HAL
open science

Main Memory XML Update Optimization: algorithms and experiments.

Sahakyan Marina

To cite this version:

Sahakyan Marina. Main Memory XML Update Optimization: algorithms and experiments.. Databases [cs.DB]. Université Paris Sud - Paris XI, 2011. English. NNT : . tel-00641579

HAL Id: tel-00641579

https://theses.hal.science/tel-00641579

Submitted on 16 Nov 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITY PARIS SUD 11 - ORSAY

P H D T H E S I S

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 ASTSATRYAN
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 unrelevant 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

4 Enabling XML Update Optimization ... 55
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 *NoMerge* . . . 71
4.3.2 Procedure *OlbMerge* . . . 75
4.4 Implementation and Experiments . . . 80
4.4.1 Implementation issues . . . 80
4.4.2 Experiments . . . 87
4.5 Conclusion . . . 104

5 Extending the Type Projection based evaluation... 117
5.1 Introduction . . . 117
5.2 Extending the Type Projector for Update Optimization . . . 120
5.2.1 Case analysis: update operation in isolation . . . 120
5.2.2 Case analysis: mixing update operations of different kinds . . . 128
5.3 Definition of the Extended Projection . . . 146
5.3.1 Merge . . . 148
5.3.2 Function *TreeMerge* - one projector component at a time - . . . 149
5.3.3 Function *TreeMerge* - general case - . . . 159
5.3.4 Conclusion . . . 164

6 Conclusion 167

References 169

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 π
- 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 π 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 issuses.

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

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

Organization 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 $Merge$ algorithm through examples. The last section of the Chapter reports the implementation and experiments of the $Merge$ algorithm.
Chapter 5 introduces the extension of the method both for the $three{-}level$ type projector and $Merge$ 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 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 $\mathbf{i}$, $\mathbf{j}$ ...; ()

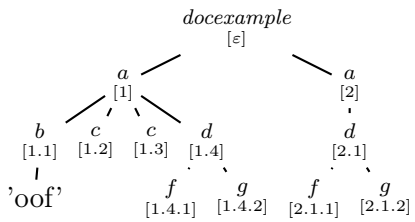


Figure 2.2: Tree representation t of *docexample.xml*

denotes the empty id-seq; $I{\cdot}I'$ denotes id-seq composition, and the intersection of I and J preserving the order in the id-seq I is denoted by $I_{|J}$.

A *store* σ over the id-set I is a mapping associating each identifier $\mathbf{i}{\in}I$ 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:
– $lab(\mathbf{i}){=}a$ if $\sigma(\mathbf{i}){=}a[J]$, and $lab(\mathbf{i}){=}String$ if $\sigma(\mathbf{i}){=}text[st]$,
– $child(\sigma, K){=}\{\mathbf{j} \mid \exists \mathbf{i}{\in}K, \sigma(\mathbf{i}){=}a[J]\ and\ \mathbf{j}{\in}J\}$,
– $roots(\sigma){=}\{\mathbf{i} \mid \neg\exists \mathbf{j}, \mathbf{i}{\in}child(\sigma, \{\mathbf{j}\})\}$.

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 ε. A node of a document t whose identifier is $\mathbf{i}$ is next denoted by $t@\mathbf{i}$.

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

Example 2.1.1. The document t of Figure 2.2 is a store σ.
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 $\sigma(\varepsilon){=}doc[1, 2]$, $\sigma(1){=}a[1.1, 1.2, 1.3, 1.4]$, $\sigma(1.4){=}d[1.4.1, 1.4.2]$, $\sigma(1.1){=}text[oof]$, $\sigma(2){=}a[2.1]$, $\sigma(2.1){=}d[2.1.1, 2.1.2]$.
We have that $lab(\varepsilon){=}docexample$, $lab(1){=}a$, $lab(2){=}a$, $lab(1.1){=}b$, $lab(1.2){=}c$, $lab(1.3){=}c$, $lab(1.4){=}d$ 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, σ) where σ is as above and $J{=}roots(\sigma)$. We write $dom(f)$ for I and σ_f for σ and $f{\circ}f'$ for the concatenation of two disjoint forests f and f'.

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 σ 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,

denoted $subfor(t)$, is defined by $\Pi_{I \setminus \{roots(t)\}}(t)$.

For the sake of simplicity we often use t in place of σ_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:
<!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.

```
<!DOCTYPE docexample[
    <!ELEMENT docexample (a*)>
    <!ELEMENT a (b*, c*, d?)>
    <!ELEMENT b (#PCDATA)>
    <!ELEMENT c (#PCDATA)>
    <!ELEMENT d ( f | g)*>
    <!ELEMENT f ( EMPTY)>
    <!ELEMENT g ( EMPTY)>

]>
```

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:

<!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.

<!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.

doc	$\rightarrow$	*a*,e**
a	$\rightarrow$	*b*,c*,e*,d?*
b	$\rightarrow$	*String*
d	$\rightarrow$	$(f \mid g)*$

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 Σ, and the reserved symbol *String*, a DTD over Σ is a tuple (D, s_D) where D is a total function from Σ 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 Σ occurring in r, is denoted by $\mathcal{L}(r)$, 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 Σ: doc$\rightarrow$a*,e* ; a$\rightarrow$b*,c*,e*,d? etc, where lab(ε)=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 ϵ is an empty regular expression.
Note that Σ contains all the labels occurring in the XML document t in Figure 2.2. □

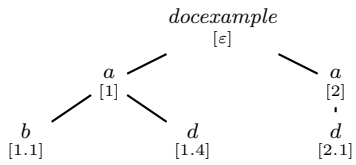


Figure 2.5: The projection t' of t

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

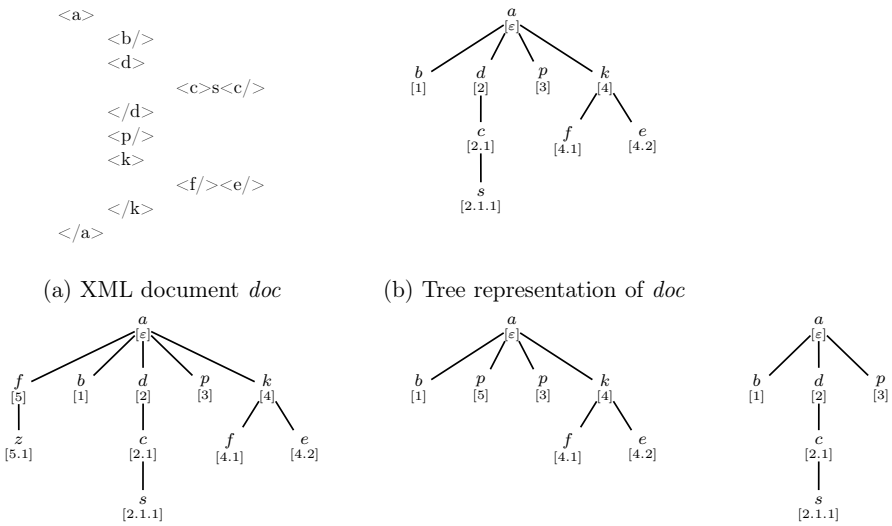
- insertion of new XML fragments,

SimpleUpdate ::= **InsertExpr** | **DeleteExpr** | **ReplaceExpr**

InsertExpr ::= *"insert"* (*"node"* | *"nodes"*)
SourceExpr InsertExprTargetChoice TargetExpr

InsertExprTargetChoice ::= *"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



(a) XML document *doc* (b) Tree representation of *doc*

(c) insert (d) replace (f) delete

Figure 2.7: Simple updates execution

- 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**)$^+$
WhereClause? **SimpleUpdate**$^+$
ConditionalUpdate ::= *"update"* *"if"*(**XQueryExpr** | *"then"*)$^+$
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:
insert
`<f><z/></f>`
as first into *doc(doc.xml)/a*.

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:
replace
doc(doc.xml)/d
with *doc(doc.xml)/p*.

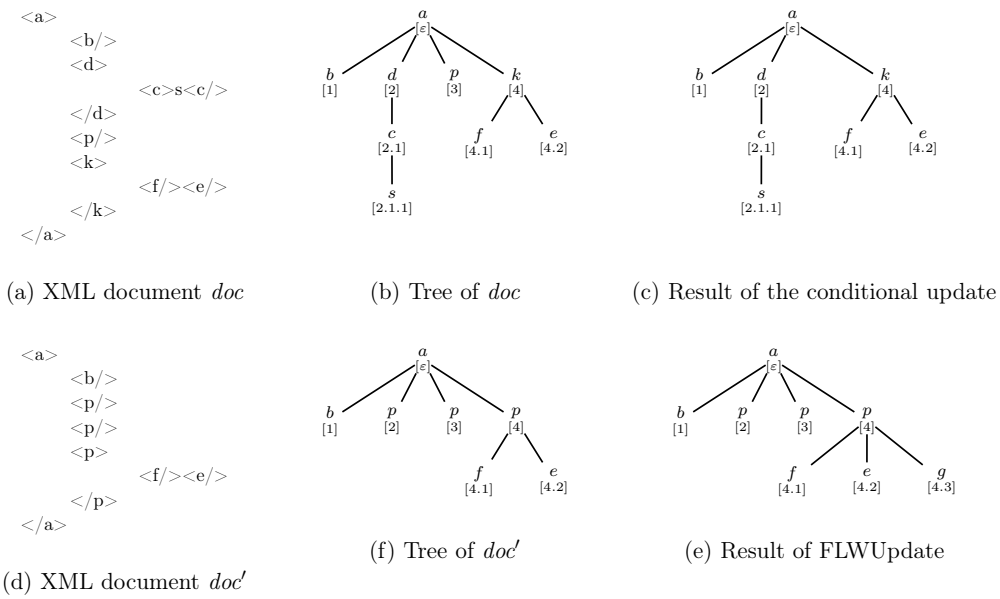
Figure 2.7-(f) illustrates the updated *doc* after the execution of the simple update expression su_3 specified by:
delete
doc(doc.xml)/k.
This expression deletes the last element of the a-node.

2.2.2 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:
update
if *empty doc*(doc.*xml)/p/f*
then



(a) XML document *doc* (b) Tree of *doc* (c) Result of the conditional update

(d) XML document *doc′* (f) Tree of *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 cu_1 over the document *doc* is illustrated in Figure 2.9-(c).

FLWUpdate expression used to apply simple updates throughout iterations. Fo instance, let us consider the FLWUpdate statement cu_2 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 cu_2 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=($roots(t)$, σ), where the store σ is over I, the $target variable is bound to a node having identifier $\mathbf{i} \in I$, 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.

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.

Chapter 3

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].

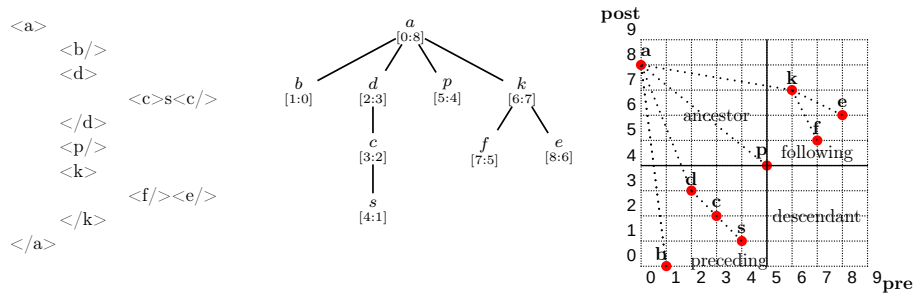
It is worth noticing that we can consider MonetDB/XQuery both as XML-enabled DBMS and as native XML database, since the XML documents are mapped into a relational representation. From a technical viewpoint, MonetDB/XQuery is an XML database "implemented on the top of a relational storage". As described 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



(a) XML document *doc*

(b) Tree representation with pre/post numbers

(c)XPath axe /a/p in the pre/post plane

pre	size	level	kind	prop
0	8	0	elem	a
1	0	1	elem	b
2	2	1	elem	d
3	1	2	elem	c
4	0	3	text	s
5	0	1	elem	p
6	2	1	elem	k
7	0	2	elem	f
8	0	2	elem	e

(d) Relational storage

Figure 3.1: XML encoding used in 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 pre-order and post-order traversal ranks, is used to encode the XML tree structure. In *pre/post* encoding the *pre* value describes the pre-order 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*, bellow 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 $\mathtt{n}_a.pre$ property is set to 0. When the next child $\mathtt{n}_b$ labelled by b is parsed, the *pre* value is incremented and thus $\mathtt{n}_b.pre$ 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 *n*ode "s" it is true that: $\mathtt{n}_s.pre{=}4$.

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 $\mathtt{n}_a.size$ is equal to 8. For the first child of the a-root we have that $\mathtt{n}_b.size{=}0$: it has no children; while for the k-node *size* is equal to 2. For the text node $\mathtt{n}_s$ 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, $\mathtt{n}_a.level{=}0$, $\mathtt{n}_b.level{=}1$ and $\mathtt{n}_s.level{=}3$.

Some additional properties of a node are stored in *doc*, like:

kind - The kind of a node $\mathtt{n}$ is either an element, a text or an attribute. For the a-root we have that $\mathtt{n}_a.kind{=}elem$, while for the text node $\mathtt{n}_s.kind{=}text$.

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.

Axis	Relational characterization
ancestor($\mathtt{n}$,$\mathtt{n}'$)	$\mathtt{n}.pre < \mathtt{n}'.pre\ AND\ \mathtt{n}'.pre \leq \mathtt{n}.pre + \mathtt{n}.size$
descendant($\mathtt{n}$,$\mathtt{n}'$)	$axis(\mathtt{n}',\ \mathtt{n},\ descendant)\ AND\ \mathtt{n}.level = \mathtt{n}'.level + 1$
child($\mathtt{n}$,$\mathtt{n}'$)	$\mathtt{n}.pre < \mathtt{n}'.pre\ AND\ \mathtt{n}'.pre \leq \mathtt{n}.pre + \mathtt{n}.size$
following($\mathtt{n}$,$\mathtt{n}'$)	$\mathtt{n}.pre > \mathtt{n}'.pre + \mathtt{n}'.size$
preceding($\mathtt{n}$,$\mathtt{n}'$)	$\mathtt{n}.pre + \mathtt{n}.size < \mathtt{n}'.pre$

Table 3.1: Relational characterization

Axes Relationships For instance, given a tree t and two nodes $\mathtt{n}_1$ and $\mathtt{n}_2$ it is true that:

$$\mathtt{n}_1{\in}\mathtt{n}_2/ancestor \Leftrightarrow \mathtt{n}_1.pre{<}\mathtt{n}_2.pre \quad AND \quad \mathtt{n}_2.pre{\leq}\mathtt{n}_1.pre{+}\mathtt{n}_1.size$$

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($\mathtt{n}$,$\mathtt{n}'$) from the table 3.1, where we have:

$$\begin{aligned} &k.pre{=}6, k.size{=}2, f.pre{=}7 \\ &k.pre < f.pre\ AND\ f.pre \leq k.pre{+}k.size \end{aligned} \tag{3.1}$$

$$\begin{aligned} &k.pre{=}6, k.size{=}2, e.pre{=}8 \\ &k.pre < e.pre\ AND\ e.pre \leq k.pre{+}k.size \end{aligned} \tag{3.2}$$

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 ($\mathtt{n}_a.pre$) works by

checking the first child: $\mathtt{n}_b.pre$=$\mathtt{n}_a.pre$+1=1; then skipping to its siblings: $\mathtt{n}_d$=$\mathtt{n}_b$+$size$[$\mathtt{n}_b.pre$]+1=1+0+1=2 (d-node) until the last child a-node is reached: ($\mathtt{n}_a.pre$+size[$\mathtt{n}_a.pre$]).

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 *pre* values starting from the inserted point. The second issue is that the *size* values of the ancestors of the inserted node must be recalculated.

Figure 3.2 illustrates an example how the *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 *pre* values of all *following* nodes after the insert point must be changed, as well as, the *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 *pre* values of the *following* nodes of p-node must be re-calculated: *pre* value of the k-node is changed to 8, of f-node to 9 and for e-node to 10. All *pre* values of the *followings* are augmented by two (two elements have been inserted). The next change must be applied on the *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 $\mathtt{n}_a.size$ 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

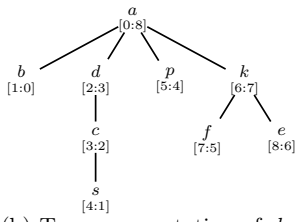
To support the updatable representation of the XML Encoding in [20] the following changes on the table *pre/size/level* have been proposed:

1 - The table is called *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.

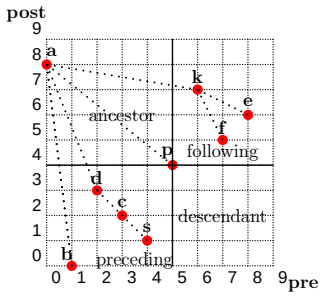
4 - New logical pages are appended at the end.



(b) Tree representation of *doc*

pre	size	level	prop
0	8	0	a
1	0	1	b
2	2	1	d
3	1	2	c
4	0	3	s
5	0	1	p
6	2	1	k
7	0	2	f
8	0	2	e

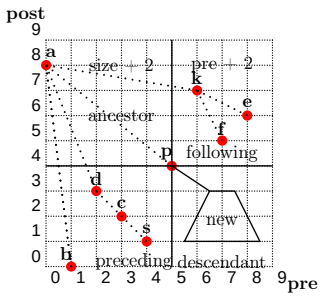
(a) Relational storage before the structural update



(b) Tree representation with pre/post numbers before

pre	size	level	prop
0	10	0	a
1	0	1	b
2	2	1	d
3	1	2	c
4	0	3	s
5	2	1	p
6	0	2	n
7	0	2	m
8	2	1	k
9	0	2	f
10	0	2	e

(c) Relational storage after the structural update



(d) Tree representation with pre/post numbers after

Figure 3.2: The impact of structural updates on pre/size/level XML storage

	rid	size	level	prop
page 0	0	**10**	0	a
	1	0	1	b
	2	2	1	d
	3	1	2	c
	4	0	3	s
	5	**2**	1	p
	6	1	null	
	7	0	null	
page 1	8	2	1	k
	9	0	2	f
	10	0	2	e
	11	4	null	
	12	3	null	
	13	2	null	
	14	1	null	
	15	0	null	

(a) before the update

	rid	size	level	prop
page 0	0	**10**	0	a
	1	0	1	b
	2	2	1	d
	3	1	2	c
	4	0	3	s
	5	**2**	1	p
	6	**0**	**2**	**n**
	7	**0**	**2**	**m**
page 1	8	2	1	k
	9	0	2	f
	10	0	2	e
	11	4	null	
	12	3	null	
	13	2	null	
	14	1	null	
	15	0	null	

(b) after the update

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

page 0 — page 1

rid	size	level	prop
0	**10**	0	a
1	0	1	b
2	2	1	d
3	1	2	c
4	0	3	s
5	**2**	1	p
6	1	null	
7	0	null	
8	2	1	k
9	0	2	f
10	0	2	e
11	4	null	
12	3	null	
13	2	null	
14	1	null	
15	0	null	

(a) before the update

page 0 — page 1 — new page 2

rid	size	level	prop
0	**16**	0	a
1	0	1	b
2	2	1	d
3	1	2	c
4	0	3	s
5	**10**	1	p
6	0	2	n
7	0	2	m
8	2	1	k
9	0	2	f
10	0	2	e
11	4	null	
12	3	null	
13	2	null	
14	1	null	
15	0	null	
16	0	2	l
17	6	null	
18	5	null	
19	4	null	
20	3	null	
21	2	null	
22	1	null	
23	0	null	

(b) after the update

Figure 3.4: Insert with a new logical page insertion

page has certain percentage of tuples stored as "unused". For our example we keep at least two unused tuples (see the tuples colored gray). `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

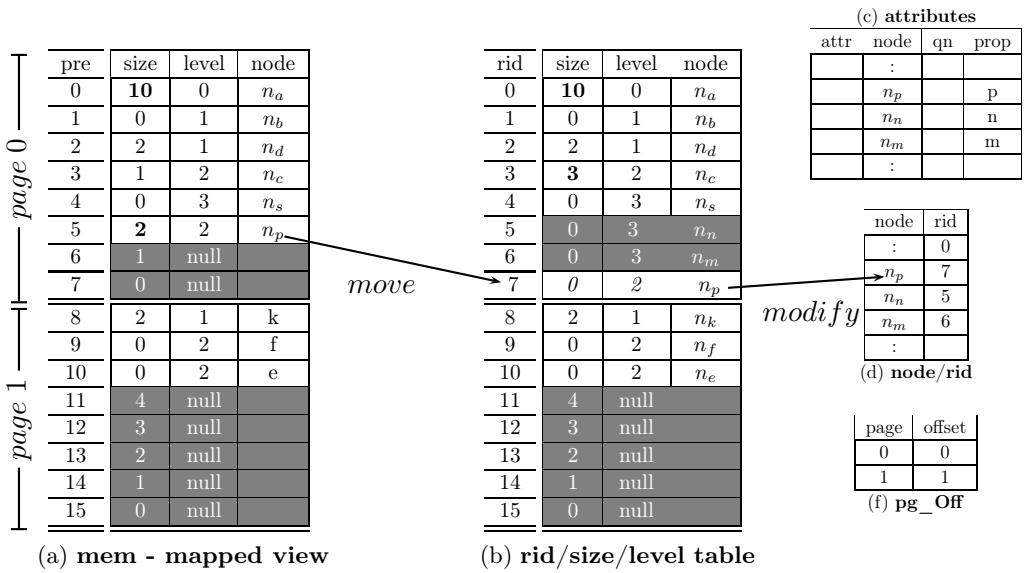


Figure 3.5: Update schema (within a page)

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 rid=6 and 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: $\mathtt{n}_l.size$=0, $\mathtt{n}_l.level$=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_Off and $node/rid$ tables.

- **pg_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_Off and $node/rid$ tables, for two update scenarios.

Example 3.2.5. *Insert within the page:* pg_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

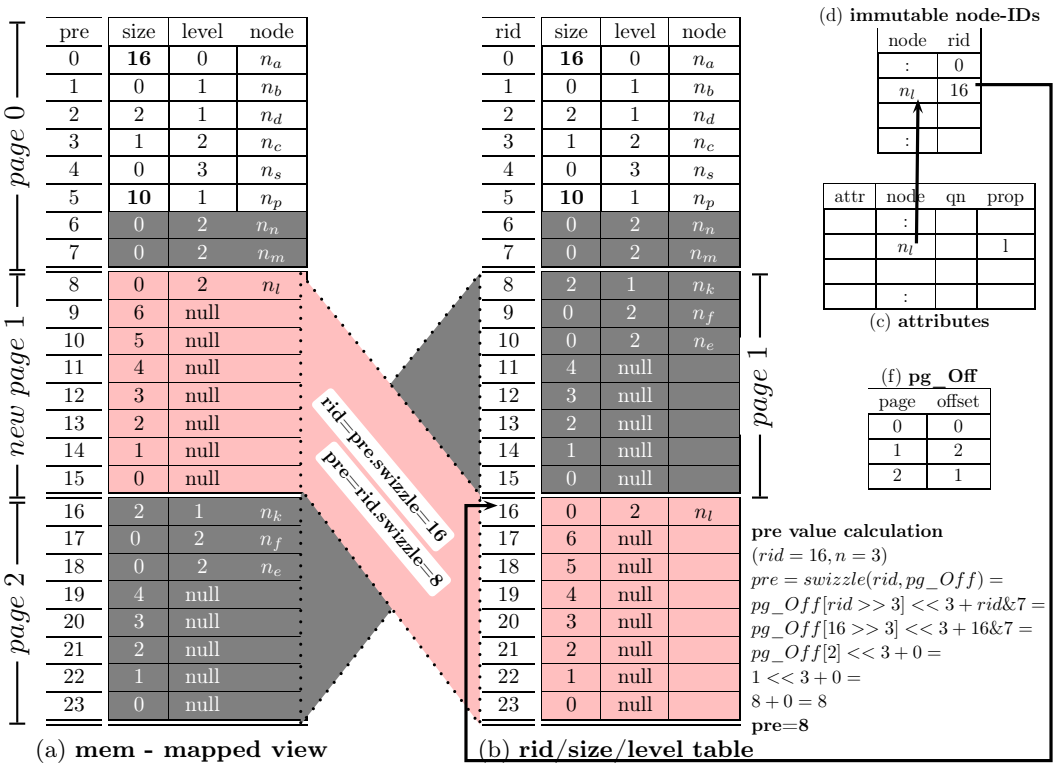


Figure 3.6: Update schema (with a new page insertion)

moved further in the page: new $p.rid$=7. Second, the new rid value of the p-node is modified in the $node/rid$ table (see fig. 3.5-(d)). Finally, newly inserted nodes labelled n and m are stored into the $rid/size/level$ (see the italic lines in the *page* 0 and in the *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 $node/rid$ table by searching for the $node$ values of the entries where rid=$null$. In the case that we do not find any node number, new tuples are appended to the $node/rid$ table and new $node$ values are assigned to them in $rid/size/level$.

Example 3.2.6. *Insert with new logical page insertion: pg_Off and $node/rid$.*

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 $rid/size/level$. Next, a new entry for the *page* 2 is appended to the pg_Off table, and the offsets of all pages after the insert point are incremented (see fig. 3.6-(f)). The $size$ values of the *ancestor* a-node is set to 16. Finally the $node$ and the rid values of newly inserted elements are added to the $node/rid$ table.

It is worth noticing that when we said that a new logical page is appended to $rid/size/level$ it is appended to the end, while for the $pre/size/level$ 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:

$$pre{=}pg_Off[rid >> 3] << 3{+}rid\ \&00000111; \tag{3.3}$$

As it is illustrated in Figure 3.6, the *pre* value of the *l*-node is calculated as followes:
The *swizzle* procedure takes as an input two parameters: the *rid* of the node and the pg_Off table. We have that $l.rid$=16 and n=3 hence:
$l.pre{=}swizzle(l.rid, pg_Off){=}pg_Off[rid >> 3] << 3{+}rid\&7$.

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 $l.rid$=16 we have that:

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

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 $l.pre$=1 << 3+0=8+0=8 hence the $l.pre$=8.

To calculate *rid* from *pre* we use the following formula:

$$rid{=}pre\&00000111{+}pg_Off[(pre\ \&11111000) >> 3] << 3; \tag{3.4}$$

The $l.rid$=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

	rid	size	level	prop
page 0	0	**10**	0	a
	1	0	1	b
	2	2	1	d
	3	1	2	c
	4	0	3	s
	5	**2**	1	p
	6	1	null	
	7	0	null	
page 1	8	2	1	k
	9	0	2	f
	10	0	2	e
	11	4	null	
	12	3	null	
	13	2	null	
	14	1	null	
	15	0	null	

(a) without projection

	rid	size	level	prop
page 0	0	7	0	a
	1	6	1	p
	2	5	null	
	3	4	null	
	4	3	null	
	5	2	null	
	6	1	null	
	7	0	null	
page 1				

(b) before the update

	rid	size	level	prop
page 0	0	7	0	a
	1	6	1	p
	2	0	2	n
	3	0	2	m
	4	3	2	l
	5	2	null	
	6	1	null	
	7	0	null	
page 1				

(c) after the update

Figure 3.7: Relational storage with projection

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 moves 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, diffrently 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

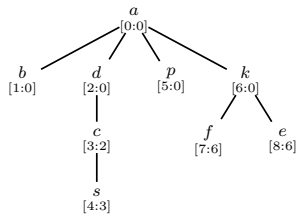
Example 3.3.1. *Document encoding in BaseX (pre/par encoding).*
Figure 3.8-(a) shows the XML document *doc*. (We choose the *doc* from the example 3.2.1, to make the BaseX storage comparable to the MonetDB/XQuery). Figure 3.8-(b) shows the tree of *doc* assigned with *pre* values. Important to note that, 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 $\mathtt{n}_a.pre{=}\mathtt{n}_b.par$

```
<a>
    <b/>
    <d>
        <c>s<c/>
    </d>
    <p/>
    <k>
        <f/><e/>
    </k>
</a>
```

(a) XML document



(b) Tree with *pre/par* values

pre	par	token	kind	att	attVal
0	0	a	elem		
1	0	b	elem		
2	0	d	elem		
3	2	c	elem		
4	3	s	text		
5	0	p	elem		
6	0	k	elem		
7	6	f	elem		
8	6	e	elem		

(c) Relational storage

id	tag
....0000	a
....0001	b
....0010	d
....0011	c
....0100	p
....	...

TagIndex

id	tag
....0000	s
....	...

TextIndex

par	kind/token	attribute
...0000	0.....0000	null
...0000	0.....0001	null
...0000	0.....0010	null
...0010	0.....0011	null
...0011	1.....0000	null
...0000	0.....0100	null

id	att
....	...

attNameIndex

id	attVal
....	...

attValIndex

(d) **Internal representation**

Figure 3.8: XML encoding in BaseX (pre/par)

- **token** - This value represents a tag name or a text contend. 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 give, 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 $\mathtt{n}_b.par$=.....0000(0 *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 $\mathtt{n}_s.par$=.....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).*

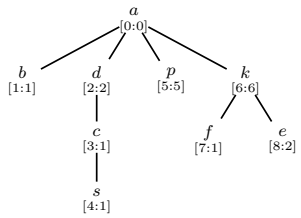
The *doc* is mapped into *pre/dist/size* where:

- **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), $\mathtt{n}_a.pre$=0, $\mathtt{n}_b.pre$=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 $\mathtt{n}_s$ we do not store the *size* value.

```
<a>
    <b/>
    <d>
            <c>s<c/>
    </d>
    <p/>
    <k>
            <f/><e/>
    </k>
</a>
```

(a) XML document



(b) Tree with *pre/dist* values

id	tag
1	a
2	b
3	d
4	c
5	p
6	k
7	f
8	e

TagIndex

id	tag
00	s

Text

id	Kind
0	doc
1	elem
2	text
3	att
4	com
5	pi

Kind

id	att
....	...

attName

id	attVal
....	...

attVal

pre	dist	size	kind	tag	txt	atS	atN	atV
0	0	8	elem	a				
1	1	0	elem	b				
2	2	2	elem	d				
3	1	1	elem	c				
4	1		text		s			
5	5	0	elem	p				
6	6	2	elem	k				
7	1	0	elem	f				
8	2	0	elem	e				

(c) Relational storage

	kind	tag	attS/Size/txt/attVal	dist
0000	01	0001	00 00 00 00 08	00 00 00 00
0010	01	0002	00 00 00 00 00	00 00 00 01
0020	01	0003	00 00 00 00 02	00 00 00 02
0030	01	0004	00 00 00 00 01	00 00 00 01
0040	02	0000	00 00 00 00 00	00 00 00 01
0050	01	0005	00 00 00 00 00	00 00 00 05
0060	01	0006	00 00 00 00 02	00 00 00 06
0070	01	0007	00 00 00 00 00	00 00 00 01
0080	02	0008	00 00 00 00 00	00 00 00 02

(d) **Internal representation**

Figure 3.9: XML encoding in BaseX (*pre/dist/size*)

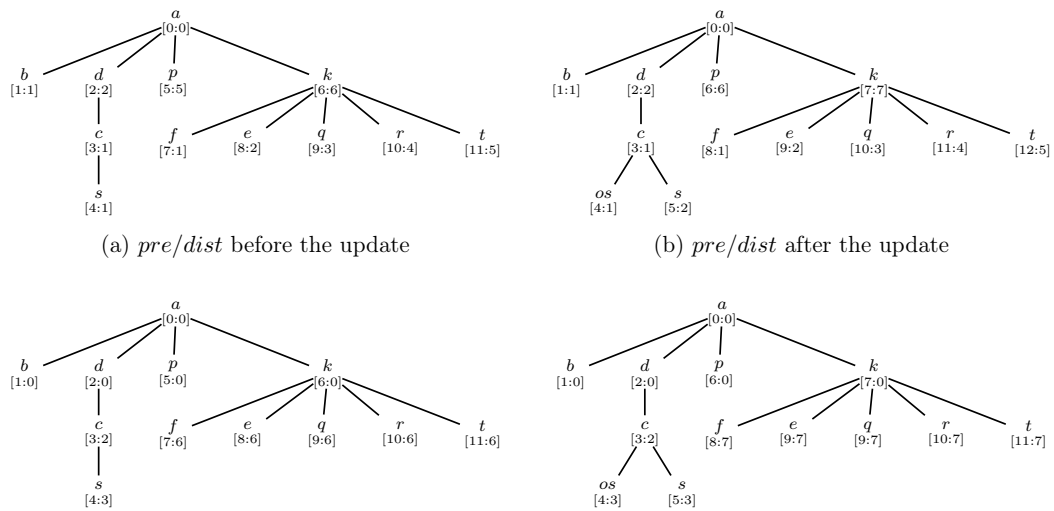
- **kind** - This value preserves six different node kinds. For instance, $\mathtt{n}_b.kind$=01 and $\mathtt{n}.s.kind$=2. All six kinds are stored in the $Kind$ table.
- **dist** - This value stores the distance to the parent node, allowing access to parents and ancestors. The *dist* value for the 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: $\mathtt{n}_p.dist$=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, $\mathtt{n}_a.tag$=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 $\mathtt{n}_s.txt$=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 give, 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 - $\mathtt{n}'$ is a parent of $\mathtt{n}$ if $\mathtt{n}'.pre$=$\mathtt{n}.pre-\mathtt{n}.dist$. For our example we have that d-node is a parent of c-node since $\mathtt{n}_d.pre$=$\mathtt{n}_c.pre-\mathtt{n}_c.size$=3−1

ancestors - The *ancestors* step is evaluated using $\mathtt{n}'.pre$=$\mathtt{n}.pre-\mathtt{n}.dist$ to access next parent $\mathtt{n}'$ and traversal is completed if $\mathtt{n}'$=*null*.
The evaluation procedure:
foreach c **in** *context* **do**
  $n \leftarrow node(c.pre-c.dist)$
  while $n! = null$ **do**
    add n to *results*
    $n \leftarrow node(n.pre-n.dist)$
return ordered *results* without duplicates.



(a) *pre/dist* before the update

(b) *pre/dist* after the update

(c) *pre/par* before the update

(d) *pre/par* after the update

Figure 3.10: *dist* value recalculation

next sibling - $\mathtt{n}'$ is a sibling of $\mathtt{n}$ if $pre(\mathtt{n})+size(\mathtt{n})+1$. For our example, h-node is a next sibling of d-node since $pre(\mathtt{n}_p)+size(\mathtt{n}_p)=5+0+1=6$.

