18.021</td>
<td>0.015</td>
<td>0.015</td>
<td>0.015</td>
</tr>
<tr>
<td><strong>importing time</strong></td>
<td>18.149</td>
<td>159.572</td>
<td>221.512</td>
<td>298.273</td>
</tr>
<tr>
<td><strong>update</strong></td>
<td>0.386</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td><strong>exporting time</strong></td>
<td>11.393</td>
<td>113.571</td>
<td>168.699</td>
<td>218.194</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td>29.928</td>
<td>291.147</td>
<td>421.368</td>
<td>565.31</td>
</tr>
<tr>
<td><strong>U18</strong></td>
<td>17.84</td>
<td>18.64</td>
<td>30.603</td>
<td>31.493</td>
</tr>
<tr>
<td><strong>importing time</strong></td>
<td>17.84</td>
<td>159.572</td>
<td>221.512</td>
<td>298.273</td>
</tr>
<tr>
<td><strong>update</strong></td>
<td>0.386</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td><strong>exporting time</strong></td>
<td>11.393</td>
<td>113.571</td>
<td>168.699</td>
<td>218.194</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td>29.928</td>
<td>291.147</td>
<td>421.368</td>
<td>565.31</td>
</tr>
</tbody>
</table>

**Table 4.8: BaseX update execution without projection and on the projection**
## Saxon total update execution on the projection

<table>
<thead>
<tr>
<th>Update query</th>
<th>128MB</th>
<th>1GB</th>
<th>1.50GB</th>
<th>2GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>projection</td>
<td>5.066</td>
<td>31.255</td>
<td>38.917</td>
<td>51.719</td>
</tr>
<tr>
<td>merge</td>
<td>14.706</td>
<td>110.897</td>
<td>154.434</td>
<td>193.142</td>
</tr>
<tr>
<td>update</td>
<td>1.151</td>
<td>7.671</td>
<td>10.252</td>
<td>13.125</td>
</tr>
<tr>
<td>total</td>
<td>20.923</td>
<td>149.823</td>
<td>203.603</td>
<td>257.986</td>
</tr>
<tr>
<td>projection</td>
<td>5.782</td>
<td>40.905</td>
<td>50.905</td>
<td>67.795</td>
</tr>
<tr>
<td>merge</td>
<td>17.857</td>
<td>142.671</td>
<td>194.75</td>
<td>240.482</td>
</tr>
<tr>
<td>update</td>
<td>2.777</td>
<td>21.604</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>26.416</td>
<td>205.18</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>projection</td>
<td>4.877</td>
<td>32.233</td>
<td>40.37</td>
<td>53.469</td>
</tr>
<tr>
<td>merge</td>
<td>14.014</td>
<td>113.041</td>
<td>154.366</td>
<td>194.162</td>
</tr>
<tr>
<td>update</td>
<td>1.049</td>
<td>5.306</td>
<td>7.01</td>
<td>8.708</td>
</tr>
<tr>
<td>total</td>
<td>19.94</td>
<td>150.58</td>
<td>201.746</td>
<td>256.339</td>
</tr>
<tr>
<td>projection</td>
<td>4.967</td>
<td>34.914</td>
<td>41.37</td>
<td>57.199</td>
</tr>
<tr>
<td>merge</td>
<td>13.042</td>
<td>96.83</td>
<td>130.459</td>
<td>183.603</td>
</tr>
<tr>
<td>update</td>
<td>1.097</td>
<td>7.294</td>
<td>9.731</td>
<td>12.215</td>
</tr>
<tr>
<td>total</td>
<td>19.106</td>
<td>139.038</td>
<td>181.56</td>
<td>235.017</td>
</tr>
<tr>
<td>projection</td>
<td>5.435</td>
<td>37.793</td>
<td>47.72</td>
<td>63.363</td>
</tr>
<tr>
<td>merge</td>
<td>17.419</td>
<td>140.027</td>
<td>181.971</td>
<td>230.477</td>
</tr>
<tr>
<td>update</td>
<td>2.367</td>
<td>16.196</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>25.221</td>
<td>194.016</td>
<td>252.316</td>
<td>328.077</td>
</tr>
<tr>
<td>projection</td>
<td>6.089</td>
<td>38.866</td>
<td>49.332</td>
<td>65.945</td>
</tr>
<tr>
<td>merge</td>
<td>17.416</td>
<td>133.789</td>
<td>183.376</td>
<td>233.264</td>
</tr>
<tr>
<td>update</td>
<td>2.336</td>
<td>40.116</td>
<td>65.029</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>25.841</td>
<td>212.771</td>
<td>298.097</td>
<td>-</td>
</tr>
<tr>
<td>projection</td>
<td>3.635</td>
<td>26.282</td>
<td>31.566</td>
<td>42.792</td>
</tr>
<tr>
<td>merge</td>
<td>8.854</td>
<td>69.706</td>
<td>98.752</td>
<td>122.516</td>
</tr>
<tr>
<td>update</td>
<td>0.261</td>
<td>0.289</td>
<td>0.277</td>
<td>0.266</td>
</tr>
<tr>
<td>total</td>
<td>12.75</td>
<td>96.277</td>
<td>130.595</td>
<td>165.574</td>
</tr>
<tr>
<td>projection</td>
<td>9.704</td>
<td>68.682</td>
<td>90.789</td>
<td>115.86</td>
</tr>
<tr>
<td>merge</td>
<td>17.611</td>
<td>142.394</td>
<td>189.66</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>4.508</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>31.823</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>projection</td>
<td>4.537</td>
<td>29.875</td>
<td>36.898</td>
<td>48.951</td>
</tr>
<tr>
<td>merge</td>
<td>11.443</td>
<td>87.942</td>
<td>122.064</td>
<td>152.708</td>
</tr>
<tr>
<td>update</td>
<td>2.995</td>
<td>235.344</td>
<td>450.104</td>
<td>725.794</td>
</tr>
<tr>
<td>total</td>
<td>18.073</td>
<td>353.161</td>
<td>609.066</td>
<td>927.453</td>
</tr>
</tbody>
</table>

## Qizx total update execution on the projection

<table>
<thead>
<tr>
<th>Update query</th>
<th>128MB</th>
<th>1GB</th>
<th>1.50GB</th>
<th>2GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>projection</td>
<td>5.066</td>
<td>31.255</td>
<td>38.917</td>
<td>51.719</td>
</tr>
<tr>
<td>merge</td>
<td>14.706</td>
<td>110.897</td>
<td>154.434</td>
<td>193.142</td>
</tr>
<tr>
<td>update</td>
<td>1.151</td>
<td>7.671</td>
<td>10.252</td>
<td>13.125</td>
</tr>
<tr>
<td>total</td>
<td>20.923</td>
<td>149.823</td>
<td>203.603</td>
<td>257.986</td>
</tr>
<tr>
<td>projection</td>
<td>5.782</td>
<td>40.905</td>
<td>50.905</td>
<td>67.795</td>
</tr>
<tr>
<td>merge</td>
<td>17.857</td>
<td>142.671</td>
<td>194.75</td>
<td>240.482</td>
</tr>
<tr>
<td>update</td>
<td>2.777</td>
<td>21.604</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>26.416</td>
<td>205.18</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>projection</td>
<td>4.877</td>
<td>32.233</td>
<td>40.37</td>
<td>53.469</td>
</tr>
<tr>
<td>merge</td>
<td>14.014</td>
<td>113.041</td>
<td>154.366</td>
<td>194.162</td>
</tr>
<tr>
<td>update</td>
<td>1.049</td>
<td>5.306</td>
<td>7.01</td>
<td>8.708</td>
</tr>
<tr>
<td>total</td>
<td>19.94</td>
<td>150.58</td>
<td>201.746</td>
<td>256.339</td>
</tr>
<tr>
<td>projection</td>
<td>4.967</td>
<td>34.914</td>
<td>41.37</td>
<td>57.199</td>
</tr>
<tr>
<td>merge</td>
<td>13.042</td>
<td>96.83</td>
<td>130.459</td>
<td>183.603</td>
</tr>
<tr>
<td>update</td>
<td>1.097</td>
<td>7.294</td>
<td>9.731</td>
<td>12.215</td>
</tr>
<tr>
<td>total</td>
<td>19.106</td>
<td>139.038</td>
<td>181.56</td>
<td>235.017</td>
</tr>
<tr>
<td>projection</td>
<td>5.435</td>
<td>37.793</td>
<td>47.72</td>
<td>63.363</td>
</tr>
<tr>
<td>merge</td>
<td>17.419</td>
<td>140.027</td>
<td>181.971</td>
<td>230.477</td>
</tr>
<tr>
<td>update</td>
<td>2.367</td>
<td>16.196</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>25.221</td>
<td>194.016</td>
<td>252.316</td>
<td>328.077</td>
</tr>
<tr>
<td>projection</td>
<td>6.089</td>
<td>38.866</td>
<td>49.332</td>
<td>65.945</td>
</tr>
<tr>
<td>merge</td>
<td>17.416</td>
<td>133.789</td>
<td>183.376</td>
<td>233.264</td>
</tr>
<tr>
<td>update</td>
<td>2.336</td>
<td>40.116</td>
<td>65.029</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>25.841</td>
<td>212.771</td>
<td>298.097</td>
<td>-</td>
</tr>
<tr>
<td>projection</td>
<td>3.635</td>
<td>26.282</td>
<td>31.566</td>
<td>42.792</td>
</tr>
<tr>
<td>merge</td>
<td>8.854</td>
<td>69.706</td>
<td>98.752</td>
<td>122.516</td>
</tr>
<tr>
<td>update</td>
<td>0.261</td>
<td>0.289</td>
<td>0.277</td>
<td>0.266</td>
</tr>
<tr>
<td>total</td>
<td>12.75</td>
<td>96.277</td>
<td>130.595</td>
<td>165.574</td>
</tr>
<tr>
<td>projection</td>
<td>9.704</td>
<td>68.682</td>
<td>90.789</td>
<td>115.86</td>
</tr>
<tr>
<td>merge</td>
<td>17.611</td>
<td>142.394</td>
<td>189.66</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>4.508</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>31.823</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>projection</td>
<td>4.537</td>
<td>29.875</td>
<td>36.898</td>
<td>48.951</td>
</tr>
<tr>
<td>merge</td>
<td>11.443</td>
<td>87.942</td>
<td>122.064</td>
<td>152.708</td>
</tr>
<tr>
<td>update</td>
<td>2.995</td>
<td>235.344</td>
<td>450.104</td>
<td>725.794</td>
</tr>
<tr>
<td>total</td>
<td>18.073</td>
<td>353.161</td>
<td>609.066</td>
<td>927.453</td>
</tr>
<tr>
<td>Update query</td>
<td>Saxon total update execution on the projection</td>
<td>Qizx total update execution on the projection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------</td>
<td>---------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>128MB</td>
<td>1GB</td>
<td>1.50GB</td>
<td>2GB</td>
</tr>
<tr>
<td>U11 projection</td>
<td>6.378</td>
<td>44.919</td>
<td>56.042</td>
<td>74.53</td>
</tr>
<tr>
<td>merge</td>
<td>23.938</td>
<td>213.411</td>
<td>280.634</td>
<td>345.573</td>
</tr>
<tr>
<td>update</td>
<td>3.324</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>33.64</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>U12 projection</td>
<td>7.357</td>
<td>47.042</td>
<td>59.944</td>
<td>77.221</td>
</tr>
<tr>
<td>merge</td>
<td>24.332</td>
<td>203.574</td>
<td>273.321</td>
<td>345.553</td>
</tr>
<tr>
<td>update</td>
<td>3.318</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>35.007</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>U13 projection</td>
<td>4.892</td>
<td>56.814</td>
<td>49.4</td>
<td>99.319</td>
</tr>
<tr>
<td>total</td>
<td>22.667</td>
<td>194.26</td>
<td>233.971</td>
<td>330.907</td>
</tr>
<tr>
<td>U14 projection</td>
<td>6.562</td>
<td>46.596</td>
<td>62.371</td>
<td>80.73</td>
</tr>
<tr>
<td>merge</td>
<td>27.657</td>
<td>247.696</td>
<td>333.826</td>
<td>415.218</td>
</tr>
<tr>
<td>update</td>
<td>3.714</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>37.933</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>U15 projection</td>
<td>3.963</td>
<td>28.035</td>
<td>35.035</td>
<td>47.39</td>
</tr>
<tr>
<td>merge</td>
<td>9.719</td>
<td>83.937</td>
<td>112.008</td>
<td>145.334</td>
</tr>
<tr>
<td>update</td>
<td>0.594</td>
<td>1.985</td>
<td>2.598</td>
<td>3.038</td>
</tr>
<tr>
<td>total</td>
<td>14.276</td>
<td>113.957</td>
<td>149.641</td>
<td>195.762</td>
</tr>
<tr>
<td>U16 projection</td>
<td>3.515</td>
<td>25.367</td>
<td>31.474</td>
<td>44.121</td>
</tr>
<tr>
<td>merge</td>
<td>8.891</td>
<td>77.743</td>
<td>104.796</td>
<td>129.893</td>
</tr>
<tr>
<td>update</td>
<td>0.285</td>
<td>0.29</td>
<td>0.29</td>
<td>0.264</td>
</tr>
<tr>
<td>total</td>
<td>12.691</td>
<td>103.4</td>
<td>136.56</td>
<td>174.278</td>
</tr>
<tr>
<td>U17 projection</td>
<td>4.062</td>
<td>27.807</td>
<td>34.688</td>
<td>47.059</td>
</tr>
<tr>
<td>merge</td>
<td>9.773</td>
<td>84.616</td>
<td>111.96</td>
<td>146.452</td>
</tr>
<tr>
<td>update</td>
<td>0.521</td>
<td>1.246</td>
<td>1.623</td>
<td>1.92</td>
</tr>
<tr>
<td>U18 projection</td>
<td>3.708</td>
<td>25.701</td>
<td>32.397</td>
<td>44.169</td>
</tr>
<tr>
<td>merge</td>
<td>11.141</td>
<td>85.986</td>
<td>118.054</td>
<td>146.325</td>
</tr>
<tr>
<td>update</td>
<td>0.489</td>
<td>0.971</td>
<td>1.132</td>
<td>1.337</td>
</tr>
<tr>
<td>total</td>
<td>15.338</td>
<td>112.658</td>
<td>151.583</td>
<td>191.831</td>
</tr>
<tr>
<td>U19 projection</td>
<td>4.023</td>
<td>28.298</td>
<td>34.63</td>
<td>48.145</td>
</tr>
<tr>
<td>update</td>
<td>0.617</td>
<td>1.752</td>
<td>2.307</td>
<td>2.755</td>
</tr>
<tr>
<td>total</td>
<td>14.303</td>
<td>110.009</td>
<td>146.287</td>
<td>187.524</td>
</tr>
<tr>
<td>U20 projection</td>
<td>6.586</td>
<td>44.538</td>
<td>55.773</td>
<td>75.88</td>
</tr>
<tr>
<td>merge</td>
<td>12.946</td>
<td>106.932</td>
<td>144.72</td>
<td>188.678</td>
</tr>
<tr>
<td>update</td>
<td>26.4</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>45.932</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.9: Saxon and Qizx total update execution on the projection
<table>
<thead>
<tr>
<th>Update query</th>
<th>BaseX update execution without the projection</th>
<th>BaseX total update execution on the projection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128MB</td>
<td>1GB</td>
</tr>
<tr>
<td>U1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>projection</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>merge</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>U2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>projection</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>merge</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>384.881</td>
<td></td>
</tr>
<tr>
<td>U3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>projection</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>merge</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>31.018</td>
<td>290.97</td>
</tr>
<tr>
<td>U4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>projection</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>merge</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>212.267</td>
<td></td>
</tr>
<tr>
<td>U5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>projection</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>merge</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>31.018</td>
<td>290.97</td>
</tr>
<tr>
<td>U6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>projection</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>merge</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>321.862</td>
<td></td>
</tr>
<tr>
<td>U7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>projection</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>merge</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>32.164</td>
<td>539.612</td>
</tr>
<tr>
<td>U8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>projection</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>merge</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>29.791</td>
<td>277.889</td>
</tr>
<tr>
<td>U9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>projection</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>merge</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>29.918</td>
<td>284.781</td>
</tr>
<tr>
<td>U10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>projection</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>merge</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>105.504</td>
<td>7168.068</td>
</tr>
</tbody>
</table>
### 4.5. Conclusion

BaseX update execution without the projection

<table>
<thead>
<tr>
<th>Update query</th>
<th>128MB</th>
<th>1GB</th>
<th>1.50GB</th>
<th>2GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>U11 projection</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>merge</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>353.768</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U12 projection</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>merge</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>30.952</td>
<td>283.974</td>
<td>415.866</td>
<td>565.703</td>
</tr>
<tr>
<td>U13 projection</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>merge</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>30.282</td>
<td>279.325</td>
<td>391.624</td>
<td>513.144</td>
</tr>
<tr>
<td>U14 projection</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>merge</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>1471.656</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U16 projection</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>merge</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>28.764</td>
<td>252.847</td>
<td>351.648</td>
<td>465.86</td>
</tr>
<tr>
<td>U17 projection</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>merge</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>29.874</td>
<td>269.802</td>
<td>378.592</td>
<td>532.762</td>
</tr>
<tr>
<td>U18 projection</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>merge</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>29.928</td>
<td>291.147</td>
<td>421.368</td>
<td>1026.277</td>
</tr>
<tr>
<td>U20 projection</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>merge</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>185.178</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

