# Partitioning XML data, towards distributed and parallel management

Noor Malla


## To cite this version:

Noor Malla. Partitioning XML data, towards distributed and parallel management. Other [cs.OH]. Université Paris Sud - Paris XI, 2012. English. NNT : 2012PA112154 . tel-00759173v2

## HAL Id: tel-00759173
## https://theses.hal.science/tel-00759173v2

Submitted on 31 Oct 2012 (v2), last revised 8 Dec 2015 (v3)

**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 OF PARIS-SUD XI - ORSAY

ECOLE DOCTORALE OF INFORMATIQUE

# PHD THESIS

to obtain the title of

## Dr. of Science

## (Informatics)

# Partitioning XML data, towards distributed and parallel management

Defended on September 21, 2012 by

**Noor MALLA**

**Thesis Advisors :**

Prof. Nicole BIDOIT-TOLLU
University Paris Sud XI - LRI

Assistant Prof. Dario COLAZZO
University Paris Sud XI - LRI

**Jury :**

| | | | |
|---|---|---|---|
| *President:* | Prof. Chantal REYNAUD | - | University Paris Sud XI |
| *Reviewers:* | Prof. Amann BERND | - | University Paris 6 - LIP6 |
| | Prof. Giovanna GUERRINI | - | University of Genova (Italy) |
| *Member:* | Associate Prof. Benjamin NGUYEN | - | University Versailles - PRiSM |

## Acknowledgments

I would like to express my gratitude to all those who gave me the possibility to complete this Thesis.

My foremost thank goes to Dr. Dario Colazzo, my thesis adviser. He has offered invaluable suggestions and advice to shape my research skills, which will help me greatly in the rest of my career. His understanding, encouraging and personal guidance have provided a good basis for this Thesis.

Secondly, I owe my most sincere gratitude to Professor Nicole Bidoit, my supervisor, who gave me her time and helped me in various aspects of my graduate study.

Also, I would like to thank Dr. Carlo Sartiani for spending time to serve my committee, and providing valuable feedback to help me improve the dissertation in many ways.

I would also like to thank my colleagues at LRI for their supports. They are Marina, Amine, Federico and Putchi. They are help me in various aspects of my graduate study.

Finally and most importantly, I thank my parents in Syria and all my family for their love and many years of support along the path of my academic pursuits.

Last but not least, I would like to express my deepest gratitude to my best friends, Yasmin, Yasser and Assem, for their unlimited support, love and encouragement to me.

# Contents


**1 Résumé en Français** **3**
- 1.1 introduction générale . . . 3
- 1.2 contributions . . . 6
- 1.3 l'organisation du manuscrit . . . 8

**2 Introduction** **13**
- 2.1 contributions . . . 15
- 2.2 structure of the thesis . . . 17

**3 Preliminaries** **19**
- 3.1 XML . . . 19
  - 3.1.1 Textual Representation . . . 20
  - 3.1.2 Well-Formedness of XML . . . 20
- 3.2 Querying XML . . . 22
  - 3.2.1 XPath Language . . . 22
  - 3.2.2 XQuery Language . . . 26
  - 3.2.3 XQuery Update Facility . . . 27
- 3.3 conclusion . . . 30

**4 XML Projection and its Limitations** **31**
- 4.1 path-based projection for queries . . . 32
  - 4.1.1 Limitations of Standard Projection for Queries . . . 35
- 4.2 type-based projection for updates . . . 39
  - 4.2.1 Limitations of Update Type-based Projection . . . 45
- 4.3 conclusion . . . 51

**5 Partitioning and Projecting XML Documents** **53**
- 5.1 preliminaries . . . 54
  - 5.1.1 Data Model . . . 54
  - 5.1.2 Query Language . . . 56
- 5.2 path extraction . . . 58
- 5.3 iterative queries and partitioning paths . . . 61
- 5.4 projection . . . 63
- 5.5 The partitioning algorithm . . . 70
  - 5.5.1 The Algorithm . . . 72
  - 5.5.2 Dealing with a Workload . . . 75
- 5.6 streaming implementation . . . 78
- 5.7 experimental evaluation . . . 88
  - 5.7.1 Experimental Setup . . . 89
  - 5.7.2 Tests Results . . . 89

5.7.3 Experiments . . . 90
5.7.4 Experiments on Queries $\{N_1,\ N_2,\ N_3\}$, and $\{D_1,\ D_2\}$ . . . 94
5.7.5 Summing Up . . . 98
5.8 CONCLUSION . . . 99

**6 Partitioning for XQuery Updates 101**
6.1 OVERVIEW . . . 102
6.2 PRELIMINARIES . . . 103
6.2.1 Simple XQuery Update Facilities (SXUF) . . . 103
6.3 ITERATIVE UPDATES . . . 105
6.4 PARTITIONING FOR ITERATIVE UPDATES . . . 115
6.4.1 Partitioning Algorithm . . . 116
6.4.2 Fusion Operation . . . 119
6.5 STREAMING IMPLEMENTATION . . . 121
6.5.1 Partitioning . . . 121
6.5.2 Fusion . . . 132
6.6 EXPERIMENTAL EVALUATION . . . 135
6.6.1 Experimental Setup . . . 136
6.6.2 Tests Results . . . 136
6.6.3 Experiments . . . 138
6.6.4 Summing Up . . . 141
6.7 CONCLUSION . . . 143

**7 Parallel Query and Update Evaluation 145**
7.1 MAPREDUCE . . . 145
7.1.1 Logical View . . . 147
7.1.2 Execution Overview . . . 148
7.2 PARALLEL EVALUATION OF ITERATIVE QUERIES AND UPDATES VIA MAPREDUCE . . . 150
7.3 CONCLUSIVE REMARKS . . . 152

**8 Related Works and Conclusion 153**
8.1 RELATED WORKS . . . 153
8.2 CONCLUSIVE REMARKS AND FUTURE DIRECTIONS . . . 154

**A XQuery Expressions and XQuery Updates 157**
A.1 XMark Queries proposed in [SWK+02a] . . . 157
A.2 Update Expressions used in [BBC+11] . . . 161
A.3 XQuery Update expressions in [Sah11] . . . 162
A.4 XQuery Update Facilities 1.0 Use Cases . . . 165

**Bibliography 169**

# List of Figures

3.1 Textual representation of an XML fragment. . . . . . . . . . . . . . . . 20
3.2 DTD of *addressBook.xml* XML document. . . . . . . . . . . . . . . . . 22
3.3 A well-formed XML document. . . . . . . . . . . . . . . . . . . . . . . . 23
3.4 Tree representation of addressBook XML document. . . . . . . . . . 24
3.5 Navigational XPath axes. . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.6 The W3C syntax of simple XQuery updates. . . . . . . . . . . . . . . . 28
3.7 Simple XQuery updates. . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.8 Complex XQuery updates. . . . . . . . . . . . . . . . . . . . . . . . . . 30

4.1 A fragment of the input XMark document $D$. . . . . . . . . . . . . . . 33
4.2 An XML document fragment. . . . . . . . . . . . . . . . . . . . . . . . 34
4.3 Loading algorithm of [MS03] for building a projection. . . . . . . . . 35
4.4 A simple example with type-based projection. . . . . . . . . . . . . . 42
4.5 DTD of the XML document $t$ illustrated in Figure 4.4. . . . . . . . . 43
4.6 Dealing with insertion. . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.7 Dealing with string and mixed-contents. . . . . . . . . . . . . . . . . 44

5.1 Projecting-partitioning scenario for an input document $D$ and a given query $Q$ and partitioning path $PP$. . . . . . . . . . . . . . . . . . . . 54
5.2 Representation of XML trees as stores and projection. . . . . . . . . 56
5.3 Two possible partitions of the XML tree $t$ of Figure 5.2. . . . . . . . 56
5.4 Query language grammar. . . . . . . . . . . . . . . . . . . . . . . . . . 57
5.5 Path extraction function. . . . . . . . . . . . . . . . . . . . . . . . . . 58
5.6 Scenario of finding candidate paths of Example 2. . . . . . . . . . . . 62
5.7 Partitioning paths of some iterative XMark queries. . . . . . . . . . . 64
5.8 Path $/child :: a/dos :: c$ transformations. . . . . . . . . . . . . . . . . 65
5.9 Partition plus projection. . . . . . . . . . . . . . . . . . . . . . . . . . 66
5.10 An input XML tree $t$. . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
5.11 Standard projections $t^{'}_{Q_1}$, $t^{'}_{Q_2}$ XML trees created from the input $t$. . 76
5.12 Partitioning scenario on $t$ for a given iterative query $Q_1$. . . . . . . . 76
5.13 Global projection $t'$ for the workload $(Q_1, Q_2)$. . . . . . . . . . . . . 77
5.14 Partitioning scenario on the global projection $t'$ of workload $(Q_1, Q_2)$. 77
5.15 An input document $t$ and its projected parts $t'_1, t'_2$. . . . . . . . . . . 84
5.16 Projection-partitioning processing: the current open-tag is `<doc>`. . . 84
5.17 Projection-partitioning processing: the current open-tag is `<a>`. . . . 84
5.18 Projection-partitioning processing: the current open-tag is `<b>`. . . . 85
5.19 Projection-partitioning processing: the current open-tag is `<c>`. . . . 85
5.20 Projection-partitioning processing: the current close-tag is `</c>`. . . 86
5.21 Projection-partitioning processing: the current close-tag is `</b>`. . . 86
5.22 Projection-partitioning processing: the current close-tag is `</a>`. . . 87

5.23 Projection-partitioning processing for parsing the subtree <a><f><c>. 87
5.24 Projection-partitioning processing for parsing the following close-tags </c></f></a>. 88
5.25 Parsing the subtree <a><b><c>to</c></b></a>, and create a new projected part $t'_2$. 88
5.26 Final projected parts $t'_1, t'_2$ produced by projection+partitioning algorithm. 89
5.27 Projection vs partitioning+projection - with input document 1GB - using Saxon. 91
5.28 Projection vs partitioning+projection - with input document 5GB - using Saxon. 92
5.29 Scalability of the partitioning approach - using Saxon. 92
5.30 Scalability of the partitioning approach: workload - using Saxon. 93
5.31 Projection vs partitioning - with input document 2GB - using Qizx. 93
5.32 Scalability of the partitioning approach: workload - using Qizx. 94
5.33 Projection vs partitioning - with input document 9GB - using Qizx. 95
5.34 Scalability of the partitioning approach - using Saxon. 95
5.35 Projection vs partitioning - with input document 1GB - using Qizx. 96
5.36 Projection vs partitioning - with input document 2GB - using Qizx. 96
5.37 Scalability of the partitioning approach - using Qizx. 97
5.38 Relation between *maxSize* (`pSize`) and the performance of the partitioning approach. 98
5.39 Scalability of the partitioning approach - using Qizx. 100

6.1 Partitioning update scenario. 102
6.2 Syntax of SXUF. 104
6.3 An XML document $t$ and a possible partition. 106
6.4 Equivalence between $U_8(t)$ and $U_8(t_1) \oplus U_8(t_2)$. 107
6.5 Another possible parts $t'_1, t'_2$ of the XML document $t$. 107
6.6 Equivalence between $U_8(t)$ and $U_8(t'_1) \oplus U_8(t'_2)$. 107
6.7 Equivalence between $U_9(t)$ and $U_9(t_1) \oplus U_9(t_2)$. 108
6.8 An XML document $D$ and two different kinds of partition. 108
6.9 Equivalence between $U_{10}(D)$ and $U_{10}(D_1) \oplus U_{10}(D_2)$. 109
6.10 Non-equivalence between $U_{10}(D)$ and $U_{10}(D'_1) \oplus U_{10}(D'_2)$. 109
6.11 Non-equivalent case between $U_{11}(t)$ and $U_{11}(t_1) \oplus U_{11}(t_2)$. 109
6.12 Non-equivalent case between $U_{11}(t)$ and $U_{11}(t'_1) \oplus U_{11}(t'_2)$. 110
6.13 Non-equivalent case between $U_{12}(t)$ and $U_{12}(t_1) \oplus U_{12}(t_2)$. 110
6.14 Path extraction function for updates. 112
6.15 An XML document $t$ and its parts $t_1, t_2, t_3$. 116
6.16 Partitioning update scenario on the input document $t$ and its parts, for a given iterative update $U$. 119
6.17 Fusion scenario on distinct (updated and non-updated) parts. 120
6.18 An input document $t$ and its created parts $t_1, t_2, t_3$. 127
6.19 Partitioning scenario: the current open-tag is <a>. 127

6.20 Partitioning scenario: parsing the open-tags `<d><c>`. . . . . . . . . . . 128
6.21 Partitioning scenario: parsing the close-tags `</c></d>`. . . . . . . . . . 128
6.22 Partitioning scenario: parsing the subtree `<b><f><c>go`. . . . . . . . 129
6.23 Partitioning scenario: parsing close-tags `</c></f>`, and create a new part $t_2$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 130
6.24 Partitioning scenario: parsing open-tags `<f><g>to` . . . . . . . . . . . 131
6.25 Parsing close-tags `</g></f></b>`, and create a new part $t_3$. . . . . . 131
6.26 Final parts $t_1, t_2, t_3$ produced by the partitioning technique. . . . . . 132
6.27 Updated parts $U(t_1)$, $U(t_2)$, non-updated $t_3$, and the fusion final result. 135
6.28 Projection vs partitioning - with input document 1GB - using Saxon. 138
6.29 Projection vs partitioning - with input document 5GB - using Saxon. 139
6.30 Projection vs partitioning - with input document 1GB - using Qizx. . 139
6.31 Projection vs partitioning - with input document 5GB - using Qizx. . 140
6.32 Projection vs partitioning - with input document 10GB - using Qizx. 140
6.33 Projection vs partitioning - with input document 15GB - using Qizx. 141
6.34 Scalability of the partitioning+update+fusion approach - using Saxon. 142
6.35 Scalability of the partitioning+update+fusion approach - using Qizx. 144

7.1 Execution overview. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149
7.2 Graphical representation of the Master-Map schema. . . . . . . . . . 151

# List of Algorithms


1 *Projection* . . . 69
2 *Partition* . . . 73
3 Projection/Partition-Init-DataStructure . . . 79
4 SAX-startDocument . . . 79
5 SAX-characters . . . 80
6 SAX-startElement . . . 81
7 SAX-endElement . . . 82

8 *Parse* . . . 117
9 *Partition* . . . 118
10 Partition-Init-DataStructures . . . 122
11 SAX-startDocument . . . 122
12 SAX-characters . . . 122
13 SAX-endDocument . . . 122
14 SAX-startElement . . . 123
15 SAX-endElement . . . 124
16 Fusion-mainProgram . . . 133
17 Fusion-startElement . . . 133
18 Fusion-endElement . . . 134

# List of Tables

4.1 Size of projected documents. . . . . 36
4.2 Qizx and Saxon performances on projected DBLP document. . . . . 36
4.3 Saxon performance on projected documents. . . . . 38
4.4 Qizx performance on projected documents. . . . . 39
4.5 The composition of 3-level type projector for 20 updates used in [Sah11]. 46
4.6 Size reduction by type projection. . . . . 49
4.7 Qizx and Saxon performances for type-based projected documents. . 50

5.1 The function $\uplus$. . . . . 67
5.2 Rewriting functions $Down(\tau)$ and $Res(\alpha; \tau)$. . . . . 68
5.3 Global projections size. . . . . 91
5.4 Qizx and Saxon performances with the partitioning approach - on DBLP database. . . . . 97

## Abstract

With the widespread diffusion of XML as a format for representing data generated and exchanged over the Web, main query and update engines have been designed and implemented in the last decade. A kind of engines that are playing a crucial role in many applications are *main-memory* systems, which distinguish for the fact that they are easy to manage and to integrate in a programming environment. On the other hand, main-memory systems have scalability issues, as they load the entire document in main-memory before processing.

This Thesis presents an XML partitioning technique that allows main-memory engines to process a class of XQuery expressions (queries and updates), that we dub *iterative*, on arbitrarily large input documents. We provide a static analysis technique to recognize these expressions. The static analysis is based on paths extracted from the expression and does not need additional schema information. We provide algorithms using path information for partitioning the input documents, so that the query or update can be separately evaluated on each part in order to compute the final result. These algorithms admit a streaming implementation, whose effectiveness is experimentally validated.

Besides enabling scalability, our approach is also characterized by the fact that it is easily implementable into a MapReduce framework, thus enabling parallel query/update evaluation on the partitioned data.



**Keywords :** XML, XQuery, XQuery updates, Projection, Data Partitioning, MapReduce.

CHAPTER 1

# Résumé en Français

**Contents**


1.1 INTRODUCTION GÉNÉRALE . . . . . . . . . . . . . . . . . . . . 3
1.2 CONTRIBUTIONS . . . . . . . . . . . . . . . . . . . . . . . . . 6
1.3 L'ORGANISATION DU MANUSCRIT . . . . . . . . . . . . . . . . 8


## 1.1 INTRODUCTION GÉNÉRALE

LA dernière décennie a vu la diffusion rapide des données semi-structurées et en particulier le standard XML (eXtensible Markup Language) dans nombreux applications qui s'appuient sur le web pour l'échange et le partage de données. XML est un successeur de SGML, il a été rapidement adopté comme format naturel pour représenter les données semi-structurées pour lesquelles le modèle relationnel et le modèle objet ne sont pas appropriés. La grande flexibilité des données XML a rendu ce format universel et a permis son utilisation pour échanger des données entre des applications différentes sur le Web.

Afin de permettre la diffusion de XML, plusieurs outils ont été défini pour la transformation, l'interrogation, la manipulation et la modélisation des données XML. En particulier, le World Wide Web Consortium (W3C) a introduit XQuery [W3S10] comme langage de requête et XQuery Update [Gro11a, Gro11b] pour mettre à jour des documents XML. Les deux langues ont été intensivement étudiées par la communauté scientifique, en particulier dans un but d'optimisation de l'exécution des requêtes et des mises à jour.

Une principale utilisation de XQuery est l'interrogation et la mise à jour des données XML qui sont simplement stockées dans des fichiers ou générées en streaming. En général, dans ces contextes, toutes ces fonctionnalités complexes qui caractérisent les DBMS traditionnels ne sont pas nécessaires. Le besoin principal dans ces contextes est la disponibilité d'un moteur de requête et mise à jour facile à installer et à intégrer dans un environnement de programmation. Pour cette motivation, de nombreux moteurs XQuery ont été mis au point pendant les dernières années, comme Galax [gal], Saxon [sax], Qizx [qiz] et eXist [exi]. Ces systèmes sont généralement conformes par rapport aux spécifications du W3C. Ils traitent les données en mémoire centrale: les données sont d'abord entièrement chargé dans la mémoire

centrale, puis traitées (interrogés ou mises à jour). Pour cette raison, ces systèmes sont généralement classés comme de systèmes mémoire-centrale.

En citant Cong et al. [GCL12], les systèmes mémoire-centrale sont le meilleur choix dans

> ... plusieurs domaines comme les sciences de la vie (par exemple, Biologie), l'astronomie, et même pour la gestion des documents XML typiques correspondant aux fichiers Microsoft Office (étant donné que les présentations PowerPoint, les fichiers Word et Excel sont actuellement stockées au format XML). Dans tous ces domaines, la gestion des documents XML est centrée sur des fichiers et aucun système de gestion des données XML traditionnels n'est mis en place.

En particulier dans les domaines tels que les sciences de la vie et de l'astronomie, les documents XML ont une taille importante (plusieurs GBs), ce qui peut compromettre la possibilité d'utiliser un moteur de mémoire-centrale pour le traitement des requêtes.