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

foreach c **in** *context* **do**
  $n \leftarrow node(c.pre+1)$
  while $n.pre! = c.pre+c.size$ **do**
    add n to *result*
    $n \leftarrow node(n.pre+n.size+1)$
return ordered *results* without duplicates.

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

foreach c **in** *context* **do**
  foreach n **in** $nodes(n.pre=c.pre;n.pre=c.pre+c.size-1)$ **do**
    add n to *results*
return 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 *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: $\mathtt{n}_s.dist$=2, $\mathtt{n}_p.dist$=6 and $\mathtt{n}_k.dist$=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

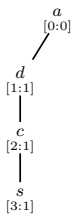
In BaseX the encoded XML document stored in the current database can be "exported" to the specified path. Similarly to MonetDB/XQuery, the implementation (Java classes: *Export*, *XMLSerialazer* and *Serializer*) scans the arrays where the nodes are stored and for each node it outputs the tags, the attributes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

3.3.5 BaseX vs. projection

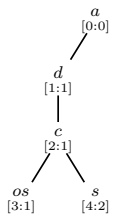
Figure 3.11 illustrates the difference between the update insertion using "pure" BaseX vs. using projection. An update specified by an insertion of a new text node "*os*" as the first child of the c-node. As it has been explained in Figure 3.10-(b) the insertion of the new node results in the recalculation 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: $\mathtt{n}_s.dist$=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

```
<a>
    <b/>
    <d>
            <c>s<c/>
    </d>
    <p/>
    <k>
            <f/><e/>
    </k>
</a>
```

(a) XML document



(b) projection with *pre/dist/size* values



(c) after insertion

	kind	tag	...
0000	01	0001	...
0010	01	0002	...
0020	**01**	**0003**	**...**
0030	01	0004	...
0040	02	0000	...
0050	01	0005	...
0060	01	0006	...
0070	01	0007	...
0080	02	0008	...

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

	kind	tag	...
0000	01	0001	...
0010	01	0002	...
0030	01	0004	...
0040	02	0000	...
0050	01	0005	...
0060	01	0006	...
0070	01	0007	...
0080	02	0008	...

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

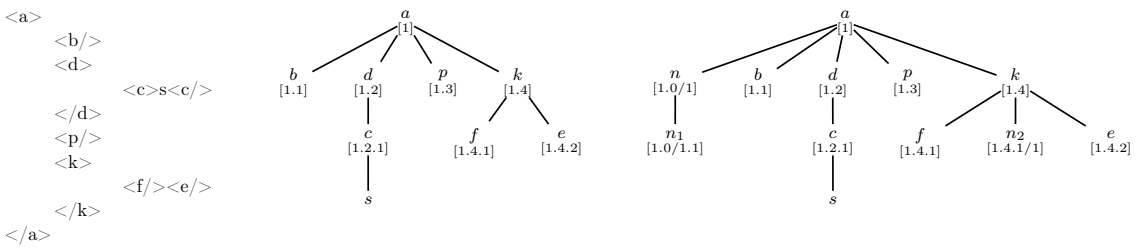
	kind	tag	...
0000	01	0001	...
0010	**01**	**0002**	**...**

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

	kind	tag	...
0000	01	0001	...

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

Figure 3.11: Optimazations in BaseX using projection



(a) XML document *doc* (b) Tree assigned IDs (c) Updated tree

Figure 3.12: XML encoding in eXist (DLN)

us assume that an update specified by a deletion of the d-node is preformed 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: $\mathtt{n}_d.parentID.\mathtt{n}_c.level$=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

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.

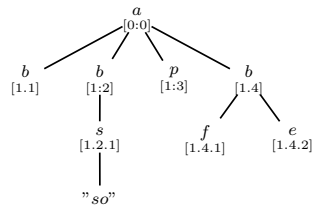
Next, we cover the general data structure used in eXist.

3.4.1 General Data Structure

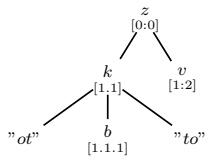
The following example illustrates the data storage architecture used in eXist.

Example 3.4.2. *Data storage architecture.*

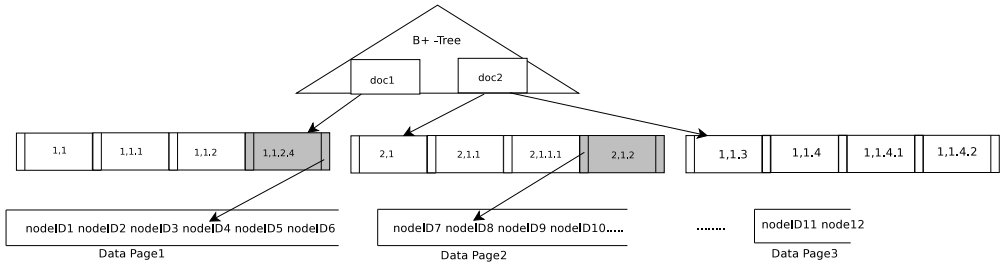
First of all it is important to note that, in eXist documents are managed in hierarchical collections, similar to storing files in a file system. This collection hierarchy is managed by the *collections.dbx*. Each collection has unique identifier ID.
The next important feature of this architecture is 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 $B+tree$. $B+tree$ 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



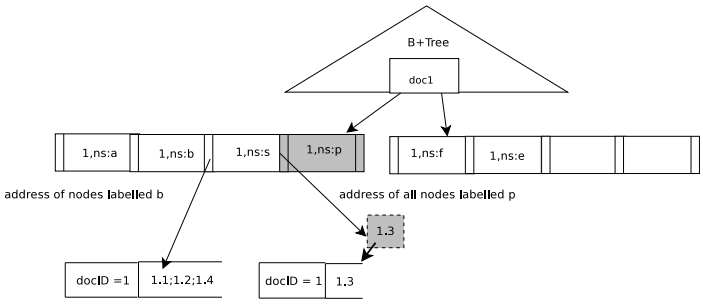
(a) doc_1



(b) doc_2



(c) *dom.dbx*



(c) *elements.dbx*

Figure 3.13: Data storage in 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 $nodeID_8$ 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 identifyer 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 leafs of $B+tree$ are pointing to the array values containing an ordered list of $nodeID$s separated by $docId$s. 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 $NodeID$s 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.

Set A	Set B	Join Result Set
1	1.1 1.2 1.4	1.1 1.2 1.4

(a) result for a//b

Set a//b	Set b[s]	Join Result Set
1.1 1.2 1.4	1.2.1	1.2.1

(b) following expression

Figure 3.14: Sets

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 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.

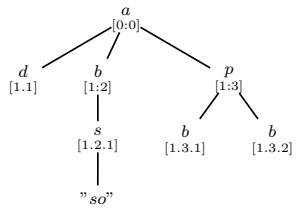
s="so" - The engine retrieves all nodes having keyword "so".

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

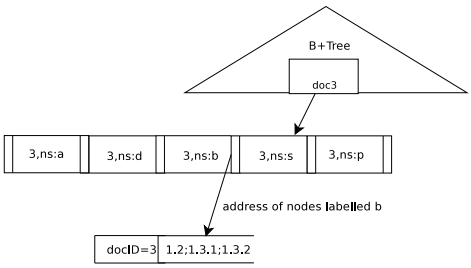
Then the engine executes structural join on the first two node sets, to find all nodes from the b set being descendants nodes in the a node set. As a result of this join a new set is created which serves as the context node set for the following expression. Therefore, this new node set for expression $a//b$ 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

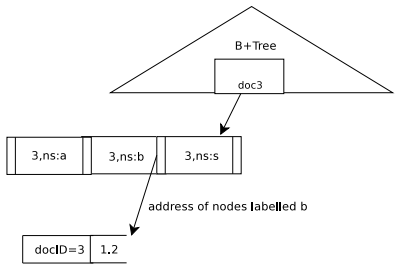
To serialize a document stored in the database eXist uses several Java classes. The *Serializer* base class, used to serialize a document or document fragment back to XML. This class offers two overloaded methods: $serialize()$ and $toSAX().serialize()$. The first one returns the XML as a string, the second one generates a stream of SAX events.



(a) doc_3



(c) *elements.dbx* without the Projection



(c) *elements.dbx* with projection

Set A	Set B	Join Result Set
1	1.2	1.2
	1.3.1	1.3.1
	1.3.2	1.3.2

(d) Sets selection without the Projection

Set A	Set B	Join Result Set
1	1.2	1.2

(f) Sets selection with projection

Figure 3.15: Data storage in eXist with and without projection

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 doc_3 (see fig. 3.15-(a)) stored in *element.dbx*, while Figure 3.15-(c) stores the projected doc_3. 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 $nodeID$s 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 doc_3 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:

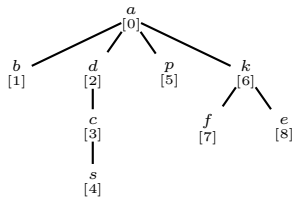
Example 3.5.1. *TinyTree array.*

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 $\mathtt{n}_1$. 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 l 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.



XML document *doc*

Array Name — Array Values

name kind	1	1	1	1	3	1	1	1	1
next sibling		2	5			6		8	
alpha	-1	- 1	-1	-1	0	-1	-1	-1	-1
beta	-1	-1	-1	-1	1	-1	-1	-1	-1
depth	0	1	1	2		1	1	2	2
node code	n_1	n_2	n_3	n_4		n_5	n_6	n_7	n_8
array Indexes	0	1	2	3	4	5	6	7	8

(a) TinyTree Arrays

0	"s"
index	text value

(b) StringBuffer

name code	URI	Local Name	Prefix
n_1		a	
n_2		b	
n_3		d	
n_4		c	
n_5		p	
n_6		k	
n_7		f	
n_8		e	

(c) NamePool

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 *mutabe 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: $\mathtt{n}_b$, $\mathtt{n}_d$, $\mathtt{n}_p$ and $\mathtt{n}_k$. 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*=$\mathtt{n}_a$, 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 $\mathtt{n}_s$, where *value*="s", while *name*=*null*.

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

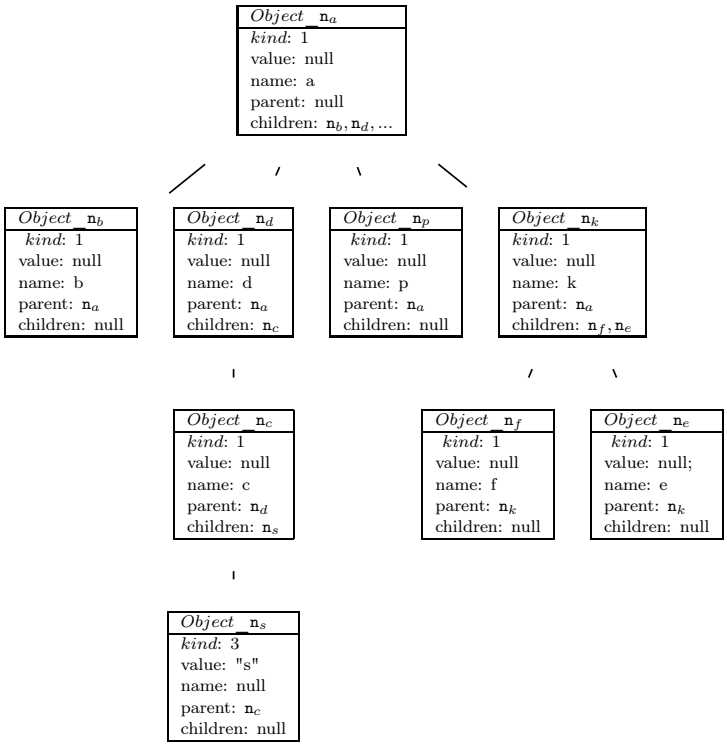


Figure 3.17: Mutable linked tree before update

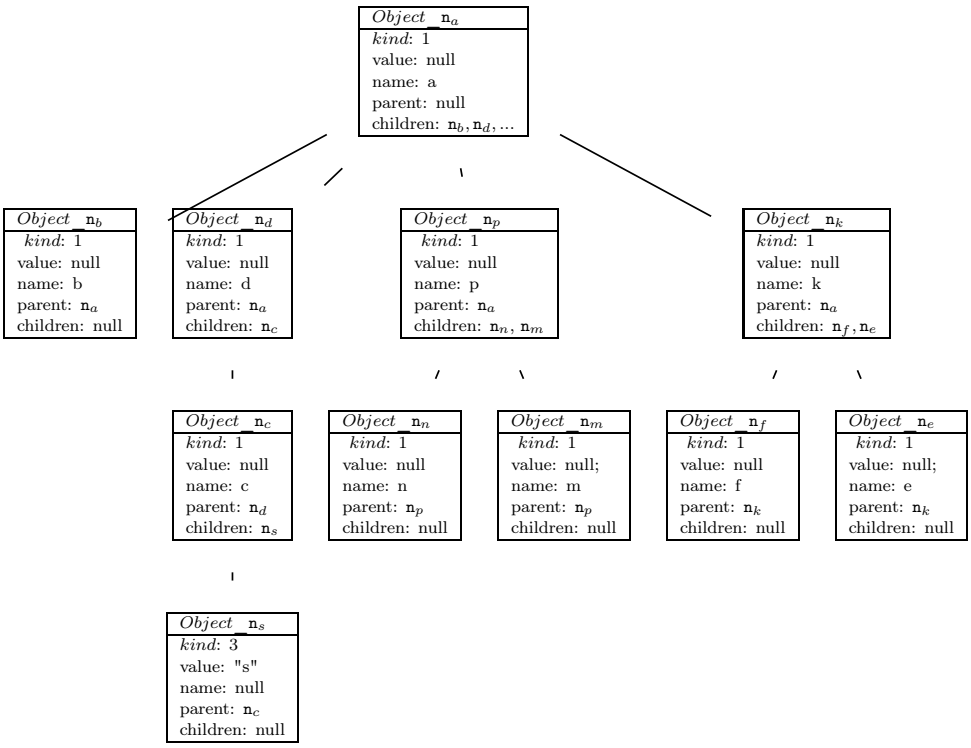


Figure 3.18: Updated mutable linked tree

The result of this update is illustrated in Figure 3.18. As the reader can observe, new Objects: $Object_\mathtt{n}_n$ and $Object_\mathtt{n}_m$ are created with *parent* variables containing the reference to $Object_\mathtt{n}_p$.

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 requeres 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 *NoMerge* 71
4.3.2 Procedure *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 *Merge* Algorithm.
In Section 4.2 we introduce some basic notions of the *three−level* type projector. In Section 4.3 we introduce the Merge algorithm together with its two procedures *NoMerge* and *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 unrelevant 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 π. 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 π 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:

Step1 - an update type projector π for u is inferred and t is projected wrt π. 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 `for $x in /doc/a where $x/d 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 ϵ 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 π.

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

doc	$\rightarrow$	*a*,e**
a	$\rightarrow$	*b*,c*,e*,d?*
b	$\rightarrow$	*String*
d	$\rightarrow$	$(f \mid g)*$

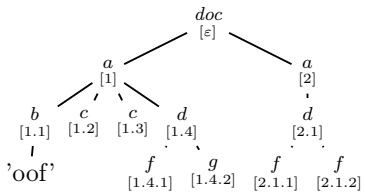
(1) The DTD D

```
for $x in /doc/a
where $x/d
return delete node $x/b
```

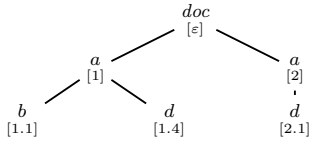
(2) The update u

$\pi_{\mathbf{no}}=\{doc, a, b, d\}$
$\pi_{\mathbf{olb}}=\pi_{\mathbf{eb}}=\emptyset$

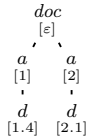
(3) The projector π for u



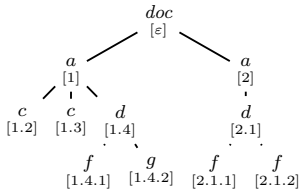
(4) The XML document t



(5) The projection t' of t wrt π



(6) The partial result $u(t')$



(7) The final result $u(t)$

Figure 4.1: A motivating example of the update scenario

whose identifier is **i** is denoted by $t@\mathbf{i}$. 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 π (see fig. 4.1-5). The projection first outputs node $t@\varepsilon$ labelled by $doc \in \pi_{\mathbf{no}}$. Next, it selects node $t@1$ labelled by $a \in \pi_{\mathbf{no}}$ and its children $t@1.1$ and $t@1.4$ labelled by $b \in \pi_{\mathbf{no}}$ and $d \in \pi_{\mathbf{no}}$. 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 $b \in \pi_{\mathbf{no}}$ 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 π 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 π (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* π, used to prune out the tree t is a 3-level projector which consists of components $\pi_{\mathbf{no}}$, $\pi_{\mathbf{olb}}$ and $\pi_{\mathbf{eb}}$, where **no** stands for "node only", **olb** stands for "one level below" and **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* π.

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 $\pi_{\mathbf{no}}$, 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 $\pi_{\mathbf{olb}}$, it is projected as well as its children. Each child will be visited and if its label belongs to π its children will be examined for projection *wrt* to the semantics of the projector,

- if a node is labelled by a type in $\pi_{\mathbf{eb}}$, 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 $\pi_{\mathbf{olb}}$ and $\pi_{\mathbf{eb}}$ 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 $\pi_{\mathbf{olb}}$ and $\pi_{\mathbf{eb}}$ and its label does not belong to π, then its children are not projected.

In the following paragraph, we provide examples motivating each component of π. First we provide example explaining the use of the $\pi_{\mathbf{no}}$ 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 $\pi_{\mathbf{no}}$, 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 $doc/a/d$ and the types of nodes traversed by this path are doc, a, d. The type

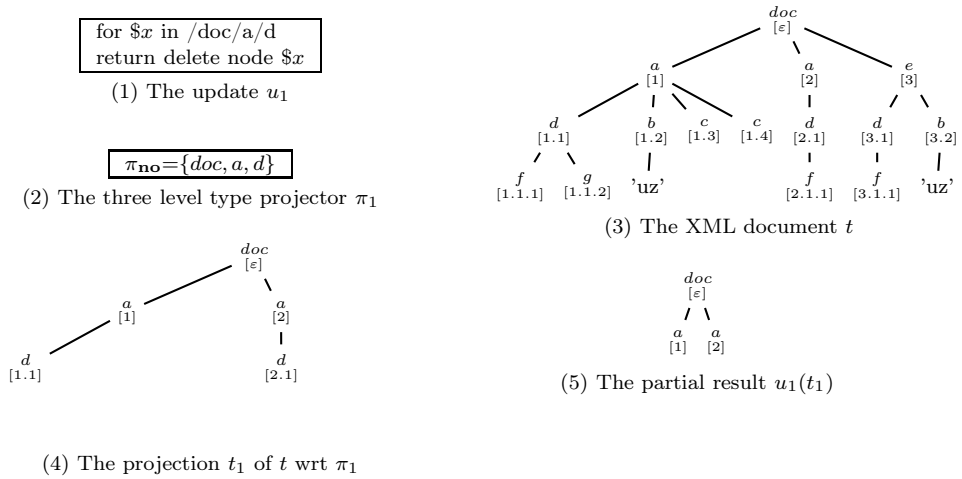


Figure 4.2: The projector component $\pi_{\mathbf{no}}$, illustrated for "delete"

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

projection - During projection only the nodes labelled by types contained in $\pi_{\mathbf{no}}$ are projected. In our example projection first outputs node $t@\varepsilon$ (labelled by $doc \in \pi_{\mathbf{no}}$) of the tree t (see fig. 4.2-3,4). Next, it selects node $t@1$ labelled by $a \in \pi_{\mathbf{no}}$ and node $t@1.1$ labelled by $d \in \pi_{\mathbf{no}}$. 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 $d \in \pi_{\mathbf{no}}$, since the parent node $t@3$ is labelled by $e \notin \pi_{\mathbf{no}}$ 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 $\pi_{\mathbf{no}}$ component (4.3-2).

projection - The projection outputs the following nodes: $t@\varepsilon$ labelled by $doc \in \pi_{\mathbf{no}}$, $t@1$ labelled by $a \in \pi_{\mathbf{no}}$ followed by $t@1.3$ and $t@1.4$ labelled by $c \in \pi_{\mathbf{no}}$, finally node $t@2$ labelled by $a \in \pi_{\mathbf{no}}$ (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 $e \notin \pi_{\mathbf{no}}$ thus not projected.

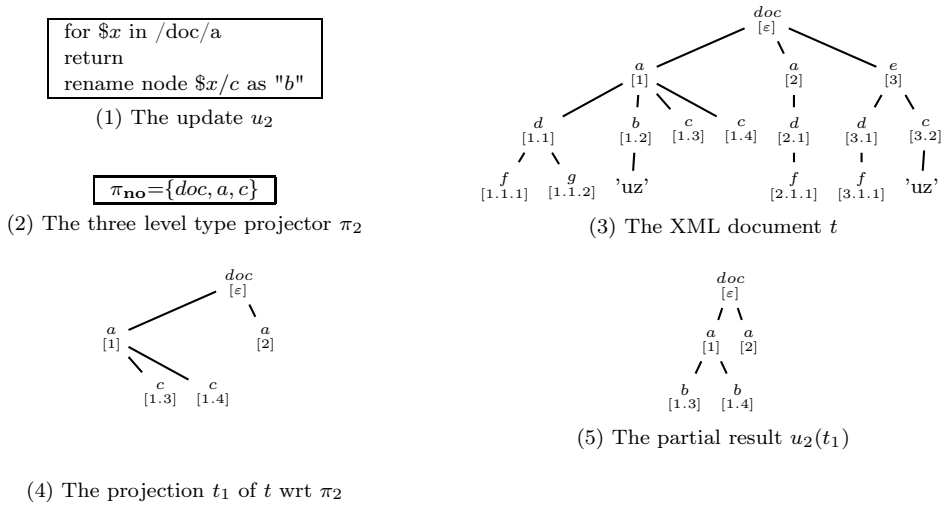


Figure 4.3: The projector component $\pi_{\mathbf{no}}$, illustrated for "rename"

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 $c \in \pi_{\mathbf{no}}$ 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 $\pi_{\mathbf{olb}}$ 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.

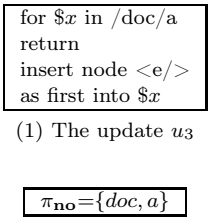
Example 4.2.3. The example given in Figure 4.4 motivates the need of the "one level below" component of the projector. Let us show that the "node only" projection is not adequate here by showing the whole scenario. We start with by treating the example using only $\pi_{\mathbf{no}}$ component, to show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

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

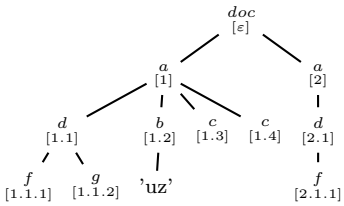
projection extraction - The update query u_3 involves an *"insert as first"* operation (see fig. 4.4-1). Intuitively, the path corresponding to data relevant for the update u_3 is doc/a and the types of nodes traversed by this path are doc, a. Thus, let us consider the projector containing one component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



(1) The update u_3

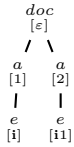


(3) The XML document t

$\pi_{\mathbf{no}}=\{doc, a\}$

(2) The three level type projector π_3

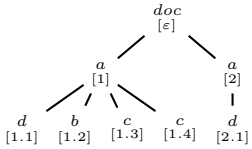
(4) The projection t_1 of t wrt to π_3



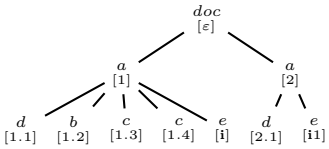
(5) The partial result t_3=$u_3(t_1)$

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

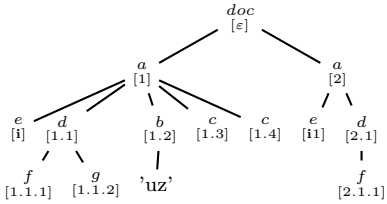
(6) The three level type projector π'_3



(7) The projection t_2 of t wrt π'_3



(8) The partial result t'_3=$u_3(t_2)$



(9) The final result

Figure 4.4: The projector component $\pi_{\mathbf{olb}}$, 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 $\pi_{\mathbf{olb}}$ (see fig. 4.4-6).

Introducing the $\pi_{\mathbf{olb}}$ 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 $\pi_{\mathbf{olb}}$ 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@ε followed by node t@1 labelled by $a{\in}\pi_{\mathbf{olb}}$ 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 **i**1 (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 $a{\in}\pi_{\mathbf{olb}}$ 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 π_4 has two components: the "node only" component $\pi_{\mathbf{no}}{=}\{doc\}$ and the "one level below" component $\pi_{\mathbf{olb}}{=}\{a\}$ (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@ε labelled by $doc{\in}\pi_{\mathbf{no}}$ followed by node t@1 labelled by $a{\in}\pi_{\mathbf{olb}}$ (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 $u_4(t_1)$@**i**1 has been inserted before the nodes $u_4(t_1)$@**i** and $u_4(t_1)$@**i**1 respectively.

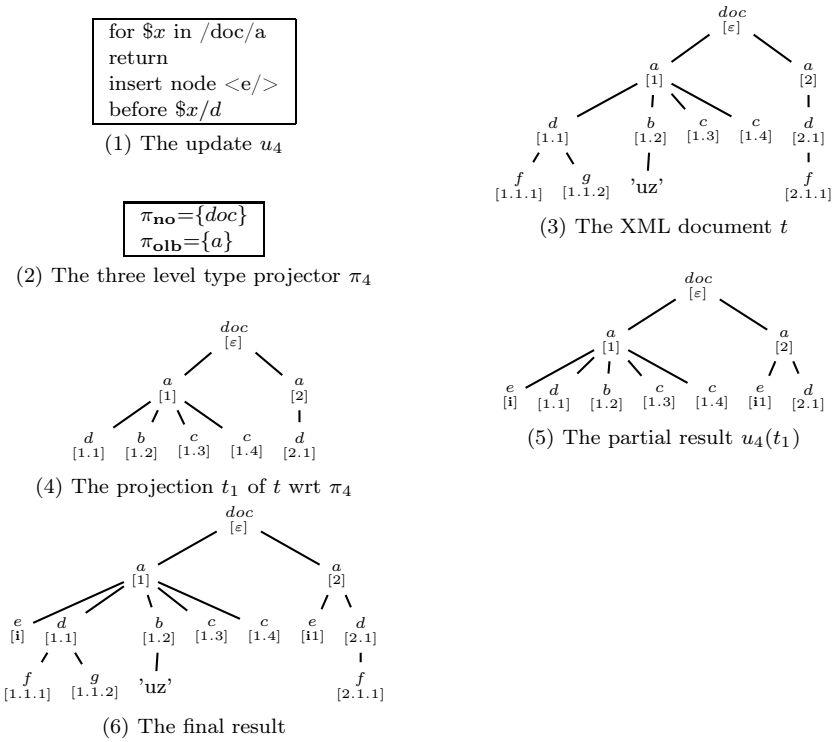


Figure 4.5: The projector component $\pi_{\mathbf{olb}}$, illustrated for "insert before"

merge - Similarly to the previous example, while synchronizing nodes t@1.1 and t'_4@i, t'_4@i is output as the first child (see fig. 4.5-6) based on the fact that t@1 is of type $a \in \pi_{\mathbf{olb}}$. □

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 $\pi_{\mathbf{no}}$ projection

Consider the update u_5 specified by `for $x in /doc/a where $x/b/text()='foot' 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 π_5={$doc, a, b, String, d$} (see fig. 4.6-2).

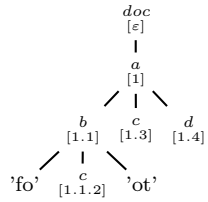
projection - Let us consider the document t given in Figure. 4.6-3 and its projection $\pi_5(t)$. Notice that projecting t wrt π_5 has the side effect to concatenate the two *Strings* 'fo' and 'ot' (see fig. 4.6-4).

```
for $x in /doc/a
where $x/b/text()='foot'
return delete node $x/d
```

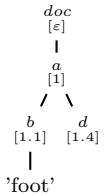
(1) The update u_5

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

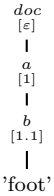
(2) The type projector π_5



(3) The XML document t



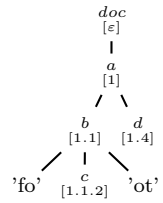
(4) The projection t_1 of π_5



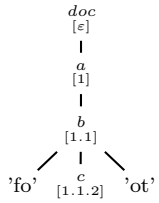
(5) The partial result $u_5(t_1)$

$\pi_{\mathbf{no}}=\{doc, a, d\}$
$\pi_{\mathbf{olb}}=\{b\}$

(6) The three level type projector π'_5



(7) The projection t_2 of π'_5



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

Figure 4.6: The projector component $\pi_{\mathbf{olb}}$, 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 $\pi_{\mathbf{olb}}$ component (see fig. 4.4-6).

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

projection extraction - The new projector π'_5 generated for the example will have two components: $\pi_{\mathbf{no}}$=$\{doc, a, d\}$ and $\pi_{\mathbf{olb}}$=$\{b\}$ (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 π'_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 $doc \rightarrow (a \mid String)*$, 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 π'_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 $\pi_{\mathbf{olb}}$ 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 π_6 is composed of three sets of types (see fig. 4.7-2); $\pi_{\mathbf{no}}$=$\{doc\}$ of category "node only", $\pi_{\mathbf{olb}}$=$\{a\}$ of category "one level below", and $\pi_{\mathbf{eb}}$=$\{d\}$ of category "everything below".

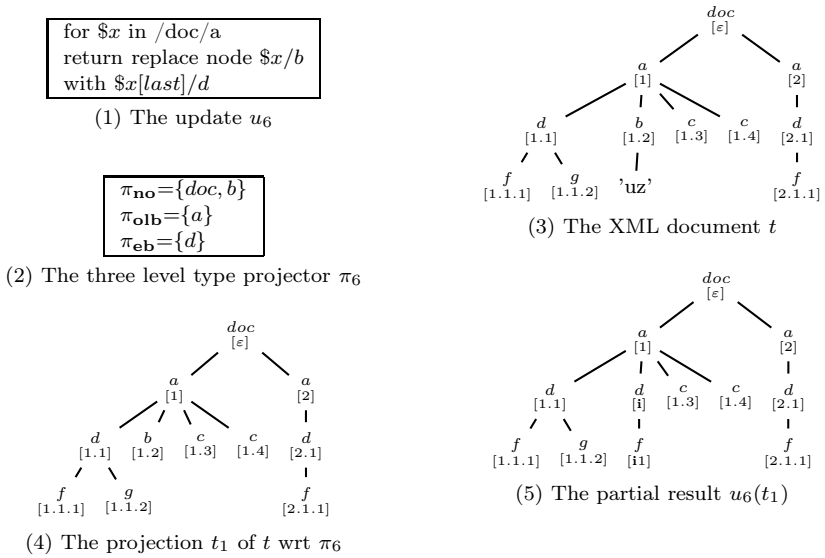


Figure 4.7: The projector component $\pi_{\mathbf{eb}}$, illustrated for "replace"

projection - Applying the projector π_6 to the document t (see fig. 4.7-3) proceeds as follows: first root $t@\varepsilon$ is projected, followed by node $t_i@1$. Because $t@1$ is labelled by $a{\in}\pi_{\mathbf{olb}}$ 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* Σ*, a type projector* π *is a triple* $(\pi_{\mathbf{no}}, \pi_{\mathbf{olb}}, \pi_{\mathbf{eb}})$ *such that (*π *also denotes* $\pi_{\mathbf{no}}{\cup}\pi_{\mathbf{olb}}{\cup}\pi_{\mathbf{eb}}$*):*
i) $\pi{\subseteq}\Sigma$*,*
ii) $\pi_{\mathbf{no}}$*,* $\pi_{\mathbf{olb}}$ *and* $\pi_{\mathbf{eb}}$ *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 $\pi_{\mathbf{no}}$ (resp. $\pi_{\mathbf{olb}}$ and $\pi_{\mathbf{eb}}$) component of π contains "node only" types (resp. "one level below" and "∀ 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 π_4 below). Notice that the *String* type itself never belongs to a type projector π: 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* $\pi=(\pi_{\mathbf{no}}, \pi_{\mathbf{olb}}, \pi_{\mathbf{eb}})$ *and the document* $t \in D$ *with* $roots(t)=\{r_t\}$ *and* $subfor(t)=F$. *The projection of* t *wrt* π, *denoted* $\pi(t)$, *is the tree* $\Pi_{K(t,\pi)}(t)$ *where* $K(t,\pi)$ *is recursively defined by:*
– *if* $lab(r_t) \notin \pi$ *then* $K(t,\pi)=\emptyset$,
– *if* $lab(r_t) \in \pi_\alpha$ *then* $K(t,\pi)=\{r_t\} \cup K_\alpha(F)$ *for* $\alpha \in \{\mathbf{no}, \mathbf{olb}, \mathbf{eb}\}$ *with:*
$K_\alpha(F)=\emptyset$ *if* $F=()$ *and otherwise, assuming* $F=t' \circ F'$,
$K_{\mathbf{no}}(F)=K(t',\pi) \cup K_{\mathbf{no}}(F')$,
$K_{\mathbf{olb}}(F)=K(t',\pi) \cup K_{\mathbf{olb}}(F')$ *if* $lab(r_{t'}) \in \pi$
$K_{\mathbf{olb}}(F)=\{r_{t'}\} \cup K_{\mathbf{olb}}(F')$ *if* $lab(r_{t'}) \notin \pi$
$K_{\mathbf{eb}}(F)=dom(F)$.