BaseX total update execution on the projection

<table>
<thead>
<tr>
<th>Update query</th>
<th>128MB</th>
<th>1GB</th>
<th>1.50GB</th>
<th>2GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>U11 projection</td>
<td>6.378</td>
<td>23.938</td>
<td>248.732</td>
<td>279.048</td>
</tr>
<tr>
<td>merge</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U12 projection</td>
<td>7.357</td>
<td>24.332</td>
<td>2.408</td>
<td>34.097</td>
</tr>
<tr>
<td>merge</td>
<td>47.942</td>
<td>203.574</td>
<td>27.119</td>
<td>277.735</td>
</tr>
<tr>
<td>update</td>
<td>273.321</td>
<td>345.553</td>
<td>11.791</td>
<td>375.025</td>
</tr>
<tr>
<td>total</td>
<td>59.944</td>
<td>77.221</td>
<td>40.764</td>
<td>482.848</td>
</tr>
<tr>
<td>U13 projection</td>
<td>4.892</td>
<td>16.379</td>
<td>1.983</td>
<td>23.254</td>
</tr>
<tr>
<td>merge</td>
<td>56.814</td>
<td>128.597</td>
<td>8.849</td>
<td>194.26</td>
</tr>
<tr>
<td>update</td>
<td>172.78</td>
<td>217.321</td>
<td>11.791</td>
<td>233.971</td>
</tr>
<tr>
<td>total</td>
<td>49.4</td>
<td>99.319</td>
<td>14.267</td>
<td>330.907</td>
</tr>
<tr>
<td>U14 projection</td>
<td>6.562</td>
<td>27.657</td>
<td>220.921</td>
<td>255.14</td>
</tr>
<tr>
<td>merge</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>update</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U16 projection</td>
<td>3.515</td>
<td>8.891</td>
<td>0.291</td>
<td>12.697</td>
</tr>
<tr>
<td>merge</td>
<td>25.367</td>
<td>77.743</td>
<td>0.306</td>
<td>103.416</td>
</tr>
<tr>
<td>update</td>
<td>31.474</td>
<td>104.796</td>
<td>0.291</td>
<td>136.561</td>
</tr>
<tr>
<td>total</td>
<td>44.121</td>
<td>129.893</td>
<td>0.316</td>
<td>174.33</td>
</tr>
<tr>
<td>U17 projection</td>
<td>4.062</td>
<td>9.773</td>
<td>0.435</td>
<td>4.062</td>
</tr>
<tr>
<td>merge</td>
<td>27.807</td>
<td>84.616</td>
<td>0.134</td>
<td>112.557</td>
</tr>
<tr>
<td>update</td>
<td>34.688</td>
<td>111.96</td>
<td>1.214</td>
<td>147.803</td>
</tr>
<tr>
<td>total</td>
<td>47.059</td>
<td>146.452</td>
<td>1.491</td>
<td>195.002</td>
</tr>
<tr>
<td>U18 projection</td>
<td>3.708</td>
<td>11.141</td>
<td>0.401</td>
<td>3.708</td>
</tr>
<tr>
<td>merge</td>
<td>25.701</td>
<td>85.986</td>
<td>11.041</td>
<td>122.728</td>
</tr>
<tr>
<td>update</td>
<td>32.397</td>
<td>118.054</td>
<td>19.352</td>
<td>169.803</td>
</tr>
<tr>
<td>total</td>
<td>44.169</td>
<td>146.325</td>
<td>31.493</td>
<td>221.987</td>
</tr>
<tr>
<td>U20 projection</td>
<td>6.586</td>
<td>12.946</td>
<td>110.051</td>
<td>129.583</td>
</tr>
</tbody>
</table>

Table 4.10: BaseX update execution without projection and total update on the projection
Figure 4.37: Sequence Diagram
In this Chapter we present the extension of the type projector for update optimization. This extension aims to optimize memory savings. In Section 5.1 we explain the motivation of this extension. In Section 5.2 we introduce extensions applied on the type projector. In Section 5.3 we provide the definitions and explain the usage of new procedures added to the Merge algorithm.

5.1 Introduction

In the previous chapter, we introduced and discussed the update optimization method based on type projection where the projector used is a three-level type projector $\pi = \{\pi_{no}, \pi_{olb}, \pi_{eb}\}$. In our setting, optimization is essentially space oriented. The goal is to be able to process very large documents that do not fit in main memory and which cannot be handled by query engines. Thus, improving the update optimization method means providing a more precise projector. The aim of this chapter is to modify the type projector in order to further prune documents.

The starting point is that the "one level below" component of the projector may lead to projecting nodes that are not necessary for the update execution. In order to overcome this problem, the type projector is extended based on a careful analysis of update operations. These operations are classified into seven kinds: "insert as last", "insert as first", "insert before", "insert after", "replace", "delete" and "rename".
The analysis made for extracting the projector (extraction of the projector is out of the scope of this work) is not only based on the paths relevant for evaluating the update. The analysis also classifies the extracted paths with respect to the kinds of updates (or other access) they are involved with.

In order to motivate our approach, we propose to start by an example showing that the three level type projector may be improved and how it can be improved.

**Example 5.1.1 (Motivating example).**

Figure 5.2 provides an example of the application of the projection technique using the three-level type projector for a given DTD (see fig. 5.1), a document \( t \) (see fig. 5.2-3) and a given update \( u \) (see fig. 5.2-1) resulting in the insertion of a new element labelled by \( e \) as the last child of the element labelled by \( a \) (see fig. 5.2-9).

**projection** - The three-level projector for this update query is given in the Figure 5.2-2. The resulting projected tree \( t_1 \) is depicted in Figure 5.2-4. Note here, that the projector \( \pi \) selects all children of the nodes labelled by \( a \) because \( a \in \pi_{olb} \). In section 4.3 we have motivated this by showing that it is necessary for ensuring the correctness of the Merge phase.

**extended projection** - The purpose of the extended projector is to avoid projecting all children of nodes labelled by \( a \). For this example, we propose to use a new projector given in (see fig. 5.2-6). This type projector has a new component called \( \pi_{aslast} \). Its execution is depicted in (see fig. 5.2-7). Notice that this time, the nodes labelled by \( a \) are projected without their children.

**new behaviour of merge** - The Merge algorithm is changed as follows, taking into account the new projector component \( \pi_{aslast} \): when processing \( t@1 \) (see fig. 5.2-3) and \( t^{ext}@1 \) (see fig. 5.2-8), because \( a \in \pi_{aslast} \), Merge will first output in the final result all subtrees of \( t@1 \). Then, because \( a \in \pi_{aslast} \), all the subtrees of \( t^{ext}@1 \), which are the "as last" inserted elements (see fig. 5.2-9). Merging \( t@2 \) and \( t^{ext}@2 \) is done with the same rules.

The presentation is decomposed in two steps:

- First, we will introduce the new projector for update expressions involving only one kind of update operation at a time.
5.1. Introduction

for $x$ in /doc/a
return insert node <e/>
as last into $x$

(1) The update $u$

\[ \pi_{no} = \{doc\} \]
\[ \pi_{ahl} = \{a\} \]
\[ \pi_{ah} = \emptyset \]

(2) The three level type projector

\[ \text{doc} \]
\[ \text{a} \]
\[ \text{b} \]
\[ \text{c} \]
\[ \text{d} \]
\[ \text{f} \]
\[ \text{g} \]

(3) The XML document $t$

\[ \text{doc} \]
\[ \text{a} \]
\[ \text{b} \]
\[ \text{c} \]
\[ \text{d} \]
\[ \text{f} \]
\[ \text{g} \]

(4) The projection $t_1$ of $t$ wrt $\pi$

\[ \pi_{no} = \{doc\} \]
\[ \pi_{ahl} = \{a\} \]
\[ \pi_{ah} = \emptyset \]

(5) The partial update $u(t_1)$

\[ \text{doc} \]
\[ \text{a} \]
\[ \text{b} \]
\[ \text{c} \]
\[ \text{d} \]
\[ \text{f} \]
\[ \text{g} \]

(6) The extended projector $\pi_{ext}$ for $u$

\[ \text{doc} \]
\[ \text{a} \]
\[ \text{b} \]
\[ \text{c} \]
\[ \text{d} \]
\[ \text{f} \]
\[ \text{g} \]

(7) The projection $t_{1}^{ext}$ of $t$ wrt extended $\pi_{ext}$

\[ \text{doc} \]
\[ \text{a} \]
\[ \text{b} \]
\[ \text{c} \]
\[ \text{d} \]
\[ \text{f} \]
\[ \text{g} \]

(8) The partial update $u(t_{1}^{ext}) = t_{2}^{ext}$

\[ \text{doc} \]
\[ \text{a} \]
\[ \text{b} \]
\[ \text{c} \]
\[ \text{d} \]
\[ \text{f} \]
\[ \text{g} \]

(9) The Final result $u(t)$

Figure 5.2: Three level type projector versus the extended type projector
Next, we will analyze the impact of having several kinds of update operations involved in the same update expression.

It is worth noticing that the extension of the type projector entails several changes of the Merge algorithm. Recall that, on the one hand, in order to ensure correctness of the Merge phase, the type projector needs to contain enough "types" and, on the other hand, the Merge phase uses the type projector to decide which nodes to output and also to control the synchronized parsing of the initial document and the partial updated document. The case analysis below focuses on the new type projector and also provides the intuition on the new behavior of the Merge phase which will be more formally described in the last section.

5.2 Extending the Type Projector for Update Optimization

A careful analysis shows that it is possible to improve the precision of the type projector, in the way presented by the Example 5.2, for each kind of update operation. As already said, we start by presenting the case analysis for simple situations where the updates generate a unique kind of update operations. In the presentation of the examples the type projectors are specified by their non-empty components.

5.2.1 Case analysis: update operation in isolation

Case "insert as first" .................................................................

To explain the application of the extended projector during the projection of a document, we will consider the example of Figure 5.3.

Example 5.2.1. See Figure 5.3.
The update (see fig. 5.3-1) involves several update operations, each of them being of the same kind "insert as first". In this case, similarly to the motivating example (see fig. 5.2) we aim that it is sufficient to project nodes labelled by a without their children.

projector extraction - The type projector has a new component πasfirst (see fig. 5.3-2) capturing the types of node which are potentially target of "insert as first" operations.

projection - The projection wrt πext is very simple: first it projects node t@ε labelled by doc, then its children t@1 and t@2 labelled by a (see fig. 5.3-4). The distinction between nodes labelled by doc and nodes labelled by a is needed for the purpose of the Merge phase.

update - The result of the execution of the update u1 on the document t is given in the Figure 5.3-5. As the reader can observe, the subtree of the tree t2, rooted at t2@1 contains "as first" inserted subtrees rooted at t2@i (labelled by c), t2@i2 (labelled by b) and t2@i3 (labelled by c). It is similar for the subtree rooted at t2@2.
5.2. Extending the Type Projector for Update Optimization

for $x$ in /doc/a
return
{ insert nodes ( <e>new</e>,<b/>,<c/>)
as first into $x$ }

(1) The update query $u_1$

$$\pi_{ns}^\to = \{ \text{doc} \}
\pi_{asfirst}^\to = \{ a \}$$

(2) The extended projector $\pi^{ext}$ for $u_1$

(4) The projection $t_1$ wrt to $\pi^{ext}$

(3) The XML document $t$

(5) The partial update $t_2$

(6) Merging $t$ with $t_2$

Figure 5.3: Dealing with "insert as first" in isolation
**merge** - While processing nodes \( t@1 \) of \( t \) (see fig. 5.3-3) and \( t_2@1 \) of the partially updated tree \( t_2 \) (see fig. 5.3-5), the information given by the type projector that \( t@1 \) is labelled by \( a \in \pi_{as\text{\text{first}}} \) is used by the *Merge* phase which consequently gives priority to output the children of \( t_2@1 \) whose nodes are identified by \( i1, i2, i3 \) (see fig. 5.3-6). After that *Merge* continues to output all subtrees of \( t@1 \), since none of their root labels belongs to \( \pi_{ext} \). □

**Case "insert as last"** .................................................
This case is similar to the previous one. A \( \pi_{asl\text{\text{ast}}} \) component is introduced in the projection in order to deal with such case, as already illustrated in the motivation example.

**Case "insert before"** .................................................

**Example 5.2.2. See Figure 5.4.**
The update (see fig. 5.4-1) involves several update operations of the same kind "insert before" resulting in the insertion of new elements before the elements labelled by \( d \) and \( c \).

**projector extraction** - To project the document \( t \) (see fig. 5.4-3) we introduce a new projector component \( \pi_{bef} \) (see fig. 5.4-2). Notice here that the \( \pi_{bef} \) component is a set of pairs of types. This new specification is made to be able to refine the projector: intuitively, if \( (x, y) \) is a pair of labels in \( \pi_{bef} \), it means that nodes labelled by \( x \) are potentially parents of nodes labelled \( y \) which are potential targets of "insert before" operations; it also allows for avoiding to project nodes labelled by \( y \) when their parent is not labelled by \( x \). The use of pairs to specify the projector component is also going to be useful when considering the general case and mixing different kinds of insertion (for instance "before" and "after") as explained in Section 5.2.2.

Next, we use the following notation:

\[
\begin{align*}
\text{par}(\pi_{bef}) &= \{x | (x, y) \in \pi_{bef} \} \quad \text{and} \quad \text{ch}(\pi_{bef}) = \{y | (x, y) \in \pi_{bef} \}.
\end{align*}
\]

For the example, we have that \( \text{par}(\pi_{bef}) = \{a\} \) and \( \text{ch}(\pi_{bef}) = \{d, c\} \).

**projection** - The projection \( \text{wrt} \ \pi_{ext} \) (see fig. 5.4-2) of the document \( t \) is depicted in Figure 5.4-4. Its execution first selects node \( t@2 \) labelled \( doc \in \pi_{no} \) of the document \( t \). Next, because \( a \in \text{par}(\pi_{bef}) \), the node \( t@1 \) is projected. After that, the children of \( t@1 \) are parsed in order to project the node \( t@1.1 \) labelled by \( d \in \text{ch}(\pi_{bef}) \) and the nodes \( t@1.3 \) and \( t@1.4 \) labelled by \( c \in \text{ch}(\pi_{bef}) \). For the subtree \( t@2 \), the projection selects only the node \( t_2@2.1 \) labelled by \( d \in \text{ch}(\pi_{bef}) \). Once again, the distinction between nodes labelled by \( doc \) and nodes labelled by \( a, d \) and \( c \) is important for the *Merge* phase.

**update** - The result of the execution of the update \( u_2 \) on the document \( t_1 \) is given in Figure 5.4-5. The subtree rooted at \( t_2@1 \) contains the elements \( t_2@i1 \) and \( t_2@i2 \) "inserted before" the target node \( t_2@1.1 \), the elements \( t_2@i3 \) and \( t_2@i4 \) "inserted before" the target node \( t_2@1.3 \) and, finally, the elements \( t_2@i5 \) and \( t_2@i6 \) "inserted before" the target node \( t_2@1.4 \).
merge - Merge starts the synchronized parsing by examining the nodes $t@1$ of $t$ and $t_2@1$ of the partially updated tree $t_2$. Because $t@1$ is labelled by $a \in \text{par}(\pi_{\text{bef}})$, it outputs node $t_2@1$ (see fig. 5.4-6). Next the children of $t@1$ and $t_2@1$ are parsed and processed based on analyzing their labels with respect to $\pi_{\text{ext}}$. If the label of a node of $t$ does not belong to $\text{ch}(\pi_{\text{bef}})$, the node is output, otherwise, Merge is guided by $t_2$. For instance, Merge detects that the node $t@1.1$ is labelled by $d \in \text{ch}(\pi_{\text{bef}})$ thus it outputs the "inserted before" elements rooted at $t_2@1.1$ and $t_2@1.2$. When Merge encounters node $t_2@1.1$ labelled by $d$ it outputs it. After that, Merge is guided by $t$. The next subtree rooted at $t@1.2$ and labelled by $b \in \pi_{\text{ext}}$ is output. Similarly to the node $t@1.1$, when Merge detects that $t@1.3$ is labelled by $c \in \text{ch}(\pi_{\text{bef}})$, it outputs the "inserted before" elements rooted at $t_2@1.3$, $t_2@1.4$ from $t_2$, followed by the node $t_2@1.3$.

The Merge phase continues based on the same principles.

**Case "insert after"** ........................................

This case is treated in a similar manner. A $\pi_{\text{af}}$ component is used during the Projection and Merge phases.
Here we start by treating a rather simple example involving an update leading to a single "replace" operation. This first example investigates a simple solution to deal with "replace" operation which consists in introducing a special new component $\pi_{rep}$ in the style of the previous cases. It turns out that this choice fails to provide a solution.

Example 5.2.3. See Figures 5.5.

The update (see fig. 5.5-1) involves one update operation of a kind "replace".

**projector extraction** - Following the approach used for the previous cases, let us consider a type projector enriched by a new component $\pi_{rep}$ (see fig. 5.5-2), which is given by pairs of labels.

Next, $\text{par}(\pi_{rep})$ and $\text{ch}(\pi_{rep})$ are defined as expected.

**projection** - The projection of the document $t$ (see fig. 5.5-3) proceeds as follows: first it projects node $t@\varepsilon$ labelled by $\text{doc} \in \pi_{no}$; then node $t@1$ labelled by $a \in \text{par}(\pi_{rep})$ followed by its children $t@1.2$ and $t@1.5$ labelled by $b \in \text{ch}(\pi_{rep})$ (see fig. 5.5-4), where $\pi_{rep} = \{(a, b)\}$. Note that, the projection prunes out the subtrees...
rooted at $t@1.1$, $t@1.3$ and $t@1.4$. This is due to the fact that their roots are labelled by $d$ and $c$, and that these types do not belong to $\pi_{ext}$.

**update** - The result of the execution of the update $u_3$ is given in Figure 5.5-5. The subtree rooted at $t_2@1$ contains only the "replaced" subtrees rooted at $t_2@i1$, $t_2@i2$, $t_2@i3$ and $t_2@i4$ labelled by $k$.