Actuellement, les systèmes mémoire-centrale qui sont très flexibles et faciles à installer et à utiliser, ne peuvent pas passer à l'échelle.

Une solution partielle pour ce problème est proposée. Cette solution est basée sur la projection. La projection XML est une technique d'optimisation proposée dans le but de surmonter les limitations des moteurs mémoire-centrale pour l'interrogation des documents XML. Cette technique repose sur une observation simple selon laquelle les requêtes sont en général sélectives cad qu'elles ciblent seulement une sous-partie des documents interrogés. L'idée consiste alors à identifier de manière statique les parties nécessaires à l'évaluation des requêtes et à utiliser cette information pour ne charger en mémoire centrale que les parties du document qui sont accédées par la requête. La projection permet ainsi de traiter des documents volumineux même sous des contraintes de mémoire importantes.

La projection a été utilisée pour la première fois dans [MS03] puis étendue dans [BCCN06, KSS08] en prenant en compte le schéma du document interrogé. L'utilisation des schémas permet de réduire la taille de la projection en exploitant la possibilité d'inférer de manière précise les données nécessaires à l'évaluation d'une requête. Dans les techniques de [BCCN06, KSS08], l'information inférée consiste en l'ensemble des étiquettes des noeuds nécessaires à l'évaluation des requêtes. Cet ensemble est appelé *type-projecteur*.

Les approches précédentes et basées sur la projection ne fournissent qu'une solution partielle aux problèmes de scalabilité des systèmes mémoire-centrale, et les documents d'entrées projetées pourraient encore dépasser la capacité de la mémoire centrale. Cela peut être le cas lorsque (i) le fichier d'entrée est énorme, (ii) la sélectivité de la requête est faible (elle a besoin d'une grande partie du document d'entrée), ou (iii) cas d'évaluation d'un workload (par exemple, un ensemble de requêtes qui doivent être évaluée sur le document d'entrée). Dans ce dernier cas, la

taille de la projection globale peut dépasser la taille de la mémoire centrale. La projection globale peut être inutile puisque tout le document en entrée peut être nécessaire pour le workload.

Il est important de dire que les problèmes de scalabilité dépendent également du type particulier de moteur qu'on veut utiliser, et sur les paramètres de la mémoire interne. En fait, la plupart des systèmes mémoire-centrale sont implémentés en Java, et leur scalabilité dépend de la quantité de mémoire centrale précisée en paramètre de la JVM (Java Virtual Machine). Dans tous les cas, même pour les grandes quantités, les problèmes de scalabilité de la projection standard sont toujours optimisés, la taille de la projection de documents augmente lorsque la taille du document en entrée augmente.

L'objectif principal de cette Thèse est de proposer une technique qui assure la scalabilité pour les requêtes et les mise à jours indépendamment:

- du type du système mémoire-principal.
- de la quantité de mémoire centrale qui est valable.
- de l'utilisation du schéma d'informations de schéma.

À cette fin, dans cette Thèse, nous proposons une technique d'optimisation basée sur le partitionnement des données XML. Cette technique repose sur l'observation que, dans plusieurs cas pratiques, les requêtes XQuery et les mises à jour sélectionnent d'abord une séquence de sous-arbres à l'aide d'une sous-requête (par exemple, une expression XPath), puis évaluent des opérations sur cette séquence des sous-arbres. Par exemple, en ce qui concerne les requêtes, 13 des 20 requêtes de XMark Benchmark [SWK+02b] vérifient cette propriété et pour les mises à jour, 16 des 20 mises à jour qui ont été proposées dans [BBC+11, Sah11] sont itératives.

Dans le cas de requêtes, lorsque cette propriété est satisfaite par une requête $Q$, le document d'entrée peut être divisé en un ensemble de parties $\{D_1, \ldots, D_\kappa\}$, de sorte que l'évaluation $Q(D)$ de la requête $Q$ sur le document d'entrée $D$ est égale à la concaténation des évaluations $Q(D_i)$ de la requête $Q$ sur les parties $D_i$ du document d'entrée $D$.

Dans le cas des mises à jour, la même stratégie peut être adoptée, à la différence que les mises à jour partielles $U(D_i)$ doivent être recombinées pour obtenir le document mis à jour $U(D)$. Alors que dans le cas de requêtes, une simple concaténation des résultats partiels est suffisant. En particulier, nous utilisons la commande `cat` pour fusionner ces résultats partiels afin de produire le résultat final. Pour les mises à jour, et puisque nous utilisons des informations supplémentaires lors de la création des partitions afin de s'assurer que les parties créées sont bien formées, des informations supplémentaires par rapport des balises supplémentaires sont nécessaires afin de correctement re-combiner des parties mises à jour et éliminer ces balises pour obtenir le résultat final $U(D)$. Ces informations auxiliaires sont opportunément mises en place pendant le partitionnement.

Avec la scalabilité, notre technique de partitionnement peut être facilement adaptée dans un environnent MapReduce [DG08], ce qui permet l'interrogation et la mise à jour parallèle des parties. Cette évaluation parallèle est possible puisque dans le cas des requêtes et des mises à jour itératives, l'évaluation de chaque partie peut se faire indépendamment de l'évaluation des autres parties. Par conséquent, cette approche peut aisément transposée dans un environnement MapReduce qui joue un rôle très important dans les plates-formes basée sur le cloud.

## 1.2 CONTRIBUTIONS

Cette Thèse propose une nouvelle technique de partitionnement basé sur l'évaluation de requêtes XQuery et les mises à jour.

La première contribution de cette Thèse se concerne les requêtes. Dans ce contexte, les contributions principales sont les suivantes et sont également présentés dans [Nic12]:

- Nous présentons d'abord une caractérisation formelle de la classe de requêtes qui satisfont la propriété de division décrite ci-dessus: nous appelons ces requêtes *requêtes itératives*. En s'appuyant sur cette caractérisation formelle, nous développons une technique d'analyse statique qui extrait des chemins et des informations sur les variables liées à la requête, et puis les analyse afin de détecter statiquement comment le document d'entrée est navigué par la requête. En se fondant sur les informations de chemin nous pouvons éviter l'utilisation d'informations de schéma qui n'est pas toujours disponible.

- Nous présentons ensuite un algorithme de partitionnement qui exploite les chemins extraites lors de l'analyse statique pour identifier la partition correcte pour le document d'entrée. Nous présentons d'abord une spécification d'algorithme basée sur la représentation DOM puis nous utilisons le parseur SAX qui permet la possibilité d'effectuer le partitionnement en streaming, en utilisant peu de mémoire. Pour améliorer encore les avantages de notre approche, nous combinons le partitionnement avec la projection standard, de sorte que lors de la création de parties de document, les sous-arbres qui ne sont pas nécessaires par la requête sont éliminées. L'utilisation de la projection standard n'est pas cruciale pour assurer la scalabilité, ce qui est notre objectif principal puisque dans notre approche, la taille maximale de chaque partie peut réglée par l'utilisateur. La projection contribue à réduire le coût du partitionnement, car elle accélère l'exécution des requêtes sur la partition.

- Ensuite, nous présentons une évaluation expérimentale intensive qui confirme que, lors que de l'utilisation de notre approche de partitionnement, des moteurs mémoire centrale peuvent traiter des documents de taille arbitraire, au prix d'un coût d'exécution légèrement supérieur à celui des approches de projection qui n'utilisent pas de schéma. Nos expériences montrent également que le

partitionnement permet la scalabilité pour les workloads, car dans ce cas le document en entrée est divisé une fois pour toutes les requêtes (ou les mises a jour) du workload.

La deuxième contribution de cette Thèse se concerne les mises à jour. Dans ce contexte, les contributions principales sont les suivantes:

- Nous analysons d'abord les cas où l'évaluation des mises à jour peut être correctement appliquée sur les partitions, puis nous fournissons une analyse statique pour caractériser ces mises à jour, que nous appelons *mises à jour itératives*. Cette caractérisation exige des restrictions sur les mécanismes d'interrogation qui sont utilisés dans les expressions *source* et *target* des mises à jour. Nous allons montrer que ces restrictions sont acceptables puisque une large classe de mises à jour peut être traitée avec notre approche.

- Et puis, nous présentons une technique de partitionnement qui se distingue de la technique des requêtes par les aspects suivants:

  Premier aspect: la projection n'est pas utilisée, afin d'avoir une recombinaison simple et efficace des mises à jour partielles. Ceci est également justifié par le fait que le partitionnement est déjà suffisant pour générer suffisamment de petites pièces (parties du document d'entrée). L'utilisation de la projection exige un processus sophistiqué de la recombinaison (puisque les sous-arbres élagués au cours de partitionnement doivent être reconnus) et de remettre dans le résultat final du processus. Ce type d'opération a été fait par [BBC+11], où l'utilisation des informations de schéma a été cruciale pour assurer une formalisation claire et efficace.

  Deuxième aspect: les chemins utilisés au cours de partitionnement sont déduite en le mettant en compte la nature particulière de mises à jour. Ces chemins sont utilisés pour assurer que les sous-arbres qui éventuellement été sélectionnées par les chemins *Target* ne sont jamais divisés pendant le partitionnement. L'atomicité de ces sous-arbres est nécessaire pour assurer que l'évaluation de la mise à jour peut être correctement répartir sur toutes les parties d'entrée.

- Ensuite, nous présentons les résultats des tests étendus montrant l'efficacité de notre technique. A la différence du cas des requêtes, la sur-coût du au partitionnement n'est pas négligeable. Toutefois, les résultats de ces tests montrent que notre objectif principal, la scalabilité est largement réalisée.

Concernant les résultats des tests, nous avons utilisé deux moteurs mémoire-centrale principaux, Saxon [sax] et Qizx [qiz]. Notre choix est motivé par le fait que Saxon est un système très populaire, qui se distingue pour son exhaustivité dans la couverture de la plupart des normes du W3C pour le traitement XML (par exemple, le schéma XML, XSLT, XQuery et les mises à jour). Différemment, Qizx

est spécialisée dans la requête XQuery et la mise à jour, et soutient des techniques sophistiquées pour optimiser le temps d'exécution et la consommation de mémoire.

La troisième contribution de cette Thèse montre est le fait que la technique proposée est été facilement adapté pour être exécuté dans un cadre MapReduce [DG08]. À cette fin, les notions principales de ce paradigme sont introduites puis l'architecture de la mise en oeuvre de notre technique sur MapReduce est été illustrée et discutée.

## 1.3 L'organisation du manuscrit

Ce manuscrit est composé de huit chapitres dont un chapitre de résumé en français, et un autre chapitre introduction.

Les six autres chapitres sont organisés comme suit:

- **Chapitre 3** Le chapitre préliminaire est consacré à la présentation des notations et des langages (XPath et XQuery [Gro03, W3S10]) de requêtes et de mises à jour (XQuery update Facility [Gro11a]) utilisés tout au long de ce manuscrit.

- **Chapitre 4** Dans ce chapitre, nous examinons les principales caractéristiques des deux approches principales proposées pour la projection XML. La première approche [MS03] concerne les requêtes, et est basé sur l'extraction des chemins de la requête et l'utilisation de ces chemins pour projeter le document en entrée. La deuxième approche pour les requêtes a été proposé dans [BCCN06], et exige des informations sur le schéma des données. Nous ne parlerons pas par rapport a cette approche car cette thèse n'utilise pas le schéma des données, et, pour le fragment XQuery que nous considérons, les performances de [BCCN06] sont très proches à celle proposée dans [MS03] en termes de la réduction de la taille des documents.

  La deuxième technique que nous allons discuter concernant des mises à jour [BBC+11, BCMS09a, BCMS09b],qui et est la seule technique de projection existant pour les mises à jour. Elle est basé sur les informations de schéma et sur l'inférence des types, plus une opération *Merge* qui, comme nous le verrons, est nécessaire pour recombiner la mise à jour de la projection avec le document original.

  Dans ce chapitre, en plus d'illustrer comment la projection peut être utilisée pour traiter une large classe de requêtes et mises à jour XML pour des documents de grande taille, nous allons montrer que ces techniques, même si elles sont assez efficaces, ne passent pas à l'échelle. Ceci a motivé notre intérêt pour des technique de partitionnement.

Le chapitre est organisé comme suit. La section 4.1 introduit la projection standard XML qui est proposée par [MS03] avec quelques définitions principales, l'algorithme analyse du chemin qui extrait l'ensemble des chemins de la projection à partir d'une requête XQuery arbitraire. Ensuite, nous expliquons l'algorithme de chargement dans la mémoire utilisé pour créer la projection. La section 4.1.1 illustre les limitations de la technique de projection standard XML en testant plusieurs requêtes sur des documents XMark et de base de données DBLP. Dans la section 4.2, nous introduisons, à travers des exemples, le concept de la technique de projection basee sur le typage et proposé par [BBC+11]. Et puis, dans la section 4.2.1, nous illustrons les limitations de cette technique dans la utilisant des mises à jour. Enfin, nous concluons ce chapitre dans la section 4.3.

- **Chapitre 5** Dans ce chapitre, nous avons présenté une nouvelle technique de la projection de partitionnement de document d'entrée XML. Cette technique se généralise des approches existantes et basées sur le chemin, et s'applique à une large classe de requêtes.

  L'approche proposée analyse une requête d'entrée et, si la requête est *itérative*, l'approche va extraire tous les chemins pertinents et les utilise pour exécuter la projection et le partitionnement sur le document d'entrée, et puis obtenir des petites parties. Notre étude expérimentale assure que l'exécution de la requête d'entrée sur chaque partie indépendamment et en combinant les résultats partiels obtenus par ces parties, n'importe quel moteur mémoire-centrale existant peut traiter une requête itérative sur des très grand documents d'entrée.

  Ce chapitre contient trois parties principales. La première partie (les sections 5.1, 5.2, 5.3) présente notre technique d'analyse statique utilisée pour caractériser des *requêtes itératives*, pour lesquels les données XML peuvent être partitionnés pour l'évaluation de la requête. La deuxième partie (Section 5.5) présente notre algorithme de partitionnement. D'abord, une spécification précise est formalisée en s'appuyant sur une représentation basée sur DOM formalisation pour des arbres d'entrée. Et puis une version basée sur SAX est fournie. Comme indiqué dans l'introduction, pour accentuer les avantages de notre stratégie, la projection est utilisée pendant le partitionnement. La troisième partie (les sections 5.6, 5.7) explique la mise en oeuvre des algorithmes basés sur SAX parseur, et présente les résultats des tests obtenus à partir d'expériences que nous avons menées en utilisant deux moteurs principaux pour XQuery. Enfin, nous concluons ce chapitre dans la section 5.8.

- **Chapitre 6** Dans ce chapitre, nous présentons une technique de partitionnement pour les mises à jour XUF (XQuery Update Facility). Comme le cas des requêtes, le partitionnement permettant le traitement des grands documents, et qui ne pouvait pas être mise à jour en utilisant des moteurs mémoire-centrale existants comme [qiz, exi, bas], même en utilisant la technique de la

projection standard basée sur la technique proposée dans [BBC+11].

Dans ce chapitre, nous caractérisons une classe des mises à jour, appelées mises à jour *itératives*, pour lesquelles une évaluation basée sur le partitionnement est possible : tout d'abord, les documents sont partitionnés en plusieurs parties puis les parties sont mises à jour indépendamment, et enfin les parties mises à jour sont fusionnées en utilisant une opération de *fusion* afin d'obtenir le résultat final cad le document en entrée mis à jour.

Pour caractériser des mises à jour itératives, nous utilisons une analyse basée sur des chemins. Les chemins extraits seront également utilisés pour le partitionnement. A la différence des requêtes, le partitionnement ne s'appuiera pas sur la projection, les chemins sont utilisés pour s'assurer uniquement que chaque partie contient tout ce qui est nécessaire pour chaque opération de mise à jour. La projection n'est pas utilisée, afin d'éviter les opérations de fusion complexes sur des parties mises à jour, opération nécessaires pour récupérer les sous-arbres élagués lors de la construction du document global actualisé. L'efficacité de l'approche proposée est démontrée par des expériences approfondies comparant notre approche basée sur le partitionnement avec la projection proposé dans [BBC+11, MS03]. Il est important de dire que cette dernière approche basée sur le type des données est la seule approche de projection pour traiter les mises à jour XQuery.

Le chapitre est structuré comme suit. Dans la section 6.2, nous introduisons quelques notations préliminaires sur le langage des mises à jour utilisées dans cette approche, et puis nous présentons notre fonction d'extraction de chemins. Dans ls section 6.3, nous décrivons formellement les mises à jour itératives. Ensuite, dans la section 6.4, nous présentons notre technique de partitionnement pour les mises à jour itératives, et introduisons les définitions formelles et les spécifications basés sur DOM du partitionnement et de la fusion. Dans la section 6.5, nous fournissons les algorithmes (basés sur le streaming) de partitionnement et de fusion utilisés pour exécuter notre scénario de partitionnement pour les mises à jour. Le chapitre se termine avec les résultats des tests dans la section 6.6 et quelques conclusions présentées dans la section 6.7.

- **Chapitre 7** Avec la scalabilité, notre technique de partitionnement présentée dans les chapitres précédents possède un autre avantage celui de pouvoir exécuter les requêtes et les mises à jour en parallèle. Ceci est possible puisque une large classe des requêtes et des mises à jour sont itératives et permettent l'évaluation de celles ci sur chaque partie indépendamment de l'autre.

  Dans ce chapitre, nous présentons les idées essentielles d'une mise en oeuvre parallèle possible de notre technique de partitionnement à l'aide du modèle de programmation MapReduce [DG08]. Nous tenons à souligner que l'architecture que nous proposons est le résultat d'une collaboration avec Carlo

Sartiani (professeur adjoint à l'Università Basilicate della, Italie) et Maurizio Nole (étudiant du Master à l'Università Basilicate della, Italie).

Nous présentons d'abord les bases du paradigme MapReduce dans la section 7.1, puis nous montrons comment notre technique peut être mise en oeuvre dans une plate-forme de MapReduce dans la section 7.2. Enfin, nous tirons notre conclusion dans la section 7.3.

- **Chapitre 8** Conclusion et perspectives: Dans ce chapitre, nous avons présenté une nouvelle technique de partitionnement pour de document XML. Cette technique généralise les approches existantes et basées sur le chemin, et s'applique à une large classe de requêtes et mises à jour.

  Une des particularités de notre approche est qu'elle n'utilise pas le schéma. Il utilise les informations de chemin provenant de la requête / mise à jour afin d'effectuer l'analyse statique nécessaire pour reconnaître la nature itérative de la requête / mise à jour et utilise les informations de chemin pour effectuer le partitionnement. Une autre particularité de cette approche est qu'elle peut s'appuyer sur n'importe quel système mémoire-centrale, car aucune intervention dans le mécanisme interne du système n'est nécessaire. Enfin, nous avons vu que notre approche peut être mise en oeuvre dans une plate-forme parallèle comme MapReduce de manière aisée permettant ainsi à l'interrogation et la mise à jour en parallèle. Pour les ensembles de documents de taille importante, et pour de grands cluster de machines, cette utilisation permet de réduire considérablement le temps comparé à une exécution sequentielle des requêtes/mises à jour.