Example 4.2.7. *Example of the projection.*
For our example illustrated in Figure 4.7 the projector π *is well-defined: it consists of three components* $\pi_{\mathbf{no}}$, $\pi_{\mathbf{olb}}$ *and* $\pi_{\mathbf{eb}}$. *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:

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

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

$K_{\mathbf{no}}(F)=K(t_1,\pi) \cup K_{\mathbf{no}}(F')$
$K(t_1,\pi)=\{1\} \cup K_{\mathbf{olb}}(F_1)$ *where* F_1 *is the sub-forest of* t_1 *and* $F_1=t_{11} \circ F_1'$
$K_{\mathbf{olb}}(F_1)=K(t_{11},\pi) \cup K_{\mathbf{olb}}(F_1')$
$K(t_{11},\pi)=\{1.1\} \cup K_{\mathbf{eb}}(F_{11})$ *where* F_{11} *is the sub-forest of* t_{11}
$K_{\mathbf{eb}}(F_{11})=\{1.1.1, 1.1.2\}$

Let assume that
$F_1'=t_{12} \circ F_2'$ *where* t_{12} *is the first tree of the forest* F_1'.
$F_2'=t_{13} \circ F_3'$ *where* t_{13} *is the first tree of the forest* F_2'.
$F_3'=t_{14} \circ F_2'$ *where* t_{14} *is the first tree of the forest* F_3'.

$K_{\mathbf{olb}}(F_1')=\{1.2\} \cup K_{\mathbf{olb}}(F_2')$
$K_{\mathbf{olb}}(F_2')=\{1.3\} \cup K_{\mathbf{olb}}(F_3')$
$K_{\mathbf{olb}}(F_4')=\{1.4\} \cup K_{\mathbf{olb}}(F_4')$

etc.

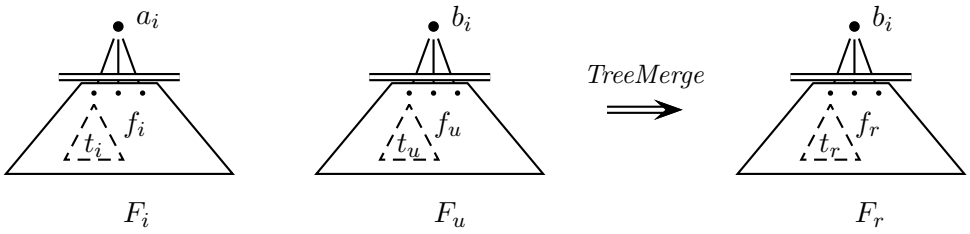


Figure 4.8: *TreeMerge* processing

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 τ and τ'. The first one, τ, is a subtree of the initial tree t. The second one, τ' is a subtree of the partially updated tree t'.
  Let us assume that:

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

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

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

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

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

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 *NoMerge* and *OlbMerge* which are formalized in Figures 4.9 and 4.15. For the sake of simplicity, the update projector π is kept implicit in the specification.

- The functions *NoMerge* and *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 *TreeMerge*)

  – *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 *String*, ii) root identifiers of top level trees in F_u belong to F_i that is $roots(F_u) \subseteq roots(F_i)$.

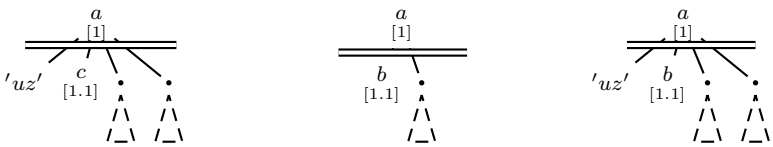
  – *OlbMerge* considers that (††) the node `n` is of category "one level below" which implies that each node in $roots(F_i)$ has been projected and that $roots(F_u)$ are exactly the top level nodes of F_u that have to be output by *OlbMerge*.

We provide explanations and examples for each line of the formalization. First we start with the procedure *NoMerge*, next we explain the procedure *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

1	$NoMerge(F_i \mid F_u) =$ F_u	if $roots(F_i)=\emptyset$, **otherwise** assume $F_i=t_i \circ f_i$
2	$t_i \circ NoMerge(f_i \mid F_u)$	if $\sigma_{t_i}(roots(t_i))=text[st]$, **otherwise** assume $\sigma_{t_i}(roots(t_i))=a[J]$,
3	$NoMerge(f_i \mid F_u)$	if $a \in \pi$ and either $roots(F_u)=\emptyset$ or $F_u=t_u \circ f_u$ with $roots(t_u) > roots(t_i)$
4	$TreeMerge(t_i \mid t_u) \circ NoMerge(f_i \mid f_u)$	if $a \in \pi$, $F_u=t_u \circ f_u$ and $roots(t_i)=roots(t_u)$
5	$t_i \circ NoMerge(f_i \mid F_u)$	if $a \notin \pi$

Figure 4.9: The function *NoMerge*



(1) sub-forest F_i (2) sub-forest F_u (3) result F_r

Figure 4.10: Example for the procedure *NoMerge*, line 2

how merge preceeds over the forest F_i and F_u. When drawing the examples, for the sake of the presentation, we keep showing the parent node of F_i (resp. F_u). It will be separated from F_i (resp. F_u) by a double horizontal line.

4.3.1 The procedure *NoMerge*

The function *NoMerge* (see fig. 4.9) proceeds as follows:

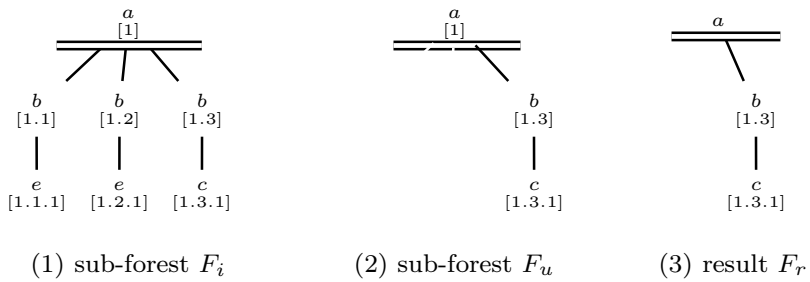
Line 2: Line 2 takes care of the case where the current parsed tree t_i of F_i is of type *String*. The assumpion † entails that it has been pruned out by π. Thus, t_i is simply output.

Example 4.3.1. Consider the update u_1 specified by `for $x in /a where $x/a return rename node $x/c with "b"` illustrated in Figure 4.10.

projector - The type projector π_1 derived from the update u_1 has one component $\pi_{\mathbf{no}} = \{a, c\}$.

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



(1) sub-forest F_i (2) sub-forest F_u (3) result F_r

Figure 4.11: Example for the procedure *NoMerge* line 3

Line 3: Line 3 deals with the case where the label a of the root $roots(t_i)$ of t_i belongs to π (thus a subtree of t_i has been projected) and $roots(t_i)$ 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_i): $roots(t_u)>roots(t_i)$ indicates that the tree t_i comes before the tree t_u in the forest F_i. Thus t_i is not output.

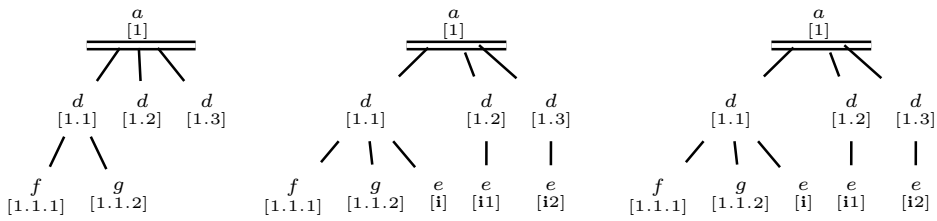
Example 4.3.2. Consider the update u_2 specified by `for $x in /a/b where not $x/c return delete node $x` illustrated in Figure 4.11.

projector - The type projector π_2 derived from the update u_2 has one component $\pi_{\mathbf{no}}=\{a, b, c\}$.

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 $a\in\pi_{\mathbf{no}}$ (see fig. 4.11-1), the forests F_i and F_u are going to be processed by *NoMerge*. First *NoMerge* examines nodes F_i@1.1 and F_u@1.3, where F_i@1.1 is labelled by $b\in\pi_{\mathbf{no}}$. 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), *NoMerge* applies line 3. Mainly, *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, *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 $b\in\pi_{\mathbf{no}}$ and the rank 3 of F_u@1.3 is strictly greater than the one of F_i@1.2. Therefore, *NoMerge* skips tree F_i@1.2, according to line 3, and parses F_i. Finally it examines nodes F_i@1.3 labelled $b\in\pi_{\mathbf{no}}$ and F_u@1.3. This time we have that the ranks of the two nodes are equal, thus *NoMerge* applies line 4, which is explained in the next paragraph. □

Line 4: Line 4 takes care of synchronization on the nodes $roots(t_u)$ and $roots(t_i)$: these nodes can only differ by their labels because of some potential renaming. In that case, the tree $TreeMerge(t_i \mid t_u)$ is output.



(1) sub-forest F_i (2) sub-forest F_u (3) result F_r

Figure 4.12: Example for the procedure *NoMerge* line 4

Example 4.3.3. Consider the update u_3 specified by `for $x in /a/d return insert node <e/> as last into $x` illustrated in Figure 4.12.

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

update - Figure 4.12-2 illustrates the changes applied by the update u_3. Nodes F_u@1.2 and F_u@1.3 contains newly inserted nodes F_u@**i**, F_u@**i**1 and F_u@**i**2 respectively, where **i**1 and **i**2 are new identifiers.

merge - Because the parent node of F_i is labelled by $a\in\pi_{\mathbf{no}}$ (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 *roots*(F_i@1.1)=*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 $d\in\pi_{\mathbf{olb}}$ 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 $d \in \pi_{\mathbf{olb}}$ 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. □

Line5: Finally, line 5 deals with the case where the label a of t_i root does not belong to the projector π 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.

update - The is identical to that of the previous example.

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

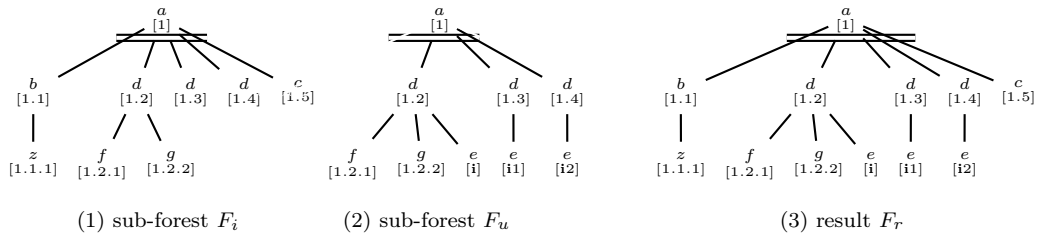


Figure 4.13: Example for the procedure *NoMerge* line 5

as a first tree of F_r and moves only on F_i (see fig. 4.13-3). After that, *NoMerge* parses forests F_i and F_u in the way explained in the previous example. Only when *NoMerge* processes the last tree of F_i rooted at F_i@1.5 and labelled by $c \notin \pi$ it is selected, once again specified by line 5. □

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

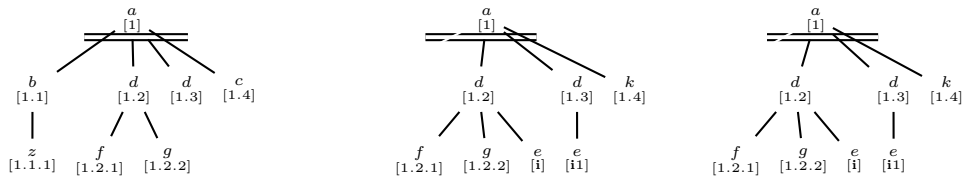
```
for $x in /a
return
{
rename node $x/c with "k",
insert node <e/> as last into $x/d,
delete node $x/b
}
```

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 π_5 derived from the update u has two components $\pi_{\mathbf{no}}=\{a, b, c\}$ and $\pi_{\mathbf{olb}}=\{d\}$.

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 $a \in \pi_{\mathbf{no}}$ the forests F_i and F_u are going to be processed by *NoMerge*. Because the root F_i@1.1 is labelled by $b \in \pi_{\mathbf{no}}$ and $roots(F_i@1.1) < roots(F_u@1.2)$ (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



(1) sub-forest F_i (2) sub-forest F_u (3) result F_r

Figure 4.14: Example for the procedure *NoMerge*, mixing cases

$c.1$	$OlbMerge(F_i \mid F_u) =$	F_u	if $roots(F_i)=\emptyset$,
$c.1'$		$()$	if $roots(F_u)=\emptyset$,
		otherwise	assume $F_u=t_u \circ f_u$
$c.2$	$t_u \circ OlbMerge(F_i \mid f_u)$		if $\sigma_{t_u}(roots(t_u))=text[st]$ or $new(roots(t_u))=true$,
		otherwise	assume $\sigma_{t_u}(roots(t_u))=b[K]$ and $F_i=t_i \circ f_i$
$c.3$	$OlbMerge(f_i \mid F_u)$		if $\sigma_{t_i}(roots(t_i))=text[st]$ or $\sigma_{t_i}(roots(t_i))=a[J]$ with $a \in \pi$ and $roots(t_u) > roots(t_i)$
$c.4$	$TreeMerge(t_i \mid t_u) \circ OlbMerge(f_i \mid f_u)$		if $a \in \pi$, $\sigma_{t_i}(roots(t_i))=a[J]$, and $roots(t_i)=roots(t_u)$
$c.5$	$t_i \circ OlbMerge(f_i \mid f_u)$		if $a \notin \pi$ and $\sigma_{t_i}(roots(t_i))=a[J]$

Figure 4.15: The function *OlbMerge*

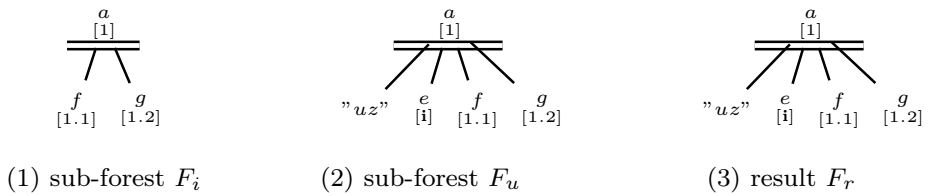
F_u@1.2, where F_i@1.2 is labelled by $d \in \pi_{\mathbf{olb}}$. This time, since the ranks are equal, *NoMerge* applies line 4. Mainly it builds a tree t_r having the root F_u@1.2 (see fig. 4.14-3). Because $lab(roots(F_i@1.2)) \in \pi_{\mathbf{olb}}$ the sub-forest of t_r is defined by *OlbMerge*.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

4.3.2 Procedure *OlbMerge*

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

Similarly to the procedure *NoMerge*, we provide examples in order to illustrate each line of the formalization of *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 *String* or a newly inserted element. This latter case is identified by checking whether the identifier $roots(t_u)$ is new ($\notin dom(t)$). Hence, the tree t_u is output. The reader may notice that no move on F_i is performed.



(1) sub-forest F_i (2) sub-forest F_u (3) result F_r

Figure 4.16: Example for the procedure *OlbMerge*, line c.2

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

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

update - Figure 4.16-2 illustrates the changes made by the updated. 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 ††, even if they are labelled by $f \notin \pi$ and $g \notin \pi$.

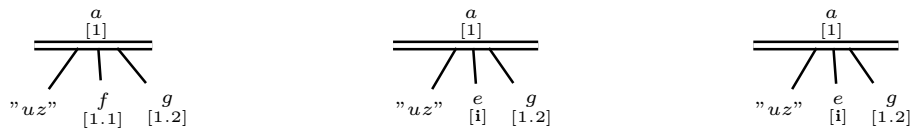
merge - We have that the parent node F_i@1 of F_i (see fig. 4.16-1) is labelled by $a \in \pi_{\mathbf{olb}}$, thus the first level nodes of F_i and F_u must be synchronized following the procedure *OlbMerge*. First *OlbMerge* processes nodes F_i@1.1 and t_u of type *String* and since the condition $\sigma_{t_i}(roots(t_i))=text[st]$ 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 $new(roots(F_u@\mathbf{i}))=true$ 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 latter, 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 `for $x in /a return replace node $x/f with <e/>` 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 $\pi_{\mathbf{olb}}=\{a\}$. Note that, the update u_7 replaces only the nodes having the child labelled by f, thus the second component of the projector is $\pi_{\mathbf{no}}=\{f\}$.

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**.



(1) sub-forest F_i (2) sub-forest F_u (3) result F_r

Figure 4.17: Example for the procedure *OlbMerge*, line c.3

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

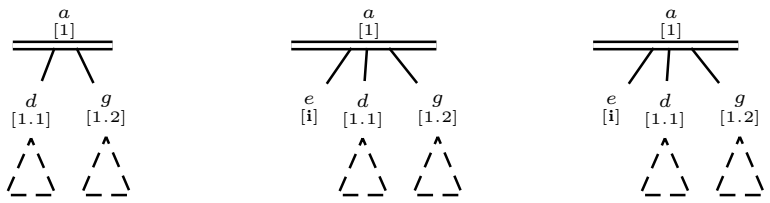
Lines c.4, c.5 are the dual of lines 4,5 of the *NoMerge* definition. The reader should pay attention to line c.5 where, although implicit, the equality $roots(t_i)=roots(t_u)$ holds (as opposed to the case line 5 of *NoMerge*): even if $a \notin \pi$, because of (††), the node identified by $roots(t_i)=roots(t_u)$ 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 π_8 derived from it contains the components $\pi_{\mathbf{olb}}=\{a\}$ and $\pi_{\mathbf{no}}=\{d\}$.

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 $a \in \pi_{\mathbf{olb}}$ (see fig. 4.18-1) thus merging top level nodes is specified by *OlbMerge*. While processing nodes F_i@1.1 and F_u@**i** *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 $d \in \pi_{\mathbf{no}}$, while nodes F_i@1.2 and F_u@1.2 are merged according to line c.5, because F_i@1.2



(1) sub-forest F_i (2) sub-forest F_u (3) result F_r

Figure 4.18: Example for the procedure *OlbMerge*, Lines c.4 and c.5

is labelled by $g \notin \pi$. □

The following paragraph explains the behavior of *OlbMerge* for the case where $lab(roots(t_i) \in \pi_{\mathbf{eb}})$:

Example 4.3.9. Let us consider the update u_7 specified by `for $x in /a return replace node $x/b with $x/c` illustrated in Figure 4.19.

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 $\pi_{\mathbf{olb}}=\{a\}$. The second one: $\pi_{\mathbf{eb}}=\{c\}$ is necessary to extract the sub-tree whose root is labelled by c.

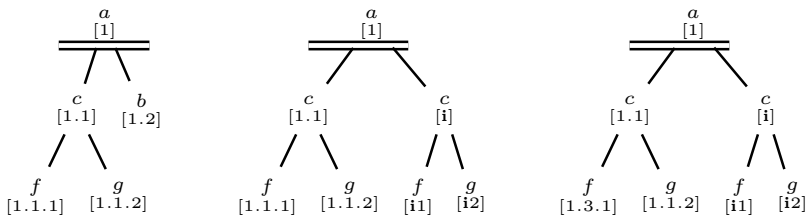
update - Figure 4.19-2 illustrates the partially updated forest F_u, where node F_u@**i** is the result of the deletion F_i@1.2 and the insertion of a new tree being the copy of the tree rooted at F_i@1.1. Note that, the identifier of F_u@**i** is not the same as F_i@1.1.

merge - Because the parent node of the F_i is labelled by $a \in \pi_{\mathbf{olb}}$, its children are processed by *OlbMerge*. While parsing them *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 $c \in \pi_{\mathbf{eb}}$ (see fig. 4.19-1,2) the action is specified by line c.4. Thus *OlbMerge* selects F_u@1.1 as the first tree of F_u and calls *TreeMerge* to determine which procedure to apply on its sub-forests. Because we have that F_i@1.1 is labelled by $c \in \pi_{\mathbf{eb}}$ all its descendants are output into into F_r (see fig. 4.19-3) and *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 $c.1'$, *OlbMerge* skips F_i. □

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

Example 4.3.10. Let us consider the update u_8 specified by

```
for $x in /a
return
{
```



(1) sub-forest F_i (2) sub-forest F_u (3) result F_r

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

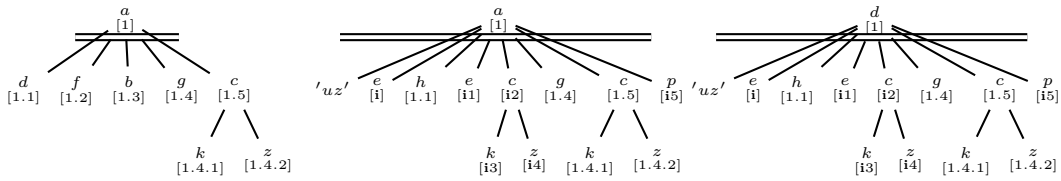
```
insert nodes ("uz"<e/>) as first into $x,
insert node <p/> as last into $x,
rename node $x/d with "h",
replace node $x/f with <e/>,
replace node $x/b with $x/c,
}
```

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 π_9 derived from this update contains three components $\pi_{\mathbf{olb}}=\{a, f, b\}$, $\pi_{\mathbf{no}}=\{d\}$ and $\pi_{\mathbf{eb}}=\{c\}$.

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@**i**. Next, the label d of the node F_i@1.1 is renamed to h. The node F_u@1.2 labelled f is replaced by a new node F_u@**i**1 labelled by e. The node F_i@1.3 labelled b is replaced by the subtree having root node F_i@1.5. Note that, the identifiers are not the same. The nodes F_i@1.4 and F_i@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 $a\in\pi_{\mathbf{olb}}$, 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 $\sigma_{t_u}(roots(t_u))=text[st]$ is true. *OlbMerge* outputs the tree t_u as the first tree of the forest F_r (see fig. 4.20-3) and moves on F_u only: processing the nodes t_i@1.1 and t_u@**i**. Once again, the action is specified by line c.2, since this time we have that the condition $new(roots(t_u))=true$ is satisfied. Therefore, *OlbMerge* selects the tree with the root F_u@**i** 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 $d\in\pi_{\mathbf{no}}$ 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 $d\in\pi_{\mathbf{no}}$, *NoMerge* must be applied. *OlbMerge* moves on both F_i and F_u. This time, nodes F_i@1.2 and F_u@**i**1 are processed as specified by line c.2 (tree rooted at F_i@1.2 has been replaced by F_u@**i**1). *OlbMerge* selects tree rooted at F_u@**i**1 and



(1) sub-forest F_i (2) sub-forest F_u (3) result F_r

Figure 4.20: Example for Mixing Cases

moves on F_u, parsing nodes F_i@1.2 and F_u@**i**2. Once again, the action is that of line c.2, hence *OlbMerge* outputs the tree rooted at F_u@**i**2 and moves on F_u. Now the nodes F_i@1.2 and F_u@1.4 are processed specified by line c.3, since the rank 4 of F_u@1.4 is strictly greater then the rank 2 of F_i@1.2 labelled by $f \in \pi_{\mathbf{no}}$. According to line c.3, *OlbMerge* skips the tree rooted at F_i@1.2 and processes nodes F_i@1.3 and F_u@1.4 as specified by line c.3. After skipping the tree rooted at F_i@1.3, *OlbMerge* is applied on nodes F_i@1.4 and F_u@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@1.4. None of the trees rooted at F_i@1.4 and F_u@1.4 has children, hence *OlbMerge* examines nodes F_i@1.5 and F_u@1.5. They have equal ranks, but because node F_i@1.5 is labelled by $c \in \pi_{\mathbf{eb}}$ *Merge* outputs the tree rooted at F_u@1.5 into F_r, and moves both on F_i and F_u. Finally, *OlbMerge* outputs node F_u@**i**5, according to line c.1.

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

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.

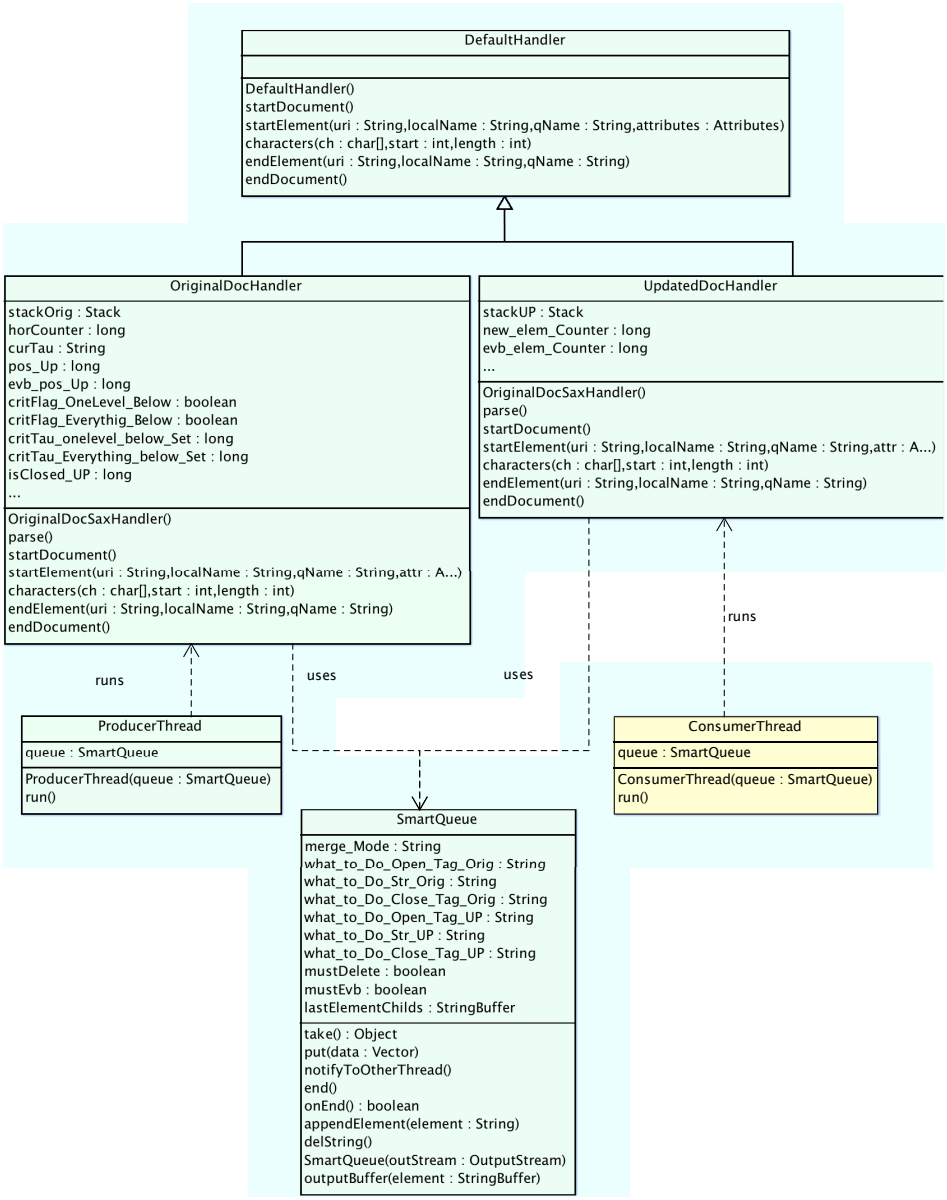


Figure 4.21: Class Diagram

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 **i** 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 $a \in \pi_{\mathbf{olb}}$, *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'@**i**2, because t'@**i**2 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 $d \in \pi_{\mathbf{eb}}$, 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

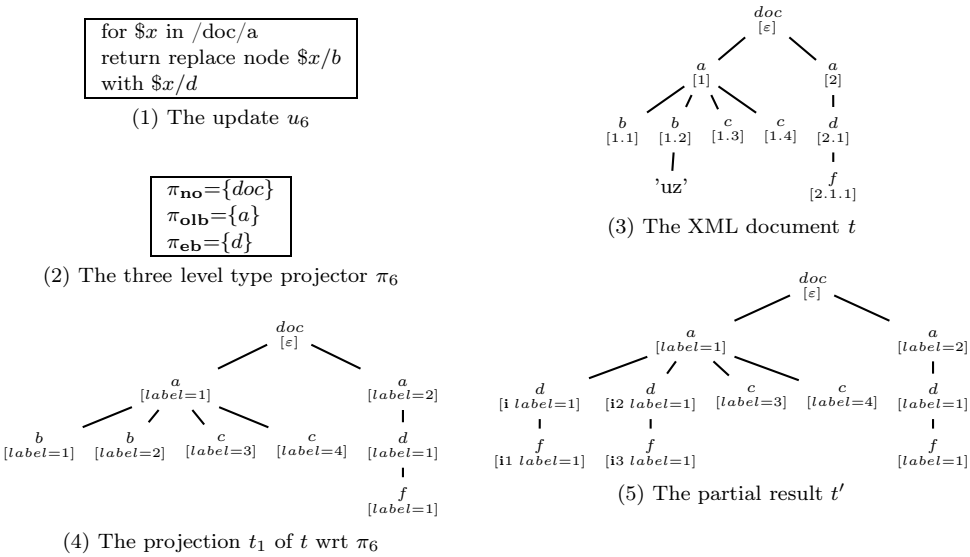


Figure 4.22: The projector component $\pi_{\mathbf{eb}}$ and *label* attribute

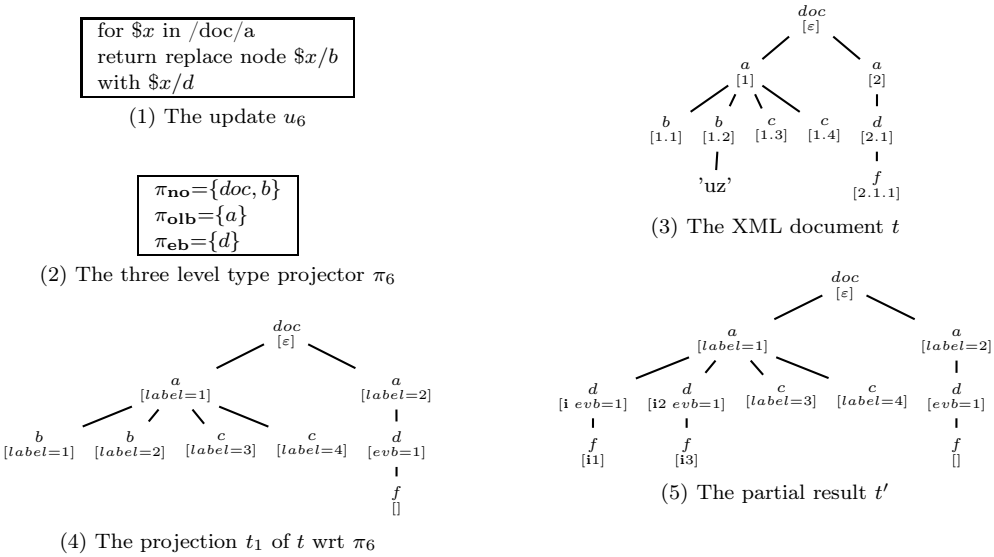


Figure 4.23: The projector component $\pi_{\mathbf{eb}}$ and *evb* attribute

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'@**i**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 $\pi_{\mathbf{no}}$ 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 $\pi_{\mathbf{no}}$ 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 *notifyToOtherTread*() (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 *SmartQueque* 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*="*OlbMerge*", 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 *OlbMerge* 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 *OlbMerge* formalazation.
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*="*OlbMerge*" 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 *OlbMerge* 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 "*OlbChild*" and is set while parsing the first level children of nodes typed by $\pi_{\mathbf{olb}}$, 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 to line c.5 of the *OlbMerge* 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

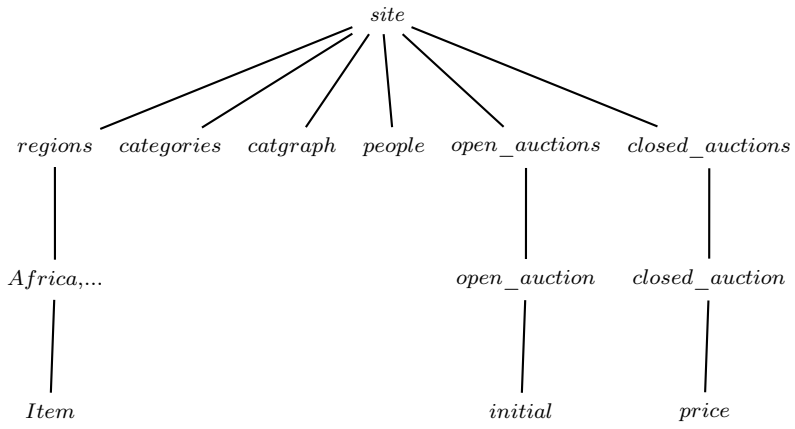


Figure 4.24: Structure of XMark documents

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

	Saxonee	Qizx F-E	eXist	MXQuery
MB	128	580	148	52

Figure 4.25: Maximal input sizes

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_proj1, reading the input document, t_proj2, projecting and storing the projected data in a buffer and t_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