**merge** - While processing nodes $t@1$ of $t$ and $t_2@1$ of the partially updated tree $t_2$, $Merge$ outputs the node $t_2@1$, since $t@1$ is labelled $a\in\text{par}(\pi_{rep})$. Then $Merge$ checks if the first child of $t@1$ is labelled by a type in $\pi_{ext}$. As it is not the case, $Merge$ outputs the subtree rooted at $t@1.1$ (see fig.5.5-6). The next child $t@1.2$ is labelled by $b\in\text{ch}(\pi_{rep})$, thus $Merge$ outputs the "replaced" elements rooted at $t_2@i1$, $t_2@i2$, $t_2@i3$ and $t_2@i4$.

As the reader can observe, there is no information enabling $Merge$ to separate $t_2@i1$, $t_2@i2$ from $t_2@i3$, $t_2@i4$. Therefore this merging process fails to produce the expected result. □

The solution proposed to solve the problem outlined in the previous example (lack of separator between inserted nodes) is to introduce a new projector component $\pi_{next}$ specified by pairs $(x, y)$ of labels. Intuitively, such pairs are used to capture nodes labelled by $y$ and to specify that they need to be projected together with their immediate sibling. It is the projection of the immediate sibling which is going to provide the separators between inserted elements and make our $Merge$ phase succeeding. The $x$ part of the pair is used as before to make the projector more precise and to prepare dealing with mixed update operations.

**Example 5.2.4.** See Figure 5.6

**projector extraction** - This time, the type projector for the update $u_4$ has a new component $\pi_{next}$ which replaces $\pi_{rep}$ (see fig. 5.6-2). Next, $\text{par}(\pi_{next})$ and $\text{ch}(\pi_{next})$ are defined as expected.

**projection** - The new component $\pi_{next} = \{(a, b), (a, e)\}$ (see fig. 5.6-2) indicates that the projection must project not only $b$- and $e$-nodes, but also the immediate siblings of these nodes. For our example the projected sibling is $t@1.3$ labelled by $c$ (see fig. 5.6-4). It is worth noticing that these nodes will play the role of separators during the $Merge$ phase.

**update** - The result of the execution of the update $u_4$ is given in Figure 5.6-5. The elements rooted at $t_2@1$, besides the "replaced" elements rooted at $t_2@i1$, $t_2@i2$ labelled by $k$ and $t_2@i3$ labelled by $f$, contain the node $t@1.3$, the next sibling of $t@1.2$.

**merge** - While merging the children of $t@1$ and $t_2@1$, $Merge$ first outputs the tree rooted at $t@1.1$, because it is labelled by $d\notin\pi$ (see fig. 5.6-3). Next $Merge$ examines nodes $t@1.2$ and $t_2@1$. Because $t@1.2$ is labelled by $b\in\text{ch}(\pi_{next})$ merging is guided by $t_2$, thus the element rooted at $t_2@i1$ is output. Next, $Merge$ examines $t@1.2$ and $t_2@i2$ and, for the same reason, outputs the element rooted at $t_2@i2$. After that, nodes $t@1.2$ and $t_2@1.3$ are processed and because $t@1.2$ is labelled by $b\in\text{ch}(\pi_{next})$
for $x$ in /doc/a
  return {
    replace node $x/b$ with $(<k/>,<k/>)$
    replace node $x/e$ with $<f/>$
  }

(1) The update query $u_4$

\[
\begin{align*}
\pi_{\text{no}} &= \{ \text{doc} \} \\
\pi_{\text{next}} &= \{ (a, b), (a, e) \}
\end{align*}
\]

(2) The extended projector $\pi_{\text{next}}$

(3) The XML document $t$

(4) The projection $t_1$ with $\pi_{\text{next}}$

(5) The partial result $t_2$

(6) Merging $t$ and $t_2$

Figure 5.6: Dealing with "replace" in isolation

(witness of a "replace") and position 1.2 preceeds position 1.3, Merge skips the tree rooted at $t@1.2$ and merges node $t@1.3$ and $t_2@1.3$ and outputs the element rooted at $t@1.3$. As the reader can see here, the node $t_2@1.3$ plays the role of separator enabling to separate nodes $t_2@1.1$, $t_2@1.2$ from $t_2@1.3$.

Case "delete" .................................................................

Example 5.2.5. See Figure 5.7

The update $u_5$ (see fig. 5.7-1) involves several update operations of the kind "delete". It is treated in a way similar to the "insert before" and "insert after" cases.

**projector extraction** - The type projector is enriched with a new component $\pi_{\text{del}}$ (see fig. 5.7-2). As for the previous cases, the projector component $\pi_{\text{del}}$ is specified by pairs of labels. Next, $\text{par}(\pi_{\text{del}})$ and $\text{ch}(\pi_{\text{del}})$ are defined as expected.

**projection** - The projection first outputs the node $t@\varepsilon$ labelled by $\text{doc} \in \pi_{\text{no}}$. After that it projects the node $t@1$, since it is labelled by $a \in \text{par}(\pi_{\text{del}})$, followed by the nodes $t@1.3$, $t@1.4$ labelled by $c \in \text{ch}(\pi_{\text{del}})$ and the node $t@1.5$ labelled by $e \in \text{ch}(\pi_{\text{del}})$ (see fig. 5.7-4). Finally the $a$-node of $t@2$ is projected. Note that, the remaining nodes of $t$ have been pruned out since none of them is labelled by types in $\pi_{\text{ext}}$. 

5.2. Extending the Type Projector for Update Optimization

(1) The update query $u_5$

$$\text{for } x \in /doc/a \
\text{return} \{ \
delete \text{node } x/c \
delete \text{node } x/e \}$$

(2) The extended projector $\pi_{\text{ext}}$ for $u_5$

$$\pi_{\text{no}} = \{ \text{doc}, \}$$
$$\pi_{\text{del}} = \{(a, c), (a, e)\}$$

(3) The XML document $t$

(4) The projection $t_1$ of $t$

(5) The partial update result $t_2$

(6) Merging $t$ and $t_2$

Figure 5.7: Dealing with "delete" in isolation
update - The result of the execution of the update \( u_5 \) is given in Figure 5.7-5. The nodes \( t_1@1.3, t_1@1.4 \) labelled by \( c \) and \( t_1@1.5 \) labelled by \( e \) have been deleted.

merge - Merge parses the two subtrees rooted at \( t@1 \) (see fig. 5.7-3) and \( t_2@1 \) (see fig. 5.7-5). Because the types of the nodes \( t@1.1 \) and \( t@1.2 \) are not in \( ch(\pi_{del}) \) Merge outputs the trees rooted at \( t@1.1 \) and \( t@1.2 \) (see fig. 5.7-6). After that, Merge skips the trees rooted at \( t@1.3, t@1.4 \) and \( t@1.5 \), since they are labelled by \( c \in ch(\pi_{del}) \) and \( e \in ch(\pi_{del}) \) and have been deleted from the tree \( t_2 \).

5.2.2 Case analysis: mixing update operations of different kinds

In this Section, we build on the previous case analysis, updates involving more than one kind of update operation. These cases, as it will be illustrated, need a special treatment, because they may imply position conflict during the Merge phase. The presentation is based on examples, as before. We consider mixing update operations two by two. Dealing with the general case (mixing more than two kinds of updates) is directly inferred from this analysis. In the presentation of the examples the type projectors are specified by giving their non-empty components.

Mixing insertion "as first" with insertion "as last" ........................

Let us consider the example illustrated by Figure 5.8. This example involves an update operation of the kind "insert as first" and an update operation of the kind "insert as last" (see fig. 5.8-1). We start by investigating a solution provided by the case analysis for update operation in isolation and show that such an approach does work here.

Example 5.2.6. See Figure 5.8

projector extraction - The type projector components \( \pi_{asfirst} \) and \( \pi_{aslast} \) for this update are given in Figure 5.8-2.

projection - The projection selects the root \( t@\$ \) labelled \( doc \), then the nodes \( t@1 \) and \( t@2 \) both labelled by \( a \in \pi_{asfirst} \) and \( a \in \pi_{aslast} \) (see fig. 5.8-4).

update - The result \( t_2 \) of the execution of the update \( u_6 \) on \( t_1 \) is given in Figure 5.8-5. The tree rooted at \( t_2@1 \) contains elements \( t_2@i1, t_2@i2 \) labelled by \( b \) (inserted "as first") and elements \( t_2@i3, t_2@i4 \) labelled by \( e \) (inserted "as last"). The tree rooted at \( t_2@2 \) contains as well the new elements rooted at \( t_2@i5, t_2@i6 \) (inserted "as first") and \( t_2@i7, t_2@i8 \) (inserted "as last").

merge - While merging the children of \( t@1 \) (see fig. 5.8-3) and \( t_2@1 \) (see fig. 5.8-5) since \( t@1 \) is labelled by \( a \in \pi_{asfirst} \) Merge gives the priority to outputting the new inserted elements having identifiers \( i1, i2, i3, i4 \) (see fig. 5.8-6). Here, the issue is that there is no information enabling Merge to separate \( t_2@i1, t_2@i2 \) from \( t_2@i3, t_2@i4 \). Therefore, the Merge process fails to produce the expected result (see fig. 5.8-6). □
for $\pi \in \mathcal{X}$
\[
\text{return } \{ \text{insert nodes } (<b/>, <b/>) \text{ as first into } \mathcal{x/a} \}
\]
\[
\text{as last into } \mathcal{x/a} \}
\]
(1) The update query $u_6$

$\pi_{\text{no}} = \{ \text{doc} \}$

$\pi_{\text{asfirst}} = \{ a \}$

$\pi_{\text{aslast}} = \{ a \}$

(2) The extended projector $\pi_{\text{ext}}$ for $u_6$

(3) The XML document $t$

(4) The projection $t_1$ of $t$ wrt $\pi_{\text{ext}}$

(5) The partial update $t_2$

(6) Attempt to merge $t$ and $t_2$

Figure 5.8: Mixing "insert as first" and "insert as last"

In order to solve the problem outlined in the previous example (separation of inserted "as first" elements and inserted "as last" elements), a new projector component $\pi_{\text{first}}$ is introduced. Intuitively, if the label of a node belongs to $\pi_{\text{first}}$, it is projected together with its first child. The projection of the first child is going to play the role of separator (when necessary) during the Merge phase.

**Example 5.2.7.** See Figure 5.9

- **Projector extraction** - The additional component $\pi_{\text{first}} = \{ a \}$ (see fig. 5.9-2) is obtained as the intersection of $\text{par}(\pi_{\text{asfirst}})$ and $\text{par}(\pi_{\text{aslast}})$. Note that, the type $a$ is removed from the component $\pi_{\text{asfirst}}$.

- **Projection** - This time the projection selects the nodes $t@1$ and $t@2$ together with the first child of each one: $t@1.1$ labelled by $d$ and $t@2.1$ labelled by $d$.

- **Update** - The trees rooted at $t@1$ contains "as first" inserted elements $t_2@i1$, $t_2@i2$ and "as last" inserted elements $t_2@i3$, $t_2@i4$. As the reader can observe these nodes are separated by $t@1.1$ and $t@2.1$ (see fig. 5.9-4).

- **Merge** - Because the node $t@1$ is labelled by $a\in\pi_{\text{asfirst}}$, while processing the nodes $t@1$ (see fig. 5.9-1) and $t_2@1$ (see fig. 5.9-4), Merge outputs $t_2@1$. While merging the children of $t@1$ and $t_2@1$, Merge first examines nodes $t@1.1$ and $t_2@1$. Because the node $t@1.1$ is the first child of $t@1$ and the node $t_2@i1$ is the result of "insert as first" operation Merge outputs $t_2@i1$. Next merging is processed on $t@1.1$ and $t_2@i2$ and, for the same reason, $t_2@i2$ is output. After that, Merge examines the nodes
The projection first selects the nodes that it projects the nodes \( t \). It is important to note that the node \( t \) is labelled by \( \epsilon \). After that, \( \text{Merge} \) outputs the tree rooted at \( t \). When \( t \) is empty \( \text{Merge} \) outputs the elements \( t_2 @ i_3 \) and \( t_2 @ i_4 \). It is important to note that the node \( t \) is labelled by \( \epsilon \). The \( \text{Merge} \) process continues in a similar manner for the trees rooted at \( t @ 2 \) and \( t_2 @ 2 \). □

**Mixing insertion "before" with insertion "after"**

Let us consider the example illustrated by Figure 5.10. This example involves an update operation of the kind "insert before" and an update operation of the kind "insert after" (see fig. 5.10-1). As for the previous case, we start by investigating a solution provided by the case analysis for update operation in isolation and show that such an approach does not work here.

**Example 5.2.8.** See Figure 5.10.

**projector extraction** - The projector extracted for the update \( u_7 \) is given in Figure 5.10-2.

**projection** - The projection first selects the nodes \( t @ \varepsilon \) labelled by \( \text{doc} \in \pi_{\text{no}} \). After that it projects the nodes \( t @ i + 1 \) labelled by \( a \in \text{par}(\pi_{\text{bef}}) \) followed by its children \( t @ i + 1.2 \) and \( t_2 @ i + 1.4 \) labelled by \( b \in \text{ch}(\pi_{\text{af}}) \) and \( c \in \text{ch}(\pi_{\text{bef}}) \) resp. (see fig. 5.10-4). Finally it projects node \( t @ 2 \), since it is labelled by \( a \in \text{par}(\pi_{\text{bef}}) \). The remaining nodes are pruned out, since none of them belongs to \( \pi_{\text{ext}} \).
5.2. Extending the Type Projector for Update Optimization

for $x$ in /doc/a
return
{
    insert nodes (<k/>, <k/>)
    after $x/b$
    insert nodes (<z/>, <z/>)
    before $x/c$
}

(1) The update query $u_7$

$$
\begin{align*}
\pi_{no} &= \{\text{doc}\} \\
\pi_{af} &= \{(a, b)\} \\
\pi_{bef} &= \{(a, c)\}
\end{align*}
$$

(2) The extended projector $\pi_{ext}$ for $u_7$

(3) The XML document $t$

(4) The projection $t_2$ of $t$

(5) The partial update result $t_2$

(6) Attempt to merge $t$ and $t_2$

Figure 5.10: Mixing "insert before" and "insert after"
update - The result $t_2$ of the execution of the update $u_7$ on $t_1$ is given in Figure 5.10-5. The tree rooted at $t_2@1$ contains the new elements $t_2@i_1$, $t_2@i_2$ labelled by $k$ (inserted "after") and the new elements $t_2@i_3$, $t_2@i_4$ labelled by $z$ (inserted "before").

merge - Merge processes as follows: first it examines the node $t@1$ and $t_2@1$ and because $t@1$ is labelled by $a \in \text{par}(\pi_{\text{bef}})$, the node $t_2@1$ is output (see fig.5.10-6). After that, Merge examines the nodes $t@1.1$ and $t_2@1.2$. Because $t@1.1$ is labelled by $a \notin \pi$, the tree rooted at $t@1.1$ is output. Next Merge continues with the nodes $t@1.2$ and $t_2@1.2$. The node $t@1.2$ is labelled by $b \in \text{ch}(\pi_{\text{af}})$, thus the node $t_2@1.2$ is output followed by the new inserted elements having identifiers $i_1,i_2,i_3,i_4$. Her once again, Merge is unable to separate the nodes $t_2@i_1$, $t_2@i_2$ (inserted "after") from the nodes $t_2@i_3$, $t_2@i_4$ (inserted "before"). Thus the Merge phase fails to produce the expected result. □.

In order to solve the problem outlined in the previous example (separation of inserted "after" elements and inserted "before" elements), we are going to make use of the projector component $\pi_{\text{next}}$ instead of $\pi_{\text{bef}}$ and $\pi_{\text{af}}$. Recall that, when a pair of types $(x,y)$ belongs to $\pi_{\text{next}}$, the nodes labelled by $y$ are projected together with their immediate sibling. It is the projection of the immediate sibling which is going to provide the separator between inserted elements.

Example 5.2.9. See Figure 5.11.

projector extraction - The contents of $\pi_{\text{next}}$ (see fig. 5.11-2) is deduced from $\text{par}(\pi_{\text{af}}) \cap \text{par}(\pi_{\text{bef}})$. Here, we have that $\text{par}(\pi_{\text{af}}) \cap \text{par}(\pi_{\text{bef}})=\{a\}$ which gives the information that at the children level of nodes labelled by $a$ there may be some conflict between "insert after" and "insert before" operations. Therefore $\pi_{\text{next}}$ contains both pairs $(a,b)$ and $(a,c)$. Indeed, the pairs $(a,b)$ and $(a,c)$ may be deleted from $\pi_{\text{bef}}$ and $\pi_{\text{af}}$ respectively.

projection - The projected tree $t_1$ (see fig. 5.11-3) contains not only nodes having types in $\pi_{\text{ext}}$, but contains node labelled by $e \notin \pi_{\text{ext}}$, which is the immediate sibling of the node $t@1.2$ labelled by $b \in \text{ch}(\pi_{\text{next}})$.

update - (see fig. 5.11-4) The tree rooted at $t_2@1$ besides the "after" inserted elements $t_2@i_1$, $t_2@i_2$ and the "before" inserted elements $t_2@i_3$, $t_2@i_4$, contains the separator node $t_2@1.3$.

merge - While merging the nodes $t@1$ (see fig. 5.11-1) and $t_2@1$ (see fig. 5.11-4), because $t@1$ is labelled by $a \in \text{par}(\pi_{\text{next}})$, Merge outputs $t_2@1$. Then, Merge examines the node $t@1.1$ and $t_2@1.2$ and since the first one is labelled by $d \notin \pi_{\text{next}}$, Merge outputs the tree rooted at it. The next pair of nodes being merged are $t@1.2$ labelled by $b \in \text{ch}(\pi_{\text{next}})$ and $t_2@1.2$. Merge selects the tree rooted at $t_2@1.2$. Merge continues processing the nodes $t@1.3$ and $t_2@i_1$. Because $t@1.3$ is labelled by $e \notin \text{ch}(\pi_{\text{next}})$ and $t_2@i_1$ is a new node, Merge outputs $t_2@i_1$. Next Merge examines the nodes $t@1.3$ and $t_2@i_2$, following the same reasoning, the node $t_2@i_1$ is output. Then Merge examines the nodes $t@1.3$ and $t_2@1.3$ and outputs the trees rooted at
5.2. Extending the Type Projector for Update Optimization

The XML document $t_\pi = \{ \text{doc} \}$

The extended projector $\pi$ for $u_\gamma$

The projection $t_1$ of $t$ with $\pi_{\text{next}}$

Figure 5.11: Mixing "insert before" and "insert after"

$t@1.3$. As the reader can observe, the node $t@1.3$ separates $t_2@i1$, $t_2@i2$ (inserted "after") from $t_2@i3$, $t_2@i4$ (inserted "before"). □

Mixing insertion "as first" with insertion "before" .................