  Il existe plusieurs perspectives. Tout d'abord, nous prévoyons d'étendre cette approche aux autres fragments de XQuery en particulier à des requêtes contenant des opérateurs d'agrégation (telles que le group-by). En plus, nous prévoyons d'étendre cette technique dans le cas où les requêtes effectuent des jointures. Dans ce cas, des tests effectués ont révélé que le temps d'exécution peut être important en utilisant des systèmes mémoire-centrale. Pour permettre le partitionnement de la requête / mise à jour on doit redéfinir l'analyse statique pour tenir compte des conditions de jointure et probablement recourir à la réécriture des requêtes /mises à jour. À notre avis, dans ce scénario une approche MapReduce pourrait aider à réduire le temps d'exécution.

  Comme deuxième perspective, nous aimerions explorer les possibilités de manipulation des workloads constitués de requêtes et de mises à jour. Une fois l'analyse de chemin effectuée pour caractériser la nature itérative du workload, le partitionnement peut être effectué pour l'ensemble des requêtes et mises à jour composant ce workload.

  Enfin, nous prévoyons d'utiliser la plate-forme MapReduce pour la mise en oeuvre de notre approche, en utilisant le schéma illustré dans le chapitre 7.

En particulier, nous allons nous concentrer sur notre implémentation, pour adapter notre code dans la plate-forme MapReduce. Dans ce contexte, nous allons également nous concentrer sur les tests expérimentaux afin de définir pour quel type de requête / mise à jour l'exécution de MapReduce est plus rapide plus que l'exécution traditionnelle centralisée.

Chapter 2

# Introduction

## Contents

2.1 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.2 Structure of the Thesis . . . . . . . . . . . . . . . . . . . . . 17

The last decade has seen the rapid diffusion of the *eXtensible Markup Language* in many application fields. XML is a successor of SGML, and was rapidly adopted as a natural format for representing semi-structured data, whose structure can not be easily modeled according to standard relational and object-oriented data models. The great flexibility which is behind the XML data model made it a universal data representation format, and allowed the use of XML as a convenient medium for exchanging data between different Web applications.

To support the diffusion of XML, several tools for transforming, querying, manipulating, and modeling XML data have been defined. In particular, the *World Wide Web Consortium* (W3C) introduced XQuery [W3S10] as the standard query language for XML data, and, more recently, XQuery Update Facility [Gro11a, Gro11b] as an extension of XQuery to update XML documents. Since their introduction, both languages have been intensively studied by the research community, in particular in directions aiming at optimizing query and update execution.

One of the main use of XQuery, is to query and update XML data that are simply stored in files or generated by a stream. Generally, in these contexts all those complex functionalities characterizing traditional DBMSs are not needed. The main need in these context is the availability of a query/update engine which is easy to install and to integrate in a programming environment. With such motivation many light-weight XQuery processors have been devised in recent years, like Galax [gal], Saxon [sax], Qizx [qiz], and eXist [exi]. These systems usually provide full compliance with respect to the W3C specifications, and process data in main memory fashion: data are first entirely loaded in the main-memory and then processed (queried or updated). For this reason, these systems are usually classified as *main-memory* systems.

By quoting Cong and al. [GCL12], main-memory systems are the best choice in

> ... domains like Life sciences (e.g., Biology), Astronomy, and even for the management of typical XML documents corresponding to Microsoft

> Office files (since powerpoint presentations, Word files, and Excel spreadsheets are all currently stored as XML). In all these domains, the management of XML documents is file-system centric and no traditional XML data management systems is yet in place (since non-expert users often find these latter systems to be hard to use and maintain).

Especially in domains like Life science and Astronomy, XML documents are likely to be huge (several GBs), which can jeopardize the possibility of using a main-memory engine for query processing. In other words, main-memory systems, while very flexible and easy to set-up and use, cannot scale up with document size. A partial solution to this problem is offered by projection-based techniques [BCCN06, KSS08, MS03] that allow one to prune out, at loading time, parts of the data that are not necessary for the query or the workload being processed. For some of the existing projection techniques, schema information in the form of DTDs or XML Schema definition is needed [BCCN06, KSS08].

Projection-based approaches provide only a partial solution to the scalability issues of main-memory systems, as the projected input documents may still exceed the main-memory capacity. This may be the case when (i) the input file is huge, (ii) the query selectivity is low and it needs a large part of the input, or (iii) a workload (i.e., a set of queries) has to be evaluated on the document. In the last case, a single global projection meeting the query needs of the whole workload is likely to exceed the main-memory size, while running a query at a time, and projecting (and loading) data for each run would result in a quite inefficient and still failure-prone process. This due to that the global projection normally will be huge, and in the worst case it will be contained the whole input document for satisfy all queries composed the workload. Therefore, the standard projection still failure in case of processing a query workload.

It is worth observing that scalability issues also depend on the particular kind of engine one wants to use, and on internal memory settings. In fact, most of main-memory system are implemented in Java, and their scalability depends on the amount of main-memory given to the Java Virtual Machine. In any case, even for large amounts, scalability problems of standard projection still persist, as the size of document projection increases as the size of the input document increase.

The main objective of this Thesis is to offer a technique that ensures scalability for both queries and updates independently of:

- the kind of main-memory system.
- the amount of available main-memory.
- the presence of schema information.

To this end, in this Thesis, we propose an optimization technique based on *data partitioning*. This technique relies on the observation that, in many practical

cases, XQuery queries and updates first select a sequence of subtrees by means of a subquery (e.g, an XPath expression), and then iterate operations on this sequence of subtrees. For instance, concerning queries, 13 out of 20 queries of the XMark benchmark meet this property, while concerning updates, 16 out of 20 updates in the benchmark adopted in [BBC+11, Sah11] are iterative.

In the case of queries, when this property is satisfied by a query $Q$, the input document can be split into a collection of parts $\{D_1, \ldots, D_\kappa\}$, so that the evaluation $Q(D)$ of the query $Q$ over the document $D$ turns out to be equal to the concatenation of the evaluations $Q(D_i)$ of the query $Q$ over the document parts $D_i$.

For updates, the same strategy can be adopted, with the difference that partial updates $U(D_i)$ have to be recombined so that the updated document $U(D)$ can be obtained. While in the case of queries a simple concatenation of partial result is sufficient. In particular we use the command `cat` to combine these partial results in order to produce the final one. For updates, and since we use additional tags during the creation of the partitions in order to hold the well-formedness of the created parts, auxiliary information about these additional tags is needed in order to *correctly* re-combine updated parts and eliminate these tags to obtain the final update result $U(D)$. This auxiliary information is opportunely built up during partitioning.

Besides scalability, our partitioning technique can be easily adapted to be adopted in a MapReduce [DG08] framework, enabling parallel querying or updating of parts composing a partition. This is due to the fact that iterative queries and updates enjoy the property that evaluation on each part does not need information coming from evaluation on another part. The possibility of an easy transposition in a MapReduce framework plays an important role nowadays, given the currently rapid and large diffusion of cloud-based platform based on this paradigm.

## 2.1 CONTRIBUTIONS

This Thesis proposes a novel technique for partitioning-based evaluation of XQuery queries and updates.

The first contribution of this Thesis focuses on queries. In this context, main contributions are the following ones, and are also reported in [Nic12]:

- We first present a formal characterization of the class of queries that enjoy the above described splitting property: we dub these queries as *iterative queries*. By relying on this formal characterization, we develop a static analysis technique that first extracts paths and information about bound variables from the query, and then analyses them in order to statically detect how the document is navigated by the query. Relying on path information allows us to avoid the use of schema information, which is not always available.

- We then present a partitioning algorithm that exploits the paths extracted during the static analysis to identify the correct partitioning for the input document. We first present DOM-based specification of the algorithm, and then a SAX based on enabling the possibility of performing partitioning in a streaming fashion, with a very limited memory footprint. To further improve the benefits of our approach, we combine partitioning with standard projection, so that during the creation of document parts, sub-trees not needed by the query are pruned out. The use of projection is not crucial to ensure scalability, which is our main purpose, since our approach is so that the maximal size of each part can be tuned by the user. Projection helps in reducing the overhead of partitioning, since it speeds up query execution on the partition.

- Then, we present extensive experimental evaluation that corroborates that, when using our partitioning approach, main-memory engines can process documents of arbitrary size, at the price of a modest overhead with respect to schema-less projection techniques; our experiments also show that partitioning allows for a scalable management of workloads, as the input document is partitioned once for all.

The second contribution of this Thesis concerns updates. In this context, main contributions are the following ones:

- We first analyze cases in which update evaluation can be correctly done on partitions, and then provide a static analysis to characterize such updates, which we call *iterative updates*. This characterization requires restrictions on the querying mechanisms that can be used in *source* and *target* expressions of updates. We will show that these restrictions are mild, in the sense that a wide class of updates can be dealt with our approach.

- We then present a partitioning technique which distinguishes from that of queries for the following two aspects.

  First, projection is not used, in order to have a simple and efficient re-combination process of partial updates. This is also justified by the fact that partitioning is already sufficient to generate small enough parts. The use of projection would require a sophisticate re-combination process, since subtrees pruned out during partitioning should be recognized and reported in the final result of the process. This kind of operation has been done [BBC+11], where the use of schema information was crucial to ensure a clear formalization and efficiency.

  Second, paths used during partitioning are inferred by keeping into account the particular nature of updates. These paths are used in order to ensure that subtrees eventually selected by target paths are never split during partitioning. Atomicity of these subtrees is necessary to ensure that the update evaluation can be correctly distributed over all the input parts.

- Then, we present extensive test results showing the effectiveness of out technique. Differently from the case of queries, the overhead due to partitioning is not negligible. However test results show that our main goal, scalability is largely attained.

Concerning test results, we used two main-memory engines, Saxon [sax] and Qizx [qiz]. Our choice is motivated as follows. Saxon is a very popular system, which distinguishes for its exhaustiveness in covering most W3C standards for XML processing (e.g., XML Schema, XSLT, XQuery queries and updates). Differently, Qizx is specialized in XQuery query and update, and supports sophisticated techniques to optimize both execution time and memory consumption.

As a third contribution, this Thesis shows that the proposed framework can be easily adapted in order to be run in a MapReduce framework [DG08]. To this end, main notions behind this paradigm are introduced first, and then the architecture of the MapReduce implementation of our framework is illustrated and discussed.

## 2.2 STRUCTURE OF THE THESIS

The Thesis is organized as follows:

- **Chapter 2** Introduces XML and XQuery Update Facility and provides some basic notions and definitions.
- **Chapter 3** Presents standard projection techniques and shows limitations of these ones in terms of scalability.
- **Chapter 4** Presents our partitioning technique for XQuery queries, together with experimental results.
- **Chapter 5** Presents our partitioning technique for XQuery updates, together with experimental results.
- **Chapter 6** Illustrates how our partitioning techniques can ensure parallel query and update evaluation by means of the MapReduce paradigm.
- **Chapter 7** Discusses related works, conclusive remarks and directions for future works.

Chapter 3

# Preliminaries

**Contents**


**3.1 XML . . . 19**
3.1.1 Textual Representation . . . 20
3.1.2 Well-Formedness of XML . . . 20
**3.2 Querying XML . . . 22**
3.2.1 XPath Language . . . 22
3.2.2 XQuery Language . . . 26
3.2.3 XQuery Update Facility . . . 27
**3.3 conclusion . . . 30**


This chapter has two essential sections. In the first one, we present some basic notions about XML data and its characteristics. In the second section, we first introduce the XML query languages: XPath and XQuery, and then introduce the update extensions provided by XQuery Update Facility language. All of these languages are W3C standards [Gro03, Gro11a, W3S10].

## 3.1 XML

XML (eXtensible Markup Language) is among the most popular data formats for representing data generated and exchanged by Web application. In particular, XML is widely adopted to describe different kinds of data such as HTML (HyperText Markup Language) data, relational and object database, multimedia files (audio, video), and so on.

XML actually is a simplified form of SGML (Standard Generalized Markup Language), and it is a W3C standard 1998 [BPMM08]. The syntax of XML data is very similar to that of HTML. However, there are some deep differences between both of them. The most important one is that HTML has predefined element tags and attributes whose behavior is well specified, while XML does not. For instance, in XML the user can adopt a `<name>` tag, while in HTML the user is obliged to use predefined tags such as `<body>`, `<head>`, `<title>`, `<p>`, etc.

The possibility of using non-predefined tags makes XML data *self-describing*. This, together with the possibility of free element nesting and mixed contents, make XML an high flexible language for data representation.

### 3.1.1 Textual Representation

According to the W3C, the basic component of an XML document is the *element*, which consists of a piece of text enclosed by an *open-tag* and its corresponding *close-tag*. The content of each XML element can be simple text value, a sequence of elements, or a mixed sequence which includes the two previous forms (text values and elements). Figure 3.1 represents a simple fragment of an XML document. It shows that elements are denoted by markup tags. For example, the *open-tag* `<name>` and the *close-tag* `</name>` represent an XML element, and the text value `Jean Scott` included between both of them refers to the content of this XML element. Elements with empty content are called *empty elements*, and have an abbreviated notation, as indicated by the empty element `<email/>`. The element `<note>` contains a complex sequence which includes elements such as `<telephone>` and text values. Elements can be annotated with attributes that contain meta data about the element and its contents. For example, the element `<person>` has a single attribute named `gender` with a simple value `M`.

```
<person gender = "M">
   <name> Jean Scott </name>
   <age> 35 </age>
   <email/>
   <note> The personal phone of Jean is :
      <telephone> 0033110203040 </telephone>
   </note>
</person>
```

Figure 3.1: Textual representation of an XML fragment.

### 3.1.2 Well-Formedness of XML

According to the W3C, an XML document is considered as well-formed if the following constraints are met. We summarize below the main ones.

- An XML document must be contain at least one element.
- Only one element must be contain the whole XML document; this element is called the *root element*.
- All element tags must be nested properly, and there is no overlap between them.
- Tags in XML are case sensitive. This means that `<Name>`, `<NAME>` and `<name>` are not the same.
- Attribute values must always be quoted.

Here, we have a list of non well-formed examples of XML elements:

- `<name>Jean Scott</lastName>`

  The *open-tag* and *close-tag* do not match.

- `<person><age></person></age>`

  The element tags are not nested properly.

- `<country> </couNtry>`

  Due to case sensitivity, *open-tag* and *close-tag* do not match.

- `<person gender = M>`

  The attribute value misses quotes.

As already said, in this Thesis we focus on a schema-less approach, in the sense that we do not rely on schema information. However we briefly introduce DTD (Document Type Definition) which is a widely used schema language. This introduction will help in understanding related works on updates [BBC+11] that make use of schemas in the form of DTD.

In a nutshell, A DTD schema consists of a set of declarations used for describing the structure of elements and attributes. The content of each element is described by means of regular expressions. elements, attributes and another constructors are used to describe the formal structure of the content for a well-formed XML document. To this end, regular expressions are used.

DTD declarations have the following form:

```
<!ELEMENT element-name (element-content)>
```

where `element-name` represents the name of element tag in an XML document (such as person, name, email, etc.) while `element-content` is either an empty content or a regular expression over tags and text-symbols representing the structure form of the element-content.

Each DTD starts with the declaration of the root element, and then it continues with specification of other elements. A DTD for our (*addressBook.xml*) document is described in Figure 3.2. In particular, the declaration says that its content has to be a sequence of zero or more of elements tagged as `person`. The DTD also specifies that content of each element `person` consists of two elements `name` and `age`, followed by two optional `telephone` and `email` elements, and finally an essential `note` element. The value `#PCDATA` is used to declare the text-content of each element node in the document (*addressBook.xml*). This text-content consists of a sequence of characters (string values) without interleaved XML element nodes. The declaration for the `person` attribute says that two possible values are admitted, and that “M” is the default one.

In many contexts, it is convenient to have a tree representation of an XML documents. In many examples that we use in next chapters, we rely on tree representation. Any XML document is actually tree shaped. The root corresponds to

```
<!DOCTYPE addressbook[
    <!ELEMENT addressbook (person *)>
    <!ELEMENT person (name, age, telephone?, email?, note)>
    <!ATTLIST person gender (M|F) "M">
    <!ELEMENT name (#PCDATA | (firstname, lastname))>
    <!ELEMENT firstname (#PCDATA)>
    <!ELEMENT lastname (#PCDATA)>
    <!ELEMENT age (#PCDATA)>
    <!ELEMENT telephone (#PCDATA)>
    <!ELEMENT email (#PCDATA)>
    <!ELEMENT note (#PCDATA | email | telephone)*>
]>
```

Figure 3.2: DTD of *addressBook.xml* XML document.

the root element, children of this elements correspond to sub-elements and textual nodes, and so on. A tree representation of our addressBook element is given in Figure 3.4.

In the next chapters, we will mainly focus on documents only containing elements. This is to simplify the formal treatment; our approaches easily extend to attributes. As a consequence, figures will be simpler too, as only element nodes will occur.

Figure 3.4 uses a graphical tree representation to describe the addressBook document. In this Thesis, we will often rely on graphical tree representation to illustrate our concepts.

## 3.2 Querying XML

This section introduces two XML query Languages: XPath and XQuery, both W3C standards. An excellent overview about the XQuery language is presented in [KCD+03], and another overview about XPath language is introduced by [Gro03]. A formal introduction to these languages is out of the scope of this Thesis. In this section, we only focus on the basic structures of XPath expressions and XQuery languages, and introduce them mainly by means of examples. Subsequent chapters will then provide formal characterizations of the fragments of these languages we will deal with.

### 3.2.1 XPath Language

XML Path Language (XPath) is one of the most popular languages used in XML technologies. It provides support for navigating through XML trees in order to select nodes satisfying some structural and value-based properties.

The main constructor in XPath language is the expression. Essentially, an XPath expression consists of a sequence of *steps* separated by the symbol /. Each *step*

```
<addressbook>
   <person gender = "M">
      <name> Jean Scott </name>
      <age>35</age>
      <email/>
      <note>The personal phone of Jean is :
         <telephone>+33110203040</telephone>
      </note>
   </person>
   <person>
      <name>
        <firstname>Steven</firstname>
        <lastname>Wesley</lastname>
      </name>
      <age>38</age>
      <telephone>+33155209940</telephone>
      <email>steven.wesley@ITcompany.com</email>
      <note>
         Work administrator, his mobile phone:
         <telephone>+33811773700</telephone>
         his email:<email>steven.boss@speedymail.com</email>
      </note>
   </person>
</addressbook>
```

Figure 3.3: A well-formed XML document.

consists of three parts; two mandatory parts are *axis* and *node test*, while an optional part is *predicate*.

Informally, the three components of *step* are defined as follows:

1. an *axis* defines the relationship between the context node and the nodes selected by the step.

2. a *node test* specifies the node type and the expanded-name of the selected nodes.

3. zero or more *predicates*, which use arbitrary expressions to further refine the set of selected nodes.

The evaluation of each *step* returns a sequence of nodes. The current node over which a *step* is evaluated is called *context node*, and the value returned by an XPath expression is the value returned by the last *step* of this expression.