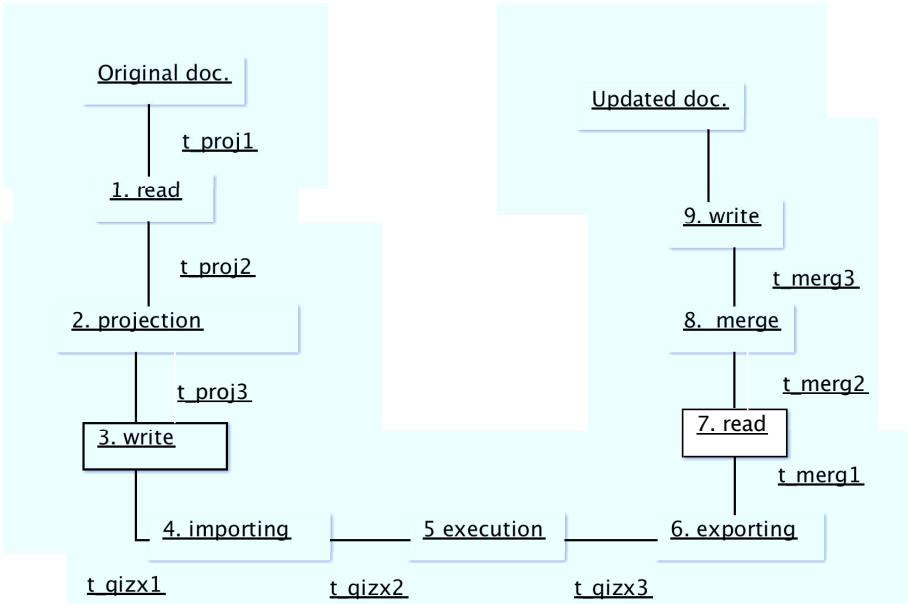


Figure 4.26: Execution with Projection using Qizx

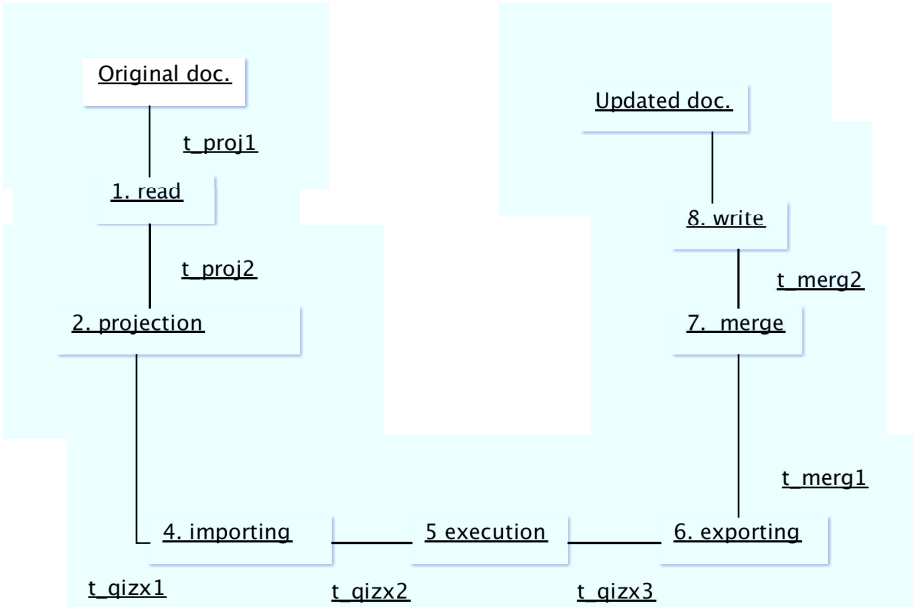


Figure 4.27: Execution with Projection using Qizx Optimization

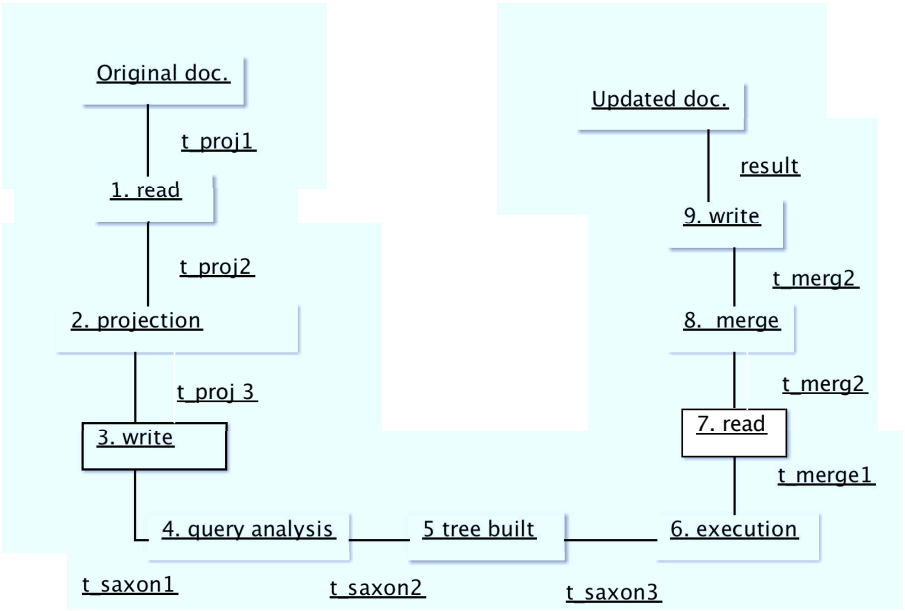


Figure 4.28: Execution with Projection using Saxon

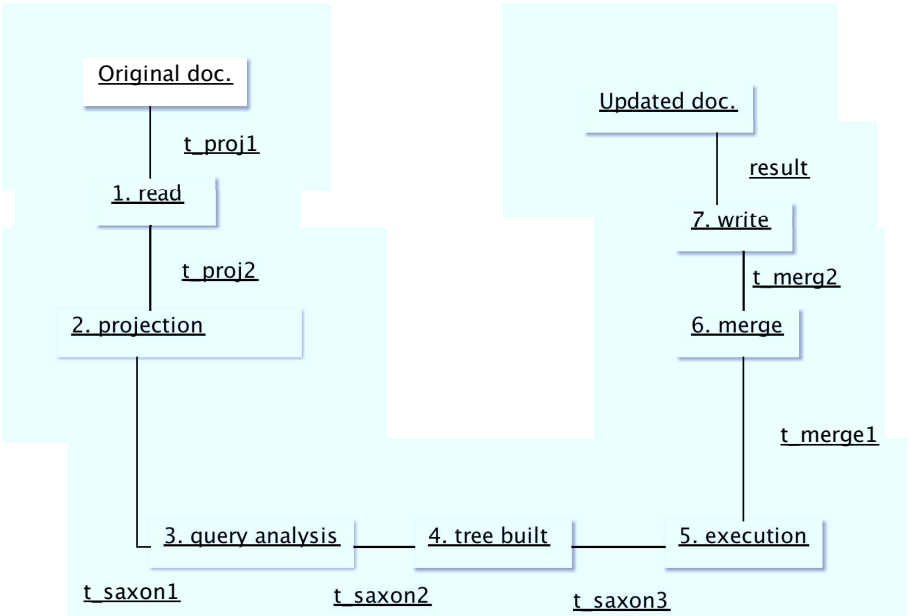


Figure 4.29: Execution using Projection using Saxon, Optimization

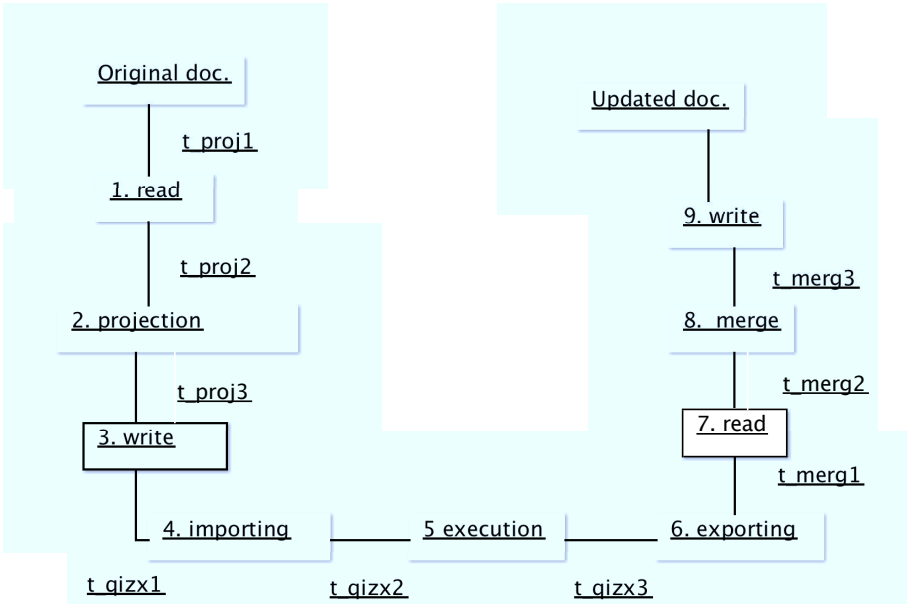


Figure 4.30: Execution with Projection using BaseX

is imported: t_qizx1. 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_qizx2 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_qizx3.
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_qizx1+ t_qizx2).
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_qizx1+ t_qizx2 + t_qizx3).

The update execution using Saxon process as follows. First, the update query is analyzed: t_saxon1. Next, the tree mapping XML document is built: t_saxon2. Finally, the update is executed on that tree: t_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_saxon1 + t_saxon2 + t_saxon3).

The Merge step is the same for the both systems. First, we read the original

and the updated documents: t_merg1. Next, we process Merge: t_merg2, storing the intermediate results in a buffer and finally writing the result of Merge to a file: t_merg3.

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.

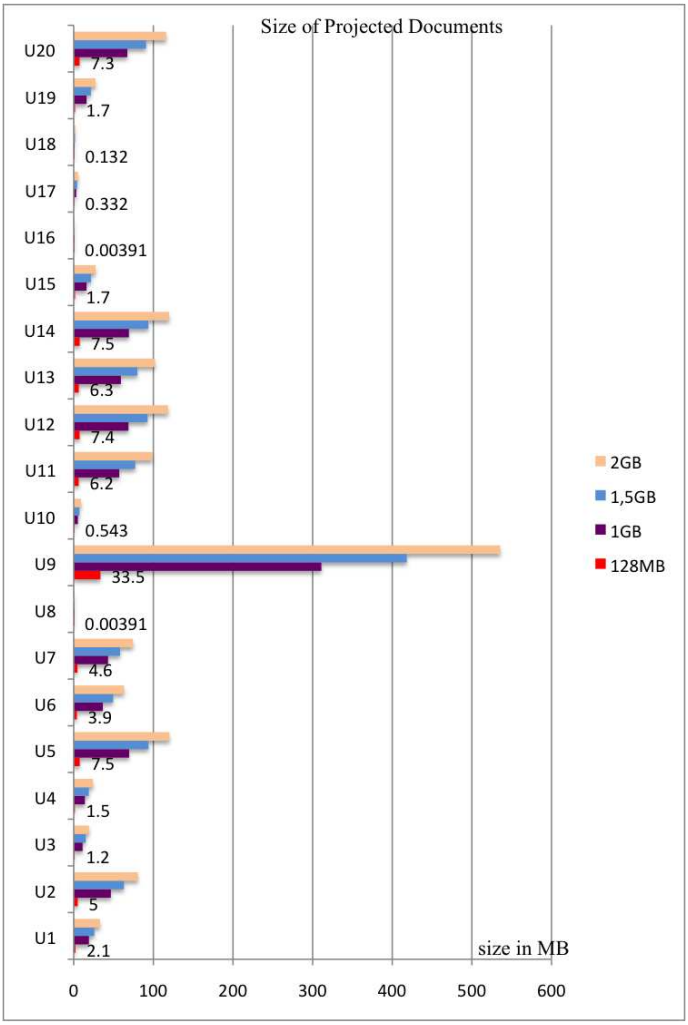


Figure 4.31: Documents size reduction after pruning

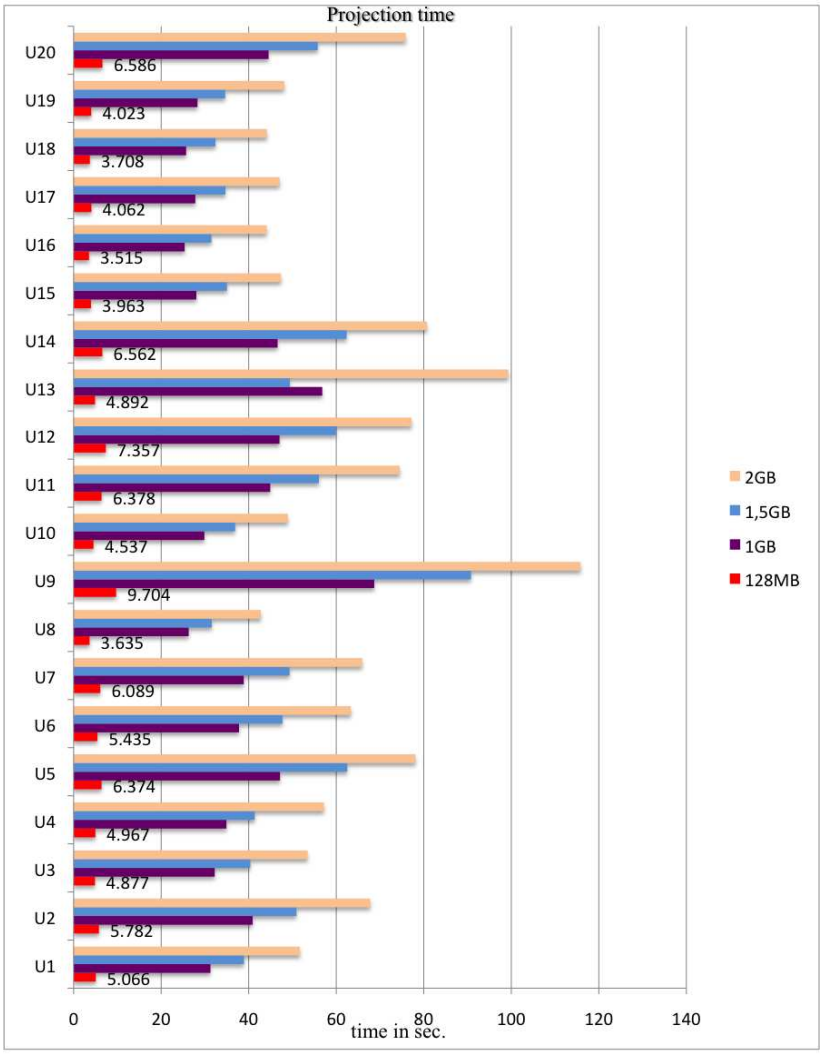


Figure 4.32: Execution time of Projection

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

below" component, which, as well, selects the text values.

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.

Or. Size MB	U1	U2	U3	U4	U5	U6	U7	U8	U9	U10
128MB	2.1	5	1.2	1.5	7.5	3.9	4.6	0.00391	33.5	0.543
1GB	19.1	46.6	11.1	14	69.6	36.6	43.1	0.00391	311	5.2
1,5GB	25.8	63	15	18.9	93.9	49.4	58.2	0.00391	418.2	7.1
2GB	33	80.5	19.2	24.2	120.2	63.2	74.4	0.00391	535.6	9.1
nodes for 128MB	75 899	170 952	37 634	55 767	261 233	132 585	159 783	3	296 794	15 606
$\pi_{\mathbf{olb}}$ descendants	59 031	99 535	0	0	0	77 300	55 318	0	59 031	0
$\pi_{\mathbf{eb}}$ descendants	0	0	0	0	0	0	49 358	0	200 134	0
nodes for 2GB	1 210 952	2 726 240	600 308	887 185	4 169 353	2 112 414	2 549 102	3	4 742 717	249 306
$\pi_{\mathbf{olb}}$ descendants	941 850	1 585 004	0	0	0	1 231 928	880 458	0	941 850	0
$\pi_{\mathbf{eb}}$ descendants	0	0	0	0	0	0	789 334	0	3 200 564	0
Or. Size MB	U11	U12	U13	U14	U15	U16	U17	U18	U19	U20
128MB	6.2	7.4	6.3	7.5	1.7	0.00391	0.332	0.132	1.7	7.3
1GB	57.2	68.7	59.2	69.3	16.1	0.00391	3.1	1.2	16.1	67.3
1,5GB	77.2	92.7	79.9	93.6	21.7	0.00391	4.2	1.7	21.6	90.8
2GB	98.7	118.5	102.1	119.8	27.4	0.00391	5.4	2.1	27.3	116.1
nodes for 128MB	207 524	257 939	52 253	259 884	7 118	2	8 435	4 439	6 514	292 839
$\pi_{\mathbf{olb}}$ descendants	83 046	184 039	0	0	604	0	8 433	1 065	0	23 167
$\pi_{\mathbf{eb}}$ descendants	0	0	0	0	4 521	0	0	0	3 765	207 432
nodes for 2GB	3 309 669	4 114 232	839 814	4 150 967	112 992	2	134 552	70 166	103 432	4 667 385
$\pi_{\mathbf{olb}}$ descendants	1 325 161	2 940 170	0	0	9 560	0	134 550	16 746	0	368 518
$\pi_{\mathbf{eb}}$ descendants	0	0	0	0	71 590	0	0	0	59 370	3 306 612

Table 4.1: Size reduction by projection

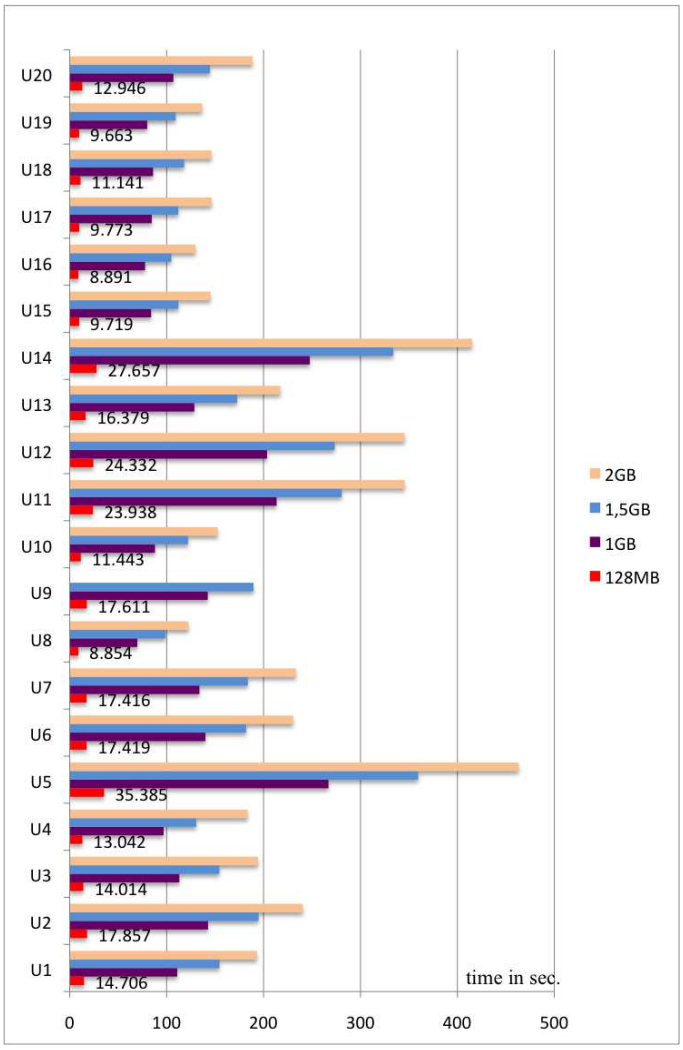
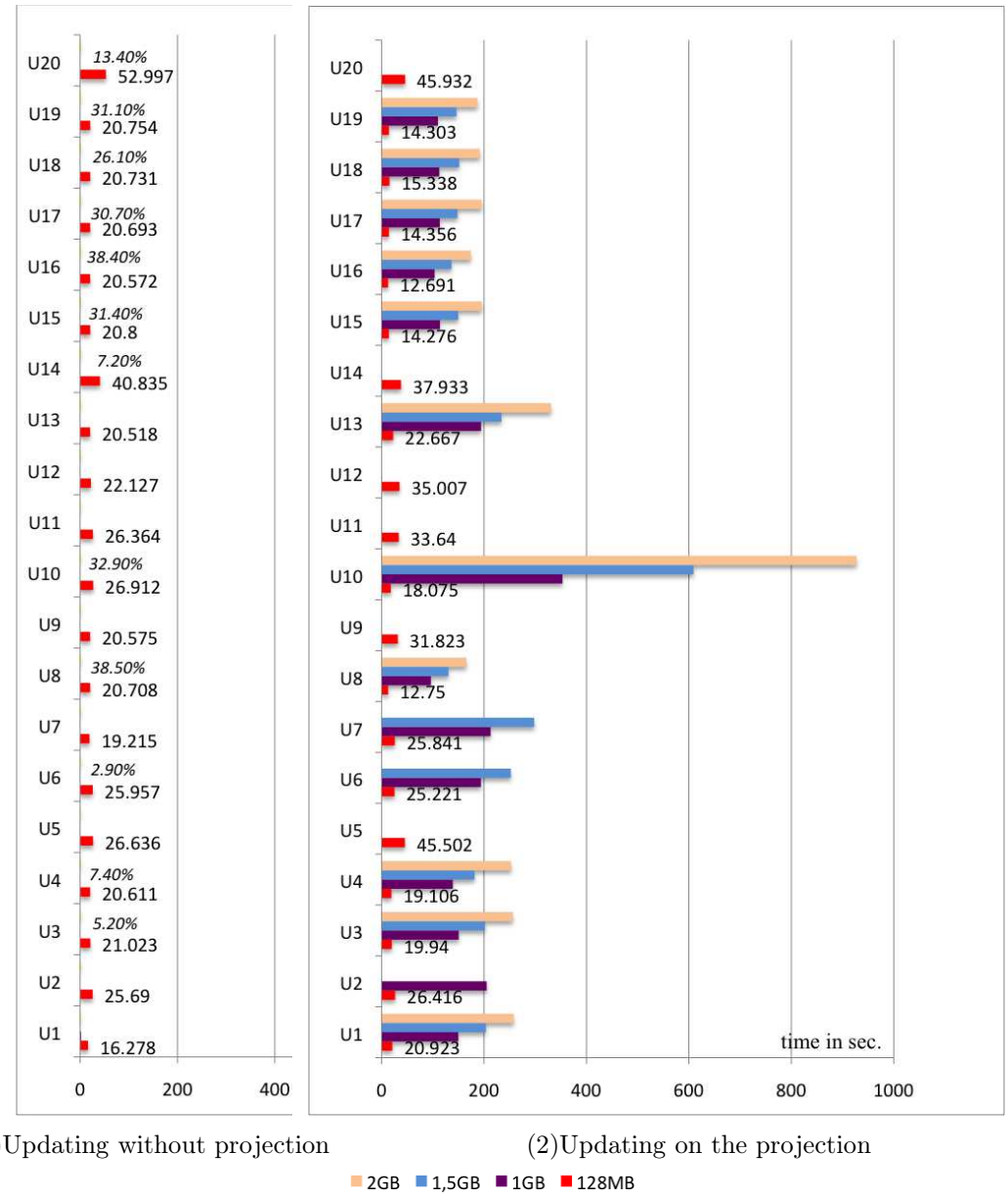


Figure 4.33: Merge process execution time

Saxon 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 128MB document, we have the following reductions of execution times, expressed in percentages: U3 (5.20%),



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

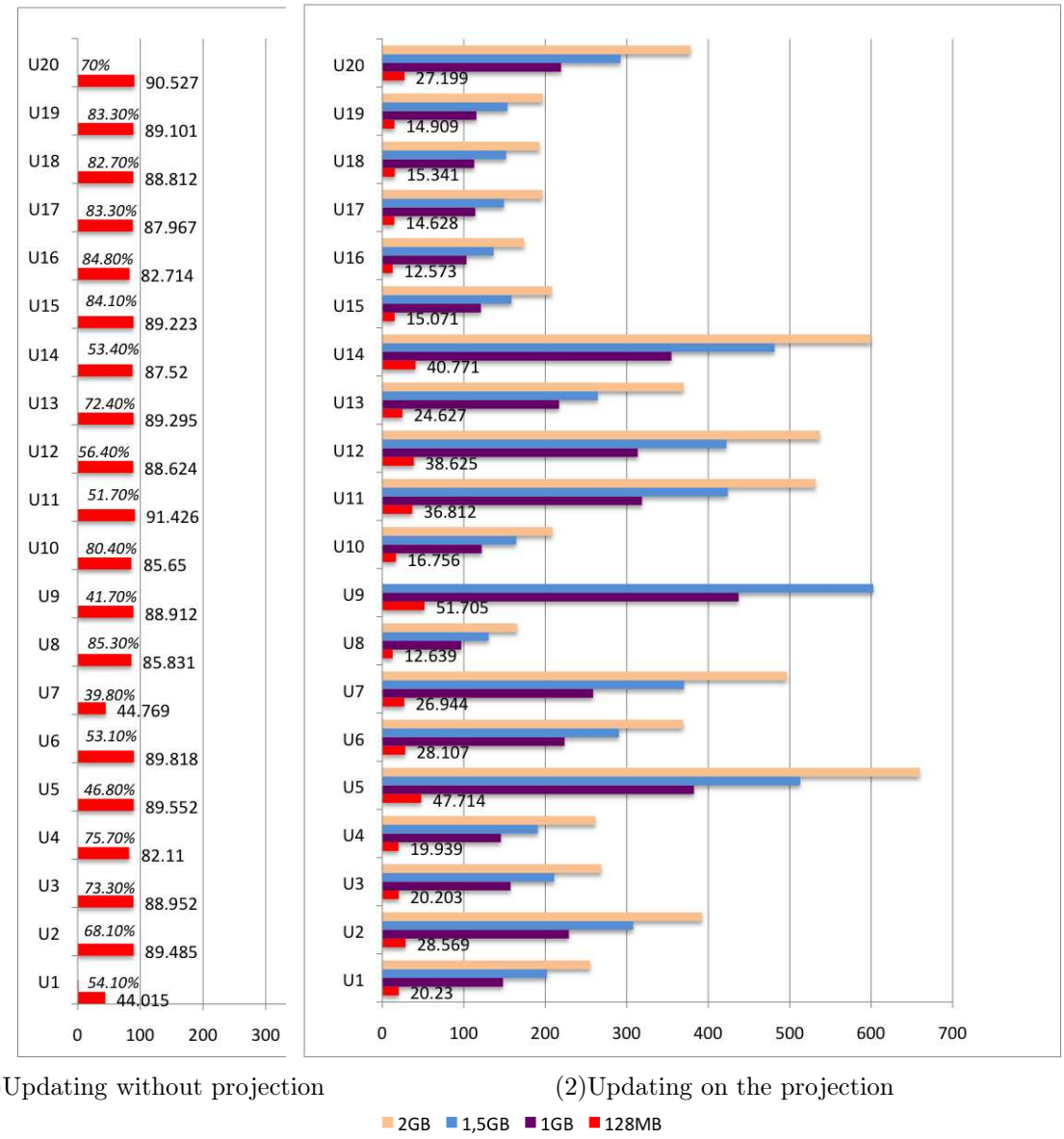
Figure 4.34: Results of the tests performed on Saxon

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 ($\pi_{\mathbf{eb}}$={*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.

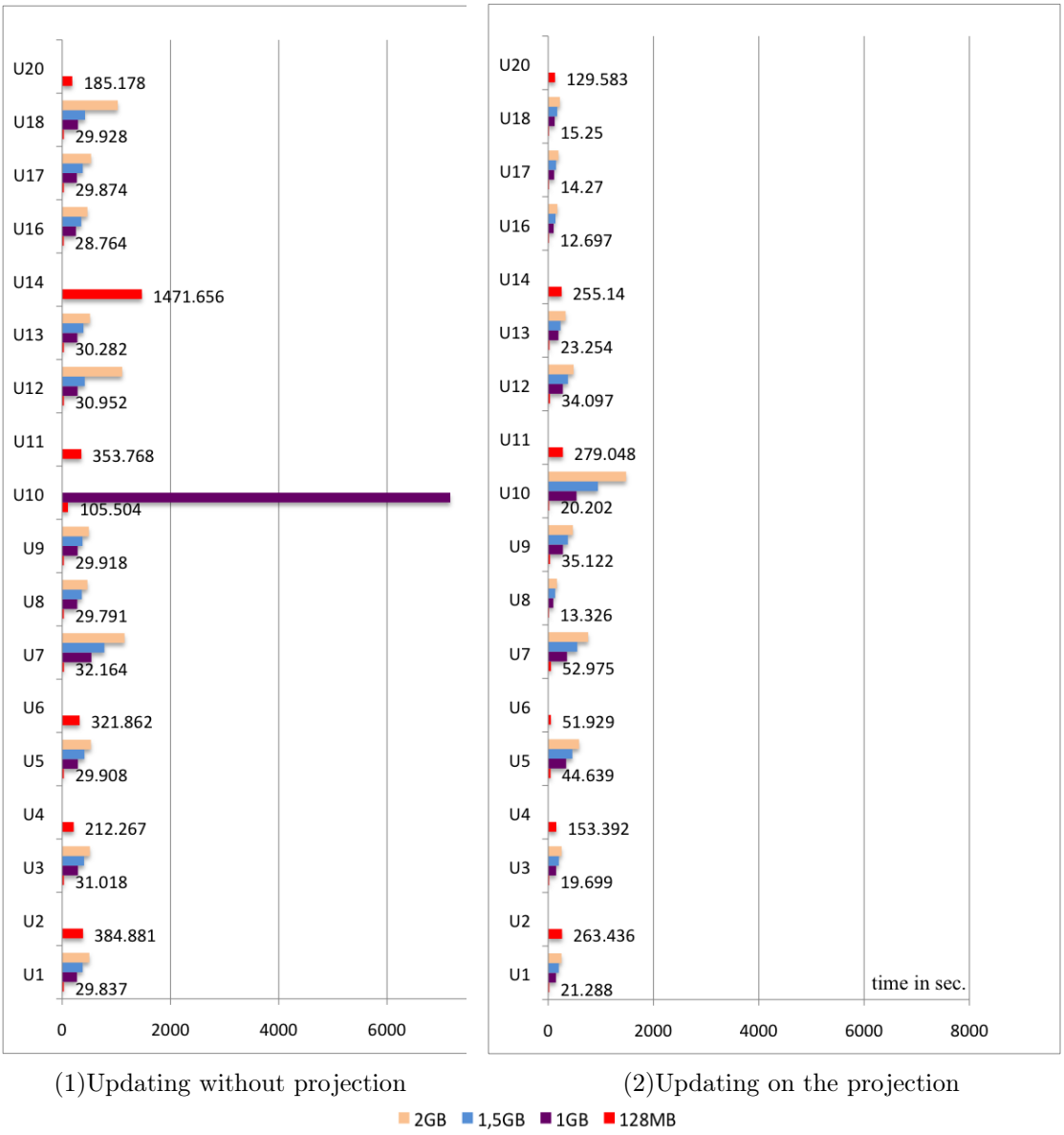
Percentage of gain using projection									
Or. Size MB	U1	U2	U3	U4	U5	U6	U7	U8	U9
128MB	28.65	31.56	36.49	27.74	-49.25	83.87	-64.74	55.25	-17.38
1GB	46.14	-	48.48	-	-17.83	-	33.85	65.16	1.93
1,5GB	47.11	-	50.14	-	-11.78	-	29.06	63.73	0.31
2GB	49.53	-	50.14	-	-10.73	-	34.07	64.34	4.6
Or. Size MB	U10	U11	U12	U13	U14	U16	U17	U18	U20
128MB	80.85	21.12	-10.17	23.22	82.66	55.84	52.23	49.05	30.02
1GB	92.5	-	2.19	30.46	-	59.1	58.28	57.85	-
1,5GB	-	-	9.82	40.26	-	61.17	60.95	59.7	-
2GB	-	-	15	35.51	-	62.58	63.4	78.37	-

Table 4.2: Gain in terms of the execution time with projection using BaseX



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

Figure 4.35: Results of the tests performed on Qizx



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

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 π_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.