The following example involves an update operation of the kind "insert as first" and an update operation of the kind "insert before" (see fig. 5.12-1). As for the previous case, we start by investigating a solution provided by the case analysis for update operation in isolation and show that such an approach does not work here.

Example 5.2.10. See Figure Figure 5.12.

Projector extraction - Following the solution given for the case analysis of update operations in isolation, the projector has components $\pi_{\text{asfirst}}$ and $\pi_{\text{bef}}$ (see fig. 5.12-2).

Projection - The projection first outputs the root node $t@e$ labelled by $\text{doc} \in \pi_{\text{no}}$. Then it projects $t@1$, since it is labelled by $a \in \pi_{\text{asfirst}}$. Note that $a \in \text{par}(\pi_{\text{bef}})$, thus the projection continues to parse the children of $t@1$. It selects the child $t@1.2$ since, $b \in \text{ch}(\pi_{\text{bef}})$ (see fig. 5.12-4). Finally it parses the subtree rooted at $t@2$, but prunes its children since their type do not belong to $\pi_{\text{ext}}$. 
for $x$ in /doc/a
    return {
        insert nodes (<e/>, </e>)
        as first into $x$
        insert nodes (<k/>, <k/>)
        before $x/b$
    }

(1) The update query $u_9$

\[
\pi_{no} = \{ \text{doc} \} \\
\pi_{asfirst} = \{ a \} \\
\pi_{bef} = \{ (a, b) \}
\]

(2) The extended projector $\pi$ for $u_8$

(3) The XML document $t$

(4) The projection $t_1$ of $t$

(5) The partial update result $t_2$

(6) Attempt to merge $t$ and $t_2$

Figure 5.12: Mixing "insert as first" and "insert before"
5.2. Extending the Type Projector for Update Optimization

update - The result of the execution of the update \( u_8 \) is given in Figure 5.12-5. The tree rooted at \( t_2@1 \) contains the "as first" inserted elements \( t_2@i1, t_2@i2 \) labelled by \( e \) and the "before" inserted elements \( t_2@i3, t_2@i4 \) labelled by \( k \).

merge - While merging children of \( t@1 \) (see fig. 5.12-3) and \( t_2@1 \) (see fig. 5.12-5) since \( t@1 \) is labelled by \( a \in \pi_{asfirst} \), \( Merge \) gives the priority to output the new inserted elements having identifiers 1, 2, 3, 4 (see fig. 5.12-6). The issue is that there is no information enabling to separate \( t_2@i1, t_2@i2 \) ("as first" inserted elements) from ("before" inserted elements) \( t_2@i3, t_2@i4 \). Therefore, this merging process fails to produce the expected result. □

In order to solve the problem outlined by the previous example (separation of inserted "as first" elements from inserted "before" elements), we use the projector component \( \pi_{first} \) in order to generate separators. As already explained, if a node belongs to \( \pi_{first} \), the node is projected together with its first child.

Example 5.2.11. See Figure 5.13.

projector extraction - The intersection of \( par(\pi_{asfirst}) \) and \( par(\pi_{bef}) \) (see fig. 5.13-1) leads to the specification \( \pi_{first}=\{a\} \). Note that, the type \( a \) is removed from the component \( \pi_{asfirst} \).

projection - Note that this time the projection not only outputs the node \( t@1.2 \) labelled by \( b \in ch(\pi_{bef}) \), but also the first child of \( t@1 \) the node \( t@1.1 \) and the first child of \( t@2 \) labelled \( d \) (see fig. 5.13-3).

update - The tree rooted at \( t_2@1 \) besides the "as first" inserted elements \( t_2@i1, t_2@i2 \) and the "before" inserted elements \( t_2@i3, t_2@i4 \), contains the separator node \( t_2@i1.1 \) (see fig. 5.13-4).

merge - Because the node \( t@1 \) is labelled by \( a \in \pi_{first} \), while processing the nodes \( t@1 \) (see fig. 5.13-1) and \( t_2@1 \) (see fig. 5.13-4), \( Merge \) outputs \( t@1.1 \). While merging the children of \( t@1 \) and \( t_2@1 \), \( Merge \) first examines nodes \( t@1.1 \) and \( t_2@i1 \). Because the node \( t@1.1 \) is the first child of \( t@1 \) and the node \( t_2@i1 \) is the result of "insert as first" operation, \( Merge \) outputs \( t@1.1 \). Next \( Merge \) considers the nodes \( t@1.1 \) and \( t_2@i2 \) and for the same reason \( t@2@i2 \) is output. Then, \( Merge \) examines the nodes \( t@1.1 \) and \( t_2@i1.1 \) and because \( t@1.1 \) is labelled by \( d \in ch(\pi_{ext}) \) \( Merge \) outputs \( t@1.1 \). The next nodes being merged are \( t@1.2 \) and \( t_2@i3 \). Because \( t@1.2 \) is labelled by \( b \in ch(\pi_{bef}) \) \( Merge \) outputs the "inserted before" node \( t_2@i3 \). The same is true when merging \( t@1.2 \) and \( t_2@i4 \). Finally, \( Merge \) examines the nodes \( t@1.2 \) and \( t_2@i2 \) and because \( t@1.2 \) is labelled by \( b \in ch(\pi_{bef}) \) outputs the node \( t_2@1.3 \). Therefore the first child \( t@1.1 \) of \( t@1 \) separates the nodes \( t_2@i1, t_2@i2 \) from \( t_2@i3, t_2@i4 \). □

Mixing insertion "as first" with insertion "after" ........................

The next example involves an update generating operations of the kind "insert as first" and operations of the kind "insert after". As opposed, to the previous cases, mixing these two kinds of operation can be dealt with using the solution provided by the analysis of update operations in isolation.
(1) The XML document \( t \)

\[ \pi_{\text{no}} = \{ \text{doc} \} \]
\[ \pi_{\text{bef}} = \{ (a, b) \} \]
\[ \pi_{\text{first}} = \{ a \} \]

(2) The extended projector \( \pi \) for \( u_g \)

(3) The projection \( t_1 \) of \( t \) with \( \pi_{\text{first}} \)

(4) The Partial Update result \( t_2 \)

(5) Merging \( t \) and \( t_2 \)

Figure 5.13: Mixing "insert as first" and "insert before"
The projection first outputs the root node \( \pi \cdot 1 \). Then it projects \( \pi \cdot 2 \), since it is labelled by \( a \in \pi \cdot 1 \). Note that \( a \in \pi \cdot 2 \), thus the projection continues to parse the children of \( \pi \cdot 1 \). It projects the child \( \pi \cdot 2 \), since \( b \in \pi \cdot 2 \) (see fig. 5.14-4). Finally it parses the subtree rooted at \( \pi \cdot 2 \), but prunes out its children, because their types do not belong to \( \pi \cdot 2 \).

**merge** - Merge processes as follows (see fig. 5.14-5): first it examines the node \( \pi \cdot 1 \) and \( \pi \cdot 2 \) and because \( \pi \cdot 1 \) is labelled by \( a \in \pi \cdot 1 \). Merge outputs \( \pi \cdot 1 \) followed by the nodes \( \pi \cdot 2 \) and \( \pi \cdot 3 \) (inserted "as first"). Then, Merge outputs the tree rooted by \( \pi \cdot 1.1 \), since it is labelled by \( d \in \pi \cdot 1.2 \). The nodes \( \pi \cdot 1.2 \) and \( \pi \cdot 2 \) are merged as follows: because \( \pi \cdot 1.2 \) is labelled by \( b \in \pi \cdot 2 \), Merge outputs the node \( \pi \cdot 1.2 \). Finally, the nodes \( \pi \cdot 3 \) and \( \pi \cdot 4 \) (inserted "as after") are output. As
the reader can see, the final result is correct. There is no need to use a separator
mechanism here as, intuitively, the projector component $\pi_{af}$ is sufficient to separate
the nodes $t_2@i1$, $t_2@i2$ from $t_2@i3$, $t_2@i4$. □

Mixing insertion "as last" with insertion "after" .........................

This case is the dual of the one dealing with mixing insertion "as first" with insertion
"before". Thus, in order to ensure the correctness of the Merge process, we
introduce a new projector component $\pi_{last}$ (the dual of the projector component
$\pi_{first}$). Intuitively, if a node type belongs to $\pi_{last}$, the node is projected together
with its last child.

Example 5.2.13. See Figure 5.15.

projector extraction - The projector is first generated as for the update operations
in isolation. This leads to the following components: $\pi_{no} = \{doc\}$, $\pi_{af} = \{(a, d)\}$ and
$\pi_{aslast} = \{a\}$. Then, noticing that $\pi_{aslast} \cap \text{par}(\pi_{af}) = \{a\}$, the new component $\pi_{last}$
is set to $\{a\}$ and $a$ is removed from $\pi_{aslast}$ (see Fig. 5.15-2).

projection - The projection outputs not only the node $t@1.1$, since it is labelled by
d$\in \text{ch}(\pi_{af})$, but also the last child of $t@1$, that is the node $t@1.2$ labelled $b$ (see fig.
5.15-3).

update - The result of the execution of the update $u_{11}$ on the projected document
t$_1$ is given in Figure 5.15-5. The tree rooted at $t_2@1$ contains the elements $t_2@i1$, $t_2@i2$ labelled by $k$ (inserted "after") and the elements $t_2@i3$, $t_2@i4$ labelled by $c$
(inserted "as last").

merge - Because the node $t@1$ is labelled by $a \in \text{par}(\pi_{af})$, while parsing the nodes
t@1 and $t_2@1$, Merge outputs $t_2@1$ (see fig. 5.15-6). Then, Merge examines the
nodes $t@1.1$ and $t_2@1.1$. Because $t@1.1$ is labelled by $d \in \text{ch}(\pi_{af})$, the node $t_2@1.1$
is output. The next nodes parsed are $t@1.2$ and $t_2@1$ and Merge gives priority to
outputting $t_2@i1$ since it is a new element. While processing $t@1.2$ and $t_2@i2$, for
the same reason Merge outputs $t_2@i2$. The next step processes the nodes $t@1.2$ and
$t_2@1.2$. Because $t@1.2$ is the last child of $t@1$ and because it is labelled by $b \notin \pi_{ext}$,
Merge outputs the node $t@1.2$. Finally, the nodes $t_2@i3$ and $t_2@i4$ (inserted "as last") are output. Thus the expected result of the update $u_{11}$ is obtained. The main
point here is the projection of the node $t@1.2$ to separate the nodes $t_2@i1$ and $t_2@i2$
(inserted "after") from the nodes $t_2@i3$ and $t_2@i4$ (inserted "as last"). □

Mixing insertion "as last" with insertion "before" .........................

This case is dual to mixing "insert as first" and "insert after". It does not require
any additional mechanism for the projector.

Mixing "replace" with insertion "before" ..........................

As it has been discussed in the subsection analyzing the update operation in iso-
lation, when the update expression involves some "replace" operation, the type
for $x$ in /doc/a
return
{
insert nodes (<c/>, <c/>)
as last into $x$
insert nodes (<k/>, <k/>)
after $x/d$
}

(1) The update query $u_{11}$

\[\pi_{\text{no}}=\{\text{doc}\}\]
\[\pi_{\text{af}}=\{(a, d)\}\]
\[\pi_{\text{last}}=\{a\}\]

(2) The extended projector $\pi$ for $u_{11}$

(3) The XML document $t$

(4) The projection $t_1$ of $t$ with $\pi_{\text{last}}$

(5) The partial update result $t_2$

(6) Merging $t$ and $t_2$

Figure 5.15: Mixing "insert as last" and "insert after"
for $x$ in /doc/a
return {
  replace node $x$/d with (<k/>,<k/>)
insert node (<z/> before $x$/e
}

(1) The update query $u_{12}$

$$\pi_{\text{no}} = \{\text{doc}\}$$
$$\pi_{\text{next}} = \{(a, d), (a, e)\}$$

(2) The extended projector $\pi_{\text{ext}}$ with $\pi_{\text{next}}$

(3) The XML document $t$

(4) The projection $t_1$

(5) The partial result $t_2$

(6) Merging $t$ and $t_2$

Figure 5.16: Mixing "replace" and "insert before"

projector requires the use of the (separator) projector $\pi_{\text{next}}$. When mixing "replace" operations with other kinds of operation, the use of the component $\pi_{\text{next}}$ is going to be sufficient to ensure the correct behavior of the Merge process most of the time (Mixing "replace" and "insert as first" is going to require an additional mechanism).

Example 5.2.14. See Figure 5.16.

**projector extractor** - A first analysis of the update expression leads to derive $\pi_{\text{no}} = \{\text{doc}\}$, $\pi_{\text{bef}} = \{a, e\}$ and $\pi_{\text{next}} = \{a, d\}$. Then, because $\text{par}(\pi_{\text{bef}}) \cap \text{par}(\pi_{\text{next}}) = \{a\}$, the pair of types $(a, e)$ is removed from $\pi_{\text{bef}}$ and added to $\pi_{\text{next}}$. Thus finally, $\pi_{\text{next}} = \{(a, d), (a, e)\}$.

**projection** - The projection outputs nodes $t@ε$, $t@1$ together with its children $t@1.1$ labelled by $d\in \text{ch}(\pi_{\text{next}})$, $t@1.2$, because it is the next sibling of $t@1.1$ and $t@1.5$ labelled by $e\in \text{ch}(\pi_{\text{next}})$. It projects as well nodes $t@2$ and $t@2.1$ (see fig.5.16-4).

**update** - The result of the execution of the update $u_{12}$ on the projected document $t_1$ is given in Figure 5.16-5. The tree rooted at $t_2@1$ contains the elements $t_2@i1$, $t_2@i2$ labelled by $k$ (inserted "in place of") and the elements $t_2@i3$ labelled by $z$ (
5.2. Extending the Type Projector for Update Optimization

merge - While merging the nodes \( t@1 \) (see fig.5.16-3) and \( t_2@1 \) (see fig.5.16-5), because \( t@1 \) is labelled by \( a\in par(\pi_{next}) \). Merge outputs \( t@1 \) (see fig.5.16-6). Then, Merge examines the node \( t@1.1 \) and \( t_2@i_1 \). Because the first one is labelled by \( d\in ch(\pi_{next}) \) and the identifier \( i_1 \) of the second one indicates that it is a new node, Merge outputs \( t_2@i_1 \). The next pair of the nodes being merged are \( t@1.1 \) and \( t_2@i_2 \). For the same reason, Merge outputs \( t_2@i_2 \). After that Merge examines the nodes \( t@1.1 \) and \( t_2@1.2 \) and Merge skips the tree rooted at \( t@1.1 \) (it has been replaced) and examines the nodes \( t@1.2 \) and \( t_2@1.2 \). While merging the nodes \( t@1.2 \) and \( t_2@1.2 \), Merge outputs \( t_2@1.2 \). Next, Merge processes the nodes \( t@1.3 \) and \( t_2@i_3 \) labelled by \( z \) and because the first one is labelled by \( e\notin\pi_{ext} \) Merge outputs \( t@1.3 \). The same is true for the pairs \( t@1.4 \) and \( t_2@i_3 \). The next two nodes to be merged are \( t@1.5 \) and \( t_2@i_3 \). Because \( t@1.5 \) is labelled by \( e\in ch(\pi_{next}) \) Merge outputs \( t_2@i_3 \). Finally, Merge process the nodes \( t@1.5 \) and \( t_2@1.5 \) and outputs \( t_2@i_5 \). □

Mixing "replace" with insertion "after" ............................................

This case is of course similar to the previous one and the use of the projector component \( \pi_{next} \) is developed in the same manner.

Mixing "replace" with insertion "as first" ............................................

Dealing with this case requires more than the projector component \( \pi_{next} \) as the following example shows.