For example, when the following *step* `child::person` is evaluated, the axis `child` selects all children nodes of the *context node*. Then, among these nodes, the condition `person` selects only children nodes corresponding to elements named as `person`. It is very important to note that nodes are resulted according to the

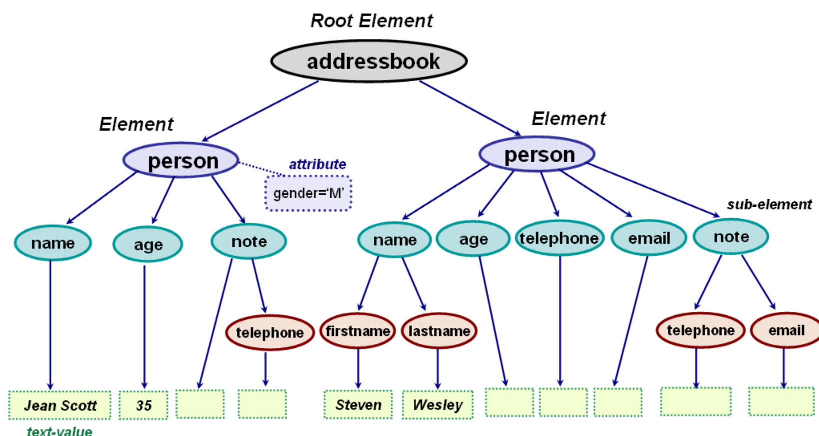



Figure 3.4: Tree representation of addressBook XML document.

document order. Also, it is important to note that XPath assumes that navigation through a document always starts from what is called the *document root*, which can be seen as a virtual node having as only child the document root element. The document root is selected by the simple expression /, so for our previous addressBook document `/child::addressbook` selects the root element `addressbook`, while `/child::addressbook/child::person` select the sequence of all `person` elements.

The following brief description presents some of available axes in XPath (Figure 3.5 illustrates these navigating axes):

- `self` axis selects the context node itself.
- `child` axis selects all children of the context node.
- `descendant` axis selects all descendants (children, grandchildren, etc.) of the context node.
- `descendant-or-self` axis selects all descendants of the context node and the context node itself.
- `parent` axis selects the parent of the context node, which is either an element node or the root node (or an empty sequence if the context node is the root node).
- `ancestor` axis selects all ancestors (parent, grandparent, etc.) of the context node, from its parent to the root node.
- `ancestor-or-self` axis selects all ancestors of the context node, from its parent to the root, and the *context node* itself.

As said before, the second essential part used to compose an XPath *step* is the *node test*, which has one of the following forms:

- `node()`: selects nodes of any type.
- `text()`: selects text nodes.
- `tag`: selects only nodes that have the element-name `tag`. For example, the element-name `age` in the step `child::age`, which selects only nodes corresponding to elements named as `age`.

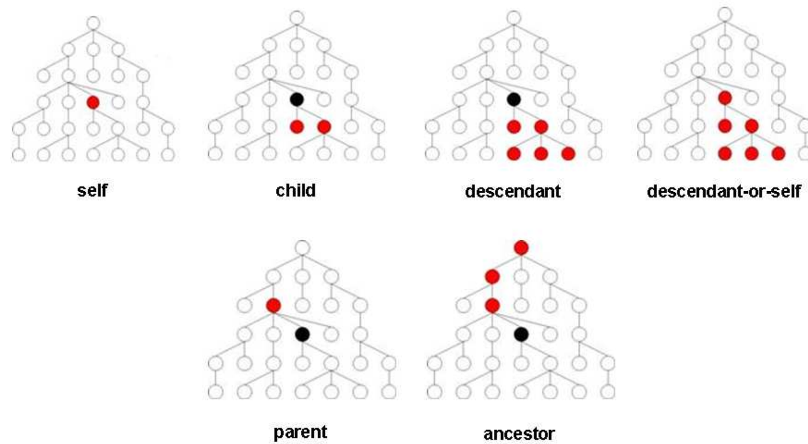



Figure 3.5: Navigational XPath axes.

In the following we give some examples of XPath expressions. The next query selects all `email` elements that are children of `person` elements. This is performed by using a specific path to be followed in order to select the requested `email` elements:

```
/child::addressbook/child::person/child::email
```

which can have the following abbreviated version (the `child::` part is omitted)

```
/addressbook/person/email
```

Another abbreviation that is admitted is that allowing the use of `//a` instead of `/descendant-or-self::node()/child::a`. So the following query selects all email elements in the document `addressBook.xml`.

```
//email
```

XPath uses predicates in its query syntax to limit the extracted data from an input XML document. The following predicate is used to select all `person` elements that have an attribute `gender` with a value `"M"`:

```
doc("addressbook.xml")//person[gender = "M"]
```

### 3.2.2 XQuery Language

The XQuery language is a flexible and powerful query language for XML data. XQuery language is built on XPath expressions, and can be used in several tasks, such as:

- Extract information from an XML database to use in a Web service.
- Generate summary reports about data stored in an XML database.
- Search textual documents on the Web for relevant information.
- Transform XML data to XHTML to be published on the Web.

In all these contexts, XPath is not sufficient, as mechanisms to select tuples of nodes, and build new ones are needed. The most used fragment of XQuery consists of FLWR expressions. The name FLWR comes from the initial letters of the following clauses:

- `for`-clauses first select a sequence of nodes, and then perform some query operations on each node;
- `let`-clauses bind a sequence of nodes to a specific variable, which can be used into another expression;
- `where`-clauses filter nodes depending on a boolean expression;
- `returtn`-clauses build values resulted by a query.

Most of these clauses are optional, except the `return` clause .This clause is always attached with at least one `for` or `let` clause. In general, a FLWR expression may contain many `for/let` clauses before the `return` clause.

The simplest FLWR expression containing a `for` clause has the following form:

```
for $x in Q1 return Q2
```

First of all, this query evaluates `Q1`, and then for each node in the resulting sequence, it binds this node to the variable `$x` and evaluates `Q2` accordingly. Note that the evaluation of `Q2` is performed according to the sequence order of `Q1` result. The final result is obtained by concatenating all `Q2` results.

The following examples illustrate a query returns the sequence `age` element of all `person` elements in the document `addressBook.xml` presented in Figure 3.3:

```
for $x in doc("addressbook.xml")//person
return $x/age
```

The following example uses a `where` clause to select exactly the same result of the previously seen query `doc("addressbook.xml")//person[gender = "M"]`

```
for $x in doc("addressbook.xml")//person
     where $x/@gender = "M"
return $x
```

XQuery also provides `if-then-else` expressions. For instance, the above query is equivalent to the following one using this kind of expressions:

```
for $x in doc("addressbook.xml")//person
return
     if $x/@gender = "M" then $x else ()
```

where () denotes the empty sequence.

The following query produces two kinds of elements depending of the gender of persons:

```
for $x in doc("addressbook.xml")//person
return
     if $x/@gender = "M" then <m/> else <f/>
```

An example illustrating how multiple `for/let` clauses can be combined is the following one:

```
let $x := doc("addressbook.xml") return
for $y in $x//person
let $w := $y/age
  where $w > 35
return $y/note
```

In the above example, each `for/let` clause is evaluated in a scope determined by previous clauses. The query above will return the following data:

```
<note>
   Work administrator, his mobile phone:
   <telephone>+33811773700</telephone>
   and private email:<email>steven.boss@speedymail.com</email>
</note>
```

### 3.2.3 XQuery Update Facility

The XQuery language is provided with a powerful extension, called *XQuery Update Facility* (XUF), for updating XML documents. The XUF language became a W3C candidate recommendation in 2009, and was finalized as recommendation in 2011 [Gro11a]. Basic updating operations provided by XUF are the following ones:

1. delete one or several nodes.

| | | |
|---|---|---|
| **DeleteExpre** | ::= | *"delete"* (*"node"* \| *"nodes"*) **TargetExpr** |
| **RenameExpre** | ::= | *"rename"* *"node"* **TargetExpr** *"as"* string-value |
| **ReplaceExpr** | ::= | *"replace"* (*"value of node"*\|*"node"*) **TargetExpr** *"with"* **SourceExpr** |
| **InsertExpre** | ::= | *"insert"* (*"node"* \| *"nodes"*) **SourceExpr** **InsertExpreTargetChoise** **TargetExpr** |
| **InsertExprTargetChoice** | ::= | *"as"* (*"first"*\|*"last"*) *"into"* \| *"after"* \| *"before"* |

Figure 3.6: The W3C syntax of simple XQuery updates.

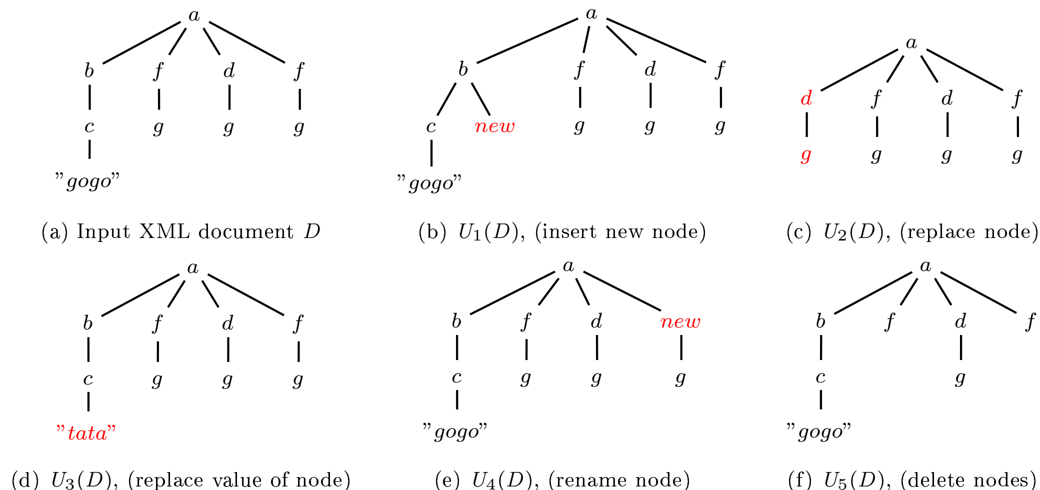
2. rename a name of an element node.

3. replace an existing node with a new node or several new nodes.

4. insert a node or several nodes into an existing node.

The syntax of the XUF language, according to the W3C recommendation, is reported in Figure 3.6. In this syntax, the **TargetExpr** computes the target location where the update operation is taking place, while the **SourceExpr** returns a new fragment which will be inserted or replaced in the target location.

In Figure 3.7, we illustrate the main update mechanism by means of some examples. The input document $D$ is reported in Figure 3.7-(a).




(a) Input XML document $D$ (b) $U_1(D)$, (insert new node) (c) $U_2(D)$, (replace node)

(d) $U_3(D)$, (replace value of node) (e) $U_4(D)$, (rename node) (f) $U_5(D)$, (delete nodes)

Figure 3.7: Simple XQuery updates.

The result of the first simple update $U_1(D)$ on the input document $D$ reported in Figure 3.7-(a) is illustrated in Figure 3.7-(b). This update inserts an empty new node `<new/>` after the last `/a/b/c` in $D$, by using the following expression:

$U_1 =$ `insert node` <new/> `after` $doc(D.xml)/child :: a/child :: b/child :: c[last()]$

Figure 3.7-(c) illustrates the update result $U_2(D)$ of the document $D$ produced by a simple update $U_2$. This update replaces the node (and its subtree) selected by `/a/b` with another subtree selected by `/a/d`.

$U_2 =$ `replace node` $doc(D.xml)/child :: a/child :: b$
`with` $doc(D.xml)/child :: a/child :: d$

Figure 3.7-(d) illustrates the updated result $U_3(D)$ after evaluating the simple update $U_3$ on $D$, which replaces the text-value of the last `c`-node located after the node selected by `/a/b` with a new value `"tata"`, as follows:

$U_3 =$ `replace value of node` $doc(D.xml)/child :: a/child :: b/child :: c[last()]$
`with` "*tata*"

Figure 3.7-(e) illustrates the updated result $U_4(D)$ produced by evaluating the simple update $U_4$ on the document $D$. This update renames the label-name of the last `f`-child node as `"new"`, as follows:

$U_4 =$ `rename node` $doc(D.xml)/child :: a/child :: f[last()]$ `as` "*new*"

The last update result $U_5(D)$ illustrated in Figure 3.7-(f) which deletes all subtrees rooted at `g`-node of `f`-nodes existed in the document $D$, as follows:

$U_5 =$ `delete nodes` $doc(D.xml)/child :: a/child :: f/child :: g$

A second form of XQuery updates relies on conditional or FLWR expressions. For example, consider the following conditional update:

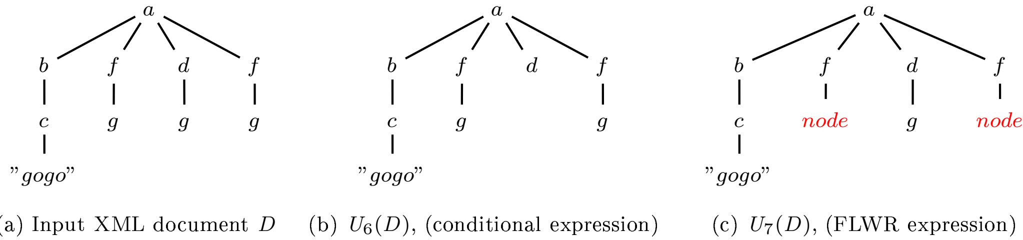
```
U6 = let $x := doc(D.xml)/child :: a/child :: d return
       if $x/child :: g then
         delete node $x
       else
         replace value of node $x with "node"
```

This update deletes each child `g`-node of `d`-node if it exists, otherwise it replaces the label-name of `d`-node with `"node"`. The result of evaluating this update on the document $D$ is illustrated in Figure 3.8-(b).

Another example is used to apply a simple update `rename` during an iteration:

```
U7 = let $i := doc(D.xml) return
       for $x in $i/child :: a/child :: f
         where $x/child :: g
       return rename node $x/child :: g as "node"
```

This update navigates the whole document $D$ and checks each `/a/f` subtree whether it contains a child `g`-node, if it exists then the update will rename the




(a) Input XML document $D$ (b) $U_6(D)$, (conditional expression) (c) $U_7(D)$, (FLWR expression)

Figure 3.8: Complex XQuery updates.

label-name of `g`-node with a new label `node`, otherwise no update will be performed in $D$. The final update result $U_7(D)$ is illustrated in Figure 3.8-(c).

It is worth noticing that according the W3C semantics, some constraints must be preserved during the update execution. For previous examples of simple/complex update expressions, these constraints are held and described in the following remarks.

**Remark 1** In order to execute a simple insertion, the **TargetExpr** must be a single node, otherwise if it is an empty sequence or contains a set of nodes, a run-time error will rise and the insert update will not be performed.

**Remark 2** In order to perform a simple deletion, the **TargetExpr** must be a single expression to avoid getting a dynamic error during the execution.

**Remark 3** In order to perform a simple replacement, the **SourceExpr** must be a content sequence which is either an empty sequence, a set of element nodes or string values. Otherwise a runtime error is risen.

Actually these constraints are orthogonal to our work, and we assume that our update language satisfies these constraints.

## 3.3 CONCLUSION

This chapter has provided a simple introduction to XML, the query language XQuery and the update language XQU. The presented introduction is far from being exhaustive. However, we have focused on a significant fragment covering mechanisms used in practice. In this Thesis we focus on such fragment.

Chapter 4

# XML Projection and its Limitations

## Contents


**4.1 Path-based projection for queries** . . . . . . . . . . . . . **32**

4.1.1 Limitations of Standard Projection for Queries . . . . . . . . 35

**4.2 Type-based projection for updates** . . . . . . . . . . . . . **39**

4.2.1 Limitations of Update Type-based Projection . . . . . . . . . 45

**4.3 Conclusion** . . . . . . . . . . . . . . . . . . . . . . . . . . . **51**


As we said in the introduction, XML data projection is one of the most important techniques used for reducing the memory consumption of main-memory XML (query/update) engines. The main idea behind this technique is quite simple and productive as well: given a query $Q$ on an XML document $t$, instead of evaluating $Q$ on $t$, the query $Q$ is evaluated on a smaller document $t'$ obtained from $t$ by pruning out, at loading-time, all subtrees of $t$ that are not necessary to evaluate $Q$. The projection $t'$ is often much smaller than the original $t$ due to the high selectivity of queries. This technique ensures a big improvement in terms of the execution memory consumption, as it allows the main-memory engine to query large documents, and also ensures gains in terms of querying time.

In this chapter, we discuss main features of two main approaches proposed for XML projection. The first one [MS03] concerns queries, and is based on query path extraction and on the use of extracted paths to project the input document. Another approach for queries has been proposed in [BCCN06], and requires schema information about data. We will not discuss it as this Thesis is in a schema-less setting, and, for the XQuery fragment we consider, performances of [BCCN06] are closed to that of [MS03] in terms of size reduction.

The second technique we will discuss concerns updates [BBC+11, BCMS09a, BCMS09b], and is the only existing projection technique for updates. It is based on schema information and on type inference, plus a novel *Merge* operation that, as we will see, is needed to recombine the updated projection with the original document.

In this chapter, besides illustrating how projection can be used to process a wide class of XML queries and updates on large XML documents, we will show that these

techniques, even if quite effective, do not scale up with respect to document size. This has motivated our investigation towards partitioning techniques.

The chapter is organized as follows. Section 4.1 introduces the XML standard projection proposed by [MS03] and some principal definitions, the path analysis algorithm which extracts the set of projection paths from an arbitrary XQuery query. Then explain the loading algorithm used to create the projection. Section 4.1.1 illustrates the limitations of the XML standard projection technique by testing several queries on XMark documents and DBLP database. In Section 4.2, we introduce, through examples, the concept of the type-based projection technique proposed by [BBC+11]. Then in Section 4.2.1, we illustrate the limitations of this technique with updates. Finally, we draw our conclusion in Section 4.3.

## 4.1 PATH-BASED PROJECTION FOR QUERIES

The path-based, and schema-less, approach for XML projection has been proposed by Marian and Siméon in [MS03]. The main contribution of this work is a static analysis algorithm used to extract paths from an XQuery query. Extracted paths specify which parts of an input XML document are sufficient to execute the XQuery query, and are used by a streaming algorithm to prune out parts of the document that are not needed by the query.

To illustrate, consider the following query on XMark documents [SWK+02a]:

```
Q1 = for $b in /site/people/person[@id="person0"] return $b/address
```

By evaluating this query on the input XML document $D$ illustrated in Figure 4.1, we have that this query does not need to process all parts in the original document. Actually, it only needs to process parts corresponding to the following projection paths (we will see later the meaning of #):

$$
\begin{aligned}
P_1 &= /\text{site}/\text{people}/\text{person}/@\text{id} \\
P_2 &= /\text{site}/\text{people}/\text{person}/\text{address}\#
\end{aligned}
$$

The resulting document obtained by using these paths for projection is illustrated in Figure 4.1.

In [MS03], a simple fragment of XPath [Dra02] is used to define the syntax of the projection paths. Each projection path starts from the root and consists of a simple path expression followed by an optional "#" flag. This optional flag is used to indicate whether the descendant subtrees returned by the whole path expression should be kept in the projected document. In Figure 4.1 it can be observed that the whole subtree selected by the projection path $P_2$ is kept in the projection.

The syntax of a simple path expression is defined by the following grammars:

```
<site>
 <regions>...</regions>
 <people>
  <person id="person0">
    <name>Xiulin Poch</name>
    <emailaddress>mailto:Poch@unizh.ch</emailaddress>
    <phone>+0 (847) 37140499</phone>
    <homepage>http://www.unizh.ch/ Poch</homepage>
    <creditcard>1655 3174 7975 9805</creditcard>
    <watches>
      <watch open_auction="open_auction124"/>
    </watches>
  </person>
  <person id="person1">
    <name>Remco Sevcikova</name>
    <emailaddress>mailto:Sevcikova@edu.sg</emailaddress>
    <phone>+0 (628) 90891260</phone>
    <address>
      <street>69 Yaru St</street>
      <city>Brunswick</city>
      <country>United States</country>
      <province>Maine</province>
      <zipcode>23</zipcode>
    </address>
    <homepage>http://www.edu.sg/ Sevcikova</homepage>
  </person>
  ...
 </people>
  ...
</site>
```

Figure 4.1: A fragment of the input XMark document $D$.

$$
\begin{array}{rcl}
SimplePath & ::= & Axis :: NT \mid SimplePath/Axis :: NT \\
Axis & ::= & child \mid self \mid descendant \\
 & \mid & descendant\text{-}or\text{-}self \mid attribute \\
NT & ::= & node() \mid text()
\end{array}
$$

As it can be seen, this technique assumes that XQuery queries use only downward axes.