The updates and the corresponding projectors

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

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

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

U4.delete nodes $doc/site/regions//item/mailbox/mail

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

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

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

	Workload evaluation using a global projector			Update execution without the projection		
	Saxon	Qizx	BaseX	Saxon	Qizx	BaseX
π_gb				-	-	-
projection	9.903	9.903	9.903			
update	33.858	39.57	272.465	115.055	442.965	442.507
merge	44.108	44.108	44.108	-	-	-
total	87.869	93.581	326.476	115.055	442.965	442.507

Table 4.3: Workload evaluation

	π_{no}	π_{olb}	π_{eb}
U1	site, closed_auctions, annotation	closed_auction	$\emptyset$
U2	site, people, address	person, country, street, province, zipcode	$\emptyset$
U3	site, regions, africa, asia, australia, europe, namerica, samerica, item	location	$\emptyset$
U4	site, regions, africa, asia, australia, europe, namerica, samerica, item, mailbox, mail	$\emptyset$	$\emptyset$
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	$\emptyset$	$\emptyset$
U6	site, people, homepage, emailaddress	person, name	$\emptyset$
U7	site, people,emailaddress	person, name, country	address
U8	site, regions, australia	$\emptyset$	$\emptyset$
U9	site, open_auctions, open_auction, closed_auctions	closed_auction	annotation
U10	site, open_auctions, open_auction	privacy	$\emptyset$
U11	site, open_auctions, bidder, initial	open_auction,increase	$\emptyset$
U12	site, regions, africa, asia, australia, europe, namerica, samerica, mailbox, mail	item,date	$\emptyset$
U13	site, open_auctions,open_auction, annotation, description, keyword,bold	text,emph	$\emptyset$
U14	site, regions, africa, asia, australia, europe, namerica, samerica, item, description, parlist, listitem, mailbox, mail, closed_auctions, closed_auction, annotation, open_auctions, open_aucton, text, emph	$\emptyset$	$\emptyset$
U15	site, categories, category, listitem	description	parlist
U16	site, closed_auctions	$\emptyset$	$\emptyset$
U17	site	closed_auctions	$\emptyset$
U18	site, categories, category, description, parlist	listitem	$\emptyset$
U19	site, categories, category, description	parlist	listitem
U20	site, open_auctions, increase	open_auction	bidder

Table 4.4: Three-level type projector for updates

	π_{no}	π_{olb}	π_{eb}
U_gb	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	location, person, name, parlist, listitem	$\emptyset$