Example 5.2.15. See Figure 5.17.

projector extraction - Let us apply the same technique as for the analysis for the update operation in isolation. The projector generated contains three components \( \pi_{no}=\{doc\} \), \( \pi_{asfirst}=\{a\} \) and \( \pi_{next}=\{(a,b)\} \) (see fig. 5.17-2).

projection - The projection first selects the node \( t@e \), followed by \( t@1 \) labelled by \( a\in\pi_{asfirst} \). The child \( t@1.2 \) of \( t@1 \) is projected since it is labelled by \( b\in ch(\pi_{next}) \). Finally, the node \( t@2 \) labelled by \( a\in\pi_{asfirst} \) is projected (see fig. 5.17-4).

update - The result of the execution of the update \( u_{13} \) on the projected document \( t_1 \) is given in Figure 5.17-5. The tree rooted at \( t_2@i_1 \) contains the elements \( t_2@i_1 \), \( t_2@i_2 \) labelled by \( d \) (inserted "as first") and the element \( t_2@i_3 \) labelled by \( z \) (inserted "in place of").

merge - Merge processes as follows (see fig.5.17-6): first it examines the node \( t@1 \) and \( t_2@1 \) and because \( t@1 \) is labelled by \( a\in\pi_{asfirst} \) it outputs \( t@1 \). Then, Merge outputs the nodes \( t_2@i_1 \), \( t_2@i_2 \) (inserted "as first") and the node \( t_2@i_3 \) (inserted "in place of"). Once again, the issue here is that there is no information enabling to separate the \( t_2@i_1 \), \( t_2@i_2 \) (inserted "as first") from the node \( t_2@i_3 \) (inserted "in place of"). Thus, Merge fails to produce the expected result. □
for $x$ in /doc/a
return
{
  insert nodes (<d/>,<d/>) as first into $x$
  replace node $x/b$ with (<k/>)
}

(1) The update query $u_{13}$

\[
\begin{align*}
\pi_{\text{no}} &= \{ \text{doc} \} \\
\pi_{\text{asfirst}} &= \{ \text{a} \} \\
\pi_{\text{next}} &= \{ \langle a, b \rangle \}
\end{align*}
\]

(2) The extended projector $\pi_{\text{ext}}$

(3) The XML document $t$

(4) The projection $t_1$

(5) The partial result $t_2$

(6) Attempt to merge $t$ and $t_2$

Figure 5.17: Mixing "replace" and "insert as first"
5.2. Extending the Type Projector for Update Optimization

To the problem outlined in the previous example, the solution is the same as for mixing “insert as first” and “insert before” and uses the separator projector \( \pi_{\text{first}} \).

Example 5.2.16. See Figure 5.18

projection extraction - The value of \( \pi_{\text{first}} \) is deduced from the intersection 
\( \pi_{\text{as first}} \cap \text{par}(\pi_{\text{next}}) = \{a\} \). Therefore, \( \pi_{\text{first}} = \{a\} \).

projection - This time, the projection of the initial document \( t \) includes the first children \( t \cdot 1.1 \) and \( t \cdot 2.1 \) of the nodes \( t \cdot 1 \) and \( t \cdot 2 \) respectively (see fig. 5.18-3).

update - The tree rooted at \( t \cdot 1 \) contains the new elements \( t_2 \cdot 1.1 \), \( t_2 \cdot 1.2 \) labelled by \( d \) (inserted "as first"), the node \( t_2 \cdot 1.1 \) and the new element \( t_2 \cdot 2.1 \) labelled by \( k \) (inserted "in place of").

merge - Because the node \( t \cdot 1 \) is labelled by \( a \in \text{par}(\pi_{\text{next}}) \), while processing the nodes \( t \cdot 1 \) (see fig. 5.18-1) and \( t_2 \cdot 1 \) (see fig. 5.18-4), \textbf{Merge} outputs \( t_2 \cdot 1 \) (see fig. 5.18-5). While parsing the children of \( t \cdot 1 \) and \( t_2 \cdot 1 \), \textbf{Merge} first examines the nodes \( t \cdot 1.1 \) and \( t_2 \cdot 1.1 \). Because the node \( t \cdot 1.1 \) is the first child of \( t \cdot 1 \) and the node \( t_2 \cdot 1.1 \) is the result of the "insert as first" operation, \textbf{Merge} outputs \( t_2 \cdot 1.1 \). For the same reason, the next node output is \( t_2 \cdot 1.2 \). Then, \textbf{Merge} examines the nodes \( t \cdot 1.1 \) and \( t_2 \cdot 2.1 \) and because \( t \cdot 1.1 \) is labelled by \( d \notin \pi_{\text{ext}} \), \textbf{Merge} outputs \( t \cdot 1.1 \). The next

Figure 5.18: Mixing "replace" and "insert as first"
nodes to be merged are $t@1.2$ and $t_2@i3$. Because $t@1.2$ is labelled by $b \in ch(\pi_{\text{next}})$, Merge outputs the new node $t_2@i3$ and skips $t@1.2$ which has been replaced. The result of the Merge phase is the expected result. □

Mixing "replace" with insertion "as last" .................................

This case is similar to mixing "replace" with insertion "before" and is treated by using the projector component $\pi_{\text{next}}$ as the separator.

Mixing "delete" with other kinds of update operation  .........................

This case is somehow slightly more intricate and we decided to solve it by using the projector component $\pi_{\text{delb}}$ introduced for the three-level projector. The following example shows the problem encountered by following the approach developed until now.

Example 5.2.17. See Figure 5.19.

projection extraction - For the update expression $u_{14}$ given in Figure 5.19-1, the first projector generated is composed of the non empty components $\pi_{\text{no}}=\{doc\}$, $\pi_{\text{af}}=\{(a,d)\}$, $\pi_{\text{bef}}=\{(a,e)\}$ and $\pi_{\text{del}}=\{(a,b)\}$. Based on the observation that $\text{par}(\pi_{\text{af}}) \cap \text{par}(\pi_{\text{bef}})=\{a\}$, the projector is modified to $\pi_{\text{no}}=\{doc\}$, $\pi_{\text{next}}=\{(a,d),(a,e)\}$ and $\pi_{\text{del}}=\{(a,b)\}$.

projection - (See Fig. 5.19-4) The projection outputs the nodes $t@\varepsilon$ labelled by $doc \in \pi_{\text{no}}$, $t@1$ labelled by $a \in \text{par}(\pi_{\text{next}})$ (note also that $a \in \text{par}(\pi_{\text{del}})$) and the children of $t@1$: $t@1.1$ labelled by $d \in ch(\pi_{\text{next}})$, $t@1.2$ labelled by $b \in ch(\pi_{\text{del}})$ and $t@1.4$ labelled by $e \in ch(\pi_{\text{next}})$. Indeed, the node $t@1.2$ is projected not only because its label belong to $ch(\pi_{\text{del}})$ but also because it is the next sibling of $t@1.1$ and the label of $t@1.1$ belongs to $ch(\pi_{\text{next}})$.

update - The partially updated tree $t_2$ (see fig. 5.19-5) is the result of applying the update $u_{14}$ on the projected document $t_1$. Mainly, the nodes $t_2@i1$, $t_2@i2$ have been inserted after the node $t_2@i1$, the node $t@1.2$ has been deleted and, finally, the nodes $t_2@i3$ and $t_2@i4$ have been inserted before the node $t_2@i4$.

The reader should pay attention to the fact that the node $t@1.2$ which has been projected with the intention to separate the elements inserted "after" from those inserted "before", has bee deleted.

merge - While merging the children of $t@1$ and $t_2@1$, Merge first processes the nodes $t@1.1$ and $t_2@1.1$. Because $t@1.1$ is labelled by $d \in ch(\pi_{\text{next}})$, Merge outputs $t_2@1.1$. Then, the nodes $t@1.2$ and $t_2@i1$ are parsed. Because $t@1.2$ labelled by $b \in ch(\pi_{\text{del}})$, Merge outputs $t_2@i1$. Next, Merge examines the nodes $t@1.2$ and $t_2@i2$ and outputs $t_2@i2$, for the same reason. Finally, while examining the nodes $t@1.2$ and $t_2@i3$, Merge outputs $t_2@i3$ and thus fails to produce expected result. The issue is that the separator $t@1.2$ has been deleted. □.
5.2. Extending the Type Projector for Update Optimization

for $x$ in /doc/a 
return 
{ 
insert nodes (<z/>, <z/>)
after $x/d$
insert nodes (<h/>, <h/>)
before $x/e$
delete node $x/b$
}

(1) The update query $u_{14}$

$\pi_{\text{no}} = \{doc\}$
$\pi_{\text{del}} = \{(a, b)\}$
$\pi_{\text{next}} = \{(a, d), (a, e)\}$

(2) The extended projector $\pi_{\text{ext}}$ for $u_{\text{ext}}$

(3) The XML document $t$

(4) The projection $t_1$ of $t$

(5) The partial update result $t_2$

(6) Attempt to merge $t$ and $t_2$

Figure 5.19: Mixing "delete" with other kinds
As explained before the example, we choose to solve this case by going back to the three-level projector and by using the $\pi_{\text{olb}}$ component. The precise solution is given in table 5.20.

### 5.3 Definition of the Extended Projection

This section is devoted to the formal presentation of the extended projector together with its semantic. We do not provide a formal presentation of the extraction of the extended projector from an update expression $u$ and a DTD $D$. We expect that the examples provided in the case analysis are sufficiently clear.

An extended projector is going to be specified by a bunch of projector components $\pi_\alpha$ where some of these components are sets of types ($\alpha \in \text{un}$) and the others components ($\alpha \in \text{bin}$) are sets of pairs of types. More precisely:

- $\text{bin} = \{\text{af}, \text{bef}, \text{del}, \text{next}\}$ and for $\alpha \in \text{bin}$, $\pi_\alpha \subseteq \Sigma \times \Sigma$
- $\text{un} = \{\text{no}, \text{olb}, \text{eb}, \text{aslast}, \text{asfirst}, \text{first}, \text{last}\}$, and for $\alpha \in \text{un}$ $\pi_\alpha \subseteq \Sigma$.

The following notations are used in the rest of the presentation:

- For $\alpha \in \text{bin}$, $x \in \text{par}(\pi_\alpha)$ iff $(x, y) \in \pi_\alpha$ for some $y$.
- For $\alpha \in \text{bin}$ and for $x \in \Sigma$, $\pi_\alpha(x) = \{y | (x, y) \in \pi_\alpha\}$
- For $\alpha \in \text{bin}$, $y \in \text{ch}(\pi_\alpha)$ iff $(x, y) \in \pi_\alpha$ for some $x$.
- $\pi^* = \bigcup_{\alpha \in \text{un}} \pi_\alpha \bigcup_{\alpha \in \text{bin}} (\text{par}(\pi_\alpha) \cup \text{ch}(\pi_\alpha))$, and
- $\pi_{\text{seed}} = \bigcup_{\alpha \in \text{un} - \{\text{first}, \text{last}\}} \pi_\alpha \bigcup_{\alpha \in \text{bin}} \text{par}(\pi_\alpha)$.

The definition below uses Table 5.20 to specify constraints. Let us explain how to read this table which indeed corresponds to the analysis of mixing update operations of different kinds. For instance, the intersection of the row $\in \pi_{\text{af}}$ with the column $\pi_{\text{bef}}$ corresponds to the constraint:

if $(x, y) \in \pi_{\text{af}}$ and $(x, z) \in \pi_{\text{bef}}$ then $(x, y)$ and $(x, z)$ should belong to $\pi_{\text{next}}$.

Note that the intersection of the row $\pi_{\text{af}}$ with the column $\pi_{\text{asfirst}}$ is marked with $\_\_$ which means that this case raises no additional element in the projector components.

**Définition 3 (Extended Type Projector).** Given a DTD $(D, s_D)$ over the alphabet $\Sigma$, an extended type projector $\pi$ is defined by $\pi = \bigcup_{\alpha \in \text{bin} \cup \text{un}} \pi_\alpha$ such that:

- the constraints summarized in Table 5.20 are satisfied (see below how to read the constraints from the Table), and
- for each $b \in \pi^*$ there exists $a \in \pi_{\text{seed}}$ such that $D(a) = r$ and $b$ occurs in $r$.

The second condition of Definition 3 expresses, like for the three level projector, a closure property wrt to the DTD $D$: a projected type $b$ cannot be deconnected from the root label $s_D$ although it does not need to be connected in all possible manners. Notice that this closure property requires that the producer type of $b$ be in $\pi_{\text{seed}}$. Notice here that we do not require disjointess of pairs of projector components.
5.3. Definition of the Extended Projection

The behavior of the extended projector is given below. This definition is written in a declarative style and does not provide a direct manner to implement the extended projector.

Définition 4 (Extended Type Projector Semantic).
Let \( \pi \) be a type projector for \((D, s_D)\), and \( t \in D \) be a tree with \( \text{roots}(t) = \{\ell_t\} \) and \( F = \text{subfor}(t) \). The \( \pi \)-projection of \( t \), denoted \( \pi(t) \), is the tree \( \Pi_{K(t, \pi)}(t) \) where \( K(t, \pi) \) is recursively defined by:

- if \( \text{lab}(\ell_t) \notin \pi \) then \( K(t, \pi) = \emptyset \) otherwise
- \( K(t, \pi) = \{\ell_t\} \cup_{a \in \pi_{\text{seed}}} K_\alpha(\text{lab}(\ell_t), F) \) where \( K_\alpha(a, F) \) is defined below for a label \( a \) and a forest \( F \).

if \( F = () \) then \( K_\alpha(a, F) = \emptyset \) otherwise let us assume that \( F = t' \circ F' \)

A. if \( \alpha \) is \text{no} or \text{as first} or \text{as last} and \( a \in \pi_\alpha \) then
   A.1 \( K_\alpha(a, F) = K(t', \pi) \cup K_\alpha(a, F') \) if \( \text{lab}(\ell_{t'}) \in \pi_{\text{seed}} \)
   A.2 \( K_\alpha(a, F) = K_\alpha(a, F') \) otherwise

B. if \( \alpha \) is \text{olb} and \( a \in \pi_{\text{olb}} \) then
   B.1 \( K_{\text{olb}}(a, F) = K(t', \pi) \cup K_{\text{olb}}(a, F') \), if \( \text{lab}(\ell_{t'}) \in \pi_{\text{seed}} \)
   B.2 \( K_{\text{olb}}(a, F) = \{\ell_{t'}\} \cup K_{\text{olb}}(a, F') \), otherwise

C. if \( \alpha \) is \text{eb} and \( a \in \pi_{\text{eb}} \) then \( K_{\text{eb}}(a, F) = \text{dom}(F) \)

D. if \( \alpha \) is \text{bef} or \text{after} or \text{del} and \( a \in \pi_{\text{par}}(\pi_\alpha) \) then
   D.1 \( K_\alpha(a, F) = K(t', \pi) \cup K_\alpha(a, F') \) if \( \text{lab}(\ell_{t'}) \in \pi_{\text{seed}} \)
   D.2 \( K_\alpha(a, F) = \{\ell_{t'}\} \cup K_\alpha(a, F') \), if \( (a, \text{lab}(\ell_{t'})) \in \pi_\alpha \)
   D.3 \( K_\alpha(a, F) = K_\alpha(a, F') \) otherwise

E. if \( \alpha \) is \text{next} and \( a \in \pi_{\text{par}}(\pi_{\text{next}}) \) then
   E.1 \( K_{\text{next}}(a, F) = K(t', \pi) \cup K_{\text{next}}(a, F') \) if \( \text{lab}(\ell_{t'}) \in \pi_{\text{seed}} \)
   E.2 \( K_{\text{next}}(a, F) = \{\ell_{t'}\} \cup K_{\text{next}}(a, F') \cup \text{Next}(F') \), if \( (a, \text{lab}(\ell_{t'})) \in \pi_{\text{next}} \) where \( \text{Next}(F) = K(t', \pi) \) if \( \text{lab}(\ell_{t'}) \in \pi_{\text{seed}} \) and \( \text{Next}(F) = \{\ell_{t'}\} \) otherwise.
   E.3 \( K_{\text{next}}(a, F) = K_{\text{next}}(a, F') \), otherwise

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
(x, y) & \pi_{\text{af}} & \pi_{\text{bef}} & \pi_{\text{del}} & \pi_{\text{next}} & \pi_{\text{as first}} & \pi_{\text{as last}} \\
\hline
(x, y) & (x, y) \in \pi_{\text{next}} & (x, y) \in \pi_{\text{next}} & (x, y) \in \pi_{\text{next}} & (x, y) \in \pi_{\text{next}} & (x, y) \in \pi_{\text{next}} & (x, y) \in \pi_{\text{next}} \\
\hline
(x, z) & x \in \pi_{\text{olb}} & x \in \pi_{\text{olb}} & x \in \pi_{\text{olb}} & x \in \pi_{\text{olb}} & x \in \pi_{\text{olb}} & x \in \pi_{\text{olb}} \\
\hline
\hline
\text{Figure 5.20: Constraints for the extended projector}
\end{array}
\]
F. if $\alpha$ is \textit{first} and $a \in \pi_{\text{first}}$ then 
$K_{\text{first}}(a, F) = K(t', \pi)$, if $\text{lab}(r_{t'}) \in \pi_{\text{seed}}$

G. if $\alpha$ is \textit{last} and $a \in \pi_{\text{last}}$ then (recall that $F = t' \circ F'$ and $F \neq ()$)

$K_{\text{last}}(a, F) = K_{\text{last}}(a, F')$ if $F' \neq ()$ otherwise - that is if $F' = ()$.

$K_{\text{last}}(a, F) = \{ r_{t'} \}$

It is important, when reading this definition to pay attention to the fact that one type $b$ may belong to several components. In such a case, several sub-items may apply and should be applied. For instance, assume that $(a, b) \in \pi_{\text{af}}$ and moreover that $b \in \pi_{\text{olb}}$. Then, at some point when using item D, because $a \in \text{par}(\pi_{\text{af}})$, the item D.1 and D.2 will be applied because $b \in \pi_{\text{olb}}$ and thus $b \in \pi_{\text{seed}}$ and because $b \in \text{ch}(\pi_{\text{af}})$.

5.3.1 Merge

This section formalizes the changes of the Merge phase to support the extension of the projection technique (see Sections 5.2.1 and 5.2.2). We proceed as for the presentation of the revised projection: a case analysis is developed. Let us first recall the global structure of the Merge process and the main elements that are useful for defining it.

We assume that:

1. The input XML document $t$ is valid with respect to the DTD $D$. For the purpose of the formal presentation, we assume that the tree $t$ is a p-store: the identifiers are the node positions (in document order).

2. The extended projector $\pi$ has been derived from the update $u$.

3. The document $t'$ is the partial update $u(\pi(t))$. We assume that the execution of the update $u$ has produced new identifiers for the purpose of node creation induced by replace and insert operations.