The path extraction algorithm proposed in [MS03] is able to extract a set of projection paths from an arbitrary XQuery expression. We omit here details about the rules, and in the sequel we focus on the projection algorithm using extracted paths, as partitioning algorithms we will present share some mechanisms with this one.

The projection algorithm processes the input in a SAX fashion [ver00]. In par-

ticular, this projection algorithm works in a recursive way. It starts to parse the original document $D$, and considers each node read from $D$ as an independent event. It uses the following specific SAX events during the process:

$$\begin{array}{rcl} SAXEvent & ::= & \texttt{OpeningTag}\ (qName) \\ & | & \texttt{Characters}\ (String) \\ & | & \texttt{ClosingTag}\ (qName) \end{array}$$

The `OpeningTag` $(qName)$ event occurs when the opening tag of an element is met; the tag value is represented by $(qName)$. The `Characters` event occurs when a text node is met during the parsing, and the text value is represented by $String$. The `ClosingTag` $(qName)$ event is dual and occurs when a closing tag is met.

When the SAX parser begins the processing operation, the loading algorithm starts to check the correspondence between the current projection paths and the `OpeningTag` token of the current node $qName$. If this $qName$ matches the first step of each projection path, this means that the loading algorithm should keep this node in the projection $D'$ which is normally smaller than the original document $D$. Moreover, the algorithm in this case will check if the creation of $D'$ needs to keep the subtree of this current $qName$ or not. If there is no match between the current $qName$ and the current projection paths, here the algorithm will skip this $qName$ together with all the ones that follow until the corresponding close-tag.

Figure 4.2 presents a simple XML example on which we will explain how the loading algorithm works:

```
<a>
  <g><b></b></g>
  <b><c><f></f></c></b>
  <d><e></e></d>
  <b></b>
  <c></c>
</a>
```

Figure 4.2: An XML document fragment.

In this example, the loading algorithm will use a certain set of projection paths /a/b/c#, /a/d to create a projected fragment from the original one presented in Figure 4.2. All operation steps of the loading algorithm are explained clearly in Figure 4.3.

It is worth observing that the algorithm is not fully specified in [MS03], since the focus is on the path-extraction algorithm. The description provided in [MS03] is limited to some examples, and the way itself and descendant axes are dealt with is not discussed in details. In the next chapters, we will formally specify both path-extraction and projection mechanism. Experimental results we provide next have been obtained by using our implementation which is presumably equivalent to that

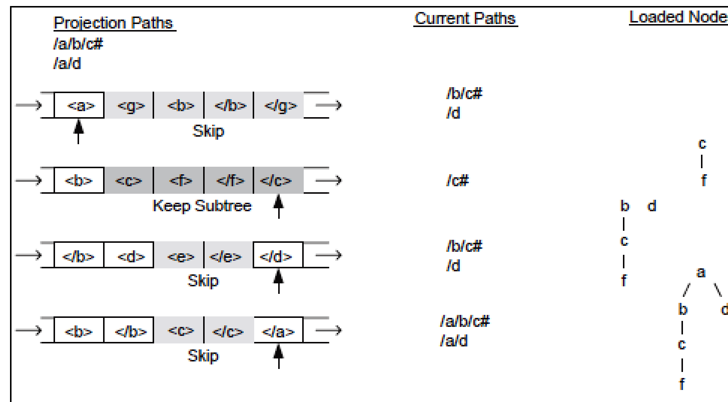



Figure 4.3: Loading algorithm of [MS03] for building a projection.

of [MS03]. In our implementation, we have followed principles outlined in [MS03] in order to minimize as much as possible the size of projection. For instance, in the presence of a projection path *descendant* :: $a$, the projection process prunes out nodes not having a descendant labeled as $a$.

### 4.1.1 Limitations of Standard Projection for Queries

The path-based XML projection technique introduced in [MS03] is an effective technique: it allows main-memory systems to query large documents. Unfortunately, as already said in the introduction, this technique still has limitation since for very large documents and for queries needing a large part of the input for evaluation, the projected document is likely to be too big to be loaded for querying by the main-memory system.

In particular, the following kinds of XQuery queries are likely to need too large projections.

- Queries performing the descendant navigation are likely to select large portions of the input. For instance, for XMark data, some performed tests revealed that for a simple query like $//text$, the projection takes around 65% of the original document.

- Full-text queries need to query textual nodes of the input. As textual content of an XML document can cover large portions of it, the needed projection is likely to be large too. An example of such queries is the query $N_2$ hereafter discussed.

- Queries producing a document with the same content of the input, but with different structure, actually need the whole document to be processed. Actually, even the structural reorganization concerns sub-parts of the document,

| Size of standard projection in (MB) | | | | | |
|---|---|---|---|---|---|
| Query | 1GB | 2GB | 3GB | 4GB | 5GB |
| $Q_1$ | 15,1 | 26,2 | 39,3 | 52,5 | 66,5 |
| $Q_2$ | 36,4 | 62,8 | 94,3 | 125,9 | 159,4 |
| $Q_3$ | 36,4 | 62,8 | 94,3 | 125,9 | 159,4 |
| $Q_4$ | 44,2 | 76,5 | 115 | 153,6 | 194,5 |
| $Q_5$ | 5,2 | 8,9 | 13,3 | 17, 8 | 22,5 |
| $Q_6$ | 6,61 | 11,5 | 17,3 | 23,1 | 29,3 |
| $Q_7$ | 30,9 | 53,4 | 80,1 | 106,8 | 135,4 |
| $Q_8$ | 21,5 | 37,2 | 55,9 | 74,5 | 94,5 |
| $Q_9$ | 25,2 | 43,5 | 65,4 | 87,3 | 110,8 |
| $Q_{10}$ | 77,1 | 133,2 | 200,1 | 266,9 | 338,6 |
| $Q_{11}$ | 28,1 | 48,7 | 73,1 | 97,6 | 123,8 |
| $Q_{12}$ | 21,3 | 36,8 | 55,4 | 73,9 | 93,8 |
| $Q_{13}$ | 33,1 | 57,2 | 85,9 | 114,5 | 145,5 |
| $Q_{14}$ | 328,4 | 567,4 | 851,2 | 1,1GB | 1,4GB |
| $Q_{15}$ | 149,8 | 258,3 | 386,5 | 514,1 | 651,2 |
| $Q_{16}$ | 153,3 | 264,3 | 395,6 | 526,2 | 666,4 |
| $Q_{17}$ | 20,9 | 36 | 54,1 | 72,2 | 91,6 |
| $Q_{18}$ | 7 | 12,1 | 18,2 | 24,3 | 30,8 |
| $Q_{19}$ | 20,5 | 35,5 | 53,3 | 71,1 | 90,2 |
| $Q_{20}$ | 12,8 | 22,2 | 33,4 | 44,5 | 56,5 |
| $N_1$ | 543,5 | 938 | 1,37GB | 1,83GB | 2,32GB |
| $N_2$ | 670,8 | 1,13GB | 1,69GB | 2,26GB | 2,86GB |
| $N_3$ | 527,6 | 910,6 | 1,33GB | 1,78GB | 2,25GB |

Table 4.1: Size of projected documents.

projection is likely to be too large. An example of such queries is the query $N_1$ discussed in the sequel.

| Query execution time on the standard projection | | | |
|---|---|---|---|
| Query | Projection size in (MB) | Saxon in (sec) | Qizx in (sec) |
| $D_1$ | 313 | - | 194,11 |
| $D_2$ | 517 | - | 381,43 |

Table 4.2: Qizx and Saxon performances on projected DBLP document.

The above queries are likely to occur in practice and need large projections even for not so big documents. However even quite selective queries, like the XMark ones, can make projection fail when the input is quite large. Next we will show experimental results reporting the size of projections obtained by means of the

path-based technique, for each XMark queries and for documents with size ranging from 1GB to 5GB. We will also show the same kind of test results for three new XMark queries ($N_1$, $N_2$, and $N_3$), also two queries ($D_1$ and $D_2$) to be evaluated on a 800MB DBLP document [ver11]. These five new queries need to very large projections. The syntax of these new queries is illustrated below.

```
N1 = let $auction := doc("xmark.xml") return
     for $i in $auction/site//item
     where $i/location/text() = "UnitedStates"
     return
            <itemInfo name ="$i/name/text()">
                      <paymentWay>$i/payment/text()</paymentWay>
                      <shippingWay>$i/shipping/text()</shippingWay>
                      <moreInfo>$i/description</moreInfo>
                      <mailboxInfo>$i/mailbox</mailboxInfo>
            </itemInfo>
```

```
N2 = let $auction := doc("xmark.xml") return
     for $i in $auction/site//description
     where contains(string(exactly-one($i)), "gold")
     return $i/node()
```

```
N3 = let $auction := doc("xmark.xml") return
     for $i in $auction/site//item
     where empty($i/payment/text())
     return
            <item id="$i/@id" name="$i/name/text()" location="$i/location/text()">
                 {$i/description, $i/mailbox}
            </item>
```

```
D1 = let $auction := doc("dblp.xml") return
     for $a in $auction/dblp//author
     return
            <AuthorName> {$a/text()} </AuthorName>
```

```
D2 = let $auction := doc("dblp.xml") return
     for $a in $auction/dblp/node()
     return
            <item>{$a/author, $a/title, $a/booktitle, $a/year}</item>
```

Test results about projection sizes are reported in Table 4.1 for what concern XMark documents, while Table 4.1 reports data about tests on queries $D_1$ and $D_2$ on DBLP data.

By analyzing XMark test results we can observe the following.

| **Saxon query execution time (sec) on the standard projection** | | | | | |
|---|---|---|---|---|---|
| Query | 1GB | 2GB | 3GB | 4GB | 5GB |
| $Q_1$ | 3,7 | 5,8 | 7,9 | 11,4 | 12,9 |
| $Q_2$ | 7 | 11,5 | - | - | - |
| $Q_3$ | 7,5 | 12,5 | - | - | - |
| $Q_4$ | 8,3 | 14 | 19,7 | - | - |
| $Q_5$ | 1,5 | 2,2 | 3,1 | 3,9 | 5,2 |
| $Q_6$ | 2,1 | 3,1 | 4,5 | 5,4 | 7,2 |
| $Q_7$ | 4,9 | 7,6 | 11,1 | 14,1 | - |
| $Q_8$ | - | - | - | - | - |
| $Q_9$ | - | - | - | - | - |
| $Q_{10}$ | - | - | - | - | - |
| $Q_{11}$ | - | - | - | - | - |
| $Q_{12}$ | - | - | - | - | - |
| $Q_{13}$ | 4,7 | 6,1 | 8,8 | 10,8 | 14,6 |
| $Q_{14}$ | - | - | - | - | - |
| $Q_{15}$ | 9,5 | - | - | - | - |
| $Q_{16}$ | 10,3 | - | - | - | - |
| $Q_{17}$ | 4,4 | 7,2 | 10 | 13,5 | 17,24 |
| $Q_{18}$ | 2 | 2,9 | 4,4 | 5,22 | 6,5 |
| $Q_{19}$ | 7,3 | 12,1 | 18 | 25,9 | - |
| $Q_{20}$ | 3,5 | 5,7 | 8,3 | 10,5 | 13,9 |
| $N_1$ | - | - | - | - | - |
| $N_2$ | - | - | - | - | - |
| $N_3$ | - | - | - | - | - |

Table 4.3: Saxon performance on projected documents.

- Queries $Q_1$, $Q_5$, $Q_6$, $Q_{13}$, $Q_{17}$, $Q_{18}$ and $Q_{20}$ are very selective, and resulting projection are likely to be processed by main-memory engines.

- Queries $Q_2$, $Q_3$, $Q_4$, $Q_7$, $Q_{19}$ are less selective, and for systems like Saxon the size of the projection is such that it can not be loaded in main-memory.

- For full-text XMark queries $Q_{14}$, $Q_{15}$, $Q_{16}$, we have that the standard projection is not effective, and all projected documents generated for these queries tend to be quite big.

- Concerning our queries $N_1, N_2$ and $N_3$, these require very big parts (nodes and text) of the input document to be evaluated. So projected documents have size that can not be handled even by powerful systems like Qizx [qiz].

The above discussion is focused on projection sizes. In the next sections, we will provide tests precisely illustrating where projection fails for the two engines Saxon [sax] and Qizx [qiz].

| **Qizx query execution time (sec) on the standard projection** | | | | | |
|---|---|---|---|---|---|
| Query | 1GB | 2GB | 3GB | 4GB | 5GB |
| $Q_1$ | 8,1 | 12,8 | 18 | 24,3 | 30,6 |
| $Q_2$ | 13,9 | 23,7 | 35,4 | 48 | 60,4 |
| $Q_3$ | 13,9 | 24,9 | 39,4 | 50,6 | 65,6 |
| $Q_4$ | 14,5 | 38,9 | 38 | 51,4 | 113,6 |
| $Q_5$ | 2,9 | 6,1 | 11,4 | 18,4 | 27,9 |
| $Q_6$ | 3,4 | 7,9 | 15,3 | 25,1 | 39,3 |
| $Q_7$ | 10,5 | 16,5 | 25,8 | 33,8 | 43,3 |
| $Q_8$ | 11,4 | 19,8 | 29,2 | 39 | 48,4 |
| $Q_9$ | 13,9 | 22,9 | 33,4 | 45 | 57,2 |
| $Q_{10}$ | 88,2 | 150,7 | 225,8 | 298 | 374,1 |
| $Q_{11}$ | 32,6 | 65,9 | 117,8 | 178 | 266,2 |
| $Q_{12}$ | 30,8 | 59,3 | 106,1 | 163,5 | 233 |
| $Q_{13}$ | 11,9 | 19,6 | 28,5 | 38 | 48,5 |
| $Q_{14}$ | 126,3 | 229,2 | - | - | - |
| $Q_{15}$ | 48,6 | 84 | 128,7 | 203,4 | 229,4 |
| $Q_{16}$ | 49,8 | 96,9 | 131,8 | 180,7 | 233,1 |
| $Q_{17}$ | 10,5 | 17,4 | 26 | 34,1 | 43,2 |
| $Q_{18}$ | 3,1 | 5 | 7,2 | 9,9 | 11,9 |
| $Q_{19}$ | 13,1 | 22,1 | 37,9 | 46,1 | 57,3 |
| $Q_{20}$ | 7,3 | 11,9 | 17,3 | 27,3 | 29,2 |
| $N_1$ | 275,2 | - | - | - | - |
| $N_2$ | 338,8 | - | - | - | - |
| $N_3$ | 213,5 | - | - | - | - |

Table 4.4: Qizx performance on projected documents.

Concerning DBLP data, we have that for the queries $D_1, D_2$ projections are quite large, making querying impossible when the engine can not rely on large amounts of main-memory. For systems like Saxon even if the allocated main-memory is large, projected files are too big to be processed. We tried with 1GB for the Java Virtual Machine memory, and for both queries projection failed to be processed.

Concerning Qizx, performed tests showed that projection worked for these queries with 512MB for the JVM memory, but since projection takes 35% and 50% of the input document, we strongly suspect that for bigger future versions of the DBLP database projections are likely to exceeds memory capacity of Qizx.

## 4.2 TYPE-BASED PROJECTION FOR UPDATES

As already said, concerning updates the only existing projection technique is the schema-based one proposed in [BBC+11] and extensively studied in Amine Baazizi

Thesis [Baa12] and Marina Sahakyan Thesis [Sah11]. So, even if our proposed approach is schema-less, we discuss here about this schema-based approach.

Schema information is used to perform a type inference operation that starts from the input update and schema yields what is called a type-projector. Essentially, this type-prjector consists of the set of types of nodes the update may need for its evaluation. As we will illustrate next, the notion of type-projector which is adopted is deeply different from that of queries proposed in [BCCN06]. Also, projection is not sufficient for the framework to work since after having updated the projected input we do not have yet global updated document. This is because, in particular, subtrees pruned during projection are missing. This motivated the adoption of a *Merge* operator allowing to merge in streaming the updated projection and the original document, in order to produce the final updated document.

More in detail, for an update $U$ and input document $t$ typed by a DTD $D$ the framework works as follows:

1. a type-projector $\pi$ is inferred from the update $U$ and with respect to the input DTD $D$.
2. a projection $t'$ of $t$ is built using a type-projector $\pi$.
3. the update $U$ is evaluated over the projection $t'$, yielding the partial updating result $U(t')$.
4. an algorithm called *Merge* is used; this algorithm parses in streaming and synchronized fashion both the input $t$ and the partial result $U(t')$ in order to produce the final result $U(t)$. This is done for recovering all nodes pruned out during the projection of $t$.

The main difference between these approaches is that the type-projector proposed in [BCCN06] is composed by one level, while a 3-level components used to build the type-projector proposed in [Baa12, BBC+11, Sah11].

The type projector adopted for queries in [BCCN06] is one-level, while the type-projector proposed in [Baa12, BBC+11, Sah11] is 3-level. The main features of using a 3-level type projector are the following ones. The first one is to optimize (minimize) the size of projections. In particular, the 3-level type projector allows to avoid keeping in the projection useless text nodes that would be kept with the 1-level type projector proposed in [BCCN06]. This feature enables an interesting improvement in case of using documents contain large parts of textual content. The second feature of using the 3-level type projector is that no rewriting of the update is required. The third feature is that this type-projector is specifically designed to deal with particular kinds of update expressions. This is done with the purpose to facilitate the complexity of *Merge* process. The last feature is that this technique is totally independent from XQuery engines.

More in detail, the 3-level type projector $\pi$ proposed in [BBC+11] is composed by the following three components $\{\pi_{no}, \pi_{olb}, \pi_{eb}\}$, where:

- the first component $\pi_{no}$ (node-only) is used to project only the nodes.
- the second component $\pi_{olb}$ (one-level-below) is used to project the nodes plus their children.
- the third component $\pi_{eb}$ (everything-below) is used to project the nodes plus all their descendants.

Next we are going to provide some examples to explain the mechanism of the update 3-level type projection technique. After this we will discuss limitations in terms of scalability.

Consider the following update $u_1$ on the input document $t$ illustrated in Figure 4.4 and the DTD $D$ illustrated in Figure 4.5:

$u_1 =$ `for` $\$x$ `in` $/doc/child :: a$
`where` $\$x/child :: d$ `return delete` $\$x/child :: b$

Suppose that the partial updated document $u_1(t')$ has been produced by updating $t'$ which is the projection of the original document $t$. In order to produce the final result $u_1(t)$, we parse, by using *merge* process, the original document $t$ and the partial updated document $u_1(t')$.

The type-based projector in [BCCN06] assumes that each node (like a,b,c, ...) of the input document $t$ is adorned with an identifier **i** inside square brackets, as illustrated in Figure 4.4. Each node in $t$ has an identifier **i** is next denoted by $t@\mathbf{i}$. The identifier **i** of each node in $t$ carries on information about the node position in $t$, according to document order.

In the projection $t'$ of $t$, the identifier of a projected node is preserved, therefore it may not reflect the new position of the node in $t'$ (it is the case, for instance, of the node $t'@1.4$ in Figure 4.4-(4)). In the partial updated document $u_1(t')$, new identifiers are assigned to inserted or replaced nodes (see next examples).