Table 4.5: Three-level type projector for workload

```
U8. delete nodes $doc/site/regions/australia

U9. let $k :=  $doc/site/closed_auctions/closed_auction[last()]
    for $b in $doc/site/open_auctions/open_auction[last()]
      return replace node $k/annotation with $b/annotation


U10. for $x in $doc/site/open_auctions/open_auction
         where ($x/privacy="Yes")
         return delete node $x

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

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

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

U14. for $x in $doc/site//text/emph
         return delete node $x


U15. for $x in $doc/site/categories/category/description/parlist
         where  ($x/listitem/parlist) return
         replace node $x with $x/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		Qizx update execution on the projection			
Update query	128MB	128MB	1GB	1.50GB	2GB
U1					
importing time	43.969	1.112	5.947	8.557	10.3
query exec. time	0.046	0.039	0.041	0.048	0.054
total exec. time	44.015	1.151	5.988	8.605	10.354
U2					
importing time	43.888	2.147	20.455	27.36	38.263
query exec. time	45.597	2.783	24.901	34.825	45.887
total exec. time	89.485	4.93	45.356	62.185	84.15
U3					
importing time	43.854	0.595	5.065	6.634	8.825
query exec. time	45.098	0.717	7.081	9.532	11.597
total exec. time	88.952	1.312	12.146	16.166	20.422
U4					
importing time	43.859	1.125	6.485	9.029	11.186
query exec. time	38.251	0.805	7.296	10.035	9.558
total exec. time	82.11	1.93	13.781	19.064	20.744
U5					
importing time	43.317	2.885	32.858	44.801	57.091
query exec. time	46.235	3.07	35.505	46.132	61.142
total exec. time	89.552	5.955	68.363	90.933	118.233
U6					
importing time	43.817	2.445	20.81	27.539	33.574
query exec. time	46.001	2.808	24.958	33.157	42.062
total exec. time	89.818	5.253	45.768	60.696	75.636
U7					
importing time	43.811	2.816	23.25	30.242	38.335
query exec. time	0.958	0.623	62.834	107.09	159.086
total exec. time	44.769	3.439	86.084	137.332	197.421
U8					
importing time	43.82	0.116	0.92	0.114	0.111
query exec. time	42.011	0.034	0.043	0.037	0.044
total exec. time	85.831	0.15	0.963	0.151	0.155
U9					
importing time	43.725	11.75	104.839	151.426	193.123
query exec. time	45.187	12.64	121.378	170.81	-
total exec. time	88.912	24.39	226.217	322.236	-
U10					
importing time	44.368	0.529	2.582	3.302	4.224
query exec. time	41.282	0.247	1.541	2.067	2.931
total exec. time	85.65	0.776	4.123	5.369	7.155

Qizx update execution without projection		Qizx update execution on the projection			
Update query	128MB	128MB	1GB	1.50GB	2GB
U11					
importing time	44.116	2.945	26.723	40.745	51.514
query exec. time	47.31	3.551	33.604	46.519	60.187
total exec. time	91.426	6.496	60.327	87.264	111.701
U12					
importing time	43.899	3.71	32.36	45.541	58.662
query exec. time	44.725	3.226	30.472	43.54	55.726
total exec. time	88.624	6.936	62.832	89.081	114.388
U13					
importing time	43.707	1.77	15.751	20.966	26.552
query exec. time	45.588	1.586	15.725	21.284	26.608
total exec. time	89.295	3.356	31.476	42.25	53.16
U14					
importing time	43.865	3.331	32.006	42.712	52.774
query exec. time	43.655	3.221	28.578	42.51	51.066
total exec. time	87.52	6.552	60.584	85.222	103.84
U15					
importing time	43.982	0.895	4.632	6.026	8.129
query exec. time	45.241	0.494	4.305	5.394	7.188
total exec. time	89.223	1.389	8.937	11.42	15.317
U16					
importing time	43.974	0.127	0.123	0.127	0.118
query exec. time	38.74	0.04	0.05	0.04	0.04
total exec. time	82.714	0.167	0.173	0.167	0.158
U17					
importing time	43.872	0.655	0.844	1.321	1.821
query exec. time	44.095	0.138	0.763	1.273	1.367
total exec. time	87.967	0.793	1.607	2.594	3.188
U18					
importing time	44.001	0.372	0.573	0.758	1.435
query exec. time	44.811	0.12	0.521	0.652	1.017
total exec. time	88.812	0.492	1.094	1.41	2.452
U19					
importing time	43.994	0.876	4.685	5.945	8.136
query exec. time	45.107	0.347	2.498	3.583	4.124
total exec. time	89.101	1.223	7.183	9.528	12.26
U20					
importing time	43.7	3.568	30.951	42.629	52.466
query exec. time	46.827	4.099	36.927	49.317	61.265
total exec. time	90.527	7.667	67.878	91.946	113.731

Table 4.6: Qizx update execution without projection and on the projection

Saxon update execution without projection		Saxon update execution on the projection			
Update query	128MB	128MB	1GB	1.50GB	2GB
U1					
analysis time	0.169	0.173	0.17	0.172	0.171
tree built time	7.933	0.432	3.62	4.879	6.238
execution time	8.176	0.546	3.881	5.201	6.716
total	16.278	1.151	7.671	10.252	13.125
memory used	498281944	23075200	176037616	241334912	302249312
U2					
analysis time	0.181	0.178	0.18	-	-
tree built time	7.933	0.906	8.297	-	-
execution time	17.576	1.693	13.127	-	-
total	25.69	2.777	21.604	-	-
memory used	512195048	53457896	475446888	Attr. Problem	-
U3					
analysis time	0.172	0.193	0.169	0.17	0.169
tree built time	8.004	0.223	1.896	2.577	3.18
execution time	12.847	0.633	3.241	4.263	5.359
total	21.023	1.049	5.306	7.01	8.708
memory used	504621904	21211328	115835288	156182640	206032488
U4					
analysis time	0.156	0.152	0.153	0.154	0.152
tree built time	8.092	0.314	2.743	3.757	4.751
execution time	12.363	0.631	4.398	5.82	7.312
total	20.611	1.097	7.294	9.731	12.215
memory used	501430264	23681936	187571320	254896728	318573656
U5					
analysis time	0.16	0.157	-	-	-
tree built time	8.078	1.358	-	-	-
execution time	18.398	2.228	-	-	-
total	26.636	3.743	-	-	-
memory used	530171776	101053248	-	-	-
U6					
analysis time	0.175	0.199	0.171	0.173	-
tree built time	8.26	0.711	6.115	8.517	-
execution time	17.522	1.457	9.91	13.935	-
total	25.957	2.367	16.196	22.625	-
memory used	516794712	40840608	362230008	485650528	-
U7					
analysis time	0.178	0.172	0.172	0.173	-
tree built time	8.035	0.753	6.901	9.419	-
execution time	11.002	1.411	33.043	55.437	-
total	19.215	2.336	40.116	65.029	-
memory used	516562888	47116112	365147848	482577720	-
U8					
analysis time	0.153	0.151	0.178	0.151	0.151
tree built time	8.157	0.008	0.008	0.01	0.008
execution time	12.398	0.102	0.103	0.116	0.107
total	20.708	0.261	0.289	0.277	0.266
memory used	497630640	11396000	11395960	11401720	11396216
U9					
analysis time	0.166	0.162	-	-	-
tree built time	7.982	1.589	-	-	-
execution time	12.427	2.757	-	-	-
total	20.575	4.508	-	-	-
memory used	498249800	115656128	-	-	-
U10					
analysis time	0.166	0.164	0.165	0.166	0.166
tree built time	7.968	0.152	0.768	1.065	1.422
execution time	18.778	1.779	234.411	448.873	724.206
total	26.912	2.095	235.344	450.104	725.794
memory used	511625264	18784608	57282744	164462968	218551088

Update query	128MB	128MB	1GB	1.50GB	2GB
U11					
analysis time	0.213	0.184	-	-	-
tree built time	8.215	1.064	-	-	-
execution time	17.936	2.076	-	-	-
total	26.364	3.324	-	-	-
memory used	499555496	70784272	-	-	-
U12					
analysis time	0.179	0.176	-	-	-
tree built time	8.491	1.306	-	-	-
execution time	13.457	1.836	-	-	-
total	22.127	3.318	-	-	-
memory used	500551456	69188488	Cannot build tree	-	-
U13					
analysis time	0.18	0.179	0.179	0.179	0.178
tree built time	7.917	0.422	3.247	4.038	5.104
execution time	12.421	0.795	5.423	7.574	8.985
total	20.518	1.396	8.849	11.791	14.267
memory used	501115400	30979736	239002832	310447312	402909512
U14					
analysis time	0.158	0.158	-	-	-
tree built time	8.495	1.36	-	-	-
execution time	32.182	2.196	-	-	-
total	40.835	3.714	-	-	-
memory used	512383104	85633200	-	-	-
U15					
analysis time	0.167	0.165	0.163	0.165	0.164
tree built time	7.976	0.129	0.561	0.723	0.899
execution time	12.657	0.3	1.261	1.71	1.975
total	20.8	0.594	1.985	2.598	3.038
memory used	498525408	8926616	53135120	64903808	85213224
U16					
analysis time	0.158	0.156	0.154	0.154	0.153
tree built time	8.132	0.01	0.011	0.008	0.008
execution time	12.282	0.119	0.125	0.128	0.103
total	20.572	0.285	0.29	0.29	0.264
memory used	497654568	11352760	11441688	11353184	11358624
U17					
analysis time	0.166	0.163	0.163	0.164	0.164
tree built time	7.975	0.102	0.443	0.581	0.738
execution time	12.552	0.256	0.64	0.878	1.018
total	20.693	0.521	1.246	1.623	1.92
memory used	498076432	15679328	24466160	32162144	39068760
U18					
analysis time	0.175	0.174	0.174	0.172	0.174
tree built time	8.014	0.081	0.275	0.316	0.412
execution time	12.542	0.234	0.522	0.644	0.751
total	20.731	0.489	0.971	1.132	1.337
memory used	499037944	15035272	23899600	24514856	26556376
U19					
analysis time	0.159	0.157	0.158	0.157	0.158
tree built time	8.079	0.126	0.578	0.734	0.897
execution time	12.516	0.334	1.016	1.416	1.67
total	20.754	0.617	1.752	2.307	2.725
memory used	499020608	9247736	47510304	68573672	79626760
U20					
analysis time	0.18	0.177	-	-	-
tree built time	8.122	1.147	-	-	-
execution time	44.695	25.076	-	-	-
total	52.997	26.4	-	-	-
memory used	511534016	66656024	Cannot build tree	-	-

Table 4.7: Saxon update execution without projection and on the projection

	BaseX update execution without the projection				BaseX update execution on the projection			
Update query	128MB	1GB	1.50GB	2GB	128MB	1GB	1.50GB	2GB
U1								
importing time	18.353	165.771	230.271	308.032	0.976	3.455	4.502	6.74
update	0.131	0.259	0.432	0.543	0.115	0.288	0.399	0.456
exporting time	11.353	106.384	146.404	192.34	0.425	0.824	1.241	0.764
total	29.837	272.414	377.107	500.915	1.516	4.567	6.142	7.96
U2								
importing time	18.858				1.434			
update	354.073				238.107			
exporting time	11.95				0.256			
total	384.881				239.797			
U3								
importing time	18.361	165.684	234.93	298.201	0.52	2.588	3.347	3.976
update	1.027	7.609	8.741	12.343	0.208	1.274	2.078	2.27
exporting time	11.63	117.677	159.593	200.932	0.08	0.762	0.908	1.159
total	31.018	290.97	403.264	511.476	0.808	4.624	6.333	7.405
U4								
importing time	17.489				0.628			
update	185.246				134.155			
exporting time	9.532				0.6			
total	212.267				135.383			
U5								
importing time	17.084	165.661	230.358	298.495	1.346	14.753	18.802	24.088
update	1.543	12.136	19.563	26.037	0.926	6.78	10.378	12.914
exporting time	11.281	111.277	159.86	205.568	0.608	4.958	6.954	8.554
total	29.908	289.074	409.781	530.1	2.88	26.491	36.134	45.556
U6								
importing time	18.882				1.228			
update	291.624				27.511			
exporting time	11.356				0.336			
total	321.862				29.075			
U7								
importing time	18.072	171.486	222.32	290.087	1.623	9.534	12.475	15.074
update	2.33	219.07	385.634	648.409	27.511	171.325	301.718	438.707
exporting time	11.762	149.056	170.388	213.134	0.336	3.432	4.859	6.243
total	32.164	539.612	778.342	1151.63	29.47	184.291	319.052	460.024
U8								
importing time	18.932	170.856	223.748	290.087	0.83	0.81	0.87	0.82
update	0.028	0.045	0.137	0.137	0.006	0.015	0.017	0.015
exporting time	10.831	106.988	137.916	175.649	0.001	0.001	0.001	0.001
total	29.791	277.889	361.801	465.873	0.837	0.826	0.888	0.836
U9								
importing time	18.008	172.562	223.386	290.481	4.411	36.317	48.755	63.395
update	0.093	0.48	0.747	0.945	0.068	0.316	0.397	0.509
exporting time	11.817	111.739	150.664	200.206	3.328	31.576	44.059	56.844
total	29.918	284.781	374.797	491.632	7.807	68.209	93.211	120.748
U10								
importing time	18.55	159.103			0.338	1.357	1.596	1.996
update	75.543	6900.808			3.865	418.669	783.36	1276.555
exporting time	11.411	108.157			0.019	0.168	0.3	0.265
total	105.504	7168.068			4.222	420.194	785.256	1278.816

	BaseX update execution without the projection				BaseX update execution on the projection			
Update query	128MB	1GB	1.50GB	2GB	128MB	1GB	1.50GB	2GB
U11								
importing time	18.59				1.577			
update	323.009				246.341			
exporting time	12.169				0.814			
total	353.768				248.732			
U12								
importing time	18.565	159.556	222.67	293.134	1.693	12.866	17.066	21.672
update	0.58	18.114	35.492	60.31	0.331	10.627	18.298	30.4
exporting time	11.807	106.304	157.704	212.259	0.384	3.626	6.396	8.002
total	30.952	283.974	415.866	565.703	2.408	27.119	41.76	60.074
U13								
importing time	17.814	160.613	229.253	294.838	1.178	7.784	10.432	13.388
update	0.206	8.663	2.578	6.399	0.102	0.576	0.701	1.059
exporting time	12.262	110.049	159.793	211.907	0.703	5.819	8.813	10.323
total	30.282	279.325	391.624	513.144	1.983	14.179	19.946	24.77
U14								
importing time	18.094				1.922			
update	1441.761				218.582			
exporting time	11.801				0.417			
total	1471.656				220.921			
U16								
importing time	18.661	160.249	225.684	299.406	0.275	0.29	0.275	0.3
update	0.026	0.106	0.196	0.185	0.015	0.015	0.015	0.015
exporting time	10.077	92.492	125.768	166.269	0.001	0.001	0.001	0.001
total	28.764	252.847	351.648	465.86	0.291	0.306	0.291	0.316
U17								
importing time	18.435	159.541	224.741	298.024	0.402	0,885	1.07	1.327
update	0.022	0.032	0.041	0.51	0.02	0.02	0.021	0.022
exporting time	11.417	110.229	153.81	234.228	0.013	0.114	0.123	0.142
total	29.874	269.802	378.592	532.762	0.435	0.134	1.214	1.491
U18								
importing time	18.149	159.572	221.512	298.273	0.205	0.585	0.632	0.797
update	0.386	18.004	31.157	509.81	0.189	10.397	18.64	30.603
exporting time	11.393	113.571	168.699	218.194	0.007	0.059	0.08	0.093
total	29.928	291.147	421.368	1026.277	0.401	11.041	19.352	31.493
U20								
importing time	17.84				1.874			
update	155.698				107.675			
exporting time	11.64				0.502			
total	185.178				110.051			

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

	Saxon total update execution on the projection				Qizx total update execution on the projection			
Update query	128MB	1GB	1.50GB	2GB	128MB	1GB	1.50GB	2GB
U1								
projection	5.066	31.255	38.917	51.719	5.066	31.255	38.917	51.719
merge	14.706	110.897	154.434	193.142	14.706	110.897	154.434	193.142
update	1.151	7.671	10.252	13.125	1.151	5.988	8.605	10.354
total	20.923	149.823	203.603	257.986	20.923	148.14	201.956	255.215
U2								
projection	5.782	40.905	50.905	67.795	5.782	40.905	50.905	67.795
merge	17.857	142.671	194.75	240.482	17.857	142.671	194.75	240.482
update	2.777	21.604	-	-	4.93	45.356	62.185	84.15
total	26.416	205.18	-	-	28.569	228.932	307.84	392.427
U3								
projection	4.877	32.233	40.37	53.469	4.877	32.233	40.37	53.469
merge	14.014	113.041	154.366	194.162	14.014	113.041	154.366	194.162
update	1.049	5.306	7.01	8.708	1.312	12.146	16.166	20.422
total	19.94	150.58	201.746	256.339	20.203	157.42	210.902	268.053
U4								
projection	4.967	34.914	41.37	57.199	4.967	34.914	41.37	57.199
merge	13.042	96.83	130.459	183.603	13.042	96.83	130.459	183.603
update	1.097	7.294	9.731	12.215	1.93	13.781	19.064	20.744
total	19.106	139.038	181.56	253.017	19.939	145.525	190.893	261.546
U5								
projection	6.374	47.151	62.509	78.131	6.374	47.151	62.509	78.131
merge	35.385	266.975	359.423	463.308	35.385	266.975	359.423	463.308
update	3.743	-	-	-	5.955	68.363	90.933	118.233
total	45.502	-	-	-	47.714	382.489	512.865	659.672
U6								
projection	5.435	37.793	47.72	63.363	5.435	37.793	47.72	63.363
merge	17.419	140.027	181.971	230.477	17.419	140.027	181.971	230.477
update	2.367	16.196	22.625	-	5.253	45.768	60.696	75.636
total	25.221	194.016	252.316	-	28.107	223.588	290.387	369.476
U7								
projection	6.089	38.866	49.332	65.945	6.089	38.866	49.332	65.945
merge	17.416	133.789	183.736	233.264	17.416	133.789	183.736	233.264
update	2.336	40.116	65.029	-	3.439	86.084	137.332	197.421
total	25.841	212.771	298.097	-	26.944	258.739	370.4	496.63
U8								
projection	3.635	26.282	31.566	42.792	3.635	26.282	31.566	42.792
merge	8.854	69.706	98.752	122.516	8.854	69.706	98.752	122.516
update	0.261	0.289	0.277	0.266	0.15	0.963	0.151	0.155
total	12.75	96.277	130.595	165.574	12.639	96.951	130.469	165.463
U9								
projection	9.704	68.682	90.789	115.86	9.704	68.682	90.789	115.86
merge	17.611	142.394	189.66	-	17.611	142.394	189.66	-
update	4.508	-	-	-	24.39	226.217	322.236	-
total	31.823	-	-	-	51.705	437.293	602.685	-
U10								
projection	4.537	29.875	36.898	48.951	4.537	29.875	36.898	48.951
merge	11.443	87.942	122.064	152.708	11.443	87.942	122.064	152.708
update	2.095	235.344	450.104	725.794	0.776	4.123	5.369	7.155
total	18.075	353.161	609.066	927.453	16.756	121.94	164.331	208.814

	Saxon total update execution on the projection				Qizx total update execution on the projection			
Update query	128MB	1GB	1.50GB	2GB	128MB	1GB	1.50GB	2GB
U11								
projection	6.378	44.919	56.042	74.53	6.378	44.919	56.042	74.53
merge	23.938	213.411	280.634	345.573	23.938	213.411	280.634	345.573
update	3.324	-	-	-	6.496	60.327	87.264	111.701
total	33.64	-	-	-	36.812	318.657	423.94	531.804
U12								
projection	7.357	47.042	59.944	77.221	7.357	47.042	59.944	77.221
merge	24.332	203.574	273.321	345.553	24.332	203.574	273.321	345.553
update	3.318	-	-	-	6.936	62.832	89.08	114.388
total	35.007	-	-	-	38.625	313.448	422.345	537.162
U13								
projection	4.892	56.814	49.4	99.319	4.892	56.814	49.4	99.319
merge	16.379	128.597	172.78	217.321	16.379	128.597	172.78	217.321
update	1.396	8.849	11.791	14.267	3.356	31.476	42.25	53.16
total	22.667	194.26	233.971	330.907	24.627	216.887	264.43	369.8
U14								
projection	6.562	46.596	62.371	80.73	6.562	46.596	62.371	80.73
merge	27.657	247.696	333.826	415.218	27.657	247.696	333.826	415.218
update	3.714	-	-	-	6.552	60.584	85.222	103.84
total	37.933	-	-	-	40.771	354.876	481.419	599.788
U15								
projection	3.963	28.035	35.035	47.39	3.963	28.035	35.035	47.39
merge	9.719	83.937	112.008	145.334	9.719	83.937	112.008	145.334
update	0.594	1.985	2.598	3.038	1.389	8.937	11.42	15.317
total	14.276	113.957	149.641	195.762	15.071	120.909	158.463	208.041
U16								
projection	3.515	25.367	31.474	44.121	3.515	25.367	31.474	44.121
merge	8.891	77.743	104.796	129.893	8.891	77.743	104.796	129.893
update	0.285	0.29	0.29	0.264	0.167	0.173	0.167	0.158
total	12.691	103.4	136.56	174.278	12.573	103.283	136.437	174.172
U17								
projection	4.062	27.807	34.688	47.059	4.062	27.807	34.688	47.059
merge	9.773	84.616	111.96	146.452	9.773	84.616	111.96	146.452
update	0.521	1.246	1.623	1.92	0.793	1.607	2.594	3.188
total	14.356	113.669	148.271	195.431	14.628	114.03	149.242	196.699
U18								
projection	3.708	25.701	32.397	44.169	3.708	25.701	32.397	44.169
merge	11.141	85.986	118.054	146.325	11.141	85.986	118.054	146.325
update	0.489	0.971	1.132	1.337	0.492	1.094	1.41	2.452
total	15.338	112.658	151.583	191.831	15.341	112.781	151.861	192.946
U19								
projection	4.023	28.298	34.63	48.145	4.023	28.298	34.63	48.145
merge	9.663	79.959	109.35	136.654	9.663	79.959	109.35	136.654
update	0.617	1.752	2.307	2.725	1.223	7.183	9.528	12.26
total	14.303	110.009	146.287	187.524	14.909	115.44	153.508	197.059
U20								
projection	6.586	44.538	55.773	75.88	6.586	44.538	55.773	75.88
merge	12.946	106.932	144.72	188.678	12.946	106.932	144.72	188.678
update	26.4	-	-	-	7.667	67.878	91.946	113.731
total	45.932	-	-	-	27.199	219.348	292.439	378.289

Table 4.9: Saxon and Qizx total update execution on the projection

	BaseX update execution without the projection				BaseX total update execution on the projection			
Update query	128MB	1GB	1.50GB	2GB	128MB	1GB	1.50GB	2GB
U1								
projection	-	-	-	-	5.066	31.255	38.917	51.719
merge	-	-	-	-	14.706	110.897	154.434	193.142
update	-	-	-	-	1.516	4.567	6.142	7.96
total	29.837	272.414	377.107	500.915	21.288	146.719	199.493	252.821
U2								
projection	-				5.782			
merge	-				17.857			
update	-				239.797			
total	384.881				263.436			
U3								
projection	-	-	-	-	4.877	32.233	40.37	53.469
merge	-	-	-	-	14.014	113.041	154.366	194.162
update	-	-	-	-	0.808	4.624	6.333	7.405
total	31.018	290.97	403.264	511.476	19.699	149.898	201.069	255.036
U4								
projection	-				4.967			
merge	-				13.042			
update	-				135.383			
total	212.267				153.392			
U5								
projection	-	-	-	-	6.374	47.151	62.509	78.131
merge	-	-	-	-	35.385	266.975	359.423	463.308
update	-	-	-	-	2.88	26.491	36.134	45.556
total	29.908	289.074	409.781	530.1	44.639	340.617	458.066	586.995
U6								
projection	-				5.435			
merge	-				17.419			
update	-				29.075			
total	321.862				51.929			
U7								
projection	-	-	-	-	6.089	38.866	49.332	65.945
merge	-	-	-	-	17.416	133.789	183.736	233.264
update	-	-	-	-	29.47	184.291	319.052	460.024
total	32.164	539.612	778.342	1151.63	52.975	356.946	552.12	759.233
U8								
projection	-	-	-	-	3.635	26.282	31.566	42.792
merge	-	-	-	-	8.854	69.706	98.752	122.516
update	-	-	-	-	0.837	0.826	0.888	0.836
total	29.791	277.889	361.801	465.873	13.326	96.814	131.206	166.144
U9								
projection	-	-	-	-	9.704	68.682	90.789	115.86
merge	-	-	-	-	17.611	142.394	189.66	232.427
update	-	-	-	-	7.807	68.209	93.211	120.748
total	29.918	284.781	374.797	491.632	35.122	279.285	373.66	469.035
U10								
projection	-	-			4.537	29.875	36.898	48.951
merge	-	-			11.443	87.942	122.064	152.708
update	-	-			4.222	420.194	785.256	1278.816
total	105.504	7168.068			20.202	538.011	944.218	1480.475

	BaseX update execution without the projection				BaseX total update execution on the projection			
Update query	128MB	1GB	1.50GB	2GB	128MB	1GB	1.50GB	2GB
U11								
projection	-				6.378			
merge	-				23.938			
update	-				248.732			
total	353.768				279.048			
U12								
projection	-	-	-	-	7.357	47.042	59.944	77.221
merge	-	-	-	-	24.332	203.574	273.321	345.553
update	-	-	-	-	2.408	27.119	41.76	60.074
total	30.952	283.974	415.866	565.703	34.097	277.735	375.025	482.848
U13								
projection	-	-	-	-	4.892	56.814	49.4	99.319
merge	-	-	-	-	16.379	128.597	172.78	217.321
update	-	-	-	-	1.983	8.849	11.791	14.267
total	30.282	279.325	391.624	513.144	23.254	194.26	233.971	330.907
U14								
projection	-				6.562			
merge	-				27.657			
update	-				220.921			
total	1471.656				255.14			
U16								
projection	-	-	-	-	3.515	25.367	31.474	44.121
merge	-	-	-	-	8.891	77.743	104.796	129.893
update	-	-	-	-	0.291	0.306	0.291	0.316
total	28.764	252.847	351.648	465.86	12.697	103.416	136.561	174.33
U17								
projection	-	-	-	-	4.062	27.807	34.688	47.059
merge	-	-	-	-	9.773	84.616	111.96	146.452
update	-	-	-	-	0.435	0.134	1.214	1.491
total	29.874	269.802	378.592	532.762	14.27	112.557	147.862	195.002
U18								
projection	-	-	-	-	3.708	25.701	32.397	44.169
merge	-	-	-	-	11.141	85.986	118.054	146.325
update	-	-	-	-	0.401	11.041	19.352	31.493
total	29.928	291.147	421.368	1026.277	15.25	122.728	169.803	221.987
U20								
projection	-				6.586			
merge	-				12.946			
update	-				110.051			
total	185.178				129.583			

Table 4.10: BaseX update execution without projection and total update on the projection

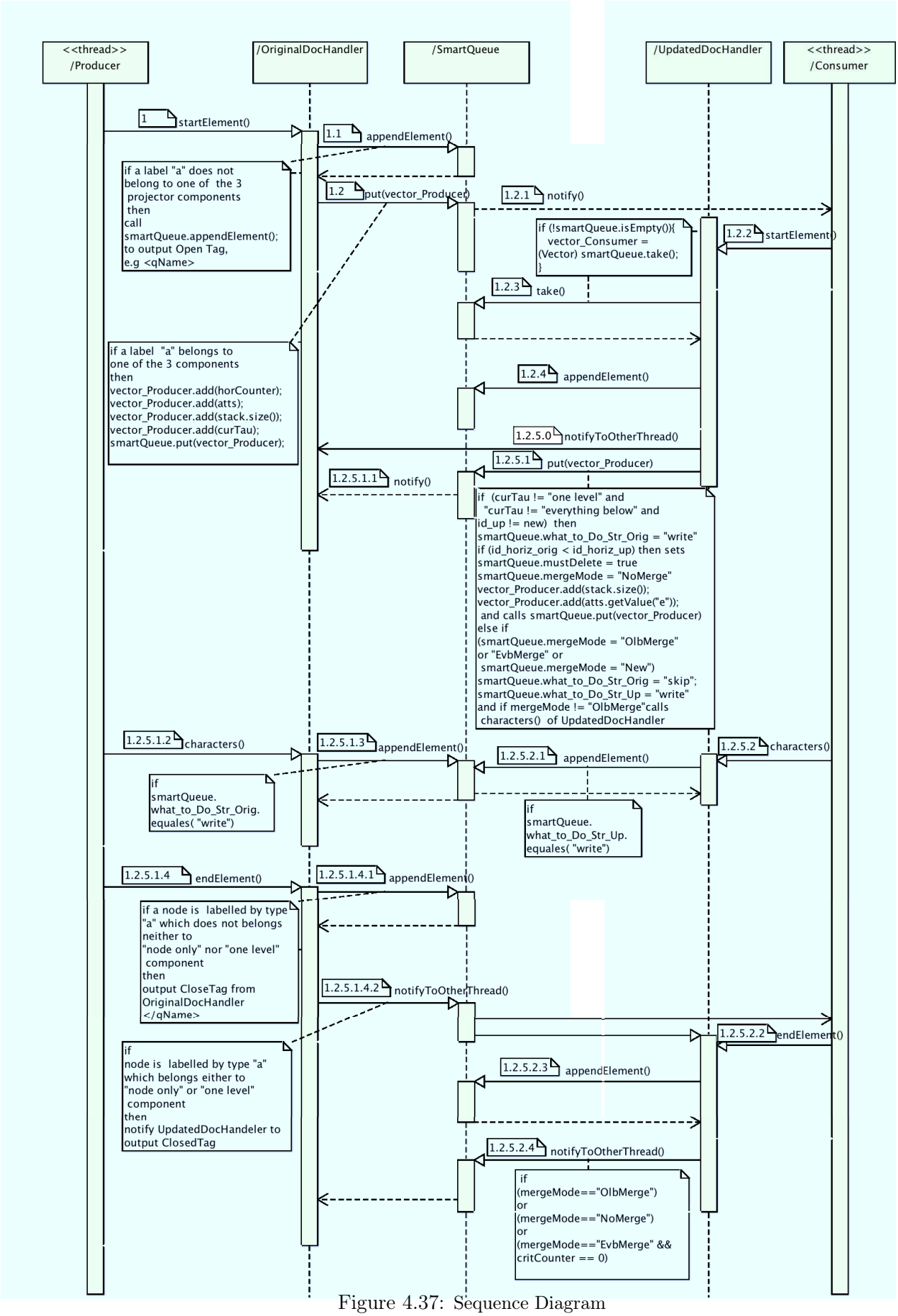


Figure 4.37: Sequence Diagram

Chapter 5

Extending the Type Projection based evaluation of Updates

Contents

5.1 Introduction 117

5.2 Extending the Type Projector for Update Optimization . . 120

5.2.1 Case analysis: update operation in isolation 120

5.2.2 Case analysis: mixing update operations of different kinds . . 128

5.3 Definition of the Extended Projection 146

5.3.1 Merge 148

5.3.2 Function *TreeMerge* - one projector component at a time - . 149

5.3.3 Function *TreeMerge* - general case - 159

5.3.4 Conclusion 164

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 $\pi=\{\pi_{\mathbf{no}}, \pi_{\mathbf{olb}}, \pi_{\mathbf{eb}}\}$. 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"*.

doc	$\rightarrow$	$a*$
a	$\rightarrow$	$d?, (a \mid k \mid b \mid c \mid z \mid c)*$
d	$\rightarrow$	$(f \mid g)*$
z	$\rightarrow$	$(f \mid g)*$
b	$\rightarrow$	$String$
e	$\rightarrow$	$String$

Figure 5.1: The *DTD D*

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 π selects all children of the nodes labelled by a because $a \in \pi_{\mathbf{olb}}$. 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 $\pi_{\mathbf{aslast}}$. 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 $\pi_{\mathbf{aslast}}$: when processing $t@1$ (see fig. 5.2-3) and $t_2^{ext}@1$ (see fig. 5.2-8), because $a \in \pi_{\mathbf{aslast}}$, Merge will first output in the final result all subtrees of $t@1$. Then, because $a \in \pi_{\mathbf{aslast}}$), all the subtrees of $t_2^{ext}@1$, which are the *"as last"* inserted elements (see fig. 5.2-9). Merging $t@2$ and $t_2^{ext}@2$ is done with the same rules.

The presentation is decomposed in two steps:

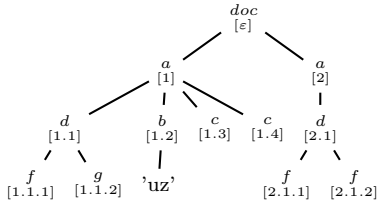
- First, we will introduce the new projector for update expressions involving only one kind of update operation at a time.

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

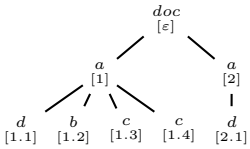
(1) The update u

$\pi_{\mathbf{no}}=\{doc\}$
$\pi_{\mathbf{olb}}=\{a\}$
$\pi_{\mathbf{eb}}=\emptyset$

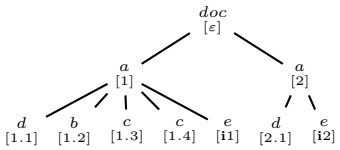
(2) The three level type projector



(3) The XML document t



(4) The projection t_1 of t wrt π

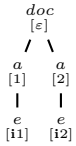


(5) The partial update $u(t_1)$

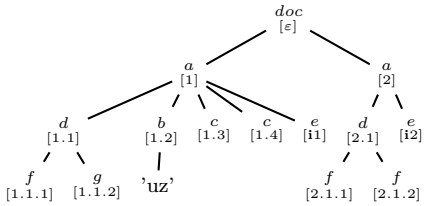
$\pi_{\mathbf{no}}=\{doc\}$
$\pi_{\mathbf{aslast}}=\{a\}$
$\pi_{\mathbf{eb}}=\emptyset$

(6) The extended projector π_{ext} for u

(7) The projection t_1^{ext} of t wrt extended π_{ext}



(8) The partial update $u(t_1^{ext})=t_2^{ext}$



(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 worth noticing that the extension of the type projector entails several changes of the *Merge* algorithm. Recall that, one 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 $\pi_{\mathbf{asfirst}}$ (see fig. 5.3-2) capturing the types of node which are potentially target of *"insert as first"* operations.

projection - The projection *wrt* π_{ext} is very simple: first it projects node $t@\varepsilon$ 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 u_1 on the document t is given in the Figure 5.3-5. As the reader can observe, the subtree of the tree t_2, rooted at $t_2@1$ contains *"as first"* inserted subtrees rooted at $t_2@\mathbf{i}$ (labelled by e), $t_2@\mathbf{i}2$ (labelled by b) and $t_2@\mathbf{i}3$ (labelled by c). It is similar for the subtree rooted at $t_2@2$.

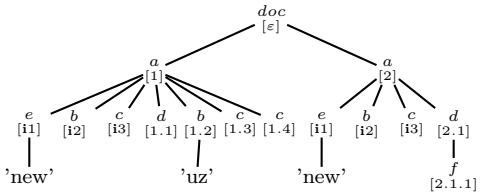
```
for $x in /doc/a
return
{ insert nodes ( <e>new</e>,<b/>,<c/>)
as first into $x }
```

(1) The update query u_1

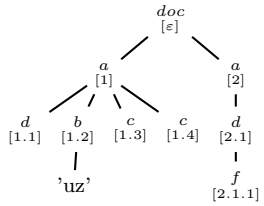
$\pi_{\mathbf{no}}=\{doc\}$
$\pi_{\mathbf{asfirst}}=\{a\}$

(2) The extended projector π^{ext} for u_1

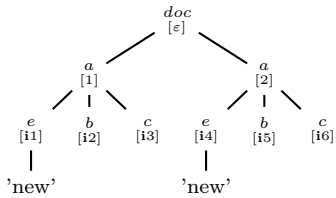
(4) The projection t_1 wrt to π_{ext}



(6) Merging t with t_2



(3) The XML document t



(5) The partial update 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 $a{\in}\pi_{\mathbf{asfirst}}$ is used by the *Merge* phase which consequently gives priority to output the children of $t_2@1$ whose nodes are identified by **i**1, **i**2, **i**3 (see fig. 5.3-6). After that *Merge* continues to output all subtrees of $t@1$, since none of their root labels belongs to π_{ext}. □

Case "insert as last" .....

This case is similar to the previous one. A $\pi_{\mathbf{aslast}}$ 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.

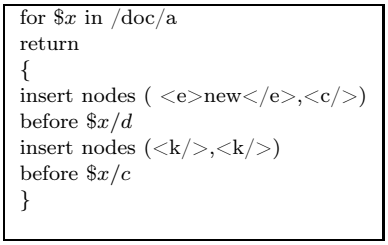
projector extraction - To project the document t (see fig. 5.4-3) we introduce a new projector component $\pi_{\mathbf{bef}}$ (see fig. 5.4-2). Notice here that the $\pi_{\mathbf{bef}}$ component is a set of pairs of types. This new specification is made to be able to refine the projector: intuitively, if (x, y) is a pair of labels in $\pi_{\mathbf{bef}}$, 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:

$par(\pi_{\mathbf{bef}}){=}\{x|(x,y){\in}\pi_{\mathbf{bef}}\}$ and $ch(\pi_{\mathbf{bef}}) =\{y|(x,y){\in}\pi_{\mathbf{bef}}\}$.

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

projection - The projection *wrt* π_{ext} (see fig. 5.4-2) of the document t is depicted in Figure 5.4-4. Its execution first selects node $t@\varepsilon$ labelled $doc{\in}\pi_{\mathbf{no}}$ of the document t. Next, because $a{\in}par(\pi_{\mathbf{bef}})$, 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 $d{\in}ch(\pi_{\mathbf{bef}})$ and the nodes $t@1.3$ and $t@1.4$ labelled by $c{\in}ch(\pi_{\mathbf{bef}})$. For the subtree $t@2$, the projection selects only the node $t_2@2.1$ labelled by $d{\in}ch(\pi_{\mathbf{bef}})$. Once again, the distinction between nodes labelled by *doc* and nodes labelled by a, d and c is important for the *Merge* phase.

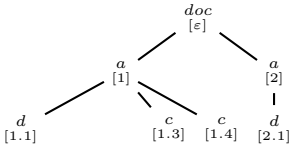
update - The result of the execution of the update u_2 on the document t_1 is given in Figure 5.4-5. The subtree rooted at $t_2@1$ contains the elements $t_2@\mathbf{i}1$ and $t_2@\mathbf{i}2$ *"inserted before"* the target node $t_2@1.1$, the elements $t_2@\mathbf{i}3$ and $t_2@\mathbf{i}4$ *"inserted before"* the target node $t_2@1.3$ and, finally, the elements $t_2@\mathbf{i}5$ and $t_2@\mathbf{i}6$ *"inserted before"* the target node $t_2@1.4$.



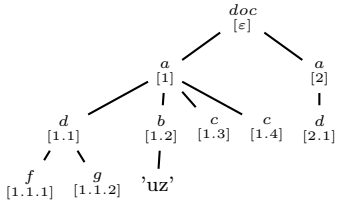
(1) The update query u_2

$\pi_{\mathbf{no}}=\{doc\}$
$\pi_{\mathbf{bef}}=\{(a,d),(a,c)\}$

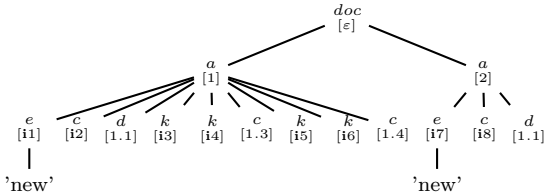
(2) The extended projector π_{ext} for u_2



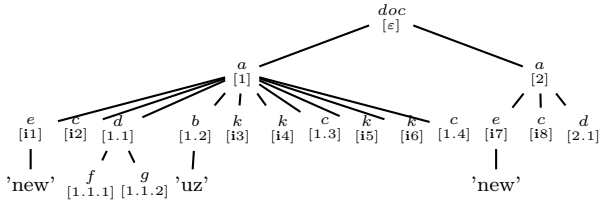
(4) The projection t_1 of t wrt π_{ext}



(3) The XML document t



(5) The partial update t_2



(6) Merging t with t_2

Figure 5.4: Dealing with "insert before" in isolation

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 $a{\in}par(\pi_{\mathbf{bef}})$, 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 π_{ext}. If the label of a node of t does not belong to $ch(\pi_{\mathbf{bef}})$, the node is output, otherwise, *Merge* is guided by t_2. For instance, *Merge* detects that the node t@1.1 is labelled by $d{\in}ch(\pi_{\mathbf{bef}})$ thus it outputs the *"inserted before"* elements rooted at t_2@**i**1 and t_2@**i**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 $b{\notin}\pi_{ext}$ is output. Similarly to the node t@1.1, when *Merge* detects that t@1.3 is labelled by $c{\in}ch(\pi_{\mathbf{bef}})$, it outputs the *"inserted before"* elements rooted at t_2@**i**3, t_2@**i**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 $\pi_{\mathbf{af}}$ component is used during the *Projection* and *Merge* phases.

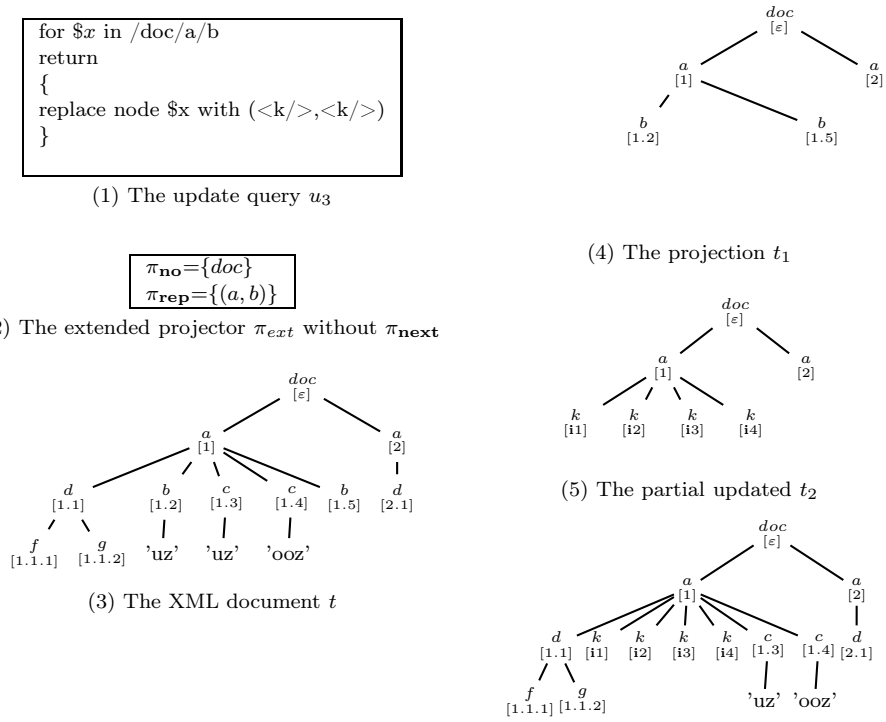


Figure 5.5: Dealing with "replace" in isolation

Case "replace" .....

Here we start by treating a rather simple example involving an update leading to a single *"replace"* operation. This first example investigate a simple solution to deal with *"replace"* operation which consists in introducing a special new component $\pi_{\mathbf{rep}}$ 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 $\pi_{\mathbf{rep}}$ (see fig. 5.5-2), which is given by pairs of labels.
Next, $par(\pi_{\mathbf{rep}})$ and $ch(\pi_{\mathbf{rep}})$ 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 $doc \in \pi_{\mathbf{no}}$; then node $t@1$ labelled by $a \in par(\pi_{\mathbf{rep}})$ followed by its children $t@1.2$ and $t@1.5$ labelled by $b \in ch(\pi_{\mathbf{rep}})$ (see fig. 5.5-4), where $\pi_{\mathbf{rep}}=\{(a, b)\}$. 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 π_{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@**i**1, t_2@**i**2, t_2@**i**3 and t_2@**i**4 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 $a{\in}par(\pi_{\mathbf{rep}})$. Then *Merge* checks if the first child of t@1 is labelled by a type in π_{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 $b{\in}ch(\pi_{\mathbf{rep}})$, thus *Merge* outputs the *"replaced"* elements rooted at t_2@**i**1, t_2@**i**2, t_2@**i**3 and t_2@**i**4.
As the reader can observe, there is no information enabling *Merge* to separate t_2@**i**1, t_2@**i**2 from t_2@**i**3, t_2@**i**4. 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 $\pi_{\mathbf{next}}$ 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 $\pi_{\mathbf{next}}$ which replaces $\pi_{\mathbf{rep}}$ (see fig. 5.6-2).
Next, $par(\pi_{\mathbf{next}})$ and $ch(\pi_{\mathbf{next}})$ are defined as expected.

projection - The new component $\pi_{\mathbf{next}}{=}\{(a, b), (a, e)\}$ (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@**i**1, t_2@**i**2 labelled by k and t_2@**i**3 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@**i**1. Because t@1.2 is labelled by $b{\in}ch(\pi_{\mathbf{next}})$ merging is guided by t_2, thus the element rooted at t_2@**i**1 is output. Next, *Merge* examines t@1.2 and t_2@**i**2 and, for the same reason, outputs the element rooted at t_2@**i**2. After that, nodes t@1.2 and t_2@1.3 are processed and because t@1.2 is labelled by $b{\in}ch(\pi_{\mathbf{next}})$

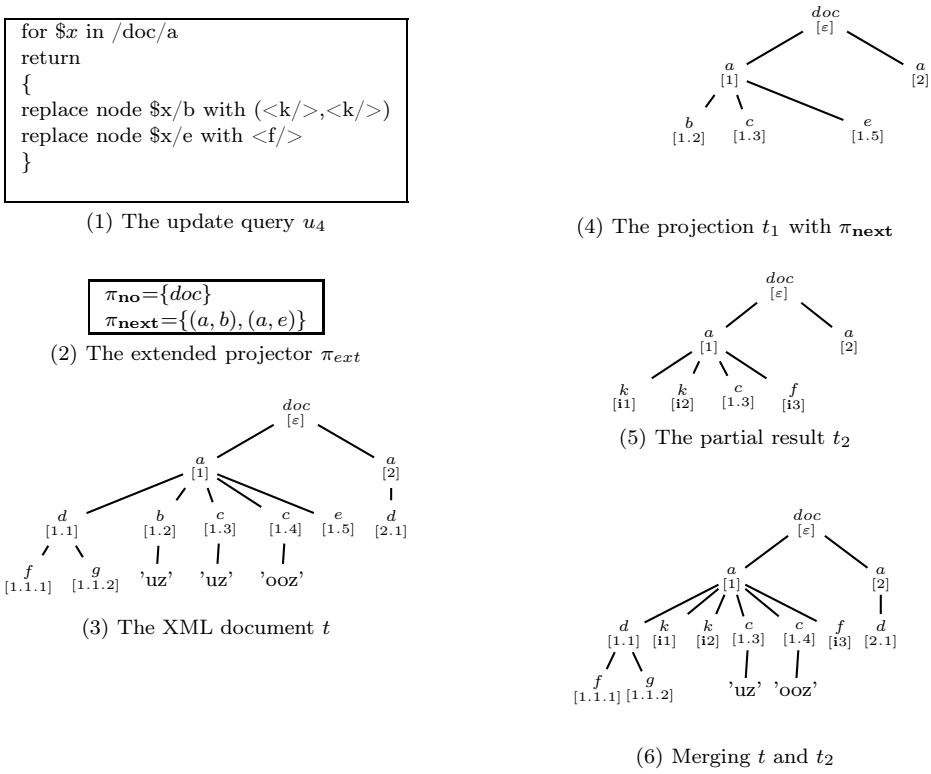


Figure 5.6: Dealing with "replace" in isolation

(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@**i**1, t_2@**i**2 from t_2@**i**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 $\pi_{\mathbf{del}}$ (see fig. 5.7-2). As for the previous cases, the projector component $\pi_{\mathbf{del}}$ is specified by pairs of labels.
Next, $par(\pi_{\mathbf{del}})$ and $ch(\pi_{\mathbf{del}})$ are defined as expected.

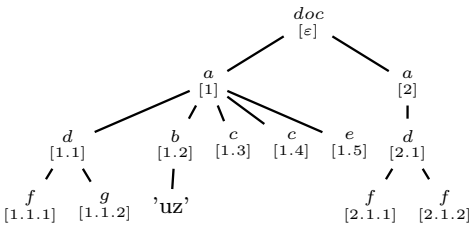
projection - The projection first outputs the node t@ε labelled by $doc \in \pi_{\mathbf{no}}$. After that it projects the node t@1, since it is labelled by $a \in par(\pi_{\mathbf{del}})$, followed by the nodes t@1.3, t@1.4 labelled by $c \in ch(\pi_{\mathbf{del}})$ and the node t@1.5 labelled by $e \in ch(\pi_{\mathbf{del}})$ (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 π_{ext}.

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

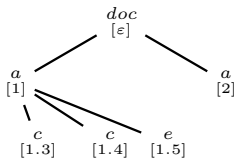
(1) The update query u_5

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

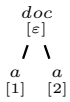
(2) The extended projector π_{ext} for u_5



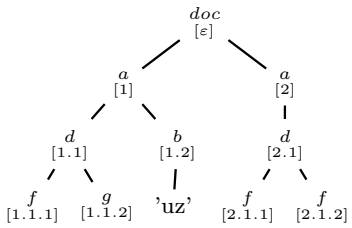
(3) The XML document t



(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 $ch(\pi_{\mathbf{del}})$ *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 $c \in ch(\pi_{\mathbf{del}})$ and $e \in ch(\pi_{\mathbf{del}})$ 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 $\pi_{\mathbf{asfirst}}$ and $\pi_{\mathbf{aslast}}$ for this update are given in Figure 5.8-2.

projection - The projection selects the root t@ε labelled *doc*, then the nodes t@1 and t@2 both labelled by $a \in \pi_{\mathbf{asfirst}}$ and $a \in \pi_{\mathbf{aslast}}$ (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@$\mathbf{i}1$, t_2@$\mathbf{i}2$ labelled by b (inserted *"as first"*) and elements t_2@$\mathbf{i}3$, t_2@$\mathbf{i}4$ labelled by e (inserted *"as last"*). The tree rooted at t_2@2 contains as well the new elements rooted at t_2@$\mathbf{i}5$, t_2@$\mathbf{i}6$ (inserted *"as first"*) and t_2@$\mathbf{i}7$, t_2@$\mathbf{i}8$ (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 $a \in \pi_{\mathbf{asfirst}}$ *Merge* gives the priority to outputting the new inserted elements having identifiers $\mathbf{i}1, \mathbf{i}2, \mathbf{i}3, \mathbf{i}4$ (see fig. 5.8 -6). Here, the issue is that there is no information enabling *Merge* to separate t_2@$\mathbf{i}1$, t_2@$\mathbf{i}2$ from t_2@$\mathbf{i}3$, t_2@$\mathbf{i}4$. Therefore, the *Merge* process fails to produce the expected result (see fig. 5.8-6). □

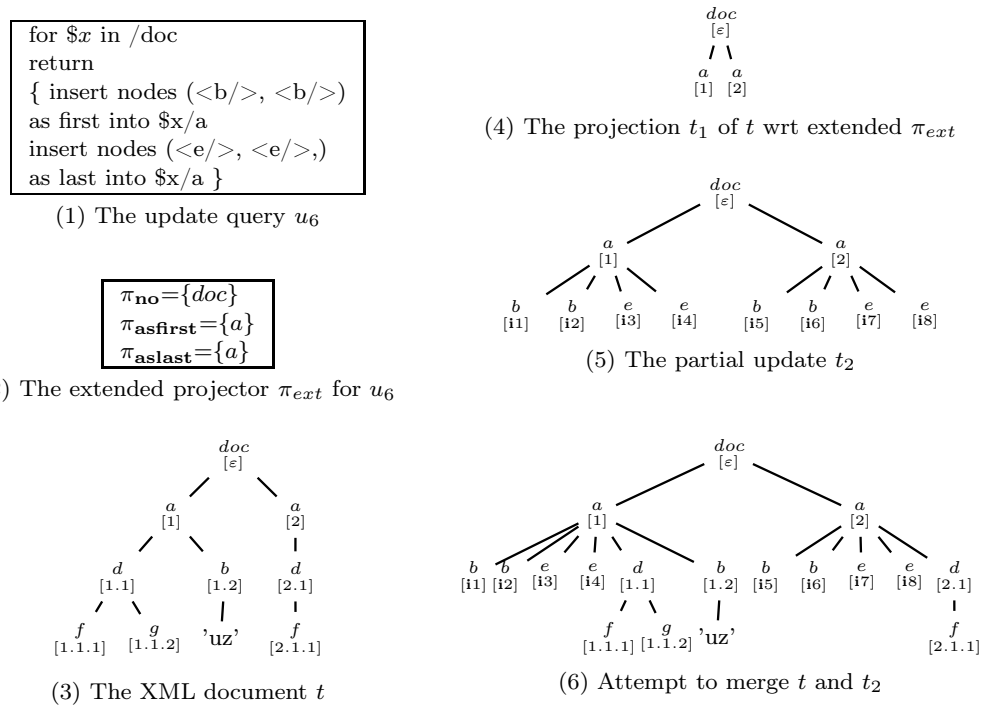


Figure 5.8: Mixing "insert as first" and "insert as last"

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 $\pi_{\mathbf{first}}$ is introduced. Intuitively, if the label of a node belongs to $\pi_{\mathbf{first}}$, 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.

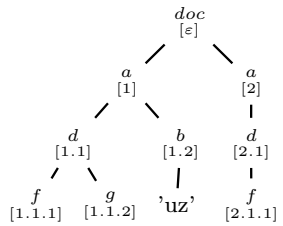
Example 5.2.7. *See Figure 5.9*

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

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_2@1 contains *"as first"* inserted elements t_2@**i**1, t_2@**i**2 and *"as last"* inserted elements t_2@**i**3, t_2@**i**4. As the reader can observe these nodes are separated by t@1.1 and t@2.1 (see fig. 5.9-4).

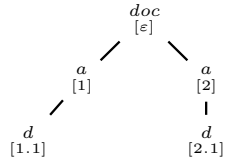
merge - Because the node t@1 is labelled by $a \in \pi_{\mathbf{asfirst}}$, while processing the nodes t@1 (see fig. 5.9-1) and t_2@1 (see fig. 5.9-4), *Merge* outputs t_2@1. While merging the children of t@1 and t_2@1, *Merge* first examines nodes t@1.1 and t_2@**i**1. Because the node t@1.1 is the first child of t@1 and the node t_2@**i**1 is the result of *"insert as first"* operation *Merge* outputs t_2@**i**1. Next merging is processed on t@1.1 and t_2@**i**2 and, for the same reason, t_2@**i**2 is output. After that, *Merge* examines the nodes



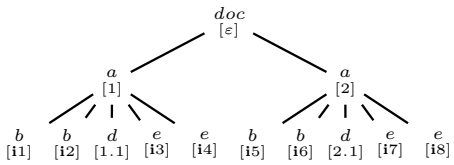
(1) The XML document t

$\pi_{\mathbf{no}}=\{doc\}$
$\pi_{\mathbf{aslast}}=\{a\}$
$\pi_{\mathbf{first}} = \{a\}$

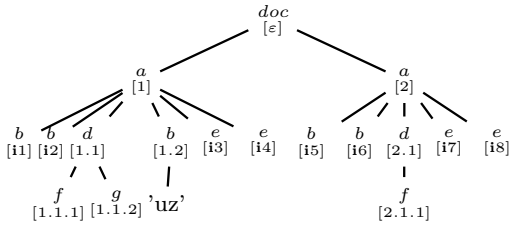
(2) The extended projector π for u_6



(3) The projection t_1 with $\pi_{\mathbf{first}}$



(4) The partial update result t_2



(5) Merging t and t_2

Figure 5.9: Mixing "insert as first" and "insert as last"

t@1.1 and t_2@1.1 and because t@1.1 is labelled by $d \notin \pi_{ext}$ *Merge* outputs the tree rooted at t@1.1. After that, *Merge* is guided by t. When t is empty *Merge* outputs the elements t_2@**i3**, t_2@**i4**. It is important to note that the node t@1.1 separates the new nodes t_2@**i1**, t_2@**i2** from t_2@**i3**, t_2@**i4**. The *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.

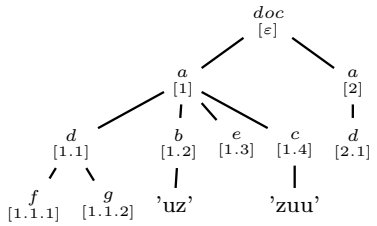
projection - The projection first selects the nodes $t@\varepsilon$ labelled by $doc \in \pi_{\mathbf{no}}$. After that it projects the nodes t@1 labelled by $a \in par(\pi_{\mathbf{bef}})$ followed by its children t@1.2 and t_2@1.4 labelled by $b \in ch(\pi_{\mathbf{af}})$ and $c \in ch(\pi_{\mathbf{bef}})$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

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

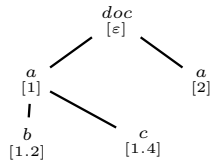
(1) The update query u_7

$\pi_{\mathbf{no}}=\{doc\}$
$\pi_{\mathbf{af}}=\{(a,b)\}$
$\pi_{\mathbf{bef}}=\{(a,c)\}$

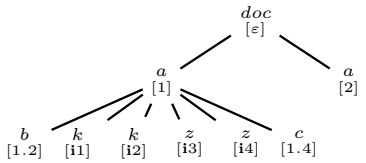
(2) The extended projector π_{ext} for u_7



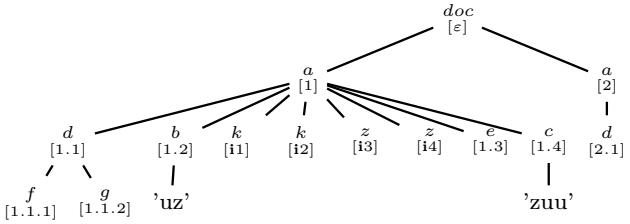
(3) The XML document t



(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 $a{\in}par(\pi_{\mathbf{bef}})$, 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 $d{\notin}\pi$, 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 $b{\in}ch(\pi_{\mathbf{af}})$, thus the node t_2@1.2 is output followed by the new inserted elements having identifiers $\mathbf{i}1, \mathbf{i}2, \mathbf{i}3, \mathbf{i}4$. 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 $\pi_{\mathbf{next}}$ instead of $\pi_{\mathbf{bef}}$ and $\pi_{\mathbf{af}}$. Recall that, when a pair of types (x, y) belongs to $\pi_{\mathbf{next}}$, 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 $\pi_{\mathbf{next}}$ (see fig. 5.11-2) is deduced from $par(\pi_{\mathbf{af}}){\cap}par(\pi_{\mathbf{bef}})$. Here, we have that $par(\pi_{\mathbf{af}}){\cap}par(\pi_{\mathbf{bef}}){=}\{a\}$ 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 $\pi_{\mathbf{next}}$ contains both pairs (a, b) and (a, c). Indeed, the pairs (a, b) and (a, c) may be deleted from $\pi_{\mathbf{bef}}$ and $\pi_{\mathbf{af}}$ respectively.

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

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 $a{\in}par(\pi_{\mathbf{next}})$, *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 $d{\notin}\pi_{\mathbf{next}}$, *Merge* outputs the tree rooted at it. The next pair of nodes being merged are t@1.2 labelled by $b{\in}ch(\pi_{\mathbf{next}})$ 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 $e{\notin}ch(\pi_{\mathbf{next}})$ 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

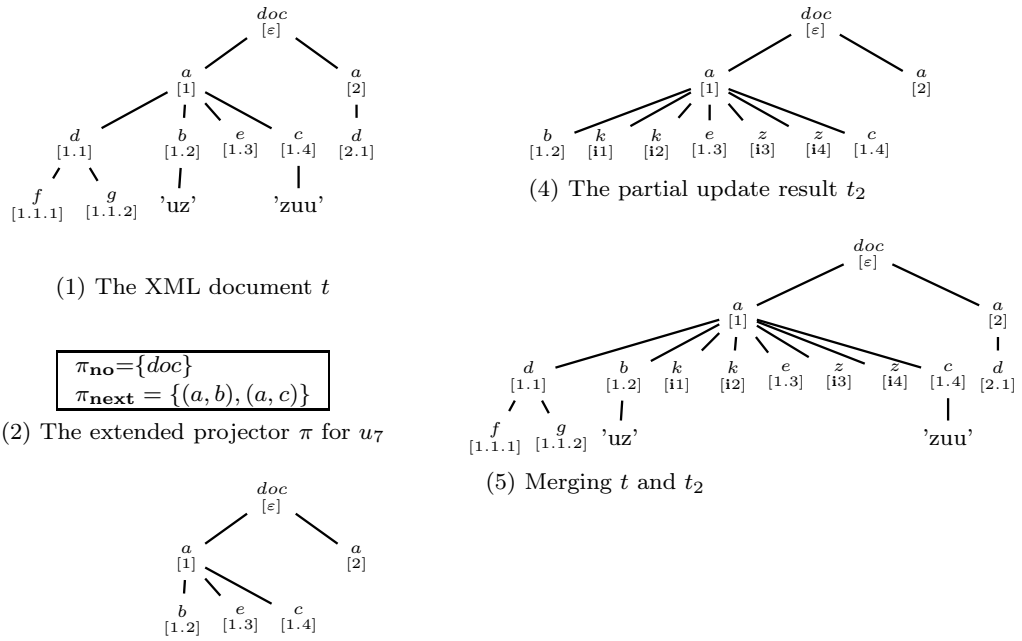


Figure 5.11: Mixing "insert before" and "insert after"

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 $\pi_{\mathbf{asfirst}}$ and $\pi_{\mathbf{bef}}$ (see fig. 5.12-2).

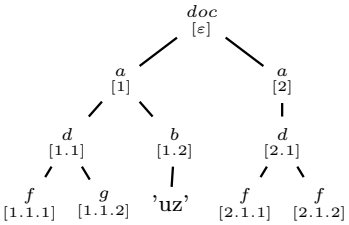
projection - The projection first outputs the root node t@ε labelled by $doc \in \pi_{\mathbf{no}}$. Then it projects t@1, since it is labelled by $a \in \pi_{\mathbf{asfirst}}$. Note that $a \in par(\pi_{\mathbf{bef}})$, thus the projection continues to parse the children of t@1. It selects the child t@1.2 since, $b \in ch(\pi_{\mathbf{bef}})$ (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 π_{ext}.

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

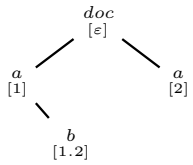
(1) The update query u_9

$\pi_{\mathbf{no}}=\{doc\}$
$\pi_{\mathbf{asfirst}}=\{a\}$
$\pi_{\mathbf{bef}}=\{(a,b)\}$

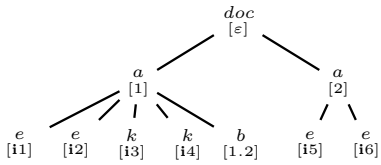
(2) The extended projector π for u_8



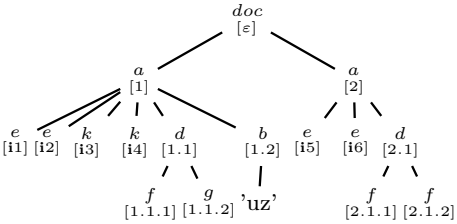
(3) The XML document t



(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"

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

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

Example 5.2.11. *See Figure 5.13.*

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

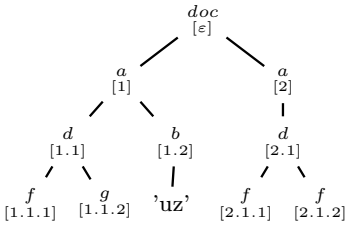
projection - Note that this time the projection not only outputs the node t@1.2 labelled by $b \in ch(\pi_{\mathbf{bef}})$, but also the first child of t@1 the node t@1.1 and the first child of t@2 labelled d (see fig. 5.13-3).

update - The tree rooted at t_2@1 besides the *"as first"* inserted elements t_2@**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@1.1 (see fig. 5.13-4).

merge - Because the node t@1 is labelled by $a \in \pi_{\mathbf{first}}$, while processing the nodes t@1 (see fig. 5.13-1) and t_2@1 (see fig. 5.13-4), *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@**i**1. Because the node t@1.1 is the first child of t@1 and the node t_2@**i**1 is the result of *"insert as first"* operation, *Merge* outputs t_2@**i**1. Next *Merge* considers the nodes t@1.1 and t_2@**i**2 and for the same reason t_2@**i**2 is output. Then, *Merge* examines the nodes t@1.1 and t_2@1.1 and because t@1.1 is labelled by $d \notin \pi_{ext}$ *Merge* outputs t@1.1. The next nodes being merged are t@1.2 and t_2@**i**3. Because t@1.2 is labelled by $b \in ch(\pi_{\mathbf{bef}})$ *Merge* outputs the *"inserted before"* node t_2@**i**3. The same is true when merging t@1.2 and t_2@**i**4. Finally, *Merge* examines the nodes t@1.2 and t_2@1.2 and because t@1.2 is labelled by $b \in ch(\pi_{\mathbf{bef}})$ outputs the node t_2@1.3. Therefore the first child t@1.1 of t@1 separates the nodes t_2@**i**1, t_2@**i**2 from t_2@**i**3, t_2@**i**4. □

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

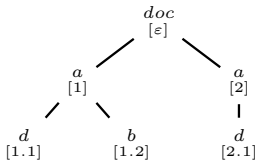
The next example involves an update generating operations of the kind *"insert as first"* and operations of the kind *"insert after"*. As opposed, to the previous cases, mixing these two kinds of operation can be dealt with using the solution provided by the analysis of update operations in isolation.



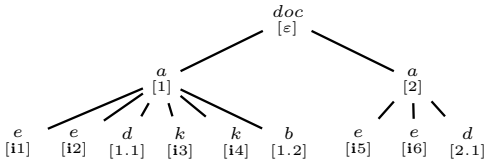
(1) The XML document t

$\pi_{\mathbf{no}}=\{doc\}$
$\pi_{\mathbf{bef}}=\{(a,b)\}$
$\pi_{\mathbf{first}}=\{a\}$

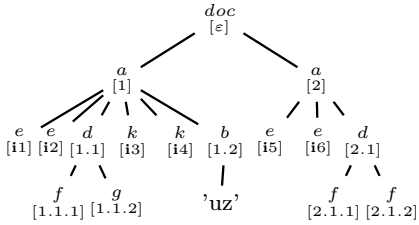
(2) The extended projector π for u_9



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



(4) The Partial Update result t_2



(5) Merging t and t_2

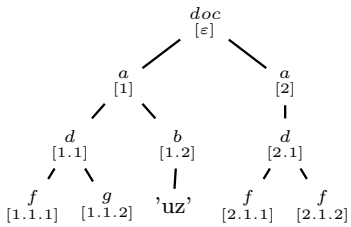
Figure 5.13: Mixing "insert as first" and "insert before"

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

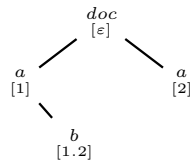
(1) The update query u_{10}

$\pi_{\mathbf{no}}=\{doc\}$
$\pi_{\mathbf{asfirst}}=\{a\}$
$\pi_{\mathbf{af}}=\{(a,b)\}$

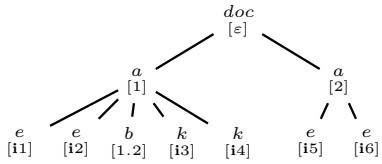
(2) The extended projector π for u_{10}



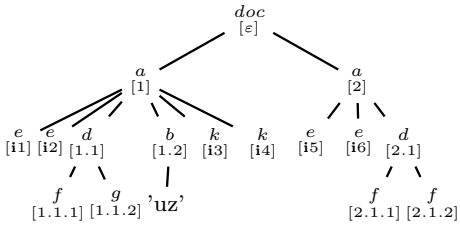
(3) The XML document t



(4) The projection t_1 of t



(5) The partial update result t_2



(6) Merging t and t_2

Figure 5.14: Mixing "insert as first" and "insert after"

Example 5.2.12. *See Figure 5.14.*

projector extraction - The projector contains components $\pi_{\mathbf{asfirst}}$ and $\pi_{\mathbf{af}}$ (see fig. 5.14-2).

projection - The projection first outputs the root node $t@\varepsilon$ labelled by $doc\in\pi_{\mathbf{no}}$. Then it projects $t@1$, since it is labelled by $a\in\pi_{\mathbf{asfirst}}$. Note that $a\in par(\pi_{\mathbf{af}})$, thus the projection continues to parse the children of $t@1$. It projects the child $t@1.2$, since $b\in ch(\pi_{\mathbf{af}})$ (see fig. 5.14-4). Finally it parses the subtree rooted at $t@2$, but prunes out its children, because their types do not belong to π_{ext}.

update - The result of the execution of the update u_{10} is given in Figure 5.14-5. The tree rooted at $t_2@1$ contains the elements $t_2@\mathbf{i}1$, $t_2@\mathbf{i}2$ labelled by e (inserted *"as first"*) and the elements $t_2@\mathbf{i}3$, $t_2@\mathbf{i}4$ labelled by k (inserted *"as after"*).

merge - *Merge* processes as follows (see fig.5.14-5): first it examines the node $t@1$ and $t_2@1$ and because $t@1$ is labelled by $a\in\pi_{\mathbf{asfirst}}$ *Merge* outputs $t_2@1$ followed by the nodes $t_2@\mathbf{i}1$ and $t_2@\mathbf{i}2$ (inserted *"as first"*). Then, *Merge* outputs the tree rooted by $t@1.1$, since it is labelled by $d\notin\pi_{ext}$. The nodes $t@1.2$ and $t_2@1.2$ are merged as follows: because $t@1.2$ is labelled by $b\in ch(\pi_{\mathbf{af}})$, *Merge* outputs the node $t_2@1.2$. Finally, the nodes $t_2@\mathbf{i}3$ and $t_2@\mathbf{i}4$ (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 $\pi_{\mathbf{af}}$ is sufficient to separate the nodes t_2@**i**1, t_2@**i**2 from t_2@**i**3, t_2@**i**4 . □

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 $\pi_{\mathbf{last}}$ (the dual of the projector component $\pi_{\mathbf{first}}$). Intuitivly, if a node type belongs to $\pi_{\mathbf{last}}$, 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: $\pi_{\mathbf{no}}$=$\{doc\}$, $\pi_{\mathbf{af}}$=$\{(a,d)\}$ and $\pi_{\mathbf{aslast}}$=$\{a\}$. Then, noticing that $\pi_{\mathbf{aslast}} \cap par(\pi_{\mathbf{af}})$=$\{a\}$, the new component $\pi_{\mathbf{last}}$ is set to $\{a\}$ and a is removed from $\pi_{\mathbf{aslast}}$ (see Fig. 5.15-2).

projection - The projection outputs not only the node t@1.1, since it is labelled by $d \in ch(\pi_{\mathbf{af}})$, 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@**i**1, t_2@**i**2 labelled by k (inserted *"after"*) and the elements t_2@**i**3, t_2@**i**4 labelled by c (inserted *"as last"*).

merge - Because the node t@1 is labelled by $a \in par(\pi_{\mathbf{af}})$, 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 $d \in ch(\pi_{\mathbf{af}})$, the node t_2@1.1 is output. The next nodes parsed are t@1.2 and t_2@**i**1 and *Merge* gives priority to outputting t_2@**i**1 since it is a new element. While processing t@1.2 and t_2@**i**2, for the same reason *Merge* outputs t_2@**i**2. 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@**i**3 and t_2@**i**4 (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@**i**1 and t_2@**i**2 (inserted *"after"*) from the nodes t_2@**i**3 and t_2@**i**4 (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" .....

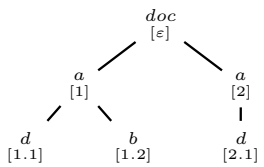
As it has been discussed in the subsection analyzing the update operation in isolation, when the update expression involves some *"replace"* operation, the type