The goal of merging the input document $t$ and the partial update $t'$ is to construct the update $u(t)$. Merging processes by parsing both trees $t$ and $t'$. The merge algorithm is decomposed as follows:

- The procedure $\text{TreeMerge}$ takes as input two subtrees $\tau_i$ and $\tau'$. The first one ($\tau_i$) is a subtree of $t$. The second one ($\tau'$) is a subtree of $t'$ (see Figure 5.21). Next, we assume that:

  $\text{roots}(\tau_i) = n$ \hspace{1cm} $\text{roots}(\tau') = m$
  $\text{lab}(\text{roots}(\tau_i)) = a_i$ \hspace{1cm} $\text{lab}(\text{roots}(\tau')) = b_i$
  $\text{subfor}(\tau_i) = F_i$ \hspace{1cm} $\text{subfor}(\tau') = F_u$

The procedure takes care of the synchronized parsing of the trees $\tau_i$ and $\tau'$. The fact that the trees $\tau_i$ and $\tau'$ are synchronized implies that we assume that the trees $\tau_i$ and $\tau'$ have identical root identifier, that is $\text{roots}(\tau_i) = n = \text{roots}(\tau') = m$. 
Their roots may have different labels if the update $u$ has renamed the label of
the node $n$. The procedure $TreeMerge$ is quite simple: it builds a tree whose
root is $\tau'$ root; it checks the label $a_i$ of $n$ with respect to $\pi$ in order to decide
how to merge the sub-forests $F_i$ and $F_u$. A first version of the procedure
$TreeMerge$ is formally presented in the next section.

- A bunch of procedures $xxMerge$ are then defined to take care of merging the
two forests $F_i$ and $F_u$, whose parent nodes $n$ and $m$ are synchronized. The
specific procedure used to merge $F_i$ and $F_u$ is determined by $TreeMerge$ and
depends on the label $a_i$ of the parent node $n$ of $F_i$. For instance, if $a_i \in \pi_{nb}$,
then merging $F_i$ and $F_u$ is done by calling $NoMerge$.

Each procedure takes advantage of the information obtained by identifying in
which projector component $a_i$ belongs to.

Next, we will always assume that the forest $F_i$ is of the form $t_i \circ f_i$ and the
forest $F_u$ of the form $t_u \circ f_u$, when they are not empty (see fig. 5.21).

The case analysis starts by examining simple cases where the label $a_i$ of the
parent node of $F_i$ belongs to only one component of the projector. We will then
examine the general case when $a_i$ may belong to more than one component.

For the sake of simplicity and in order to avoid presenting redundant defini-
tions, we make the choice here not to consider the cases where $a_i \in \pi_{olb}$ or where
$a_i \in \pi_{eb}$; these cases subsumes all cases introduced in this section ($a_i \in \text{par}(\pi_{bef})$,
$a_i \in \text{par}(\pi_{af})$, ..., $a_i \in \text{par}(\pi_{next})$) and the procedure $OlbMerge$ introduced in Chapter
4.3 applies directly for these cases.

### 5.3.2 Function $TreeMerge$ - one projector component at a time -

The procedure $TreeMerge$ has already been introduced. Given two synchronized
subtrees $\tau_i$ and $\tau'$, it produces a subtree $\tau_r$ such that:

- $\text{roots}(\tau_r) = n = m$,
- $\text{lab}(\text{roots}(\tau_r)) = \text{lab}(\text{roots}(\tau')) = b_i$, and
- $\text{subfor}(\tau_r) =$
Chapter 5. Extending the Type Projection based evaluation...

\[
\begin{align*}
\text{NoMerge}(F_i | F_u) & \quad \text{if } a_i \in \pi_{\text{no}} \\
\text{AsFirstMerge}(F_i | F_u) & \quad \text{if } a_i \in \pi_{\text{asfirst}} \\
\text{AsLastMerge}(F_i | F_u) & \quad \text{if } a_i \in \pi_{\text{aslast}} \\
\text{DelMerge}(F_i | F_u | a_i) & \quad \text{if } a_i \in \text{par}(\pi_{\text{del}}) \\
\text{BeforeMerge}(F_i | F_u | a_i) & \quad \text{if } a_i \in \text{par}(\pi_{\text{bef}}) \\
\text{AfterMerge}(F_i | F_u | a_i) & \quad \text{if } a_i \in \text{par}(\pi_{\text{af}}) \\
\text{NextMerge}(F_i | F_u | a_i) & \quad \text{if } a_i \in \text{par}(\pi_{\text{next}})
\end{align*}
\]

Recall here that we develop our analysis based on the assumption that the label \(a_i\) of the parent node of \(F_i\) belongs to only one projector component. We make the assumption that the types of the first level nodes of the forest \(F_i\) also belong to only one projector component, but not necessarily the same component as \(a_i\).

5.3.2.1 The procedure NoMerge

This procedure is meant to merge two forests \(F_i\) and \(F_u\) whose parent nodes \(n\) and \(m\) respectively are "synchronized" (\(n\equiv m\)) and such that the label \(a_i\) of the parent node of \(F_i\) only belongs to \(\pi_{\text{no}}\). It produces a sub-forest \(F_r\) of the final result \(u(t)\).

First let us comment on the properties induced by the condition \(a_i \in \pi_{\text{no}}\). Indeed, this condition ensures that the children of \(F_i\) could not be the target of update operation other than renaming. This implies that \((\dagger_1)\) the first level nodes of \(F_u\) are the first level nodes of \(F_i\) projected by \(\pi\) up to some renaming of labels.

Building the forest \(F_r\) is then very simple: trees \(t_i\) of \(F_i\) that were not projected are re-introduced and synchronized pairs of trees \(t_i, t_u\) of the forests \(F_i\) and \(F_u\) are processed by calling TreeMerge. Note that the fact that a tree \(t_i\) has not been projected is identified by checking if the type of its root belong to \(\pi_{\text{seed}}\) because we assume that \(a_i\) belongs to \(\pi_{\text{no}}\) only.

Example 5.3.1. Illustrating the behavior of NoMerge: see Figure 5.23.

Let us consider the update \(u_1\) specified by:

\[
\text{for } x \text{ in } /a \text{ return rename node } x/b \text{ with "d" .}
\]

projector - The extended projector \(\pi_{\text{ext}}\) derived for the update \(u_1\) contains one component \(\pi_{\text{no}}=\{a,b\}\).

update - Figure 5.23-2 illustrates the changes applied by the update \(u_1\). The parent node of \(F_u\) labelled by \(a\) has one child \(F_u@1.2\) labelled by \(d\) (the node \(F_i@1.2\) labelled by \(b\) has been renamed by \(d\)).

merge - (see fig. 5.23-3). Because the parent node of \(F_i\) is labelled by \(a \in \pi_{\text{no}}\) (see fig. 5.23-1), the forests \(F_i\) and \(F_u\) are merged by NoMerge.

First, NoMerge examines the nodes \(F_i@1.1\) and \(F_u@1.2\). Because \(F_i@1.1\) is labelled by \(d \in \pi_{\text{seed}}\), NoMerge executes line 2 and accordingly, it outputs the tree rooted at \(F_i@1.1\). The next two nodes processed are \(F_i@1.2\) and \(F_u@1.2\). The node \(F_i@1.2\) is labelled by \(b \in \pi_{\text{no}}\) leading to the execution of line 3: TreeMerge is going to choose how to merge the trees rooted at \(F_i@1.2\) and \(F_u@1.2\) based on the type of the node \(F_i@1.2\). After that step, since all subtrees of \(F_u\) have been processed, line 1 is executed leading to output the remaining sub-trees from \(F_i\). □
5.3. Definition of the Extended Projection

5.3.2.2 The procedure \textit{AsFirstMerge}

This procedure is meant to merge two forests $F_i$ and $F_u$ whose parent nodes $n$ and $m$ are "synchronized" ($n=m$) and such that $a_i \in \pi_{\text{asfirst}}$ only. It produces a sub-forest $F_r$ of the final result $u(t)$.

The properties induced by the condition $a_i \in \pi_{\text{asfirst}}$ are following: (†2) the first level nodes of $F_u$ are either new nodes (having fresh identifiers $i_x$) or nodes whose types are in $\pi_{\text{seed}}$ and thus have been projected. Moreover, knowing that $a_i \in \pi_{\text{asfirst}}$ implies that $F_u$ potentially starts with new trees corresponding to the "as first" insertion.

The procedure \textit{AsFirstMerge} is formally presented in Figure 5.24: line 2 outputs new inserted "as first" elements (indeed, \textit{AsFirstMerge} gives priority to outputting new elements); line 3 outputs subtrees of $F_i$ projected out; line 4 treats synchronized subtrees of $F_i$ and $F_u$.

\textbf{Example 5.3.2}. \textit{Illustrating the behavior of AsFirstMerge: see Figure 5.25.}

\begin{align*}
1 & \text{ AsFirstMerge}(F_i \mid F_u) = \begin{cases} F_i & \text{ if roots}(F_u)=\emptyset \\ \text{otherwise} (\text{neither } F_i \text{ nor } F_u \text{ are empty}) \end{cases} \\
2 & t_i \circ \text{AsFirstMerge}(f_i \mid F_u) \quad \text{if } \text{new} (\text{roots}(t_u))=\text{true} \\
3 & \text{otherwise} \quad t_i \circ \text{AsFirstMerge}(f_i \mid F_u) \quad \text{if } \text{lab} (\text{roots}(t_i)) \notin \pi_{\text{seed}} \\
4 & \text{TreeMerge}(t_i \mid t_u) \circ \text{AsFirstMerge}(f_i \mid f_u) 
\end{align*}

Figure 5.24: The procedure \textit{AsFirstMerge}
Let us consider the update $u_2$ specified by:

```xml
for $x$ in /a return insert nodes {<k/>,<k/>} as first into $x$
```

**projector** - The extended projector $\pi_{ext}$ derived for this update $u_2$ has one component $\pi_{asfirst}=\{a\}$.

**update** - Figure 5.25-2 illustrates the changes applied by the update $u_2$. The forest $F_u$ has two "as first" inserted elements $F_u@i_1$ and $F_u@i_2$ labelled by $k$.

**merge** - (see fig. 5.25-3). Because the parent node of $F_i$ is labelled by $a\in\pi_{asfirst}$, the forests $F_i$ and $F_u$ are merged by $AsFirstMerge$ (see fig. 5.24). First, $AsFirstMerge$ parses the nodes $F_i@i_1$ and $F_u@i_1$. Because the identifier of $F_u@i_1$ indicates that it is a new inserted node ($new(\text{roots}(F_u@i_1))=true$), $AsFirstMerge$ executes line 2 and outputs the tree rooted at $F_u@i_1$. Next, $AsFirstMerge$ examines the nodes $F_i@i_1$ and $F_u@i_2$ and, for the same reasons as before, $AsFirstMerge$ executes line 2 and outputs the tree rooted at $F_u@i_2$. After that step, since the forest $F_i$ has been totally parsed, line 1 is executed leading to output the remaining subtrees of $F_i$. □

### 5.3.2.3 The procedure $AsLastMerge$

This case is similar to the previous one. This time the assumption that $a_i$ belongs to $\pi_{aslast}$ only, entails that $F_u$ potentially ends with new trees corresponding to the "as last" insertion.

The properties induced by this condition are following: (†3) the first level nodes of $F_u$ are either new nodes (having fresh identifiers ix) or nodes whose types are in $\pi_{seed}$ and thus have been projected. Moreover, knowing that $a_i\in\pi_{aslast}$ implies that $F_u$ potentially ends with the new trees corresponding to the "as last" insertion. The procedure $AsLastMerge$ is formally presented in Figure 5.26: lines 3 and 4 gives priority to outputting trees of $F_i$ which have been projected out or to merging synchronized subtrees $t_i$ and $t_u$; line 2 takes care of outputting potentially new elements inserted "as last".

**Example 5.3.3. Illustrating the behavior of $AsLastMerge$: see Figure 5.27.**

Let us consider the update $u_2$ specified by:

```xml
for $x$ in /a return insert nodes {<k/>,<k/>} as last into $x$.
```
5.3. Definition of the Extended Projection

1. \[ \text{AsLastMerge}(F_i | F_u) = \begin{cases} F_i & \text{if roots}(F_u) = \emptyset \\ F_u & \text{if roots}(F_i) = \emptyset \\ \text{otherwise (neither } F_i \text{ nor } F_u \text{ are empty)} \end{cases} \]

2. \[ t_i \circ \text{AsLastMerge}(F_i | F_u) \]

3. \[ \text{otherwise} \]

4. \[ \text{TreeMerge}(t_i | t_u) \circ \text{AsLastMerge}(F_i | F_u) \]

Figure 5.26: The procedure \text{AsLastMerge}

Figure 5.27: Illustration of the behavior of \text{AsLastMerge}

**projector** - The extended projector \( \pi_{\text{ext}} \) derived from the update \( u_3 \) has one component \( \pi_{\text{aslast}} = \{a\} \).

**update** - Figure 5.27-2 illustrates the changes applied by the update \( u_3 \). The parent node of \( F_u \) labelled by \( a \) has two "as last" inserted elements \( F_u@1 \) and \( F_u@2 \) labelled by \( k \).

**merge** - (see fig. 5.27-3). Because the parent node of \( F_i \) is labelled by \( a \in \pi_{\text{aslast}} \), the forests \( F_i \) and \( F_u \) merged by \text{AsLastMerge} (see fig. 5.26).

First, \text{AsLastMerge} examines the nodes \( F_u@1 \) and \( F_u@2 \). Because \( F_u@1 \) is labelled by \( d \not\in \pi_{\text{seed}} \), line 3 is executed and \text{AsLastMerge} outputs the tree rooted at \( F_i@1 \). Next the nodes \( F_u@2 \) and \( F_u@1 \) are parsed and, for the same reason as before, line 3 is executed and outputs the tree rooted at \( F_i@2 \). The remaining subtrees of \( F_i \) are processed in the same manner. Finally, when \( F_i \) has been totally parsed, \text{AsLastMerge} executes line 2 and outputs the forest \( F_u \) containing the new trees rooted at \( F_u@1 \) and \( F_u@2 \).

5.3.2.4 The procedure \text{BeforeMerge}

This procedure is meant to merge two forests \( F_i \) and \( F_u \) whose parent nodes \( n \) and \( m \) are "synchronized" \( n = m \) and such that \( a_i \) belongs only to \( \text{par}(\pi_{\text{bef}}) \). It produces a sub-forest \( F_r \) of the final result \( u(t) \).

The properties induced by the condition \( a_i \in \text{par}(\pi_{\text{bef}}) \) are the following: \((\dagger_4)\) the first level nodes of \( F_u \) are either new nodes (having fresh identifiers \( k \)) or nodes whose types are in \( \pi_{\text{seed}} \) and thus have been projected, or nodes whose types are in \( \pi_{\text{bef}}(a_i) \) and have been projected for the purpose of potential insertions "before". The condition ensures that no delete or replace operation could be performed by the
update $u$ over the first level nodes of $F_i$ and that the only possible insert operation is "insert before".

The procedure $BeforeMerge$ is formally specified in Figure 5.28. Note that this procedure has an extra parameter $x$ which is the type of the parent node of $F_i$. This parameter is required for identifying the first level nodes of $F_i$ whose types belong to $\pi_{bef}(x)$ (with $x=a_i$ in the current presentation). Line 2 takes care of the subtrees of $F_i$ that were projected out. Line 3 takes care of synchronized subtrees $t_i$ and $t_u$ of $F_i$ and $F_u$, and finally line 4 takes care of the inserted before subtrees $t_u$ of $F_u$.

**Example 5.3.4. Illustrating the behavior of $BeforeMerge$: see Figure 5.29.**

Let us consider the update $u_4$ specified by:

for $x$ in /a/e where $x/e/z$ return insert nodes {<k/>,<k/>} before $x$.

**projector** - The extended projector $\pi_{ext}$ derived for the update $u_4$ contains one component $\pi_{bef}=$\{(a,e)\}.

**update** - Figure 5.29-2 illustrates the changes applied by the update $u_4$. The parent node of $F_u$ labelled by $a$ has two "before" inserted elements $F_u@i1$ and $F_u@i2$ labelled by $k$ and the elements $F_u@1.3$ and $F_u@1.5$ obtained by the projection. Note that the element $F_u@1.3$ is followed by its child $F_u@1.3.1$.

**merge** - (see fig. 5.29-3). Because the parent node of $F_i$ is labelled by $a\in\text{par}(\pi_{bef})$, the forests $F_i$ and $F_u$ are merged by $BeforeMerge$ (see fig. 5.28).

First, $BeforeMerge$ examines the nodes $F_i@1.1$ and $F_u@i1$. Because $F_i@1.1$ is labelled by $d\notin\pi_{bef}(a)\cup\pi_{seed}$, $BeforeMerge$ executes line 2 and accordingly, it outputs the tree rooted at $F_i@1.1$. The next two nodes processed are $F_i@1.2$ and $F_u@i1$ and, for the
same reason as before, line 2 is executed and BeforeMerge outputs the tree rooted at $F_{i@1.2}$. Next, BeforeMerge examines the nodes $F_{i@1.3}$ and $F_{u@i1}$. Because $F_{i@1.3}$ is labelled by $e\in ch(\pi_{bef})$ and the identifier of $F_{u@i1}$ indicates that it is a new inserted node ($new(roots(F_{u@i1}))=true$), BeforeMerge outputs the tree rooted at $F_{u@i1}$. Next, the nodes $F_{i@1.3}$ and $F_{u@i2}$ are parsed and, for the same reason as before, line 4 is executed and BeforeMerge outputs the tree rooted at $F_{u@i2}$. After that step, the next two nodes processed are $F_{i@1.3}$ and $F_{u@i3}$. The identifiers of the two nodes are equal ($roots(F_{i@1.3})=roots(F_{u@i3})$), leading to the execution of line 3: TreeMerge is going to choose how to merge the trees rooted at $F_{i@1.3}$ and $F_{u@1.3}$ based on the type of the node $F_{i@1.3}$. The next two nodes processed are $F_{i@1.4}$ and $F_{u@1.5}$. Because $F_{i@1.4}$ is labelled by $c\notin \pi_{bef}(a)\cup \pi_{seed}$, BeforeMerge executes line 2 and outputs the tree rooted at $F_{i@1.4}$. Finally the nodes $F_{i@1.5}$ and $F_{u@1.5}$ are examined. Because the identifiers of the two nodes are equal ($roots(F_{i@1.5})=roots(F_{u@1.5})$), BeforeMerge executes line 3: TreeMerge is going to choose how to merge the trees rooted at $F_{i@1.5}$ and $F_{u@1.5}$. 