Now the *Merge* process is presented. This process starts to parse (merging) both $t$ and $u_1(t')$, nothing special happens until the nodes (labeled `a`) $t@1$ and $u_1(t')@1$ are met. Here, the two nodes checked by *Merge* are: the first child node $t/@1.1$ labeled $b$ of $t@1$, and the first child node $u_1(t')@1.4$ labeled $d$ of $u_1(t')@1$. In the examined nodes, the child rank 4 of $u_1(t')@1.4$ is strictly greater than the child rank 1 of $t@1.1$. Also, the label $b$ belongs to the projector $\pi$, indicating that the node $t@1.1$ has been projected in $t'$. Thus, the node $t@1.1$ is not output (it has been deleted by the update $u_1$), the original document $t$ is further parsed.

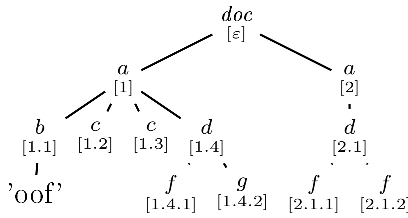
The next two nodes checked are: $t@1.2$ labeled $c$ and $u_1(t')@1.4$ labeled $d$. Once again, the child rank 4 of $u_1(t')@1.4$ is strictly greater than the child rank 2 of $t@1.2$, however this time, the label $c$ does not belong to the projector $\pi$ (the node $t@1.2$ was not needed for the partial update and thus not projected in $t'$) and thus the node $t@1.2$ is output in the final result, the original document $t$ is further parsed.

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

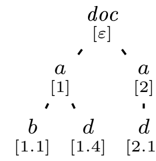
(1) The update $u_1$

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

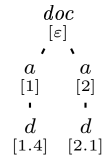
(2) The projector $\pi_1$ for $u_1$



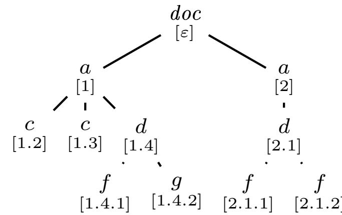

(3) XML document $t$




(4) Projection $t'$ of $t$ wrt $\pi_1$




(5) Partial update $u_1(t')$




(6) Final result $u_1(t)$

Figure 4.4: A simple example with type-based projection.

The process will continue merging $t$ and $u_1(t')$ until both documents are fully parsed. It is worth noting that positions of nodes in the original document play a crucial role in the *Merge* process.

**Dealing with insertion** Consider the following update $u_2$ over the same input document $t$ (see Figure 4.4-(3)) with respect to the same DTD $D$ (see Figure 4.5):

```
u2 = for $x in /doc/child :: a
     return insert as last <e>new<e/> into $x
```

Intuitively, the path corresponding to data needed for the update $u_2$ is $/doc/child :: a$ and the types of nodes traversed by this path are $\pi_2$=$\{doc, a\}$. The projection $\pi_2(t)$ of $t$ as well as the partial update $u_2(\pi_2(t))$ are illustrated in Figure 4.6. Recall that node identifiers in $\pi_2(t)$ correspond to node identifiers in $t$, the same holds for unchanged nodes in $u_2(\pi_2(t))$, and that new (inserted or replaced) nodes in $u_2(\pi_2(t))$ are given new identifiers. In Figure 4.6, **i** and **i**$'$ are new identifiers.

In the following, we will see how the *Merge* process parses both the original document $t$ and the partial update result $u_2(\pi_2(t))$ in order to produce the final result $u_2(t)$. After parsing the root elements of both documents, the current two nodes examined by *Merge* are: $t$@1.1 labeled $b$ and the new node $u_2(\pi_2(t))$@**i** labeled $e$. Here, the new identifier **i** does not carry any information about child rank of the new node and even if the projector tells us that the node $t$@1.1 has been projected

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

Figure 4.5: DTD of the XML document $t$ illustrated in Figure 4.4.

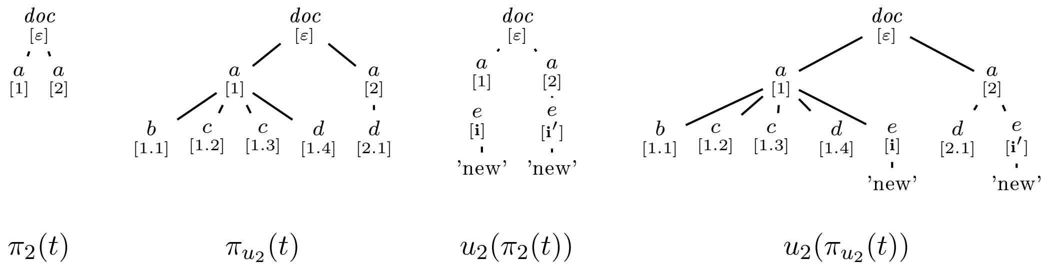



Figure 4.6: Dealing with insertion.

out, there is no way to decide whether it has to be output before the inserted node or after. Recall here the assumption made for *Merge*: information about the update $u_2$ is not available.

In order to solve this problem, related to insertion, we modify the projector, to take into account that for the update $u_2$ the path $/doc/child :: a$ is the target of an insertion. The projector $\pi_{u_2}$ will have 2 components: the type *doc* of category *node-only* and the type $a$ of category *one-level-below*. Applying this new projector to a document proceeds as follows: the nodes labeled by types of category *node-only* are projected; the nodes labeled by types of category *one-level-below* are projected together with each of their children. Descendants of these children are not projected, unless other components of the projector require this projection.

Going back to our example $u_2$, applying the projector $\pi_{u_2}$=$(\pi_{no}, \pi_{olb})$ with $\pi_{no}$=$\{doc\}$ and $\pi_{olb}$=$\{a\}$ to the document $t$ leads to the document $\pi_{u_2}(t)$ described in Figure 4.6 together with the partial update $u_2(\pi_{u_2}(t))$. Since now the new nodes are inserted inside the projection containing all their siblings, it is easy to check that the documents $t$ and $u_2(\pi_{u_2}(t))$ can be merged in a valid, and simple way.

It is worth mentioning that our type projector avoids unnecessary node projection: the projection of all children of a *one-level-below* node is forced, but labels of these children do not take part of the type projector.

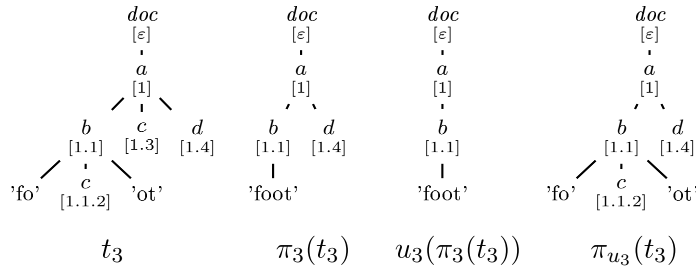



Figure 4.7: Dealing with string and mixed-contents.

**Dealing with String and mixed-content** In order to deal these cases, we will modify the DTD $D$ by redefining the rule for $b$ as <!ELEMENT b (String|c)*> and consider the following update $u_3$:

$u_3 =$ `for` $\$x$ `in` $/doc/child :: a$
`where` $\$x/child :: b/() =' foot'$ `return delete` $\$x/child :: d$

Intuitively, $/doc/child :: a/child :: d$ and $/doc/child :: a/child :: b/text()$ are the paths corresponding to data needed for the update $u_3$. The associated types are $\pi_3$=$\{doc, a, b, String, d\}$. Let us consider the document $t_3$ and its projection $\pi_3(t_3)$ both illustrated in Figure 4.7. Notice that projecting $t_3$ with respect to $\pi_3$ has the side effect to concatenate the two *Strings* `'fo'` and `'ot'` and consequently, the node $u_3(\pi_3(t_3))$@1.4 labeled $d$ is deleted when the update $u_3$ is applied on the projected document $\pi_3(t_3)$. Recall the assumption that *Merge* is not supposed to change the elements parsed in $t_3$ and $u_3(\pi_3(t_3))$, and has only access to the projector. Thus, we cannot expect that merging the original document $t_3$ and the partial updated result $u_3(\pi_3(t_3))$ will produce the final updated document.

The problem here is due to mixed-content nodes and solved by modifying the projector in the same way as for insertion. The new projector $\pi_{u_3}$ generated for the example will have 2 components: $\pi_{no}$=$\{doc, a, d\}$ of category *node-only* and $\pi_{olb}$=$\{b\}$ of category *one-level-below*.

**Dealing with element extraction** Consider the DTD $D$ and the following update $u_4$:

$u_4 =$ `for` $\$x$ `in` $/doc/child :: a$
`return replace` $\$x/child :: b$ `with` $\$x/child :: d$

First, it is clear that `replace` updates have to be treated like insert with respect to the target path $\$x/child :: b$: replace is a delete followed by an insert. Second, because the path $/doc/child :: a/child :: d$ is meant to return the element copied at the target node computed by $/doc/child :: a/child :: b$, the complete subtrees rooted at nodes of type $d$ have to be completely projected. For this update, we propose to generate a projector $\pi_{u_4}$ composed of three sets of types:

- $\pi_{no}$={$doc$} of category (node-only).
- $\pi_{olb}$={$a$} of category (one-level-below).
- $\pi_{everyb}$={$d$} of category (everything-below).

Now we will explain the behavior of the 3-level type projector with respect to the category (everything-below): a node labeled by a type of this category is projected together with its sub-forest. Indeed, applying the projector $\pi_{u_4}$ on the document $t$ of Figure 4.4-(3) produces almost the whole document with the exception of the String `'oof'` which is pruned out.

Actually, the third component of the type projector ensures higher precision and efficiency with respect to [BCCN06]. In particular, it allows avoiding to include the types of the nodes in the subtree of a (everything-below) node in the type projector, and accelerates the projection process it-self.

In Table 4.5, we provide the composition of the 3-level type projector for 20 XQuery updates proposed in Marina SAHAKYAN Thesis [Sah11].

### 4.2.1 Limitations of Update Type-based Projection

Despite the high precision of the 3-level type-projector approach, there are still problems in terms of scalability. As for queries, this is due to the fact that as the input size increases, projection increases as well, and when mechanisms already discussed for queries are present in updates, projection can soon become too large to be processed.

Differently from queries, currently there is no benchmark for updates that is widely recognized by the research community. Fortunately, a rich set of updates has been proposed in Marina Sahakyan Thesis [Sah11]; these updates use XMark documents as inputs, and a part of them has been used in [BBC+11]. These updates are below indicated:

```
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>
```

| Update | $\pi_{no}$ | $\pi_{olb}$ | $\pi_{eb}$ |
|---|---|---|---|
| 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 | open_auction | bidder, increase |

Table 4.5: The composition of 3-level type projector for 20 updates used in [Sah11].

```
       <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
```

```
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
```

Table 4.6 illustrates the dimension of projections (in MB) for each update and for XMark documents whose size ranges from 1GB to 10GB and 15GB.

| **Size of type projected documents in (MB) for 20 different updates** | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Input Size | U1 | U2 | U3 | U4 | U5 | U6 | U7 | U8 | U9 | U10 |
| 1GB | 19.1 | 46.6 | 11.1 | 14 | 69.6 | 36.6 | 43.1 | 4 KB | 311 | 5.2 |
| 2GB | 33 | 80.5 | 19.2 | 24.2 | 120.2 | 63.2 | 74.4 | 4 KB | 535.6 | 9.1 |
| 3GB | 48.1 | 120.8 | 28.9 | 36.4 | 180.3 | 85.1 | 111.9 | 4 KB | 861.5 | 13.7 |
| 4GB | 64.2 | 161.2 | 38.7 | 48.7 | 240.4 | 126.5 | 148.9 | 4 KB | 1.15 GB | 18.3 |
| 5GB | 81.4 | 204.4 | 49.1 | 61.8 | 305 | 160.4 | 188.9 | 4 KB | 1.45 GB | 23.2 |
| 6GB | 96.1 | 241.3 | 58 | 73 | 360.3 | 189.4 | 222.9 | 4 KB | 1.72 GB | 27.4 |
| 7GB | 112.9 | 283.5 | 68.2 | 85.8 | 423.2 | 222.6 | 262 | 4 KB | 2.02 GB | 32.2 |
| 8GB | 128.1 | 321.7 | 77.3 | 97.3 | 480.1 | 252.6 | 297.3 | 4 KB | 2.29 GB | 36.5 |
| 9GB | 144.4 | 362.8 | 87.3 | 109.8 | 541.2 | 284.8 | 335.2 | 4 KB | 2.58 GB | 41.2 |
| 10GB | 163 | 409.7 | 98.6 | 124 | 610.8 | 321.7 | 378.6 | 4 KB | 2.91 GB | 46.5 |
| 15GB | 233.3 | 650.1 | 586.6 | 177.6 | 874.5 | 583.2 | 578.5 | 4 KB | 4.46 GB | 66.8 |
| Input Size | U11 | U12 | U13 | U14 | U15 | U16 | U17 | U18 | U19 | U20 |
| 1GB | 57.2 | 68.7 | 59.2 | 69.3 | 16.1 | 4 KB | 3.1 | 1.2 | 16.1 | 67.3 |
| 2GB | 98.7 | 118.5 | 102.1 | 119.8 | 27.4 | 4 KB | 5.4 | 2.1 | 27.3 | 116.1 |
| 3GB | 148.1 | 177.9 | 161.1 | 179.7 | 45.4 | 4 KB | 8.1 | 3.2 | 45.3 | 174.2 |
| 4GB | 197.7 | 237.3 | 215.2 | 239.6 | 59.5 | 4 KB | 10.8 | 4.3 | 59.3 | 232.6 |
| 5GB | 250.4 | 301 | 272.8 | 304 | 75.8 | 4 KB | 13.7 | 5.4 | 75.5 | 294.5 |
| 6GB | 295.5 | 355.3 | 321.9 | 359 | 89.4 | 4 KB | 16.1 | 6.4 | 89.2 | 347.5 |
| 7GB | 347 | 417.3 | 378.5 | 421.8 | 104.7 | 4 KB | 18.9 | 7.5 | 104.4 | 408 |
| 8GB | 393.5 | 473.2 | 429.4 | 478.4 | 117.2 | 4 KB | 21.5 | 8.4 | 116.8 | 462.7 |
| 9GB | 443.5 | 533.5 | 483.4 | 539.3 | 134 | 4 KB | 24.2 | 9.6 | 133.6 | 521.5 |
| 10GB | 500.7 | 602.4 | 546.4 | 608.7 | 149 | 4 KB | 27.3 | 10.7 | 148.5 | 588.6 |
| 15GB | 716.8 | 861.9 | 780.4 | 871.5 | 226.2 | 4 KB | 39.1 | 15.5 | 225.5 | 883.9 |

Table 4.6: Size reduction by type projection.

From test results about sizes of projections we can observe that used in many cases projection have a relatively small size. However, for systems like Saxon, starting from the 1GB document and for 512MB of main-memory for the JVM, several updates can not be evaluated. Of course if we increase the JVM memory size, problems disappear for the 1GB document, but they re-appear after for bigger files. For Saxon thinks get worst for bigger sizes: for the 5GB document projection allows to execute only 6 out 20 updates (see Table 4.7).

For Qizx thinks are different. However, scalability is still not ensured as it can be seen for the 15GB file: 12 our 20 updates could be executed (see Table 4.7).

| Update | 1GB | 2GB | 3GB | 4GB | 5GB | 6GB | 7GB | 8GB | 9GB | 10GB | 15GB |
|---|---|---|---|---|---|---|---|---|---|---|---|
| U1 | | | | | | | | | | | |
| Saxon | 7.671 | 13.125 | 31.594 | - | - | - | - | - | - | - | - |
| Qizx | 5.988 | 10.345 | 14.955 | 20.119 | 25.340 | 29.401 | 34.454 | 38.072 | 42.009 | 47.176 | 59.665 |
| U2 | | | | | | | | | | | |
| Saxon | 21.604 | - | - | - | - | - | - | - | - | - | - |
| Qizx | 45.356 | 84.15 | 93.026 | 120.582 | 151.153 | - | - | - | - | - | - |
| U3 | | | | | | | | | | | |
| Saxon | 5.306 | 8.708 | 11.555 | 14.419 | - | - | - | - | - | - | - |
| Qizx | 12.146 | 20.422 | 23.925 | 31.028 | 38.522 | 44.336 | 52.067 | 58.042 | 70.074 | 78.367 | - |
| U4 | | | | | | | | | | | |
| Saxon | 7.294 | 12.215 | 29.801 | - | - | - | - | - | - | - | - |
| Qizx | 13.781 | 20.744 | 26.778 | 34.861 | 44.135 | 52.545 | 60.968 | 67.108 | 78.656 | 89.855 | 99.554 |
| U5 | | | | | | | | | | | |
| Saxon | - | - | - | - | - | - | - | - | - | - | - |
| Qizx | 68.363 | 108.233 | 119.798 | 156.766 | 197.669 | 225.574 | 275.608 | 320.105 | 367.504 | 416.487 | - |
| U6 | | | | | | | | | | | |
| Saxon | 16.196 | - | - | - | - | - | - | - | - | - | - |
| Qizx | 45.768 | 65.636 | 78.783 | 102.380 | 129.314 | - | - | - | - | - | - |
| U7 | | | | | | | | | | | |
| Saxon | 40.116 | - | - | - | - | - | - | - | - | - | - |
| Qizx | 86.084 | 197.421 | 324.657 | 523.130 | 823.594 | 1139.02 | - | - | - | - | - |
| U8 | | | | | | | | | | | |
| Saxon | 0.289 | 0.266 | 0.266 | 0.266 | 0.266 | 0.266 | 0.266 | 0.266 | 0.266 | 0.266 | 0.266 |
| Qizx | 0.54 | 0.54 | 0.54 | 0.54 | 0.54 | 0.54 | 0.54 | 0.54 | 0.54 | 0.54 | 0.54 |
| U9 | | | | | | | | | | | |
| Saxon | - | - | - | - | - | - | - | - | - | - | - |
| Qizx | 226.217 | - | - | - | - | - | - | - | - | - | - |
| U10 | | | | | | | | | | | |
| Saxon | 235.344 | 725.794 | - | - | - | - | - | - | - | - | - |
| Qizx | 4.123 | 7.155 | 12.230 | 14.051 | 17.545 | 20.112 | 22.620 | 24.711 | 26.460 | 31.196 | - |
| Update | 1GB | 2GB | 3GB | 4GB | 5GB | 6GB | 7GB | 8GB | 9GB | 10GB | 15GB |
| U11 | | | | | | | | | | | |
| Saxon | - | - | - | - | - | - | - | - | - | - | - |
| Qizx | 60.327 | 111.701 | 121.640 | 144.387 | 188.488 | 219.966 | - | - | - | - | - |
| U12 | | | | | | | | | | | |
| Saxon | - | - | - | - | - | - | - | - | - | - | - |
| Qizx | 62.832 | 103.504 | 114.388 | 130.234 | 169.113 | 191.240 | 243.874 | 272.829 | 297.499 | - | - |
| U13 | | | | | | | | | | | |
| Saxon | 8.849 | 14.267 | - | - | - | - | - | - | - | - | - |
| Qizx | 31.476 | 53.16 | 83.797 | 106.396 | 138.138 | 185.425 | 204.588 | 236.866 | 895.262 | - | - |
| U14 | | | | | | | | | | | |
| Saxon | - | - | - | - | - | - | - | - | - | - | - |
| Qizx | 60.584 | 77.854 | 108.855 | 141.928 | 187.238 | 213.861 | 254.537 | 297.178 | 343.826 | - | - |
| U15 | | | | | | | | | | | |
| Saxon | 1.985 | 3.038 | 5.789 | 6.967 | 8.210 | 9.751 | 10.224 | 12.184 | 13.349 | 14.709 | - |
| Qizx | 8.937 | 15.317 | 20.692 | 25.828 | 31.911 | 39.015 | 45.543 | 51.416 | 60.944 | 65.165 | 76.25 |
| U16 | | | | | | | | | | | |
| Saxon | 0.264 | 0.264 | 0.264 | 0.264 | 0.264 | 0.264 | 0.264 | 0.264 | 0.264 | 0.264 | 0.264 |
| Qizx | 0.158 | 0.158 | 0.158 | 0.158 | 0.158 | 0.158 | 0.158 | 0.158 | 0.158 | 0.158 | 0.158 |
| U17 | | | | | | | | | | | |
| Saxon | 1.246 | 1.92 | 2.484 | 2.89 | 3.02 | 3.30 | 3.96 | 5.57 | 5.9 | 6.29 | 7.15 |
| Qizx | 1.607 | 3.188 | 5.665 | 6.967 | 7.682 | 8.617 | 9.552 | 10.590 | 11.384 | 12.489 | 13.22 |
| U18 | | | | | | | | | | | |
| Saxon | 0.522 | 0.751 | 4.184 | 4.89 | 5.902 | 6.01 | 6.85 | 7.65 | 7.70 | 8.5 | 9.34 |
| Qizx | 1.094 | 2.452 | 4.755 | 5.067 | 6.182 | 6.857 | 9.552 | 10.590 | 11.384 | 12.489 | 13.552 |
| U19 | | | | | | | | | | | |
| Saxon | 1.752 | 2.725 | 3.775 | 4.781 | 6.803 | 8.79 | 9.874 | 10.753 | 11.852 | 12.421 | - |
| Qizx | 7.183 | 12.26 | 18.013 | 22.143 | 26.669 | 30.883 | 35.554 | 39.566 | 44.272 | 48.684 | 52.45 |
| U20 | | | | | | | | | | | |
| Saxon | - | - | - | - | - | - | - | - | - | - | - |
| Qizx | 67.878 | 113.731 | 129.587 | 173.089 | 222.251 | 287.289 | 332.112 | 376.297 | 437.769 | 483.709 | - |

Table 4.7: Qizx and Saxon performances for type-based projected documents.