```
for $x in /doc/a
return
{
insert nodes (<c/>, <c/>)
as last into $x
insert nodes (<k/>, <k/>)
after $x/d
}
```

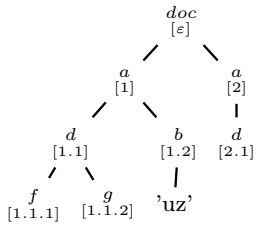
(1) The update query u_{11}

$\pi_{\mathbf{no}}=\{doc\}$
$\pi_{\mathbf{af}}=\{(a,d)\}$
$\pi_{\mathbf{last}}=\{a\}$

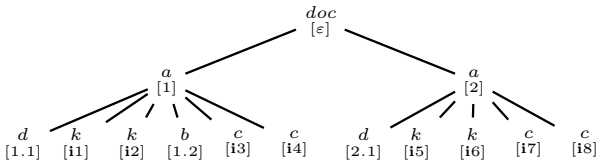
(2) The extended projector π for u_{11}



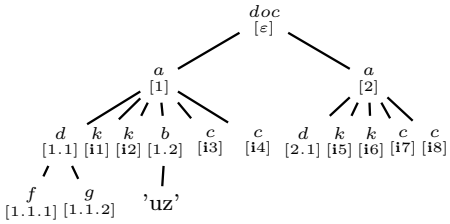
(4) The projection t_1 of t with $\pi_{\mathbf{last}}$



(3) The XML document t



(5) The partial update result t_2



(6) Merging t and t_2

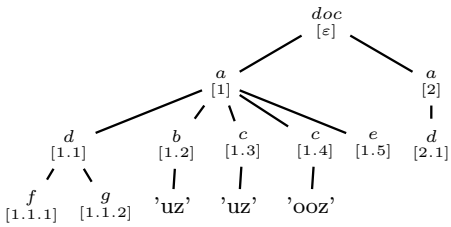
Figure 5.15: Mixing "insert as last" and "insert after"

```
for $x in /doc/a
return
{
replace node $x/d with (<k/>,<k/>)
insert node (<z/>) before $x/e
}
```

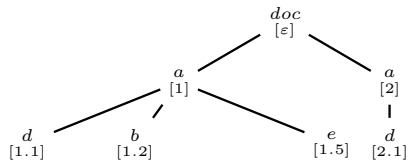
(1) The update query u_{12}

$\pi_{\mathbf{no}}$={*doc*}
$\pi_{\mathbf{next}} = \{(a,d),(a,e)\}$

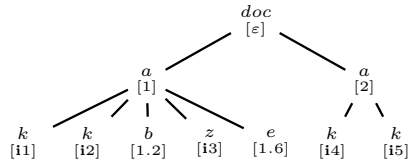
(2) The extended projector π_{ext} with $\pi_{\mathbf{next}}$



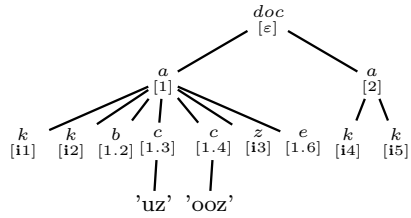
(3) The XML document t



(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 $\pi_{\mathbf{next}}$. When mixing *"replace"* operations with other kinds of operation, the use of the component $\pi_{\mathbf{next}}$ 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 $\pi_{\mathbf{no}}$={*doc*}, $\pi_{\mathbf{bef}}$={a,e} and $\pi_{\mathbf{next}}$={a,d}. Then, because $par(\pi_{\mathbf{bef}})\cap par(\pi_{\mathbf{next}})$={$a$}, the pair of types (a,e) is removed from $\pi_{\mathbf{bef}}$ and added to $\pi_{\mathbf{next}}$. Thus finally, $\pi_{\mathbf{next}}$={$(a,d),(a,e)$}.

projection - The projection outputs nodes $t@\varepsilon$, $t@1$ together with its children $t@1.1$ labelled by $d\in ch(\pi_{\mathbf{next}})$, $t@1.2$, because it is the next sibling of $t@1.1$ and $t@1.5$ labelled by $e\in ch(\pi_{\mathbf{next}})$. 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 $t_2@\mathbf{i1}$, $t_2@\mathbf{i2}$ labelled by k (inserted "in place of") and the elements $t_2@\mathbf{i3}$ labelled by z (

inserted *"before"*).

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 $a{\in}par(\pi_{\mathbf{next}})$, *Merge* outputs $t_2@1$ (see fig.5.16-6). Then, *Merge* examines the node $t@1.1$ and $t_2@\mathbf{i}1$. Because the first one is labelled by $d{\in}ch(\pi_{\mathbf{next}})$ and the identifier $\mathbf{i}1$ of the second one indicates that it is a new node, *Merge* outputs $t_2@\mathbf{i}1$. The next pair of the nodes being merged are $t@1.1$ and $t_2@\mathbf{i}2$. For the same reason, *Merge* outputs $t_2@\mathbf{i}2$. 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 $t_2@\mathbf{i}3$ labelled by z and because the first one is labelled by $c{\notin}\pi_{ext}$ *Merge* outputs $t@1.3$. The same is true for the pairs $t@1.4$ and $t_2@\mathbf{i}3$. The next two nodes to be merged are $t@1.5$ and $t_2@\mathbf{i}3$. Because $t@1.5$ is labelled by $e{\in}ch(\pi_{\mathbf{next}})$ *Merge* outputs $t_2@\mathbf{i}3$. Finally, *Merge* process the nodes $t@1.5$ and $t_2@1.5$ and outputs $t_2@1.5$. □

Mixing "replace" with insertion "after" .....

This case is of course similar to the previous one and the use of the projector component $\pi_{\mathbf{next}}$ is developed in the same manner.

Mixing "replace" with insertion "as first" .....

Dealing with this case requires more than the projector component $\pi_{\mathbf{next}}$ 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 $\pi_{\mathbf{no}}{=}\{doc\}$, $\pi_{\mathbf{asfirst}}{=}\{a\}$ and $\pi_{\mathbf{next}}{=}\{(a,b)\}$ (see fig. 5.17-2).

projection - The projection first selects the node $t@\varepsilon$, followed by $t@1$ labelled by $a{\in}\pi_{\mathbf{asfirst}}$. The child $t@1.2$ of $t@1$ is projected since it is labelled by $b{\in}ch(\pi_{\mathbf{next}})$. Finally, the node $t@2$ labelled by $a{\in}\pi_{\mathbf{asfirst}}$ 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 $t_2@1$ contains the elements $t_2@\mathbf{i}1$, $t_2@\mathbf{i}2$ labelled by d (inserted *"as first"*) and the element $t_2@\mathbf{i}3$ labelled by z (inserted "in place of").

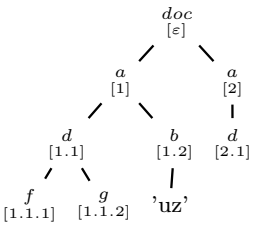
merge - *Merge* processes as follows (see fig.5.17-6): first it examines the node $t@1$ and $t_2@1$ and because $t@1$ is labelled by $a{\in}\pi_{\mathbf{asfirst}}$ it outputs $t_2@1$. Then, *Merge* outputs the nodes $t_2@\mathbf{i}1$, $t_2@\mathbf{i}2$ (inserted *"as first"*) and the node $t_2@\mathbf{i}3$ (inserted "in place of"). Once again, the issue here is that there is no information enabling to separate the $t_2@\mathbf{i}1$, $t_2@\mathbf{i}2$ (inserted *"as first"*) from the node $t_2@\mathbf{i}3$ (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/>)
}
```

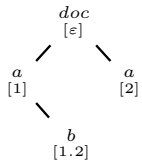
(1) The update query u_{13}

$\pi_{\mathbf{no}}=\{doc\}$
$\pi_{\mathbf{asfirst}} = \{a\}$
$\pi_{\mathbf{next}} = \{(a, b)\}$

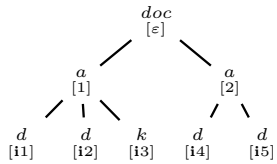
(2) The extended projector π_{ext}



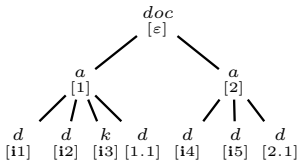
(3) The XML document t



(4) The projection t_1

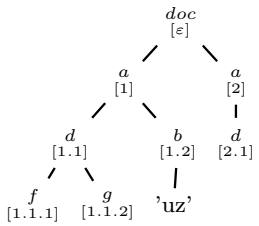


(5) The partial result t_2



(6)Attempt to merge t and t_2

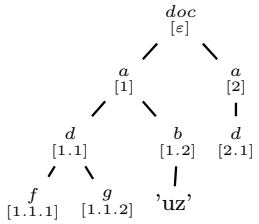
Figure 5.17: Mixing "replace" and "insert as first"



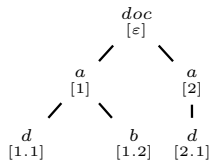
(1) Verification of $\pi_{\mathbf{first}}$

$\pi_{\mathbf{no}}$={*doc*}
$\pi_{\mathbf{next}}$={(*a*, *b*)}
$\pi_{\mathbf{first}}$={*a*}

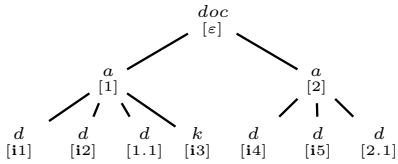
(2) The extended projector π_{ext}



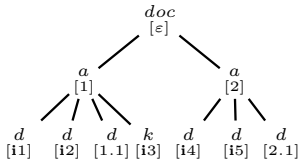
(3) The XML document t



(4) The projection t_1



(5) The partial result t_2



(6) Merging t and t_2

Figure 5.18: Mixing "replace" and "insert as first"

To the problem outlined in the previous example, the solution is the same as for mixing *"insert as fisrt"* and *"insert before"* and uses the separator projector $\pi_{\mathbf{first}}$.

Example 5.2.16. *See Figure 5.18*

projection extraction - The value of $\pi_{\mathbf{first}}$ is deduced from the intersection $\pi_{\mathbf{asfirst}} \cap par(\pi_{\mathbf{next}})$={*a*}. Therefore, $\pi_{\mathbf{first}}$={*a*}.

projection - This time, the projection of the initial document t includes the first children t@1.1 and t@2.1 of the nodes t@1 and t@2 respectively (see fig. 5.18-3).

update - The tree rooted at t_2@1 contains the new elements t_2@**i**1, t_2@**i**2 labelled by d (inserted *"as first"*), the node t_2@1.1 and the new element t_2@**i**3 labelled by k (inserted "in place of").

merge - Because the node t@1 is labelled by $a \in par(\pi_{\mathbf{next}})$, while processing the nodes t@1 (see fig. 5.18-1) and t_2@1 (see fig. 5.18-4), *Merge* outputs t_2@1 (see fig. 5.18-5). While parsing the children of t@1 and t_2@1, *Merge* first examines the nodes t@1.1 and t_2@**i**1. Because the node t@1.1 is the first child of t@1 and the node t_2@**i**1 is the result of the *"insert as first"* operation, *Merge* outputs t_2@**i**1. For the same reason, the next node output is t_2@**i**2. Then, *Merge* examines the nodes t@1.1 and t_2@1.1 and because t@1.1 is labelled by $d \notin \pi_{ext}$, *Merge* outputs t@1.1. The next

nodes to be merged are t@1.2 and t_2@**i**3. Because t@1.2 is labelled by $b \in ch(\pi_{\mathbf{next}})$, *Merge* outputs the new node t_2@**i**3 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 $\pi_{\mathbf{next}}$ as the separator.

Mixing "delete" with other kinds of update operation .....

This case is somehow sligthly more intricate and we decided to solve it by using the projector component $\pi_{\mathbf{olb}}$ 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 $\pi_{\mathbf{no}}=\{doc\}$, $\pi_{\mathbf{af}}=\{(a,d)\}$, $\pi_{\mathbf{bef}}=\{(a,e)\}$ and $\pi_{\mathbf{del}}=\{(a,b)\}$.
Based on the observation that $par(\pi_{\mathbf{af}}) \cap par(\pi_{\mathbf{bef}})=\{a\}$, the projector is modified to $\pi_{\mathbf{no}}=\{doc\}$, $\pi_{\mathbf{next}}=\{(a,d),(a,e)\}$ and $\pi_{\mathbf{del}}=\{(a,b)\}$.

projection - (See Fig. 5.19-4) The projection outputs the nodes t@ε labelled by $doc \in \pi_{\mathbf{no}}$, t@1 labelled by $a \in par(\pi_{\mathbf{next}})$ (note also that $a \in par(\pi_{\mathbf{del}})$) and the children of t@1: t@1.1 labelled by $d \in ch(\pi_{\mathbf{next}})$, t@1.2 labelled by $b \in ch(\pi_{\mathbf{del}})$ and t@1.4 labelled by $e \in ch(\pi_{\mathbf{next}})$. Indeed, the node t@1.2 is projected not only because its label belong to $ch(\pi_{\mathbf{del}})$ but also because it is the next sibling of t@1.1 and the label of t@1.1 belongs to $ch(\pi_{\mathbf{next}})$.

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@**i**1, t_2@**i**2 have been inserted after the node t_2@1.1, the node t@1.2 has been deleted and, finally, the nodes t_2@**i**3 and t_2@**i**4 have been inserted before the node t_2@1.4.
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.

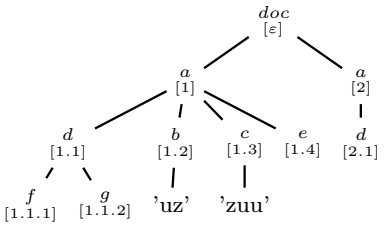
merge - While merging the children of t@1 and t_2@1, *Merge* first processes the nodes t@1.1 and t_2@1.1. Because t@1.1 is labelled by $d \in ch(\pi_{\mathbf{next}})$, *Merge* outputs t_2@1.1. Then, the nodes t@1.2 and t_2@**i**1 are parsed. Because t@1.2 labelled by $b \in ch(\pi_{\mathbf{del}})$, *Merge* outputs t_2@**i**1. Next, *Merge* examines the nodes t@1.2 and t_2@**i**2 and outputs t_2@**i**2, for the same reason. Finally, while examining the nodes t@1.2 and t_2@**i**3, *Merge* outputs t_2@**i**3 and thus fails to produce expected result. The issue is that the separator t@1.2 has been deleted. □.

```
for $x in /doc/a
return
{
insert nodes (<z/>, <z/>)
after $x/d
insert nodes (<h/>, <h/>)
before $x/e
delete node $x/b
}
```

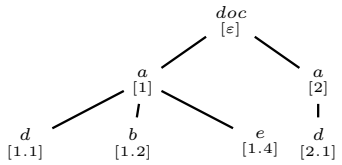
(1) The update query u_{14}

$\pi_{\mathbf{no}}=\{doc\}$
$\pi_{\mathbf{del}}=\{(a,b)\}$
$\pi_{\mathbf{next}}=\{(a,d),(a,e)\}$

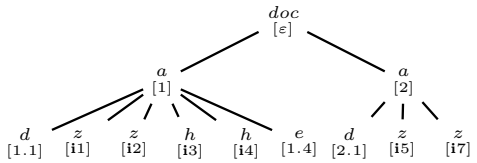
(2) The extended projector π_{ext} for u_{ext}



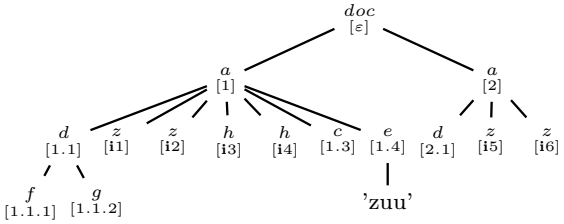
(3) The XML document t



(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 $\pi_{\mathbf{olb}}$ component. The precise solution is given in table 5.20.

5.3 Definition of the Extended Projection

This section is devoted to the formal presention 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 π_α where some of these components are sets of types ($\alpha{\in}un$) and the others components ($\alpha{\in}bin$) are sets of pairs of types. More precisely:

$bin{=}\{af, bef, del, next\}$ and for $\alpha{\in}bin, \pi_\alpha{\subseteq}\Sigma{\times}\Sigma$

$un = \{no, olb, eb, aslast, asfirst, first, last\}$, and for $\alpha{\in}un$ $\pi_\alpha{\subseteq}\Sigma$.

The following notations are used in the rest of the presentation:

– for $\alpha{\in}bin$, $x{\in}par(\pi_\alpha)$ iff $(x, y){\in}\pi_\alpha$ for some y,
– for $\alpha{\in}bin$ and for $x{\in}\Sigma$, $\pi_\alpha(x){=}\{y|(x, y){\in}\pi_\alpha\}$
– for $\alpha{\in}bin$, $y{\in}ch(\pi_\alpha)$ iff $(x, y){\in}\pi_\alpha$ for some x,
– $\pi{*}{=}\bigcup_{\alpha\in un}\pi_\alpha \bigcup_{\alpha\in bin}(par(\pi_\alpha) \cup ch(\pi_\alpha))$, and
– $\pi_{\mathbf{seed}}{=}\bigcup_{\alpha\in un-\{first,last\}}\pi_\alpha \bigcup_{\alpha\in bin} par(\pi_\alpha)$.

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 $\in\pi_{\mathbf{af}}$ with the column $\pi_{\mathbf{bef}}$ corresponds to the constraint:

if $(x, y){\in}\pi_{\mathbf{af}}$ and $(x, z){\in}\pi_{\mathbf{bef}}$ then (x, y) and (x, z) should belong to $\pi_{\mathbf{next}}$.

Note that the intersection of the row $\pi_{\mathbf{af}}$ with the column $\pi_{\mathbf{asfirst}}$ 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 Σ, an extended type projector π is defined by $\pi = \bigcup_{\alpha\in bin\cup un}\pi_\alpha$ 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 $a{\in}\pi_{\mathbf{seed}}$ 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 $\pi_{\mathbf{seed}}$. Notice here that we do not require disjointess of pairs of projector components.

(x,y) \ (x,z)	$\pi_{\mathbf{af}}$	$\pi_{\mathbf{bef}}$	$\pi_{\mathbf{del}}$	$\pi_{\mathbf{next}}$	$\pi_{\mathbf{asfirst}}$	$\pi_{\mathbf{aslast}}$
$\pi_{\mathbf{af}}$		$(x,y)\in\pi_{\mathbf{next}}$ $(x,z)\in\pi_{\mathbf{next}}$	$x\in\pi_{\mathbf{olb}}$	$(x,y)\in\pi_{\mathbf{next}}$ $(x,z)\in\pi_{\mathbf{next}}$	–	$x\in\pi_{\mathbf{last}}$
$\pi_{\mathbf{bef}}$			$x\in\pi_{\mathbf{olb}}$	$(x,y)\in\pi_{\mathbf{next}}$ $(x,z)\in\pi_{\mathbf{next}}$	$x\in\pi_{\mathbf{first}}$	–
$\pi_{\mathbf{del}}$				$x\in\pi_{\mathbf{olb}}$	–	–
$\pi_{\mathbf{next}}$					$x\in\pi_{\mathbf{first}}$	–
$\pi_{\mathbf{asfirst}}$						$x\in\pi_{\mathbf{first}}$

Figure 5.20: Constraints for the extended projector

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 π be a type projector for (D, s_D), and $t\in D$ be a tree with $roots(t)=\{r_t\}$ and $F=subfor(t)$. The π-projection of t, denoted $\pi(t)$, is the tree $\Pi_{K(t,\pi)}(t)$ where $K(t,\pi)$ is recursively defined by:

– $if\ lab(r_t) \notin \pi*$ then $K(t,\pi)=\emptyset$ otherwise
– $K(t,\pi)=\{r_t\}\bigcup_{\alpha\in\pi_{\mathbf{seed}}} K_\alpha(lab(r_t),F)$ 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 α is no or $asfirst$ or $aslast$ and $a\in\pi_\alpha$ then
A.1 $K_\alpha(a,F)=K(t',\pi)\cup K_\alpha(a,F')$ if $lab(r_{t'})\in\pi_{\mathbf{seed}}$
A.2 $K_\alpha(a,F)=K_\alpha(a,F')$ otherwise

B. if α is olb and $a\in\pi_{\mathbf{olb}}$ then
B.1 $\boldsymbol{K}_{\mathbf{olb}}(a,F)=K(t',\pi)\cup\boldsymbol{K}_{\mathbf{olb}}(a,F')$, if $lab(r_{t'})\in\pi_{\mathbf{seed}}$
B.2 $\boldsymbol{K}_{\mathbf{olb}}(a,F)=\{r_{t'}\}\cup\boldsymbol{K}_{\mathbf{olb}}(a,F')$, otherwise

C. if α is eb and $a\in\pi_{\mathbf{eb}}$ then $\boldsymbol{K}_{\mathbf{eb}}(a,F)=dom(F)$

D. if α is bef or $after$ or del and $a\in par(\pi_\alpha)$ then
D.1 $K_\alpha(a,F)=K(t',\pi)\cup K_\alpha(a,F')$ if $lab(r_{t'})\in\pi_{\mathbf{seed}}$
D.2 $K_\alpha(a,F)=\{r_{t'}\}\cup K_\alpha(a,F')$, if $(a,lab(r_{t'}))\in\pi_\alpha$
D.3 $K_\alpha(a,F)=K_\alpha(a,F')$ otherwise

E. if α is $next$ and $a\in par(\pi_{\mathbf{next}})$ then
E.1 $K_{next}(a,F)=K(t',\pi)\cup K_{next}(a,F')$ if $lab(r_{t'})\in\pi_{\mathbf{seed}}$
E.2 $K_{next}(a,F)=\{r_{t'}\}\cup K_{next}(a,F')\cup Next(F')$, if $(a,labr_{t'})\in\pi_{\mathbf{next}}$ where
$Next(F)=K(t',\pi)$ if $lab(r_{t'}))\in\pi_{\mathbf{seed}}$ and
$Next(F)=\{r_{t'}\}$ otherwise.
E.3 $K_{next}(a,F)=K_{next}(a,F')$, otherwise

F. if α is $first$ and $a \in \pi_{\mathbf{first}}$ then
$K_{first}(a, F) = K(t', \pi)$, if $lab(r_{t'}) \in \pi_{\mathbf{seed}}$

G. if α is $last$ and $a \in \pi_{\mathbf{last}}$ then (recall that $F = t' \circ F'$ and $F \neq ()$)
$K_{last}(a, F) = K_{last}(a, F')$ if $F' \neq ()$ otherwise - that is if $F' = ()$-
$K_{last}(a, F) = \{r_{t'}\}$

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 $(a, b) \in \pi_{\mathbf{af}}$ and moreover that $b \in \pi_{\mathbf{olb}}$. Then, at some point when using item D, because $a \in par(\pi_{\mathbf{af}})$, the item D.1 and D.2 will be applied because $b \in \pi_{\mathbf{olb}}$ and thus $b \in \pi_{\mathbf{seed}}$ and because $b \in ch(\pi_{\mathbf{af}})$.

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 π 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 *TreeMerge* takes as input two subtrees τ_i and τ'. The first one (τ_i) is a subtree of t. The second one (τ') is a subtree of t' (see Figure 5.21). Next, we assume that:

  $roots(\tau_i) = n$ $\quad$ $roots(\tau') = m$
  $lab(roots(\tau_i)) = a_i$ $\quad$ $lab(roots(\tau')) = b_i$
  $subfor(\tau_i) = F_i$ $\quad$ $subfor(\tau') = F_u$

  The procedure takes care of the synchronized parsing of the trees τ_i and τ'. The fact that the trees τ_i and τ' are synchronized implies that we assume that the trees τ_i and τ' have identical root identifier, that is $roots(\tau_i) = n = roots(\tau') = m$.

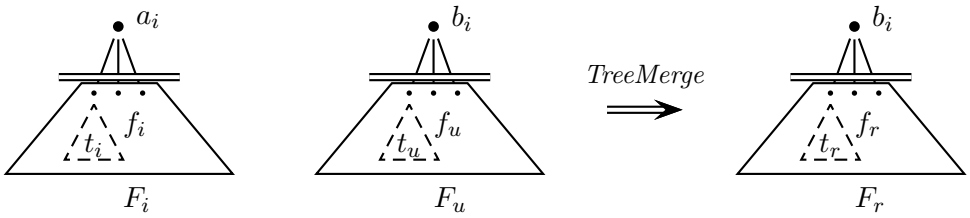


Figure 5.21: *TreeMerge* processing

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 τ' root; it checks the label a_i of n with respect to π 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 $a_i \in \pi_{\mathbf{no}}$, 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 definitions, we make the choice here not to consider the cases where $a_i \in \pi_{\mathbf{olb}}$ or where $a_i \in \pi_{\mathbf{eb}}$; these cases subsumes all cases introduced in this section ($a_i \in par(\pi_{\mathbf{bef}})$, $a_i \in par(\pi_{\mathbf{af}})$, ..., $a_i \in par(\pi_{\mathbf{next}})$) 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 τ_i and τ', it produces a subtree τ_r such that:

- $roots(\tau_r) = n = m$,
- $lab(roots(\tau_r)) = lab(roots(\tau')) = b_i$, and
- $subfor(\tau_r) =$

$NoMerge(F_i \mid F_u)$	if $a_i \in \pi_{\mathbf{no}}$
$AsFirstMerge(F_i \mid F_u)$	if $a_i \in \pi_{\mathbf{asfirst}}$
$AsLastMerge(F_i \mid F_u)$	if $a_i \in \pi_{\mathbf{aslast}}$
$DelMerge(F_i \mid F_u \mid a_i)$	if $a_i \in par(\pi_{\mathbf{del}})$
$BeforeMerge(F_i \mid F_u \mid a_i)$	if $a_i \in par(\pi_{\mathbf{bef}})$
$AfterMerge(F_i \mid F_u \mid a_i)$	if $a_i \in par(\pi_{\mathbf{af}})$
$NextMerge(F_i \mid F_u \mid a_i)$	if $a_i \in par(\pi_{\mathbf{next}})$

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=m) and such that the label a_i of the parent node of F_i only belongs to $\pi_{\mathbf{no}}$. It produces a sub-forest F_r of the final result $u(t)$.

First let us comment on the properties induced by the condition $a_i \in \pi_{\mathbf{no}}$. 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 π 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 $\pi_{\mathbf{seed}}$ because we assume that a_i belongs to $\pi_{\mathbf{no}}$ only.

Example 5.3.1. *Illustrating the behavior of NoMerge: see Figure 5.23.*
Let us consider the update u_1 specified by:

```
for $x in /a return rename node $x/b with "d" .
```

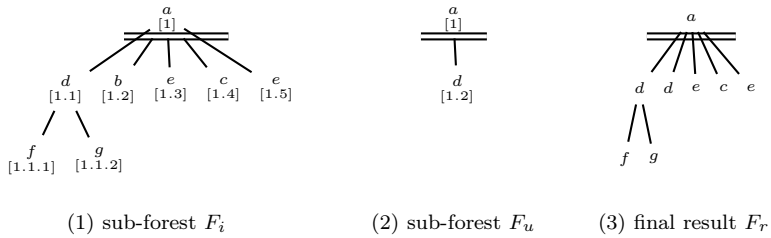
projector - The extended projector π_{ext} derived for the update u_1 contains one component $\pi_{\mathbf{no}}$=$\{a, b\}$.

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 $a \in \pi_{\mathbf{no}}$ (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 $d \notin \pi_{\mathbf{seed}}$, *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 $b \in \pi_{\mathbf{no}}$ 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. □

1 $NoMerge(F_i \mid F_u) =$ F_i if $roots(F_u)=\emptyset$
otherwise *(neither F_i nor F_u are empty)*
2 $t_i \circ NoMerge(f_i \mid F_u)$ if $lab(roots(t_i)) \notin \pi_{\mathbf{seed}}$
otherwise
3 $TreeMerge(t_i \mid t_u) \circ NoMerge(f_i \mid f_u)$

Figure 5.22: The procedure *NoMerge*



(1) sub-forest F_i (2) sub-forest F_u (3) final result F_r

Figure 5.23: Illustration of the behavior of *NoMerge*

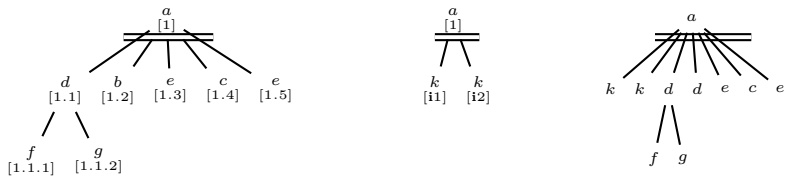
5.3.2.2 The procedure *AsFirstMerge*

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 $a_i \in \pi_{\mathbf{asfirst}}$ only. It produces a sub-forest F_r of the final result $u(t)$.
The properties induced by the condition $a_i \in \pi_{\mathbf{asfirst}}$ are following: ($\dagger_2$) the first level nodes of F_u are either new nodes (having fresh identifiers $\mathbf{i}x$) or nodes whose types are in $\pi_{\mathbf{seed}}$ and thus have been projected. Moreover, knowing that $a_i \in \pi_{\mathbf{asfirst}}$ implies that F_u potentially starts with new trees corresponding to the "as first" insertion.
The procedure *AsFirstMerge* is formally presented in Figure 5.24: line 2 outputs new inserted "as first" elements (indeed, *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.

Example 5.3.2. *Illustrating the behavior of AsFirstMerge: see Figure 5.25.*

1 $AsFirstMerge(F_i \mid F_u) =$ F_i if $roots(F_u)=\emptyset$
otherwise *(neither F_i nor F_u are empty)*
2 $t_u \circ AsFirstMerge(F_i \mid f_u)$ if $new(roots(t_u))=true$
otherwise
3 $t_i \circ AsFirstMerge(f_i \mid F_u)$ if $lab(roots(t_i)) \notin \pi_{\mathbf{seed}}$
otherwise
4 $TreeMerge(t_i \mid t_u) \circ AsFirstMerge(f_i \mid f_u)$

Figure 5.24: The procedure *AsFirstMerge*



(1) sub-forest F_i (2) sub-forest F_u (3) final result F_r

Figure 5.25: Illustration of the behavior of *AsFirstMerge*

Let us consider the update u_2 specified by:

```
for $x in /a return insert nodes {<k/>,<k/>} as first into $x
```

projector - The extended projector π_{ext} derived for this update u_2 has one component $\pi_{\mathbf{asfirst}}$=$\{a\}$.