**5.3.2.5 The procedure AfterMerge**

This procedure is meant to merge two forests $F_i$ and $F_u$ whose parent nodes $n$ and $m$ are "synchronized" ($n=m$) and such that $a_i$ belongs only to par($\pi_{af}$). It produces a sub-forest $F_r$ of the final result $u(t)$.

The properties induced by the condition $a_i\in par(\pi_{af})$ are the same as for $\uparrow_4$ except that this time, the new elements are inserted "after".

The procedure AfterMerge is formally defined in Figure 5.30 and do not present any new difficulties.

**Example 5.3.5.** **Illustrating the behavior of AfterMerge:** see Figure 5.31. Let us consider the update $u_5$ specified by :

```
for $x$ in /a return insert nodes {<k/>, <k/>} after $x/e$.
```

**projector** - The extended projector $\pi_{ext}$ derived from the update $u_4$ contains one component $\pi_{af}=$\{(a, e)\}.

**update** - Figure 5.31-2 illustrates the changes applied by the update $u_5$. The parent node of $F_u$ labelled by $a$ has four "after" inserted elements $F_u@i1$, $F_u@i2$, $F_u@i3$.
This procedure is meant to merge two forests $F_i$ and $F_u$ whose parent nodes $n$ and $m$ are "synchronized" ($n=m$) and such that $a_i$ belongs only to $\pi_{\text{next}}$. It produces a sub-forest $F_r$ of the final result $t(u)$.

The procedure $\text{DelMerge}$ is formally presented in Figure 5.32. Roughly, it behaves like the procedure $\text{NoMerge}$ specified in the context of the three-level projector.

5.3.2.7 The procedure $\text{NextMerge}$

This procedure is meant to merge two forests $F_i$ and $F_u$ whose parent nodes $n$ and $m$ are "synchronized" ($n=m$) and such that $a_i$ belongs only to $\pi_{\text{next}}$. It produces a sub-forest $F_r$ of the final result $t(u)$. 

Figure 5.31: Illustration of the behavior of $\text{AfterMerge}$

and $F_u@i4$ labelled by $k$, and the elements $F_u@1.3$ and $F_u@1.5$ obtained by the projection.

**merge** - (see fig. 5.31-3). Because the parent node of $F_i$ is labelled by $a\in \text{par}(\pi_{\text{af}})$, the forests $F_i$ and $F_u$ are merged by $\text{AfterMerge}$ (see fig. 5.30).

First, $\text{AfterMerge}$ examines the nodes $F_i@1.1$ and $F_u@1.3$. Because $F_i@1.1$ is labelled by $d\notin \pi_{\text{bef}}(a)\cap \pi_{\text{seed}}$ and the identifier of $F_u@1.3$ indicates that it is not a new inserted node (new($\text{roots}(F_u@1.3)) = \text{false}$), line 3 is executed and $\text{AfterMerge}$ outputs the tree rooted at $F_i@1.1$. The next two nodes processed are $F_i@1.2$ and $F_u@1.3$ and, for the same reason as before, line 3 is executed and $\text{AfterMerge}$ outputs the tree rooted at $F_i@1.2$. After that step, the next two nodes processed are $F_i@1.3$ and $F_u@1.3$. The identifiers of the two nodes are equal ($\text{roots}(F_i@1.3) = \text{roots}(F_u@1.3)$), leading to the execution of line 4: $\text{TreeMerge}$ is going to choose how to merge the trees rooted at $F_i@1.3$ and $F_u@1.3$ based on the type of the node $F_i@1.3$. Next, $\text{AfterMerge}$ examines the nodes $F_i@1.4$ and $F_u@i1$. Because the identifier of $F_u@i1$ indicates that it is a new inserted node (new($\text{roots}(F_u@i1)) = \text{true}$), $\text{AfterMerge}$ executes line 5 and outputs the tree rooted at $F_u@i1$. The next two nodes processed are $F_i@1.4$ and $F_u@i2$ and, for the same reason as before, line 5 is executed and $\text{AfterMerge}$ outputs the tree rooted at $F_u@i2$. After that step, $\text{AfterMerge}$ examines the nodes $F_i@1.4$ and $F_u@1.5$. Because the two nodes are equal (roots($F_i@1.4) = \text{roots}(F_u@1.5)$), $\text{AfterMerge}$ executes line 4. Finally, $\text{AfterMerge}$ executes line 2 and outputs the forest $F_u$ containing the new trees rooted at $F_u@i3$ and $F_u@i4$. 

5.3.2.6 The procedure $\text{DelMerge}$

This procedure is meant to merge two forests $F_i$ and $F_u$ whose parent nodes $n$ and $m$ are "synchronized" ($n=m$) and such that $a_i$ belongs only to $\text{par}(\pi_{\text{next}})$. It produces a sub-forest $F_r$ of the final result $t(u)$. 

The procedure $\text{DelMerge}$ is formally presented in Figure 5.32. Roughly, it behaves like the procedure $\text{NoMerge}$ specified in the context of the three-level projector.
5.3. Definition of the Extended Projection

\[
\begin{align*}
1 & \quad \text{DelMerge}(F_i \mid F_u \mid x) = F_u \quad \text{if roots}(F_i) = \emptyset \\
\text{otherwise (} F_i \text{ is not empty)} & \quad t_i \circ \text{DelMerge}(f_i \mid F_u \mid x) \quad \text{if lab}(\text{roots}(t_i)) \notin \pi_{\text{del}}(x) \cup \pi_{\text{seed}} \\
\text{otherwise, assuming roots}(F_u) \neq \emptyset \quad \text{then} & \quad \text{TreeMerge}(t_i \mid t_u) \circ \text{DelMerge}(f_i \mid F_u \mid x) \quad \text{if roots}(t_i) = \text{roots}(t_u) \\
\text{otherwise, assuming roots}(F_u) = \emptyset \quad \text{then} & \quad \text{DelMerge}(f_i \mid F_u \mid x)
\end{align*}
\]

Figure 5.32: The procedure DelMerge

\[
\begin{align*}
1 & \quad \text{NextMerge}(F_i \mid F_u \mid x) = F_u \quad \text{if roots}(F_i) = \emptyset \\
\text{otherwise (} F_i \text{ is not empty),} & \quad t_i \circ \text{NextMerge}(f_i \mid F_u \mid x) \quad \text{if roots}(t_i) = \text{roots}(t_u) \\
\text{assuming roots}(F_u) \neq \emptyset \quad \text{and lab}(\text{roots}(t_i)) \notin \pi_{\text{next}}(x) \cup \pi_{\text{seed}} & \quad \text{if roots}(t_i) = \text{roots}(t_u) \\
\text{or (} \text{new}(\text{roots}(t_u)) = \text{true} \text{)} & \quad \text{if roots}(t_i) < \text{roots}(t_u) \\
\text{otherwise still assuming roots}(F_u) \neq \emptyset & \quad t_u \circ \text{NextMerge}(F_i \mid F_u \mid x) \quad \text{if new}(\text{roots}(t_u)) = \text{true} \\
\text{or (} \text{roots}(F_u) = \emptyset \text{)} & \quad \text{NextMerge}(f_i \mid F_u \mid x) \quad \text{if roots}(t_i) < \text{roots}(t_u) \\
\text{or (} \text{roots}(F_u) = \emptyset \text{)} & \quad \text{TreeMerge}(t_i \mid t_u) \circ \text{NextMerge}(f_i \mid F_u \mid x) \quad \text{if roots}(t_i) = \text{roots}(t_u) \\
\text{otherwise (} \text{roots}(F_u) = \emptyset \text{)} & \quad F_i \quad \text{if lab}(\text{roots}(t_i)) \notin \pi \\
\text{or } & \quad () \quad \text{if lab}(\text{roots}(t_i)) \in \pi_{\text{next}}(x)
\end{align*}
\]

Figure 5.33: The procedure NextMerge

The properties induced by the condition \( a_i \in \text{par}(\pi_{\text{next}}) \) are the following: (†5) the first level nodes of \( F_u \) are either nodes whose types are in \( \pi_{\text{seed}} \) and thus have been projected or nodes whose types belong to \( \pi_{\text{next}}(a_i) \) and have not been replaced, followed by nodes that have been projected because they are the next siblings of nodes whose types belong to \( \pi_{\text{next}}(a_i) \) or finally new nodes which are roots of inserted elements "in place of".

The procedure NextMerge is formally presented in Figure 5.33. The reader should pay attention to line 2: it takes care of the case where \( t_i \) has been projected as a separator and no other reason (thus its root type does not belong to \( \pi_{\text{seed}} \)). Line 3 takes care of the subtrees of \( F_i \) that were projected out. Lines 4 and 5 takes care of the case where the subtree \( t_i \) has been replaced and line 6 takes care of synchronized subtrees \( t_i \) and \( t_u \) of \( F_i \) and \( F_u \). Finally, lines 7 and 8 are dealing with parsing termination.

**Example 5.3.6.** Illustrating the behavior of NextMerge: see Figure 5.34.
Let us consider the update $u_6$ specified by:

\[
\text{for } x \text{ in } /a \text{ return replace node } x/e \text{ with } \langle k/\rangle, \langle k/>.\]

**projector** - The extended projector $\pi_{\text{ext}}$ derived from the update $u_6$ has one component $\pi_{\text{next}}=\{(a, e)\}$.

**update** - Figure 5.34-2 illustrates the changes applied by the update $u_6$. The parent node of $F_u$ labelled by $a$ has four "in place of" inserted elements $F_u@i1$, $F_u@i2$, $F_u@i3$ and $F_u@i4$ labelled by $k$, and the separator element $F_u@1.4$ obtained by the projection.

**merge** - (see fig. 5.34-3). Because the parent node of $F_i$ is labelled by $a\in \text{par}(\pi_{\text{next}})$, the forests $F_i$ and $F_u$ are merged by $\text{NextMerge}$ (see fig. 5.33).

First, $\text{NextMerge}$ examines the nodes $F_i@1.1$ and $F_u@i1$. Because $F_i@1.1$ is labelled by $d\notin \pi_{\text{bef}}(a)\cup \pi_{\text{seed}}$ and the identifier of $F_u@i1$ indicates that it is a new inserted node ($\text{new(roots(F_u@i1))=true}$), line 3 is executed and $\text{NextMerge}$ outputs the tree rooted at $F_i@1.1$. The next two nodes processed are $F_i@1.2$ and $F_u@i1$ and, for the same reason as before, line 3 is executed and $\text{NextMerge}$ outputs the tree rooted at $F_i@1.2$. After that step, the next two nodes processed are $F_i@1.3$ and $F_u@i1$. Because $F_i@1.3$ is labelled by $e\in \text{ch}(\pi_{\text{next}})$ and the identifier of $F_u@i1$ indicates that it is a new inserted node ($\text{new(roots(F_u@i1))=true}$), $\text{NextMerge}$ executes line 4 and outputs the tree rooted at $F_u@i1$. Next, $\text{NextMerge}$ examines the nodes $F_i@1.3$ and $F_u@i2$ and, for the same reason as before, line 4 is executed and $\text{NextMerge}$ outputs the tree rooted at $F_u@i2$. The next two nodes processed are $F_i@1.3$ and $F_u@i4$. Because the identifier of $F_i@1.3$ is less than the one of $F_u@1.4$ line 5 is executed and $\text{NextMerge}$ skips the tree rooted at $F_i@1.3$. After that step, the nodes $F_i@1.4$ and $F_u@1.4$ are examined. Because $F_i@1.4$ is labelled by $c\notin \pi_{\text{bef}}(a)\cup \pi_{\text{seed}}$ and the identifiers of $F_i@1.4$ and $F_u@1.4$ are equal ($\text{roots(F_u@1.4)=roots(F_i@1.4)}$) $\text{NextMerge}$ executes line 2 and outputs the tree rooted at $F_i@1.4$. Next, $\text{NextMerge}$ parses the nodes $F_i@1.5$ and $F_u@i3$. Because $F_i@1.5$ is labelled by $e\in \text{ch}(\pi_{\text{next}})$ and the identifier of $F_u@i3$ indicates that it is a new inserted node ($\text{new(roots(F_u@i3))=true}$), $\text{NextMerge}$ executes line 4 and outputs the tree rooted at $F_u@i3$. Next, $\text{NextMerge}$ examines the nodes $F_i@1.5$ and $F_u@i4$ and, for the same reason as before, line 4 is executed and $\text{NextMerge}$ outputs the tree rooted at $F_u@i4$. Finally, since the forest $F_u$ has been totally parsed and $F_i@1.5$ is labelled by $e\in \text{ch}(\pi_{\text{next}})$ $\text{NextMerge}$ executes line 8 and skips the tree rooted at $F_i@1.5$. □
5.3.3 Function TreeMerge - general case -

This section is devoted to the general case which means that now the label $a_i$ of the parent node of $F_i$ may belong to several projector components. The section starts by revising the core of the TreeMerge procedure. Then, we revise some of the xxMerge procedures that have been introduced already.

The procedure TreeMerge of Section 5.3.2 is extended to take into account the cases where $a_i$ belongs to more than one projector component. It is of course not necessary to consider all cases as some of them cannot happen. For instance, with respect to our analysis (See Table 5.20 for a synthesis), it is not possible that the label $a_i$ belongs to $par(\pi_{\text{bef}})$ and $par(\pi_{\text{af}})$ only because in that case $a_i$ should belong to $par(\pi_{\text{next}})$ and has been deleted from $par(\pi_{\text{bef}})$ and $par(\pi_{\text{af}})$.

Below, in the presentation of the procedure TreeMerge,

1. as for the definition of TreeMerge in Section 5.3.2, we do not consider the cases where $a_i \in \pi_{\text{ob}}$ or where $a_i \in \pi_{\text{eb}}$: these cases subsumes all cases introduced in this section and the procedure $OlbMerge$ introduced in Section 4.3 applies directly.

2. we do not reintroduce the cases where $a_i$ belongs to only one projector component for the sake of simplicity, and thus the code below should be viewed as additional code.

3. for the sake of simplicity, we also make the following convention: in the conditional parts of the procedure, writing that $a_i \in A$ and $a_i \in B$, we intend that $a_i \in A$ and $a_i \in B$ only or $a_i \in A$ and $a_i \in B$ and $a_i \in \pi_{\text{no}}$; for instance, $a_i \in par(\pi_{\text{bef}})$ and $a_i \in \pi_{\text{aslast}}$ means that the label $a_i$ belongs to $par(\pi_{\text{bef}})$ and to $\pi_{\text{aslast}}$ and may be also to $\pi_{\text{no}}$.

A similar convention is made for $a_i \in A$ and $a_i \in B$ and $a_i \in C$.

Given two synchronized subtrees $\tau_i$ and $\tau'$, TreeMerge produces a subtree $\tau_r$ such that:

- $\text{roots}(\tau_r) = n = m$,
- $\text{lab}(\text{roots}(\tau_r)) = \text{lab}(\text{roots}(\tau')) = b_i$, and
- $\text{subfor}(\tau_r) =$

\[
\begin{align*}
\text{AsFirstDelMerge}(F_i \mid F_u \mid a_i) & \quad \text{if } a_i \in par(\pi_{\text{del}}) \text{ and } a_i \in \pi_{\text{asfirst}} \\
\text{AsLastDelMerge}(F_i \mid F_u \mid a_i) & \quad \text{if } a_i \in par(\pi_{\text{del}}) \text{ and } a_i \in \pi_{\text{aslast}} \\
\text{BeforeMerge}(F_i \mid F_u \mid a_i) & \quad \text{if } a_i \in par(\pi_{\text{bef}}) \text{ and } a_i \in \pi_{\text{aslast}} \\
\text{AfterMerge}(F_i \mid F_u \mid a_i) & \quad \text{if } a_i \in par(\pi_{\text{af}}) \text{ and } a_i \in \pi_{\text{aslast}} \text{ or } a_i \in \pi_{\text{aslast}} \text{ and } a_i \in \pi_{\text{last}} \\
\text{NextMerge}(F_i \mid F_u \mid a_i) & \quad \text{if } a_i \in par(\pi_{\text{next}}) \text{ and } a_i \in \pi_{\text{aslast}} \text{ or } a_i \in \pi_{\text{aslast}} \text{ and } a_i \in \pi_{\text{first}} \\
\text{FirstMerge}(F_i \mid F_u \mid a_i) & \quad \text{if } a_i \in par(\pi_{\text{bef}}) \text{ and } a_i \in \pi_{\text{aslast}} \text{ and } a_i \in \pi_{\text{first}} \\
\end{align*}
\]
AsFirstDelMerge($F_i \mid F_u \mid x$) = $F_u$ if roots($F_i$)=$\emptyset$

otherwise ($F_i$ is not empty),

$$t_i \circ AsFirstDelMerge(f_i \mid F_u \mid x)$$ if lab(roots($t_i$))$\notin \pi_{del}(x) \cup \pi_{seed}$ and

$$[\text{roots}($F_u$)=$\emptyset$ or new(roots($t_u$))=false]$$

otherwise, assuming roots($F_u$)$\neq \emptyset$

TreeMerge($t_i \mid t_u$)$\circ AsFirstDelMerge(f_i \mid F_u \mid x)$ if roots($t_i$)=roots($t_u$)

AsFirstDelMerge($f_i \mid F_u \mid x$) if roots($t_i$)<roots($t_u$)

t_u \circ AsFirstDelMerge($F_i \mid F_u \mid x$) if new(roots($t_u$))=true

otherwise, assuming roots($F_u$)=$\emptyset$

AsFirstDelMerge($f_i \mid F_u \mid x$)

Figure 5.35: The procedure AsFirstDelMerge - general case -

5.3.3.1 The procedure AsFirstDelMerge - general case -

The procedure DelMerge (see fig. 5.32) is extended to support the case where the type $a_i$ of the parent node of $F_i$ belongs to $par(\pi_{del})$ and $\pi_{asfirst}$ (and may be to $\pi_{no}$ also).