## 4.3 CONCLUSION

In this chapter, we introduce the XML projection technique, which is one of the most important technique used for reducing the memory consumption. Also, we present two mains approaches proposed for XML projection technique for queries and updates. As illustrated in this chapter, these techniques still fail in several cases, for which the projected document is still quite big to be loaded in main memory.

As we have seen, limitations are sensible to the kind of used engine, and to the amount of available main-memory allocated for the JVM. In the next chapters, we propose our technique to solve overcome such limitations, we will choose a relatively small size for the main-memory (512MB) to show that the approach behaves well in this context, by allowing querying and updating documents of arbitrary sizes under some conditions met by the query/update expression.

CHAPTER 5

# Partitioning and Projecting XML Documents

## Contents


**5.1 PRELIMINARIES** . . . . . . . . . . **54**
5.1.1 Data Model . . . . . . . . . . 54
5.1.2 Query Language . . . . . . . . . . 56
**5.2 PATH EXTRACTION** . . . . . . . . . . **58**
**5.3 ITERATIVE QUERIES AND PARTITIONING PATHS** . . . . . . . . . . **61**
**5.4 PROJECTION** . . . . . . . . . . **63**
**5.5 THE PARTITIONING ALGORITHM** . . . . . . . . . . **70**
5.5.1 The Algorithm . . . . . . . . . . 72
5.5.2 Dealing with a Workload . . . . . . . . . . 75
**5.6 STREAMING IMPLEMENTATION** . . . . . . . . . . **78**
**5.7 EXPERIMENTAL EVALUATION** . . . . . . . . . . **88**
5.7.1 Experimental Setup . . . . . . . . . . 89
5.7.2 Tests Results . . . . . . . . . . 89
5.7.3 Experiments . . . . . . . . . . 90
5.7.4 Experiments on Queries $\{N_1, N_2, N_3\}$, and $\{D_1, D_2\}$ . . . . 94
5.7.5 Summing Up . . . . . . . . . . 98
**5.8 CONCLUSION** . . . . . . . . . . **99**


THis chapter includes three main parts. The first part (Section 5.1, 5.2, 5.3) presents our static analysis technique used to characterize *iterative* queries, for which XML data can be partitioned for query evaluation. The second part (Section 5.5) presents our partitioning algorithm. First an high level specification is formalized by relying on a DOM-based representation of input trees. Then a SAX based version of the partitioning algorithm is provided. As said in the introduction, to accentuate benefits of our strategy, projection is used while partitioning. The third part (Section 5.6, 5.7) discusses about the implementation of the SAX-based algorithms, and presents test results obtained from experiments we conducted by using two main XQuery engines. Finally, we draw our conclusion in Section 5.8.

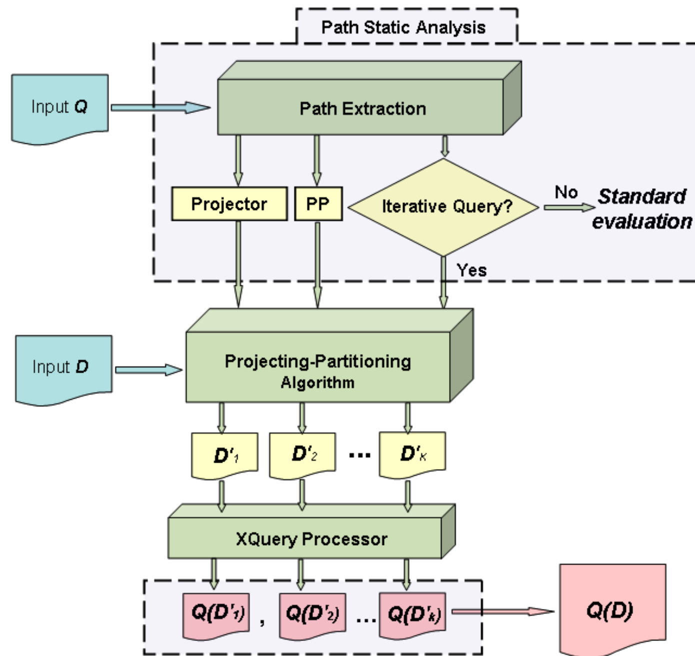



Figure 5.1: Projecting-partitioning scenario for an input document $D$ and a given query $Q$ and partitioning path $PP$.

## 5.1 PRELIMINARIES

### 5.1.1 Data Model

Following [BC09], we represent XML data by means of a store $\sigma$, which associates to each node location (or identifier) l either an element node or a text node. For simplicity we disregard attributes in the formal treatment, while they are considered in the implementation.

When l is an element node, we have $\sigma(\mathrm{l})=\mathrm{a}[\mathtt{L}]$ (also written $\mathrm{l} \leftarrow \mathrm{a}[\mathtt{L}]\in\sigma$) where a is the element tag and $\mathtt{L}=(\mathrm{l}_1, \ldots, \mathrm{l}_n)$ is the ordered sequence of the child locations for l. When l is a text node, we have $\sigma(\mathrm{l})=\mathit{text}[\mathrm{s}]$ (also written $\mathrm{l} \leftarrow \mathit{text}[\mathrm{s}]\in\sigma$) where the string s is the textual content of the l node.

An XML tree is a pair $t=(\sigma, \mathrm{l}_t)$, where $\mathrm{l}_t$ is the root location of the tree. We denote by $\mathrm{dom}(\sigma)$ the set of locations of a store (analogously $\mathrm{dom}(t)$ for a tree). Given a location $\mathrm{l}\in\mathrm{dom}(\sigma)$, $\sigma@\mathrm{l}$ denotes the subtree of $\sigma$ rooted at l. Sometimes, for simplicity, when $t=(\sigma, \mathrm{l}_t)$, we abusively use $t$ instead of $\sigma$ and, for instance, we write $\mathrm{l} \leftarrow \mathrm{a}[\mathtt{L}]\in t$ instead of $\mathrm{l} \leftarrow \mathrm{a}[\mathtt{L}]\in\sigma$, and similarly for an association of the form $\mathrm{l} \leftarrow \mathit{text}[\mathrm{s}]$.

In the following, we provide formal definitions of $\sigma$ and its components.

**Definition 5.1.1 (Location Sequence $\mathtt{L}$)** *A location sequence* $\mathtt{L}$ *is defined by the following grammar:*

$$\mathtt{L} ::= () \mid \mathrm{l} \mid \mathtt{L},\mathtt{L}$$

*where* $()$ *is the empty sequence,* $\mathrm{l}$ *is a single location, and* $\mathtt{L},\mathtt{L}$ *denotes the concatenation of location sequences.*

**Definition 5.1.2 (XML Store $\sigma$)** *A store* $\sigma$ *is a finite mapping*

$$\sigma = \{\mathrm{l}_1 \leftarrow \alpha_1, \mathrm{l}_2 \leftarrow \alpha_2, \cdots, \mathrm{l}_\mathrm{n} \leftarrow \alpha_\mathrm{n}\}$$

*each* $\alpha_\mathrm{i}$ *can be either a text value* $text[\mathrm{s}]$ *where* $\mathrm{s}$ *is a string value referred to the textual content of the node* $\mathrm{l}$*; or a an element* $\mathrm{a}[\mathtt{L}]$ *where* $\mathtt{L}$ *is a location sequence (see Definition 5.1.1).*

We use $\{\mathtt{L}\}$ to denote the set of locations in the sequence $\mathtt{L}$. Also, We say that $\mathtt{L}'$ is a projection of $\mathtt{L}$, denoted by $\mathtt{L}' \preceq \mathtt{L}$, if $\mathtt{L}'$ is obtained from $\mathtt{L}$, by erasing some of its locations. Note that sequence projection preserves ordering.

For instance $\mathrm{l}_1, \mathrm{l}_3 \preceq \mathrm{l}_1, \mathrm{l}_2, \mathrm{l}_3$, while $\mathrm{l}_3, \mathrm{l}_1 \not\preceq \mathrm{l}_1, \mathrm{l}_2, \mathrm{l}_3$ (ordering is not preserved).

In order to define XML partition, we need the following notion of XML projection.

**Definition 5.1.3 (XML Projection)** *A tree* $t'{=}(\sigma', \mathrm{l}_{t'})$ *is a* projection *of a tree* $t{=}(\sigma, \mathrm{l}_t)$*, noted as* $t' \preceq t$*, if* $\mathrm{l}_{t'}{=}\mathrm{l}_t$*, and for each location* $\mathrm{l}{\in}\mathrm{dom}(\sigma')$*:*

$$\mathrm{l} \leftarrow \mathrm{a}[\mathtt{L}']{\in}\sigma' \text{ implies } (\exists \mathtt{L}.\ \mathrm{l} \leftarrow \mathrm{a}[\mathtt{L}]{\in}\sigma \text{ and } \mathtt{L}' \preceq \mathtt{L})$$

Note that projection preserves tree roots, and it is used to define XML partition. Figure 5.2 shows a simple XML tree, its associated store, and a possible projection. In this figure, we have that the root location is $\mathrm{l}_t{=}\mathrm{l}_1$, and the set of locations in the projection $\sigma'$ is $\mathrm{dom}(\sigma'){=}\{\mathrm{l}_1, \mathrm{l}_2, \mathrm{l}_3, \mathrm{l}_5\}$, and $\mathrm{dom}(\sigma'){\subseteq}\mathrm{dom}(\sigma)$ where $\mathrm{dom}(\sigma){=}\{\mathrm{l}_1, \mathrm{l}_2, \mathrm{l}_3, \mathrm{l}_4, \mathrm{l}_5, \mathrm{l}_6\}$.

**Definition 5.1.4 (XML Partition)** *A collection of trees* $\{t_1, \ldots, t_\kappa\}$ *is a* partition *of a tree* $t$ *if, for each* $i{=}1 \ldots \kappa$*,* $t_i \preceq t$ *, and if for each location* $\mathrm{l}{\in}dom(t)$*, we have:*

$$\begin{array}{lll} \mathrm{l} \leftarrow text[\mathrm{s}]{\in}t & \text{implies} & \exists\ t_i.\ \mathrm{l} \leftarrow text[\mathrm{s}]{\in}t_i \quad \text{or} \\ \mathrm{l} \leftarrow \mathrm{a}[\mathtt{L}]{\in}t & \text{implies} & \{\mathtt{L}\}{=}\bigcup_{\mathrm{l} \leftarrow \mathrm{a}[\mathtt{L}_i]\in t_i} \{\mathtt{L}_\mathrm{i}\} \end{array}$$

A tree $t_i$ of the partition is called a *part*. The two above properties say that each text node has to belong to at least one part, and that element nodes are partitioned in such a way that no child is left out.

Figure 5.3 contains two possible partitions of the document in Figure 5.2. As a document can be partitioned in multiple ways, it is crucial to carefully design the partitioning strategy, so that the query result equals to the concatenation of query results on each part of the partition. We will see next how to choose the right partition in terms of a path analysis on the query.

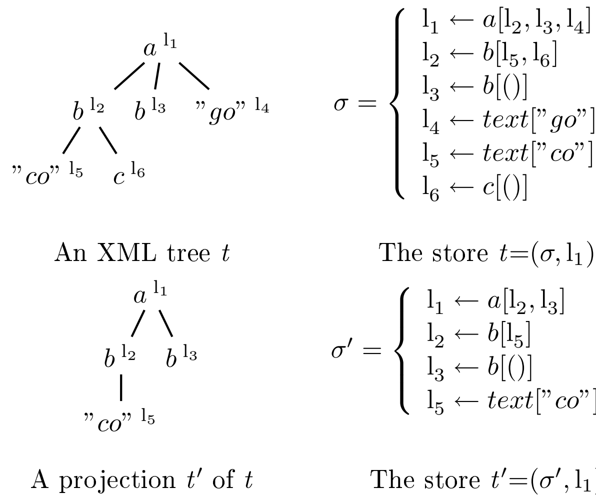



Figure 5.2: Representation of XML trees as stores and projection.

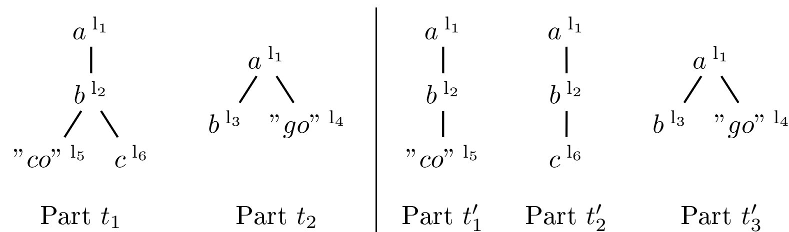



Figure 5.3: Two possible partitions of the XML tree $t$ of Figure 5.2.

### 5.1.2 Query Language

In this approach, we use the fragment of XQuery described by the grammar illustrated in Figure 5.4. This fragment comprises (`for`, `let` and `return`) clauses as well as (`if-then-else`) statements, and allows the user to specify *self*, *child*, and *descendant-or-self* XPath axes [BBC+10] (for simplicity, we will write *dos* instead of *descendant-or-self*). The grammar uses a for tag symbols.

In the grammar illustrated in Figure 5.4: () refers to the empty sequence; *Expr* is an XQuery expression; $Q_1, Q_2$ denotes the query concatenation; a[$Q$] denotes an element node with a label "a", where the content of this node is a query $Q$.

We say that a query $Q$ is *well-formed* if and only if **i)** it does not contain free variables (i.e., variables with no corresponding `for/let` binders), **ii)** no variable name is used twice in `for/let` bindings, and **iii)** it starts navigating the document by means of non-*self* step.

Condition **(i)** ensures that well-formed queries start navigating documents from their root element. For instance, the query `for` $y$ `in` $x/Step$ `return` $Q$ is not well-formed because it starts the navigation from a variable $x$ which does not represent the root element, while the query `for` $y$ `in` $/Step$ `return` $Q$ is well-formed.

| | | | | |
|---|---|---|---|---|
| Query | $Q$ | $::=$ | $()$ | {empty sequence} |
| | | $\mid$ | $Expr$ | {XQuery expression} |
| | | $\mid$ | $\mathrm{a}[Q]$ | {element node labeled by a} |
| | | $\mid$ | $Q_1, Q_2$ | {concatenation} |
| | | $\mid$ | $\texttt{if}\ (Q)\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2$ | {conditional expression} |
| | | $\mid$ | $\texttt{for}\ x\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2$ | {iteration} |
| | | $\mid$ | $\texttt{let}\ x := Q_1\ \texttt{return}\ Q_2$ | {let-binding} |
| XQuery Expression | $Expr$ | $::=$ | $x \mid x/Step \mid /Step$ | |
| XPath Step | $Step$ | $::=$ | $Axis :: NT$ | |
| XPath Axis | $Axis$ | $::=$ | $self \mid child \mid dos$ | |
| Node Test | $NT$ | $::=$ | $\mathrm{a} \mid node() \mid text()$ | |

Figure 5.4: Query language grammar.

The restriction **(ii)** simplifies the analysis, and can be always obtained by renaming. Condition **(iii)** excludes queries like $\texttt{for}\ y\ \texttt{in}\ /self :: NT\ \texttt{return}\ Q$, it is assumed to simplify the formalization, and is non restrictive, as in most practical cases queries start the navigation by means of either *child* or *dos* axis.

In this approach, we focus on queries issued on a single document. Indeed, multiple document queries are likely to be not iterative, and their treatment goes far beyond the scope of this approach. Also, we focus on `for/let` expressions using element construction only on the right-hand side expression $Q_2$, as happens in most practical cases. For instance all XMark queries are of this form, provided that in some queries `let` bindings are inlined. Inlining consists of replacing each *use* of let-variables with the query they are bound to. For instance,

$$\texttt{let}\ x := b[/child :: a]\ \texttt{return}\ res[x, /dos :: d]$$

is rewritten into $res[b[/child :: a], /dos :: d]$. Of course, this rewriting preserves query semantics.

The evaluation of a query $Q$ on an input tree $t{=}(\sigma, \mathrm{l}_t)$, denoted by $Q(t)$, yields a pair $(\sigma_Q, \mathtt{L}_Q)$, where the store $\sigma_Q$ is a forest which extends the initial store $\sigma$ with the new elements built by $Q$, while $\mathtt{L}_Q$ is the sequence of location nodes returned by the query whose contents is described in $\sigma_Q$. In order to present a formal semantics of this XQuery fragment, a concise and elegant formalization can be found in [BC09].