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 $a \in \pi_{\mathbf{asfirst}}$, the forests F_i and F_u are merged by *AsFirstMerge* (see fig. 5.24).
First, *AsFirstMerge* parses the nodes F_i@1.1 and F_u@**i**1. Because the identifier of F_u@**i**1 indicates that it is a new inserted node ($new(roots(F_u@\mathbf{i}1))$=$true$), *AsFirstMerge* executes line 2 and outputs the tree rooted at F_u@**i**1. Next, *AsFirstMerge* examines the nodes F_i@1.1 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_u has been totally parsed, line 1 is executed leading to output the remaining subtrees of F_i. □

5.3.2.3 The procedure *AsLastMerge*

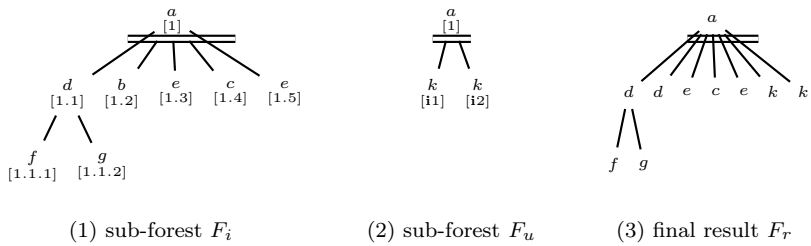
This case is similar to the previous one. This time the assumption that a_i belongs to $\pi_{\mathbf{aslast}}$ only, entails that F_u potentially ends with new trees corresponding to the "as last" insertion.
The properties induced by this condition are following: ($\dagger_3$) the first level nodes of F_u are either new nodes (having fresh identifiers ix) or nodes whose types are in $\pi_{\mathbf{seed}}$ and thus have been projected. Moreover, knowing that $a_i \in \pi_{\mathbf{aslast}}$ 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:

```
for $x in /a return insert nodes {<k/>,<k/>} as last into $x.
```

1 $AsLastMerge(F_i \mid F_u) =$ F_i if $roots(F_u)=\emptyset$
2 F_u if $roots(F_i)=\emptyset$
otherwise *(neither F_i nor F_u are empty)*
3 $t_i \circ AsLastMerge(f_i \mid F_u)$ if $lab(roots(t_i)) \notin \pi_{\mathbf{seed}}$
otherwise
4 $TreeMerge(t_i \mid t_u) \circ AsLastMerge(f_i \mid f_u)$

Figure 5.26: The procedure *AsLastMerge*



(1) sub-forest F_i (2) sub-forest F_u (3) final result F_r

Figure 5.27: Illustration of the behavior of *AsLastMerge*

projector - The extended projector π_{ext} derived from the update u_3 has one component $\pi_{\mathbf{aslast}}=\{a\}$.

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@**i**1 and F_u@**i**2 labelled by k.

merge - (see fig. 5.27-3). Because the parent node of F_i is labelled by $a \in \pi_{\mathbf{aslast}}$, the forests F_i and F_u merged by *AsLastMerge* (see fig. 5.26).
First, *AsLastMerge* examines the nodes F_i@1.1 and F_u@**i**1. Because F_i@1.1 is labelled by $d \notin \pi_{\mathbf{seed}}$, line 3 is executed and *AsLastMerge* outputs the tree rooted at F_i@1.1. Next the nodes F_i@1.2 and F_u@**i**1 are parsed and, for the same reason as before, line 3 is executed and outputs the tree rooted at F_i@1.2. The remaining subtrees of F_i are processed in the same manner. Finally, when F_i has been totally parsed, *AsLastMerge* executes line 2 and outputs the forest F_u containing the new trees rooted at F_u@**i**1 and F_u@**i**2. □

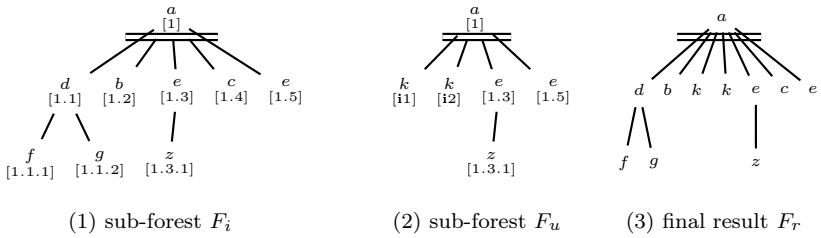
5.3.2.4 The procedure *BeforeMerge*

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 $par(\pi_{\mathbf{bef}})$. It produces a sub-forest F_r of the final result $u(t)$.

The properties induced by the condition $a_i \in par(\pi_{\mathbf{bef}})$ are the following: ($\dagger_4$) the first level nodes of F_u are either new nodes (having fresh identifiers **i**x) or nodes whose types are in $\pi_{\mathbf{seed}}$ and thus have been projected, or nodes whose types are in $\pi_{\mathbf{bef}}(a_i)$ 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

$$
\begin{array}{lrl}
 & \textit{BeforeMerge}(F_i \mid F_u \mid x) = & \\
1 & F_i & \text{if } \textit{roots}(F_u)=\emptyset \\
 & \textbf{otherwise } \textit{(neither } F_i \textit{ nor } F_u \textit{ are empty)} & \\
2 & t_i \circ \textit{BeforeMerge}(f_i \mid F_u \mid x) & \text{if } \textit{lab}(\textit{roots}(t_i)) \notin \pi_{\mathbf{bef}}(x) \cup \pi_{\mathbf{seed}} \\
3 & \textit{TreeMerge}(t_i \mid t_u) \circ \textit{BeforeMerge}(f_i \mid f_u \mid x) & \text{if } \textit{roots}(t_i)=\textit{roots}(t_u) \\
4 & t_u \circ \textit{BeforeMerge}(F_i \mid f_u \mid x) & \text{if } \textit{new}(\textit{roots}(t_u))=\textit{true}
\end{array}
$$

Figure 5.28: The procedure *BeforeMerge*



(1) sub-forest F_i (2) sub-forest F_u (3) final result F_r

Figure 5.29: Illustration of the behavior of *BeforeMerge*

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 $\pi_{\mathbf{bef}}(x)$ (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 π_{ext} derived for the update u_4 contains one component $\pi_{\mathbf{bef}}{=}\{(a, e)\}$.

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@**i**1 and F_u@**i**2 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 $a{\in}par(\pi_{\mathbf{bef}})$, 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@**i**1. Because F_i@1.1 is labelled by $d{\notin}\pi_{\mathbf{bef}}(a){\cup}\pi_{\mathbf{seed}}$, *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@**i**1 and, for the

	$AfterMerge(F_i \mid F_u \mid x) =$	
1	F_i	if $roots(F_u)=\emptyset$
2	F_u	if $roots(F_i)=\emptyset$
	otherwise *(neither F_i nor F_u are empty)*	
3	$t_i \circ AfterMerge(f_i \mid F_u \mid x)$	if $lab(roots(t_i)) \notin \pi_{\mathbf{af}}(x) \cup \pi_{\mathbf{seed}}$ and $new(roots(t_u))=false$
4	$TreeMerge(t_i \mid t_u) \circ AfterMerge(f_i \mid f_u \mid x)$	if $roots(t_i)=roots(t_u)$
5	$t_u \circ AfterMerge(F_i \mid f_u \mid x)$	if $new(roots(t_u))=true$

Figure 5.30: The procedure *AfterMerge*

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@**i**1. Because F_i@1.3 is labelled by $e \in ch(\pi_{\mathbf{bef}})$ and the identifier of F_u@**i**1 indicates that it is a new inserted node ($new(roots(F_u@\mathbf{i}1))=true$), *BeforeMerge* executes line 4 and outputs the tree rooted at F_u@**i**1. Next, the nodes F_i@1.3 and F_u@**i**2 are parsed and, for the same reason as before, line 4 is executed and *BeforeMerge* outputs the tree rooted at F_u@**i**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 ($roots(F_i@1.3)=roots(F_u@1.3)$), 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 $c \notin \pi_{\mathbf{bef}}(a) \cup \pi_{\mathbf{seed}}$, *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 $par(\pi_{\mathbf{af}})$. It produces a sub-forest F_r of the final result $u(t)$.

The properties induced by the condition $a_i \in par(\pi_{\mathbf{af}})$ are the same as for ($\dagger_4$) except that this time, the new elements are inserted "after".

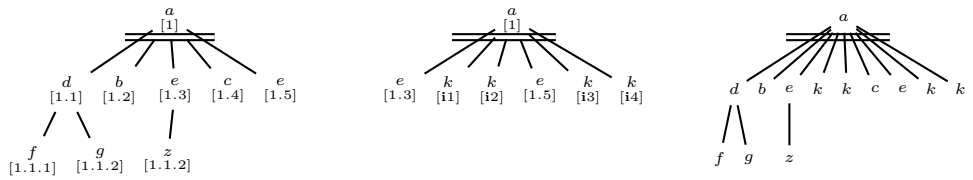
The procedure *AfterMerge* is formally defined in Figure 5.30 and do not present any new difficulties.

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 π_{ext} derived from the update u_4 contains one component $\pi_{\mathbf{af}}=\{(a, e)\}$.

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@**i**1, F_u@**i**2, F_u@**i**3



(1) sub-forest F_i (2) sub-forest F_u (3) final result F_r

Figure 5.31: Illustration of the behavior of *AfterMerge*

and F_u@**i**4 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 $a \in par(\pi_{\mathbf{af}})$, the forests F_i and F_u are merged by *AfterMerge* (see fig. 5.30).
First, *AfterMerge* examines the nodes F_i@1.1 and F_u@1.3. Because F_i@1.1 is labelled by $d \notin \pi_{\mathbf{bef}}(a) \cup \pi_{\mathbf{seed}}$ and the identifier of F_u@1.3 indicates that it is not a new inserted node ($new(roots(F_u@1.3))=false$), line 3 is executed and *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 *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 ($roots(F_i@1.3)=roots(F_u@1.3)$), leading to the execution of line 4: *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,*AfterMerge* examines the nodes F_i@1.4 and F_u@**i**1. Because the identifier of F_u@**i**1 indicates that it is a new inserted node ($new(roots(F_u@\mathbf{i}1))=true$), *AfterMerge* executes line 5 and outputs the tree rooted at F_u@**i**1. The next two nodes processed are F_i@1.4 and F_u@**i**2 and, for the same reason as before, line 5 is executed and *AfterMerge* outputs the tree rooted at F_u@**i**2. After that step, *AfterMerge* examines the nodes F_i@1.4 and F_u@1.5. Because the two nodes are equal ($roots(F_i@1.5)=roots(F_u@1.5)$), *AfterMerge* executes line 4. Finally, *AfterMerge* executes line 2 and outputs the forest F_u containing the new trees rooted at F_u@**i**3 and F_u@**i**4. □

5.3.2.6 The procedure *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 $par(\pi_{\mathbf{next}})$. It produces a sub-forest F_r of the final result $t(u)$.

The procedure *DelMerge* is formally presented in Figure 5.32. Roughly, it behaves like the procedure *NoMerge* specified in the context of the three-level projector.

5.3.2.7 The procedure *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 $par(\pi_{\mathbf{next}})$. It produces a sub-forest F_r of the final result $t(u)$.

1	$DelMerge(F_i \mid F_u \mid x) =$ F_u	if $roots(F_i)=\emptyset$
	otherwise *(F_i is not empty)*	
2	$t_i \circ DelMerge(f_i \mid F_u \mid x)$	if $lab(roots(t_i)) \notin \pi_{\mathbf{del}}(x) \cup \pi_{\mathbf{seed}}$
	otherwise, assuming $roots(F_u)\neq\emptyset$ **then**	
3	$TreeMerge(t_i \mid t_u) \circ DelMerge(f_i \mid f_u \mid x)$	if $roots(t_i)=roots(t_u)$
4	$DelMerge(f_i \mid F_u \mid x)$	if $roots(t_i)<roots(t_u)$
	otherwise, assuming $roots(F_u)=\emptyset$ **then**	
5	$DelMerge(f_i \mid F_u \mid x)$	

Figure 5.32: The procedure *DelMerge*

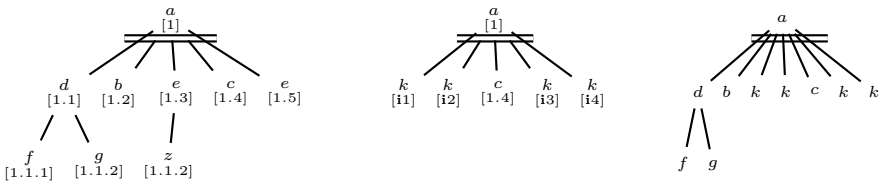
	$NextMerge(F_i \mid F_u \mid x) =$	
1	F_u	if $roots(F_i)=\emptyset$
	otherwise *(F_i is not empty),*	
	assuming $roots(F_u)\neq\emptyset$ and $lab(roots(t_i)) \notin \pi_{\mathbf{next}}(x) \cup \pi_{\mathbf{seed}}$	
2	$t_i \circ NextMerge(f_i \mid f_u \mid x)$	if $roots(t_i)=roots(t_u)$
3	$t_i \circ NextMerge(f_i \mid F_u \mid x)$	if $roots(t_i)<roots(t_u)$
		or $(new(roots(t_u))=true)$
	otherwise still assuming $roots(F_u)\neq\emptyset$	
4	$t_u \circ NextMerge(F_i \mid f_u \mid x)$	if $new(roots(t_u))=true$
5	$NextMerge(f_i \mid F_u \mid x)$	if $roots(t_i)<roots(t_u)$
6	$TreeMerge(t_i \mid t_u) \circ NextMerge(f_i \mid f_u \mid x)$	if $roots(t_i)=roots(t_u)$
	otherwise $(roots(F_u)=\emptyset)$	
7	F_i	if $lab(roots(t_i)) \notin \pi$
8	$()$	if $lab(roots(t_i)) \in \pi_{\mathbf{next}}(x)$

Figure 5.33: The procedure *NextMerge*

The properties induced by the condition $a_i \in par(\pi_{\mathbf{next}})$ are the following: ($\dagger_5$) the first level nodes of F_u are either nodes whose types are in $\pi_{\mathbf{seed}}$ and thus have been projected or nodes whose types belong to $\pi_{\mathbf{next}}(a_i)$ and have not been replaced, followed by nodes that have been projected because they are the next siblings of nodes whose types belong to $\pi_{\mathbf{next}}(a_i)$ 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 $\pi_{\mathbf{seed}}$). 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.*



(1) sub-forest F_i (2) sub-forest F_u (3) final result F_r

Figure 5.34: Illustration of the behavior of *NextMerge*

Let us consider the update u_6 specified by:

```
for $x in /a return replace node $x/e with {<k/>,<k/>}.
```

projector - The extended projector π_{ext} derived from the update u_6 has one components $\pi_{\mathbf{next}}=\{(a,e)\}$.

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@**i**1, F_u@**i**2, F_u@**i**3 and F_u@**i**4 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 $a{\in}par(\pi_{\mathbf{next}})$, the forests F_i and F_u are merged by *NextMerge* (see fig. 5.33).
First, *NextMerge* examines the nodes F_i@1.1 and F_u@**i**1. Because F_i@1.1 is labelled by $d{\notin}\pi_{\mathbf{bef}}(a){\cup}\pi_{\mathbf{seed}}$ and the identifier of F_u@**i**1 indicates that it is a new inserted node ($new(roots(F_u@\mathbf{i}1))=true$), line 3 is executed and *NextMerge* outputs the tree rooted at F_i@1.1. The next two nodes processed are F_i@1.2 and F_u@**i**1 and, for the same reason as before, line 3 is executed and *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@**i**1. Because F_i@1.3 is labelled by $e{\in}ch(\pi_{\mathbf{next}})$ and the identifier of F_u@**i**1 indicates that it is a new inserted node ($new(roots(F_u@\mathbf{i}1))=true$), *NextMerge* executes line 4 and outputs the tree rooted at F_u@**i**1. Next, *NextMerge* examines the nodes F_i@1.3 and F_u@**i**2 and, for the same reason as before, line 4 is executed and *NextMerge* outputs the tree rooted at F_u@**i**2. The next two nodes processed are F_i@1.3 and F_u@1.4. Because the identifier of F_i@1.3 is less than the one of F_u@1.4 line 5 is executed and *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 $c{\notin}\pi_{\mathbf{bef}}(a){\cup}\pi_{\mathbf{seed}}$ and the identifiers of F_i@1.4 and F_u@1.4 are equal ($roots(F_u@1.4)=roots(F_u@1.4)$) *NextMerge* executes line 2 and outputs the tree rooted at F_i@1.4. Next, *NextMerge* parses the nodes F_i@1.5 and F_u@**i**3. Because F_i@1.5 is labelled by $e{\in}ch(\pi_{\mathbf{next}})$ and the identifier of F_u@**i**3 indicates that it is a new inserted node ($new(roots(F_u@\mathbf{i}3))=true$), *NextMerge* executes line 4 and outputs the tree rooted at F_u@**i**3. Next, *NextMerge* examines the nodes F_i@1.5 and F_u@**i**4 and, for the same reason as before, line 4 is executed and *NextMerge* outputs the tree rooted at F_u@**i**4. Finally, since the forest F_u has been totally parsed and F_i@1.5 is labelled by $e{\in}ch(\pi_{\mathbf{next}})$ *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 happend. 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 $par(\pi_{\mathbf{bef}})$ and $par(\pi_{\mathbf{af}})$ only because in that case a_i should belong to $par(\pi_{\mathbf{next}})$ and has been deleted from $par(\pi_{\mathbf{bef}})$ and $par(\pi_{\mathbf{af}})$.
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 $a_i{\in}\pi_{\mathbf{olb}}$ or where $a_i{\in}\pi_{\mathbf{eb}}$; 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 $a_i{\in}\pi_{\mathbf{no}}$; for instance, $a_i{\in}par(\pi_{\mathbf{bef}})$ and $a_i{\in}\pi_{\mathbf{aslast}}$ means that the label a_i belongs to $par(\pi_{\mathbf{bef}})$ and to $\pi_{\mathbf{aslast}}$ and may be also to $\pi_{\mathbf{no}}$.
A similar convention is made for $a_i{\in}A$ and $a_i{\in}B$ and $a_i{\in}C$.

Given two synchronized subtrees τ_i and τ', *TreeMerge* produces a subtree τ_r such that:

- $roots(\tau_r){=}n{=}m$,
- $lab(roots(\tau_r)){=}lab(roots(\tau'))){=}b_i$, and
- $subfor(\tau_r) =$

$AsFirstDelMerge(F_i \mid F_u \mid a_i)$	if $a_i{\in}par(\pi_{\mathbf{del}})$ and $a_i{\in}\pi_{\mathbf{asfirst}}$
$AsLastDelMerge(F_i \mid F_u \mid a_i)$	if $a_i{\in}par(\pi_{\mathbf{del}})$ and $a_i{\in}\pi_{\mathbf{aslast}}$
$BeforeMerge(F_i \mid F_u \mid a_i)$	if $a_i{\in}par(\pi_{\mathbf{bef}})$ and $a_i{\in}\pi_{\mathbf{aslast}}$
$AfterMerge(F_i \mid F_u \mid a_i)$	if $a_i{\in}par(\pi_{\mathbf{af}})$ and $a_i{\in}\pi_{\mathbf{asfirst}}$ or $a_i{\in}par(\pi_{\mathbf{af}})$ and $a_i{\in}\pi_{\mathbf{last}}$ or $a_i{\in}par(\pi_{\mathbf{af}})$ and $a_i{\in}\pi_{\mathbf{asfirst}}$ and $a_i{\in}\pi_{\mathbf{last}}$
$NextMerge(F_i \mid F_u \mid a_i)$	if $a_i{\in}par(\pi_{\mathbf{next}})$ and $a_i{\in}\pi_{\mathbf{aslast}}$ or $a_i{\in}par(\pi_{\mathbf{next}})$ and $a_i{\in}\pi_{\mathbf{first}}$ or $a_i{\in}par(\pi_{\mathbf{next}})$ and $a_i{\in}\pi_{\mathbf{aslast}}$ and $a_i{\in}\pi_{\mathbf{first}}$
$FirstMerge(F_i \mid F_u \mid a_i)$	if $a_i{\in}par(\pi_{\mathbf{bef}})$ and $a_i{\in}\pi_{\mathbf{first}}$ or $a_i{\in}\pi_{\mathbf{aslast}}$, and $a_i{\in}\pi_{\mathbf{first}}$ or $a_i{\in}par(\pi_{\mathbf{bef}})$, and $a_i{\in}\pi_{\mathbf{aslast}}$, and $a_i{\in}\pi_{\mathbf{first}}$

1	$AsFirstDelMerge(F_i \mid F_u \mid x) =$	F_u	if $roots(F_i)=\emptyset$
	otherwise$(F_i$ *is not empty*$)$,		
2	$t_i \circ AsFirstDelMerge(f_i \mid F_u \mid x)$		if $lab(roots(t_i)) \notin \pi_{\mathbf{del}}(x) \cup \pi_{\mathbf{seed}}$ and $[roots(F_u)=\emptyset$ or $new(roots(t_u))=false]$
	otherwise, assuming $roots(F_u)\neq\emptyset$ **then**		
3	$TreeMerge(t_i \mid t_u) \circ AsFirstDelMerge(f_i \mid f_u \mid x)$		if $roots(t_i)=roots(t_u)$
4	$AsFirstDelMerge(f_i \mid F_u \mid x)$		if $roots(t_i)<roots(t_u)$
5	$t_u \circ AsFirstDelMerge(F_i \mid f_u \mid x)$		if $new(roots(t_u))=true$
	otherwise, assuming $roots(F_u)=\emptyset$ **then**		
6	$AsFirstDelMerge(f_i \mid F_u \mid x)$		

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 $par(\pi_{\mathbf{del}})$ and $\pi_{\mathbf{asfirst}}$ (and may be to $\pi_{\mathbf{no}}$ 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 $par(\pi_{\mathbf{del}})$ and $\pi_{\mathbf{aslast}}$ (and may be to $\pi_{\mathbf{no}}$ 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 $par(\pi_{\mathbf{bef}})$ and $\pi_{\mathbf{aslast}}$ (and may be to $\pi_{\mathbf{no}}$ also).

The properties induced by the condition $a_i \in par(\pi_{\mathbf{bef}})$ and $\pi_{\mathbf{aslast}}$ are the following: ($\dagger_6$) the first level nodes of F_i are either new nodes (having fresh identifiers $\mathbf{i}x$) or

1	$AsLastDelMerge(F_i \mid F_u \mid x) =$ F_u	if $roots(F_i)=\emptyset$
	otherwise$(F_i$ *is not empty*$)$,	
2	$t_i \circ AsLastDelMerge(f_i \mid F_u \mid x)$	if $lab(roots(t_i)) \notin \pi_{\mathbf{del}}(x) \cup \pi_{\mathbf{seed}}$
	otherwise, assuming $roots(F_u)\neq\emptyset$ **then**	
3	$TreeMerge(t_i \mid t_u) \circ AsLastDelMerge(f_i \mid f_u \mid x)$	if $roots(t_i)=roots(t_u)$
4	$AsLastDelMerge(f_i \mid F_u \mid x)$	if $roots(t_i)<roots(t_u)$
5	$AsLastDelMerge(f_i \mid F_u \mid x)$	if $new(roots(t_u))=true$
	otherwise, assuming $roots(F_u)=\emptyset$ **then**	
6	$AsLastDelMerge(f_i \mid F_u \mid x)$	

Figure 5.36: The procedure *AsLastDelMerge* -general case -

	$BeforeMerge(F_i \mid F_u \mid x) =$	
1	F_i	if $roots(F_u)=\emptyset$
2	F_u	if $roots(F_i)=\emptyset$
	otherwise *(neither* F_i *nor* F_u *are empty)*	
3	$t_i \circ BeforeMerge(f_i \mid F_u \mid x)$	if $lab(roots(t_i)) \notin \pi_{\mathbf{bef}}(x) \cup \pi_{\mathbf{seed}}$
4	$TreeMerge(t_i \mid t_u) \circ BeforeMerge(f_i \mid f_u \mid x)$	if $roots(t_i)=roots(t_u)$
5	$t_u \circ BeforeMerge(F_i \mid f_u \mid x)$	if $new(roots(t_u))=true$

Figure 5.37: The procedure *BeforeMerge* -general case -

nodes whose types are in $\pi_{\mathbf{seed}}$ and thus have been projected, or nodes whose types are in $\pi_{\mathbf{bef}}(a_i)$ 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* givrn 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 ($par(\pi_{\mathbf{af}})$ and $\pi_{\mathbf{asfirst}}$) or ($par(\pi_{\mathbf{af}})$ and $\pi_{\mathbf{last}}$) or ($par(\pi_{\mathbf{af}})$ and $\pi_{\mathbf{asfirst}}$ and $\pi_{\mathbf{last}}$).

$AfterMerge(F_i \mid F_u \mid x) =$

1 $\quad F_i$ if $roots(F_u)=\emptyset$

otherwise *(F_u is not empty),*
assuming $roots(F_i)\neq\emptyset$ and $lab(roots(t_i))\notin\pi_{\mathbf{af}}(x)\cup\pi_{\mathbf{seed}}$

2 $\quad t_i \circ AfterMerge(f_i \mid F_u \mid x)$ if $new(roots(t_u))=false$

3 $\quad t_u \circ AfterMerge(F_i \mid f_u \mid x)$ if $new(roots(t_u))=true$

otherwise still assuming $roots(F_i)\neq\emptyset$

4 $\quad TreeMerge(t_i \mid t_u) \circ AfterMerge(f_i \mid f_u \mid x)$ if $roots(t_i)=roots(t_u)$

otherwise $(roots(F_i)=\emptyset)$

5 $\quad AfterMerge(F_i \mid f_u \mid x)$ if $new(roots(t_u)=false$

6 $\quad F_u$ if $new(roots(t_u))=true$

Figure 5.38: The procedure *AfterMerge* - general case -

The properties induced by the conditions given above are the following: ($\dagger_7$) the first level nodes of F_u are either new nodes (having fresh identifiers $\mathbf{i}x$) or nodes whose types are in $\pi_{\mathbf{seed}}$ and thus have been projected, or nodes whose types are in $\pi_{\mathbf{af}}(a_i)$ 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 $\pi_{\mathbf{last}}$.
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 $\pi_{\mathbf{last}}$ 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 ($par(\pi_{\mathbf{next}})$ and $\pi_{\mathbf{aslast}}$) or ($par(\pi_{\mathbf{next}})$ and $\pi_{\mathbf{first}}$) or ($par(\pi_{\mathbf{next}})$ and $\pi_{\mathbf{aslast}}$ and $\pi_{\mathbf{first}}$).

The properties induced by the conditions given above are the following: ($\dagger_8$) the first level nodes of F_u are either nodes whose types are in $\pi_{\mathbf{seed}}$ and thus have been projected or nodes whose types belong to $\pi_{\mathbf{next}}(a_i)$ and have not been replaced,

$NextMerge(F_i \mid F_u \mid x) =$

1 F_u if $roots(F_i)=\emptyset$

otherwise *(F_i is not empty),*

assuming $roots(F_u)\neq\emptyset$ and $lab(roots(t_i))\notin\pi_{\mathbf{next}}(x)\cup\pi_{\mathbf{seed}}$

2 $t_i\circ NextMerge(f_i \mid f_u \mid x)$ if $roots(t_i)=roots(t_u)$

3 $t_i\circ NextMerge(f_i \mid F_u \mid x)$ if $roots(t_i)<roots(t_u)$
or $(new(roots(t_u))=true$ and $first(roots(t_i))=false)$

4 $t_u\circ NextMerge(F_i \mid f_u \mid a)$ if $first(roots(t_i)) = true$ and $new(roots(t_u))=true$

otherwise still assuming $roots(F_u)\neq\emptyset$

5 $t_u\circ NextMerge(F_i \mid f_u \mid x)$ if $new(roots(t_u))=true$

6 $NextMerge(f_i \mid F_u \mid x)$ if $roots(t_i)<roots(t_u)$

7 $TreeMerge(t_i \mid t_u)\circ NextMerge(f_i \mid f_u \mid x)$ if $roots(t_i)=roots(t_u)$

otherwise $(roots(F_u)=\emptyset)$

8 F_i if $lab(roots(t_i)) \notin \pi$

9 $()$ if $lab(roots(t_i))\in\pi_{\mathbf{next}}(x)$

Figure 5.39: The procedure *NextMerge* - general case -

followed by nodes that have been projected because they are the next siblings of nodes whose types belong to $\pi_{\mathbf{next}}(a_i)$, 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 $\pi_{\mathbf{first}}$.
This condition ensures that no delete operation could be performed by the update u over the first level nodes of F_i.

The procedure *NextMerge* is formally presented in Figure 5.39. It uses the function $first$ 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: $new(roots(t_u))=true$ and $first(roots(t_i))=false$ 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 ($par(\pi_{\mathbf{bef}})$ and $\pi_{\mathbf{first}}$) or to ($par(\pi_{\mathbf{bef}})$ and $\pi_{\mathbf{aslast}}$ and $\pi_{\mathbf{first}}$) or to ($\pi_{\mathbf{aslast}}$ and $\pi_{\mathbf{first}}$).

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 $\mathbf{i}x$) or nodes whose types are in $\pi_{\mathbf{seed}}$, or nodes whose types are in $\pi_{\mathbf{bef}}(a_i)$ and have been

1 $FirstMerge(F_i \mid F_u \mid x) =$ F_i if $roots(F_u)=\emptyset$
2 F_u if $roots(F_i)=\emptyset$
otherwise *(neither F_i nor F_u are empty)*
assuming $lab(roots(t_i)) \in \pi_{\mathbf{next}}(x) \cup \pi_{\mathbf{seed}}$
3 $TreeMerge(t_i \mid t_u) \circ FirstMerge(f_i \mid f_u \mid x)$ if $roots(t_i)=roots(t_u)$
4 $t_u \circ FirstMerge(F_i \mid f_u \mid x)$ if $new(roots(t_u))=true$
otherwise
5 $t_i \circ FirstMerge(f_i \mid F_u \mid x)$ if $first(roots(t_i))=false$
6 $t_u \circ FirstMerge(F_i \mid f_u \mid x)$ if $first(roots(t_i))=true$ and $new(roots(t_i))=true$
7 $t_i \circ FirstMerge(f_i \mid f_u \mid x)$ if $first(roots(t_i))=true$ and $roots(t_i)=roots(t_u)$

Figure 5.40: The procedure *FirstMerge* - general case -

projected for the purpose of potential insertions "before" or the first child node of n because a_i belongs to $\pi_{\mathbf{first}}$. 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 *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 prefered 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 $\pi_{\mathbf{olb}}$ 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 $\pi_{\mathbf{olb}}$ 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

[1] Dom. http://www.w3.org/DOM/. 34

[2] eXist. http://exist.sourceforge.net/. 4, 22, 88

[3] Galax. http://www.galaxquery.org. 4, 22

[4] QizX Free-Engine-3.0. http://www.xmlmind.com/qizx/free_engine.html. 4, 22, 88

[5] QizX/open. http://www.xmlmind.com/qizx/qizxopen.shtml. 4, 22

[6] SAX. http://www.saxproject.org/. 34, 81

[7] Saxon-ee. http://www.saxonica.com/. 4, 22, 88

[8] W3C. http://www.w3.org/. 9

[9] XUF. http://www.w3.org/TR/xquery-update-10/. 18

[10] Edbt 2011, 14th international conference on extending database technology, uppsala, sweden, march 21-24, 2011, proceedings. In A. Ailamaki, S. Amer-Yahia, J. M. Patel, T. Risch, P. Senellart, and J. Stoyanovich, editors, *EDBT*. ACM, 2011. 169, 170

[11] M. A. Baazizi, N. Bidoit, D. Colazzo, N. Malla, and M. Sahakyan. Projection for xml update optimization. In Ailamaki et al. [10], pages 307–318. 56, 167

[12] M. Benedikt, A. Bonifati, S. Flesca, and A. Vyas. Verification of tree updates for optimization. In *CAV*, 2005. 6

[13] M. Benedikt and J. Cheney. Schema-based independence analysis for XML updates. *VLDB*, 2009. 6

[14] M. Benedikt and J. Cheney. Semantics, types and effects for XML updates. In *DBPL*. Springer, 2009. 10, 14

[15] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen. Type-based XML projection. In *VLDB*, 2006. 3, 4, 6

[16] N. Bidoit, D. Colazzo, N. Malla, and M. Sahakyan. Projection based optimization for xml updates. *ADBIS - Local proceeding*, -:315–322, September 2009. 56

[17] M. N. Bidoit Tollu N., Colazzo D. and S. M. Optimisation de mises a jour xml par typage et projection. In *25emes journees Bases de Donnees Avancees (BDA)*, Octobre 2009. 56

[18] T. Bohme and E. Rahm. Supporting efficient streaming and insertion of xml data in rdbms. In *PROC. 3RD INT. WORKSHOP DATA INTEGRATION OVER THE WEB (DIWEB), 2004*, pages 70–81, 2004. 42

[19] P. Boncz, T. Grust, M. Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *SIGMOD*, pages 479–490, 2006. 22, 23

[20] P. Boncz, S. Manegold, and J. Rittinger. Updating the Pre/Post Plane in MonetDB/XQuery, 2005. 22, 23, 26

[21] S. Bressan, B. Catania, Z. Lacroix, Y.-G. Li, and A. Maddalena. Accelerating queries by pruning XML documents. *Data Knowl. Eng.*, 54(2), 2005. 3

[22] F. Cavalieri, G. Guerrini, and M. Mesiti. Dynamic reasoning on xml updates. In Ailamaki et al. [10], pages 165–176. 6

[23] A. Deutsch and V. Tannen. MARS: A System for Publishing XML from Mixed and Redundant) Storage. In *In VLDB*, pages 201–212, 2003. 23

[24] W. Fan, G. Cong, and P. Bohannon. Querying XML with update syntax. In *SIGMOD Conference*, 2007. 6

[25] L. Fegaras. A schema-based translation of XQuery updates. In *XSym*, 2010. 6

[26] W. Gelade, W. Martens, and F. Neven. Optimizing schema languages for XML: Numerical constraints and interleaving. In *ICDT*, 2007. 13

[27] G. Ghelli, K. H. Rose, and J. Siméon. Commutativity analysis in XML update languages. In *ICDT*, 2007. 6

[28] G. Ghelli, K. H. Rose, and J. Siméon. Commutativity analysis for XML updates. *ACM Trans. Database Syst.*, 33(4), 2008. 6

[29] C. Grun, S. Gath, A. Holupirek, and M. H. Scholl. XQuery Full Text Implementation in BaseX. 4, 22, 34

[30] C. Grun, A. Holupirek, M. Kramis, M. H. Scholl, and M. Waldvogel. Pushing XPath Accelerator to its Limits, 2006. 4, 22, 34

[31] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *VLDB*, pages 252–263, 2004. 23

[32] T. Grust and J. Teubner. Relational algebra: Mother tongue-XQuery: Fluent. 25

[33] A. Marian and J. Siméon. Projecting XML documents. In *VLDB*, 2003. 3, 6

[34] W. Meier. eXist: An Open Source Native XML Database. In *Web-Services, and Database Systems, NODe 2002 Web and Database-Related Workshops*, pages 169–183. Springer, 2002. 22

[35] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, 2002. 87

[36] M. Schmidt, S. Scherzinger, and C. Koch. Combined static and dynamic analysis for effective buffer minimization in streaming XQuery evaluation. In *ICDE*, 2007. 3

[37] J. T. Teubner. Pathfinder: XQuery compilation techniques for relational database targets, 2006. 23