The extended procedure AsFirstDelMerge is formally presented in Figure 5.35. Line 2 takes care of the subtrees of $F_i$ that were projected out. The condition on the forest $F_u$ checks that all "inserted as first" subtrees have bee treated (by line 5). Line 3 takes care of synchronized subtrees $t_i$ and $t_u$ of $F_i$ and $F_u$. Finally, lines 4 and 6 deal with the "deleted" subtrees $t_i$ of $F_i$. And, as already mentioned, line 5 takes care of the inserted "as first" subtrees $t_u$ of $F_u$.

5.3.3.2 The procedure AsLastDelMerge - general case -

The procedure DelMerge (see fig. 5.32) is extended to support the general case, when the type $a_i$ of the parent node of $F_i$ belongs to $par(\pi_{del})$ and $\pi_{aslast}$ (and may be to $\pi_{no}$ also).

The extended procedure AsLastDelMerge is formally presented in Figure 5.36. Line 1 deals with "inserted as last" subtrees $t_u$ of $F_u$. Line 2 takes care of the subtrees of $F_i$ that were projected out. Note that this line does not make any assumption on the emptiness of the partially updated subforest $F_u$. Line 3 takes care of synchronized subtrees $t_i$ and $t_u$ of $F_i$ and $F_u$. Finally, lines 4, 5 and 6 deal with the "deleted" subtrees $t_i$ of $F_i$.

5.3.3.3 The procedure BeforeMerge - general case -

The procedure BeforeMerge (see fig. 5.28) is extended to support the case where the type $a_i$ of the parent node of $F_i$ belongs to $par(\pi_{bef})$ and $\pi_{aslast}$ (and may be to $\pi_{no}$ also).

The properties induced by the condition $a_i$$\in$$par(\pi_{bef})$ and $\pi_{aslast}$ are the following: (†6) the first level nodes of $F_i$ are either new nodes (having fresh identifiers $ix$) or
5.3. Definition of the Extended Projection

\[ \text{AsLastDelMerge}(F_i \mid F_u \mid x) = \begin{cases} F_u & \text{if } \text{roots}(F_i) = \emptyset \\
\text{otherwise}(F_i \text{ is not empty}), \end{cases} \]

\[ t_i \circ \text{AsLastDelMerge}(f_i \mid F_u \mid x) \text{ if } \text{lab}(\text{roots}(t_i)) \notin \pi_{\text{del}}(x) \cup \pi_{\text{seed}} \]

\[ \text{otherwise, assuming } \text{roots}(F_u) \neq \emptyset \text{ then} \]

\[ \text{TreeMerge}(t_i \mid t_u) \circ \text{AsLastDelMerge}(f_i \mid F_u \mid x) \text{ if } \text{roots}(t_i) = \text{roots}(t_u) \]

\[ \text{AsLastDelMerge}(f_i \mid F_u \mid x) \text{ if } \text{roots}(t_i) < \text{roots}(t_u) \]

\[ \text{AsLastDelMerge}(f_i \mid F_u \mid x) \text{ if } \text{new}(\text{roots}(t_u)) = \text{true} \]

\[ \text{otherwise, assuming } \text{roots}(F_u) = \emptyset \text{ then} \]

\[ \text{AsLastDelMerge}(f_i \mid F_u \mid x) \]

Figure 5.36: The procedure AsLastDelMerge - general case -

\[ \text{BeforeMerge}(F_i \mid F_u \mid x) = \begin{cases} F_i & \text{if } \text{roots}(F_u) = \emptyset \\
F_u & \text{if } \text{roots}(F_i) = \emptyset \\
\text{otherwise (neither } F_i \text{ nor } F_u \text{ are empty)} \end{cases} \]

\[ t_i \circ \text{BeforeMerge}(f_i \mid F_u \mid x) \text{ if } \text{lab}(\text{roots}(t_i)) \notin \pi_{\text{bef}}(x) \cup \pi_{\text{seed}} \]

\[ \text{TreeMerge}(t_i \mid t_u) \circ \text{BeforeMerge}(f_i \mid F_u \mid x) \text{ if } \text{roots}(t_i) = \text{roots}(t_u) \]

\[ \text{BeforeMerge}(F_i \mid F_u \mid x) \text{ if } \text{new}(\text{roots}(t_u)) = \text{true} \]

Figure 5.37: The procedure BeforeMerge - general case -

nodes whose types are in \( \pi_{\text{seed}} \) and thus have been projected, or nodes whose types are in \( \pi_{\text{bef}}(a_i) \) and have been projected for the purpose of potential insertions "before".

The condition ensures that no delete or replace operation could be performed by the update \( u \) over the first level nodes of \( F_i \) and that the only possible insert operations are "insert before" and "insert as last".

The procedure BeforeMerge is formally specified in Figure 5.37. Line 2 takes care of outputting potentially new elements inserted "as last". Indeed, it is the only change made on the former BeforeMerge given in Figure 5.28. Line 3 takes care of the subtrees of \( F_i \) that were projected out. Line 4 takes care of synchronized subtrees \( t_i \) and \( t_u \) of \( F_i \) and \( F_u \), and finally line 5 takes care of the inserted "before" subtrees \( t_u \) of \( F_u \).

5.3.3.4 The procedure AfterMerge - general case -

The procedure AfterMerge (see fig. 5.30) is extended to support the general case, when the type \( a_i \) of the parent node of \( F_i \) belongs either to \( \langle \text{par}(\pi_{\text{af}}) \rangle \) and \( \pi_{\text{asfirst}} \) or \( \langle \text{par}(\pi_{\text{af}}) \rangle \) and \( \pi_{\text{last}} \) or \( \langle \text{par}(\pi_{\text{af}}) \rangle \) and \( \pi_{\text{asfirst}} \) and \( \pi_{\text{last}} \).
AfterMerge($F_i \mid F_u \mid x$) =

1. $F_i$ if $roots(F_u) = \emptyset$
2. $t_i \circ \text{AfterMerge}(f_i \mid F_u \mid x)$ if $new(roots(t_i)) = \text{false}$
3. $t_u \circ \text{AfterMerge}(F_i \mid f_u \mid x)$ if $new(roots(t_u)) = \text{true}$
4. $\text{TreeMerge}(t_i \mid t_u) \circ \text{AfterMerge}(f_i \mid f_u \mid x)$ if $roots(t_i) = roots(t_u)$
5. $\text{AfterMerge}(F_i \mid f_u \mid x)$ if $new(roots(t_u)) = \text{false}$
6. $F_u$ if $new(roots(t_u)) = \text{true}$

Figure 5.38: The procedure AfterMerge - general case -

The properties induced by the conditions given above are the following: (†7) the first level nodes of $F_u$ are either new nodes (having fresh identifiers $ix$) or nodes whose types are in $\pi_{\text{seed}}$ and thus have been projected, or nodes whose types are in $\pi_{\text{af}}(a_i)$ and have been projected for the purpose of potential insertions "after", or the last node of $F_i$ projected as a separator when $a_i$ belongs to $\pi_{\text{last}}$. This ensures that no delete or replace operation could be performed by the update $u$ over the first level nodes of $F_i$ and that the only possible insert operations are "insert after", "insert as first" and "insert as last".

The procedure AfterMerge is formally specified in Figure 5.38. Line 2 takes care of the subtrees of $F_i$ that were projected out. Line 3 takes care of the inserted "as first" or "after" subtrees $t_u$ of $F_u$. Line 4 takes care of synchronized subtrees $t_i$ and $t_u$ of $F_i$ and $F_u$. Finally, lines 5 and 6 are dealing with parsing termination. The reader should pay attention to line 5: it takes care of the case where $t_u$ has been projected as a separator that is $t_u$ is the last child of the parent node $a_i$ which belong to $\pi_{\text{last}}$ and its label does not necessarily belong to the projector; in that case this node has already been output by line 2 and thus it has to be skipped by AfterMerge. Line 6 takes care of outputting potentially new elements inserted "as last".

5.3.3.5 The procedure NextMerge - general case -

The procedure NextMerge (see fig. 5.33) is extended to support the case where the type $a_i$ of the parent node of $F_i$ belongs either to $(par(\pi_{\text{next}})$ and $\pi_{\text{nslast}}$) or $(par(\pi_{\text{next}})$ and $\pi_{\text{first}}$) or $(par(\pi_{\text{next}})$ and $\pi_{\text{nslast}}$ and $\pi_{\text{first}}$).

The properties induced by the conditions given above are the following: (†8) the first level nodes of $F_u$ are either nodes whose types are in $\pi_{\text{seed}}$ and thus have been projected or nodes whose types belong to $\pi_{\text{next}}(a_i)$ and have not been replaced,
5.3. Definition of the Extended Projection

![Figure 5.39: The procedure NextMerge - general case -](image)

followed by nodes that have been projected because they are the next siblings of nodes whose types belong to \( \pi_{next}(a_i) \), or new nodes which are roots of inserted elements "in place of", "as first" or "as last". Finally, the first child of \( a_i \) may have been projected, because \( a_i \) belongs to \( \pi_{first} \).

This condition ensures that no delete operation could be performed by the update \( u \) over the first level nodes of \( F_i \).

The procedure \( \text{NextMerge} \) is formally presented in Figure 5.39. It uses the function \( \text{first} \) that simply returns true when the subtree \( t_i \) is the first child of the parent node of \( F_i \) (it has an identifier \( x.1 \)).

Line 2 takes care of the case where \( t_i \) has been projected as a separator. Line 3 takes care of the subtrees of \( F_i \) that were projected out. Note that the condition: \( \text{new}(\text{roots}(t_i)) = \text{true} \) and \( \text{first}(\text{roots}(t_i)) = \text{false} \) verifies that the subtree \( t_u \) of \( F_u \) is not inserted "as first". Line 4 takes care of the inserted "as first" subtrees \( t_u \) of \( F_u \). Lines 5 and 6 take care of the case where the subtree \( t_i \) has been replaced and line 7 takes care of synchronized subtrees \( t_i \) and \( t_u \) of \( F_i \) and \( F_u \). Finally, lines 8 and 9 are dealing with parsing termination.

### 5.3.3.6 The procedure FirstMerge - general case -

This procedure is meant to merge two forests \( F_i \) and \( F_u \) whose parent nodes \( n \) and \( m \) are "synchronized" \( (n=m) \) and such that \( a_i \) belongs either to \( \text{par}(\pi_{bef}) \) and \( \pi_{first} \) or to \( \text{par}(\pi_{bef}) \) and \( \pi_{aslast} \) and \( \pi_{first} \) or to \( \pi_{aslast} \) and \( \pi_{first} \).

The properties induced by the conditions given above are the following: \( (\dagger_9) \) the first level nodes of \( F_u \) are either new nodes (having fresh identifiers \( ix \)) or nodes whose types are in \( \pi_{seed} \), or nodes whose types are in \( \pi_{bef}(a_i) \) and have been
164 Chapter 5. Extending the Type Projection based evaluation...

\[ \text{FirstMerge}(F_i \mid F_u \mid x) = \begin{cases} F_i & \text{if roots}(F_u) = \emptyset \\ F_u & \text{if roots}(F_i) = \emptyset \\ \text{otherwise} \quad \text{(neither } F_i \text{ nor } F_u \text{ are empty)} \end{cases} \]

assuming \( \text{lab}(\text{roots}(t_i)) \in \pi_{\text{next}}(x) \cup \pi_{\text{seed}} \)

\[ \text{TreeMerge}(t_i \mid t_u) \circ \text{FirstMerge}(f_i \mid f_u \mid x) \text{ if } \text{roots}(t_i) = \text{roots}(t_u) \]

\[ t_u \circ \text{FirstMerge}(F_i \mid f_u \mid x) \text{ if } \text{new}(\text{roots}(t_u)) = \text{true} \]

\[ \text{otherwise} \]

\[ t_i \circ \text{FirstMerge}(f_i \mid F_u \mid x) \text{ if } \text{first}(\text{roots}(t_i)) = \text{false} \]

\[ t_u \circ \text{FirstMerge}(F_i \mid f_u \mid x) \text{ if } \text{first}(\text{roots}(t_i)) = \text{true} \text{ and } \] \[ \text{new}(\text{roots}(t_i)) = \text{true} \]

\[ t_i \circ \text{FirstMerge}(f_i \mid F_u \mid x) \text{ if } \text{first}(\text{roots}(t_i)) = \text{true} \text{ and } \] \[ \text{roots}(t_i) = \text{roots}(t_u) \]

Figure 5.40: The procedure \text{FirstMerge} - general case -

projected for the purpose of potential insertions "before" or the first child node of \( n \) because \( a_i \) belongs to \( \pi_{\text{first}} \). The condition ensures that no delete or replace operation could be performed by the update \( u \) over the first level nodes of \( F_i \) and that the only possible insert operation is "insert before", "insert as last" and "insert as first".

The procedure \text{FirstMerge} is formally presented in Figure 5.40. Line 2 takes care of the inserted "as last" subtrees \( t_u \) of \( F_u \). Line 3 takes care of synchronized subtrees \( t_i \) and \( t_u \) of \( F_i \) and \( F_u \). Line 4 takes care of the inserted "before" subtrees \( t_u \) of \( F_u \). Line 5 takes care of the subtrees of \( F_i \) that were projected out. Line 6 takes care of the inserted "as first" subtrees \( t_u \) of \( F_u \). The reader should pay attention to line 7: it takes care of the case where \( t_i \) has been projected as a separator.

5.3.4 Conclusion

As stated at the beginning of this Chapter, extending the three−level projection based evaluation of updates is memory oriented: the goal was to decrease the size of the projected document. For example, we have showed that while executing an update query \( u \) that performs insertion "as first", using the extended projector vs. the three−level one, requires less memory usage. As the reader can observe, the proposed optimization leads to a more complex type projector. Execution of the extended projector requires performing additional tests. Intuitively, this may have an impact on the execution time, increasing it compared to three−level type projector. On the other hand, the Merge phase reflects the changes done on the type projector and is composed of a larger set of procedures. Compared to the Merge supporting three−level projector, this set of procedures is more complex to implement, since there are more cases to verify.

Therefore, it remains to analyse whether using method based the extended pro-
jection is always better than using the one based on the three–level type projector and in which cases the method based on the three–level type projector should be preferred for the purpose of saving execution time when space is not the priority.

The implementation of the extended optimization has not yet been developed, thus we cannot provide such an analysis and results concerning the changes in the execution time neither for the projection nor for the Merge phases.
Chapter 6

Conclusion

In this Thesis, we have studied the update optimization techniques for main-memory systems. To this end we have adopted techniques based on XML projection.

We have first examined internal data representation and evaluation strategies of main XQuery engines, namely: MonetDB/XQuery, BaseX, eXist and Saxon. For the experiments in the Thesis, we also used Qizx, but we did not discuss the implementation details, because there is no documentation available. Even if the systems are efficient for memory management, as we have seen projection improves a memory usage for all systems.

We have first developed a projection based optimization method for updates using a three-level projection. My contribution was the specification of the Merge phase, developing a prototype and running tests. The results of the experiments demonstrate that our technique is very efficient for memory savings: using our technique we can execute updates on documents having sizes up to 2GB. We have, as well, sensible improvements in terms of time, which is due to reducing the number of elements to be indexed while importing a document to the database.

It is important to note that our approach allows, as well, to evaluate a workload (n updates) by processing our method just once. The scenario is then the following. A global projector is inferred in a straightforward manner (it is the union of the update projectors inferred for each update). Next the document on which workload has to be applied is projected, the updates are evaluated and finally the Merge algorithm is executed without any change required. Some preliminary tests have been done ([11]), which shows that the projection is efficient for workload. Further test are needed to understand the limitations of the approach.

For Saxon we still have memory limitation using projection, due to the low selectivity of the $\pi_{olb}$ component in some cases. This was the starting point of my second contribution. An extension of the projector extracted from the update and schema has been proposed as well as the extension of the Merge phase compatible with this projection. The extension does not cover the case of mixed content element. Further analysis is required for dealing with text in a more precise manner than the $\pi_{olb}$ component does.
We are currently working on the implementation and the experiments of the extension of the method explained in Chapter 5. It is worth noticing that this extension has a bigger set of projector components and many cases to deal with, thus more complex projection and Merge process. Therefore, one of our future work is to make an analysis of both Merge and its extension, in order to determine the cases when using the extension is less effective than using the core method based on a three-level projection.

One of the future improvements of our technique is the reduction of the execution time. In order to do that, as it has been explained in Chapter 4 we plan to eliminate: (i) storing the pruned document on the disk, and (ii) storing and re-reading the partial update pruned document. This requires some strong interaction with the update processor, and hence further implementation efforts.
References


[22] F. Cavalieri, G. Guerrini, and M. Mesiti. Dynamic reasoning on XML updates. In Ailamaki et al. [10], pages 165–176. 6