In order to define equivalence among query results, we also need the following notions. Equivalence among two trees, denoted by $t \cong t'$, holds if and only if the two rooted trees are isomorphic (they possibly differ only in terms of name of locations). When $\sigma$ and $\sigma'$ are forests and $\mathtt{L}{=}(\mathrm{l}_1, \ldots, \mathrm{l}_n)$ and $\mathtt{L}'{=}(\mathrm{l}'_1, \ldots, \mathrm{l}'_n)$ are sequences of locations, we write $(\sigma, \mathtt{L}) \cong (\sigma', \mathtt{L}')$ to state that, for $i{=}1..n$, we have $\sigma@\mathrm{l}_i \cong \sigma'@\mathrm{l}'_i$.

| | | | |
|---|---|---|---|
| 1. | $E((), \Gamma, m)$ | $=$ | $()$ |
| 2. | $E((Q_1, Q_2), \Gamma, m)$ | $=$ | $E(Q_1, \Gamma, m) \cup E(Q_2, \Gamma, m)$ |
| 3. | $E(\mathrm{a}[Q], \Gamma, m)$ | $=$ | $\{P\{for\ y\} \mid P\{for\ y\} \in \Gamma\} \cup E(Q, \Gamma, 1)$ |
| 4. | $E(x, \Gamma, 0)$ | $=$ | $\{P\{for\ x\} \mid P\{for\ x\} \in \Gamma\}$ |
| 5. | $E(x, \Gamma, 1)$ | $=$ | $\{P\{for\ x\}/dos :: node() \mid P\{for\ x\} \in \Gamma\}$ |
| 6. | $E(/P, \Gamma, 0)$ | $=$ | $\{/P\}$ |
| 7. | $E(/P, \Gamma, 1)$ | $=$ | $\{/P/dos :: node()\}$ |
| 8. | $E(x/P, \Gamma, 0)$ | $=$ | $\{P'\{for\ x\}/P \mid P'\{for\ x\} \in \Gamma\}$ |
| 9. | $E(x/P, \Gamma, 1)$ | $=$ | $\{P'\{for\ x\}/P/dos :: node() \mid P'\{for\ x\} \in \Gamma\}$ |
| 10. | $E(\texttt{if}\ Q\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2, \Gamma, m)$ | $=$ | $E(Q, \Gamma, 0) \cup E(Q_1, \Gamma, 1) \cup E(Q_2, \Gamma, 1)$ |
| 11. | $E(\texttt{for}\ x\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2, \Gamma, m)$ | $=$ | $\Gamma' \cup E(Q_2, \Gamma \cup \Gamma', m)$ <br> $where\ \Gamma' = \{P\{for\ x\} \mid P \in E(Q_1, \Gamma, 0)\}$ |
| 12. | $E(\texttt{let}\ x := Q_1\ \texttt{return}\ Q_2, \Gamma, m)$ | $=$ | $\Gamma' \cup E(Q_2, \Gamma \cup \Gamma', m)$ <br> $where\ \Gamma' = E(Q_1, \Gamma, 0)$ |
| 13. | $E(P/@attr :: \mathrm{a}, \Gamma, m)$ | $=$ | $E(P, \Gamma, m)$ |

Figure 5.5: Path extraction function.

Finally, when $\sigma$ and $\sigma'$ have disjoint domains (no common location), we define the concatenation $(\sigma, \mathtt{L}) \cdot (\sigma', \mathtt{L}')$ as the pair $(\sigma \cup \sigma', (\mathtt{L}, \mathtt{L}'))$, where $\mathtt{L}, \mathtt{L}'$ denotes the concatenation of $\mathtt{L}$ and $\mathtt{L}'$.

## 5.2 PATH EXTRACTION

In our approach, paths are used for characterizing iterative queries, and for partitioning and projecting an input document. Paths are extracted from a query by using the path extraction function $E()$ of Figure 5.5; this function resembles that proposed in [BCCN06, MS03]. However, paths extracted according to $E()$ carry a richer information, as they also describe the relation with `for`-variables. Paths obey the following grammar:

$$P \ ::= \ \varepsilon \mid /S \mid P/S \qquad\qquad S \ ::= \ Step \mid Step\{for\ x\}$$

where $\varepsilon$ denotes the empty path.

For instance, when a path $P'\{for\ x\}/P''$ has been extracted from $Q$, it captures that a subquery of $Q$ has the shape `for` $x$ `in` $Q_1$ `return` $Q_2$ and (i) $P'$ is extracted from $Q_1$ and selects possible bindings for $x$ while (ii) $P''$ has been extracted from $Q_2$ in the context of the previous bindings or, in other words, $x/P''$ is extracted from $Q_1$.

Variable information in paths is important to characterize iterative queries and to identify partitioning paths (see Section 5.3), while it will be ignored for the purpose of partitioning.

Our path extraction function $E()$ is defined in Figure 5.5 by structural induction on queries defined in Figure 5.4. This Function has the form $E(Q, \Gamma, m)$. The first parameter is the query at issue. The second parameter is the environment $\Gamma$ that keeps track of bindings of the form $\{for\ x\}$ or $\{let\ x\}$ between query variables and their corresponding paths. We use $\Gamma$ because we always need to remember the set of paths corresponding to given variables in queries of the form `for` $x$ `in` $Q_1$ `return` $Q_2$ or `let` $x := Q_1$ `return` $Q_2$.

The third parameter used in $E()$ rules is a boolean flag $m$ to distinguish between subqueries that generate fragments of the result of the outer query ($m$=1) and subqueries that are only used for binding variables or filtering results ($m$=0). When ($m$=1), the terminal rules 5, 7 and 9 extend extracted paths with a $dos :: node()$ step, so to capture all the nodes required by the query to build the result.

For queries of the form `for` $x$ `in` $Q_1$ `return` $Q_2$ (rule 11 in Figure 5.5), the function $E()$ first extracts paths from $Q_1$; these paths are, then, enriched with information about variable bindings and added to the environment $\Gamma$, which is used for the recursive extraction of paths from $Q_2$. In particular $\Gamma$ is used to associate the right path to each free occurrence of the variable $x$ in $Q_2$ (rules 4 and 5 in Figure 5.5). Rules of `let` expressions are similar, with the exception that they do not keep track of information about let-variables (rule 12 in Figure 5.5). Information about let-variables is not needed because we are only interested in information telling us that there is an iteration performed by the query. Only for-variables are needed to this end.

**Example 1** Consider the following query $Q$:

Q = `for` $\$x$ `in` $/child :: a/child :: b$
`return if` $(\$x/child :: c)$ `then` $\$x/text()$ `else` $()$

This query is from the form (`for` $x$ `in` $Q_1$ `return` $Q_2$) where:

$$\begin{aligned} Q_1 &= /child :: a/child :: b \\ Q_2 &= \text{if } (\$x/child :: c) \text{ then } \$x/text() \text{ else } () \end{aligned}$$

By using our extraction function $E(Q)$ defined in Figure 5.5, we have that:

$$\begin{aligned}
&Rules\ .6.8 && E(Q_1, \Gamma, 1) = \{/child :: a,\ /child :: a/child :: b\{for\ x\}\} \\
&Rule\ .10 && E(Q_2, \Gamma, m) = E(\$x/child :: c, \Gamma, 0) \cup E(\$x/text(), \Gamma, 1) \cup E((), \Gamma, 1) \\
& && = \{/child :: a/child :: b\{for\ x\}/child :: c/dos :: node()\} \cup \\
& && \quad \{/child :: a/child :: b\{for\ x\}/child :: text()\} \cup () \\
& && = \{/child :: a,\ /child :: a/child :: b\{for\ x\}, \\
& && \quad /child :: a/child :: b\{for\ x\}/child :: c/dos :: node()\} \\
&Rule\ .11 && E(Q, \Gamma, m) = \{/child :: a,\ /child :: a/child :: b\{for\ x\}, \\
& && \quad /child :: a/child :: b\{for\ x\}/child :: c/dos :: node()\}
\end{aligned}$$

So the final set of extracted paths of this query is $\{P_1,\ P_2,\ P_3\}$, with

$$\begin{aligned}
P_1 &= /child :: a \\
P_2 &= /child :: a/child :: b\{for\ x\} \\
P_3 &= /child :: a/child :: b\{for\ x\}/child :: c/dos :: node()
\end{aligned}$$

**Example 2** Consider the following query $Q$:

Q = `for` $\$x$ `in` $/child :: a$
`for` $\$y$ `in` $\$x/child :: b$
`return` $(\$y/child :: d\ ,\ \$y/child :: e)$

The set of extracted paths of this query is $\{P_1,\ P_2,\ P_3,\ P_4\}$, with

$$\begin{aligned}
P_1 &= /child :: a\{for\ x\} \\
P_2 &= /child :: a\{for\ x\}/child :: b\{for\ y\} \\
P_3 &= /child :: a\{for\ x\}/child :: b\{for\ y\}/child :: d/dos :: node() \\
P_4 &= /child :: a\{for\ x\}/child :: b\{for\ y\}/child :: e/dos :: node()
\end{aligned}$$

□

Paths extracted from a query express properties of the query data needs. In Examples 1, 2 we have that all nodes that are either selected by the paths or traversed in order to reach a node selected by a path, form a sound projection for both query examples. By sound projection we mean a projection of the input tree that preserves query results.

We will see later that these projections can be obtained quite efficiently by opportunely matching extracted paths against nodes of the input documents, visited in a streaming fashion by means of a SAX parser.

In our work we assume the following. For queries of the form `for` $x$ `in` $Q_1$ `return` $Q_2$, and similarly in case of `let` expression, we suppose that the subquery $Q_1$ does not use concatenation. For example, the following query is not allowed:

Q = `for` $\$x$ `in` $(/child :: a/child :: b, /child :: a/child :: b)$ `return` $\$x$

We omitted this case from our study because we have two identical paths extracted from the query $Q$ (see below $E(Q)$), associated with the same binding variable $x$ and coming from different subexpressions. This could make formalizations quite cumbersome, as information about the provenance of extracted path should be gathered during extraction.

$$E(Q) = \{/child :: a/child :: b\{for\ x\},\ /child :: a/child :: c\{for\ x\}\}$$

Actually, for the purpose of partitioning (and projection) variable information in extracted paths is not needed. Partitioning (and projection) will use extracted

paths once variable information has been eliminated. For instance, rather than $(/child :: a\{for\ x\}/child :: b\{for\ y\})$ the path $(/child :: a/child :: b)$ is used.

Variable bindings are erased by means of the function $ErVar(P)$ (illustrated in Definition 5.2.1) which indicates the path obtained from $P$ by removing $\{for\ -\}$ occurrences. Hereafter, for simplicity, we will often abbreviate $E(Q, \emptyset, 1)$ with $E(Q)$.

**Definition 5.2.1** ($ErVar(P)$) *Given a well-formed query $Q$ and its set of extracted paths $P \in E(Q)$, the function $ErVar(P)$ removes all $\{for\ -\}$ occurrences in $P$ if they exist. By induction on the structure of $P$, the syntax of $ErVar(P)$ is defined as follows:*

$$\begin{array}{lcl} ErVar(\varepsilon) & = & \varepsilon \\ ErVar(/Step/P) & = & /Step/ErVar(P) \\ ErVar(/Step\{for\ x\}/P) & = & /Step/ErVar(P) \end{array}$$

*where $\varepsilon$ denotes the empty path (we assume $/P/\varepsilon = /P$).*

## 5.3 ITERATIVE QUERIES AND PARTITIONING PATHS

Our approach is based on the idea of partitioning an input document $t$ into a collection of documents $\{t_1, \ldots, t_\kappa\}$ and projecting each $t_i$ according to $Q$, so that $Q(t) \cong (Q(t'_1), \ldots, Q(t'_\kappa))$, where $t'_i$ is the projection of $t_i$. The input document is partitioned according to a partitioning path $P$, which is opportunely chosen among the paths extracted from $Q$. Indeed, paths extracted from $Q$ are also used to project each partition $t_i$. In order to guarantee the correctness of query evaluation, this approach can be applied only when $Q$ first selects a sequence of nodes $S$, and then iterates over the nodes in $S$ by exploring their corresponding subtrees. Queries satisfying this requirement are called *iterative* and are quite common in practice. The query of Example 1 is iterative. It selects the sequence $S$ of nodes specified by the subquery $/child :: a/child :: b$. Then for each node in $S$, it evaluates the if-sub query. As a concrete example, 13 out of the 20 XMark queries are iterative: namely, queries from $Q_1$ to $Q_6$, and $Q_{14}$ to $Q_{20}$ are iterative. These queries are given in Section A.1 of Appendix A.

For an iterative query over a document $t$, there may be more than one path that could be used for partitioning $t$. We first characterize this set of candidate partitioning paths and then show how to pick the best one. In the definition below, we say that the path $P \in E(Q)$ is *maximal* if no other path in $E(Q)$ contains $P$ as a prefix.

**Definition 5.3.1 (Candidate Partitioning Paths)** *Given a* well-formed *query $Q$, a candidate partitioning path for $Q$ is a path $ErVar(P)$ with $P \in E(Q)$ such that:*

*(i) $P$ is of the form $P_0\{for\ x\}$.*

*(ii) $P$ does not use text node test.*

*(iii) for each maximal path $P' \in E(Q)$, $P' = P/P''$.*

*The set of all candidate partitioning paths for $Q$, is hereafter denoted by $Candidate(Q)$.*

Condition (i) states that each candidate path is used for iterating inside the query $Q$. Condition (ii) rules out candidate paths that would iterate on text nodes (like in the query `for` $x$ `in` $/dos :: text()$ `return` $Q'$) because we want to ensure that partitioning is performed on a sequence of element nodes rather than a sequence of text nodes. The technical reason is that projection of text nodes which are sibling produces a text node (the concatenation of the text nodes) rather than a sequence of text nodes. Although this restriction can be relaxed, we give priority to presenting the core of the partitioning method here. Condition (iii) is the most important one: the restriction on maximal paths is needed since otherwise the minimal common prefix of $E(Q)$ paths would be a candidate.

As an example, for the query and extracted paths in Example 1, we have that $ErVar(P_1)$=$/child :: a$ and $ErVar(P_2)$=$/child :: a/child :: b$ are candidate paths.

As another example, for the query and extracted paths presented in Example 2, the $ErVar(P_1)$ and $ErVar(P_2)$ are candidate paths, while $ErVar(P_3)$ is not a candidate, as the prefix relation does not hold with respect to the path $P_4$. Figure 5.6 illustrates the process of finding the candidate paths of Example 2.

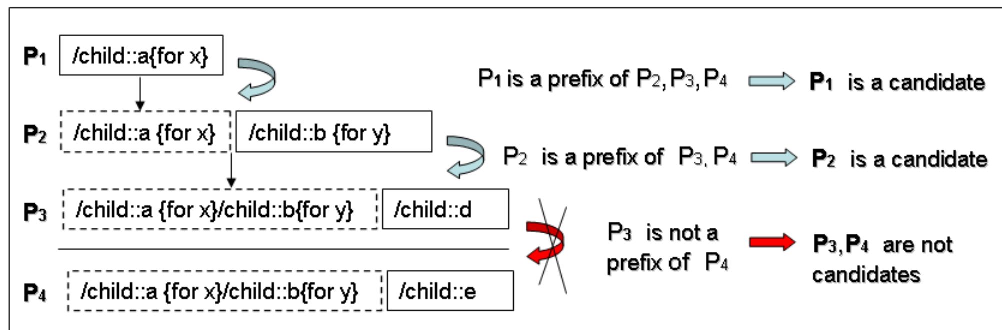



Figure 5.6: Scenario of finding candidate paths of Example 2.

Note that if we alter the query in Example 2 by considering the following new returned clause `return` ($\$x/child :: d$ , $\$y/child :: e$):

Q = `for` $\$x$ `in` $/child :: a$
`for` $\$y$ `in` $\$x/child :: b$
`return` ($\$x/child :: d$ , $\$y/child :: e$)

then the only candidate is $P_1$, while the path $P_2$ cannot be safely used for partitioning the input due to $\$x/child :: d$ in the return clause.

Also, if we change the query in Example 2 as follows (note that the path selecting nodes for the second iteration starts from the document root):

$$
\begin{aligned}
\mathrm{Q} = & \texttt{ for } \$x \texttt{ in } /child :: a/child :: b \\
& \quad \texttt{for } \$y \texttt{ in } /child :: a/child :: b \\
& \quad\quad \texttt{return } (\$x/child :: d \ , \ \$y/child :: e)
\end{aligned}
$$

then we have no candidates, because due to the presence of different variables $\$x$ and $\$y$ variables in extracted paths, condition (iii) of Definition 5.3.1 does not hold for any paths. This query is not recognized as an iterative query (there is no candidate path). In fact, the kind of partitioning we want to adopt can not be used for this query as it performs two iterations .

**Definition 5.3.2 (Iterative Queries)** *A* well-formed *query $Q$ is iterative if and only if $Candidate(Q) \neq \emptyset$.*

If the query $Q$ is iterative, then the sequence of nodes selected by a candidate path in a document $t$, can be partitioned in order to split query evaluation.

**Definition 5.3.3 (Partitioning Path)** *Given an iterative query $Q$, we say that the path $P$ is the partitioning path for $Q$ if and only if $P$ is the candidate partitioning path of $Q$ having maximum length.*

In the following, a partitioning path will be denoted *PP*. Going back to the query of Example 1, we have *PP*$=/child :: a/child :: b$.

Another example, is about the query of the Example 2, we have that :

$$Candidate(Q) \ = \ \{/child :: a, \ /child :: a/child :: b\}$$

and *PP*$=/child :: a/child :: b$ because it has maximum length comparing with the other candidate path $/child :: a$.

Several cases of XMark queries are recognized as iterative queries, some of these queries and their partitioning paths are reported in Figure 5.7.

Picking up the longest candidate as partitioning path minimizes the size of trees belonging to the sequence selected by the path, hence maximizing the likelihood that each part yielded by partitioning fits in the available main-memory.

## 5.4 PROJECTION

A particular feature of our approach is that while performing partitioning, projection is performed too, in a single pass on the input document $t$: the projected partition $\{t'_1, t'_2, \ldots, t'_\kappa\}$ is directly obtained from $t$, hence avoiding scanning the document twice and storing intermediate results on persistent storage.

In this section, we will formalize the projection process, which will be then plugged in the definition of the partitioning algorithm. As already said, projection is made in terms of paths extracted from a query, once $\{for\ x\}$ occurrences have been eliminated.

| Query | Partitioning Path $PP$ |
|---|---|
| $Q_1$ | $/child :: site/child :: people/child :: person$ |
| $Q_2$ | $/child :: site/child :: open_auctions/child :: open_auction$ |
| $Q_5$ | $/child :: site/child :: closed_auctions/child :: closed_auction$ |
| $Q_{13}$ | $/child :: site/child :: regions/child :: australia/child :: item$ |
| $Q_{14}$ | $/child :: site/dos :: item$ |
| $Q_{15}$ | $/child :: site/child :: closed_auctions/child :: closed_auction/child :: annotation$ $/child :: description/child :: parlist/child :: listitem/child :: parlist$ $/child :: listitem/child :: text/child :: emph/child :: keyword$ |
| $Q_{16}$ | $/child :: site/child :: closed_auctions/child :: closed_auction$ |
| $Q_{17}$ | $/child :: site/child :: people/child :: person$ |
| $Q_{18}$ | $/child :: site/child :: open_auctions/child :: open_auction$ |
| $Q_{19}$ | $/child :: site/child :: regions/dos :: item$ |
| $Q_{20}$ | $/child :: site/child :: people/child :: person$ |

Figure 5.7: Partitioning paths of some iterative XMark queries.

In the definition below, we will formalize our query projector, and present some examples which explain how the projection process works.

**Definition 5.4.1 (Query projector)** *Given a* well-formed *query $Q$, we define the* projector $\tau$ *of $Q$ as the set* $\tau = \{ErVar(P) \mid P \in E(Q)\}$.

Projecting an XML document $t$ according to a set of paths $\tau$ is a recursive process and works as follows. According to the document order, each node is visited and compared against the current set of paths to check whether the node matches the first step of each extracted path.The example below illustrates how projection works.

**Example 3** Consider the tree $t$ in Figure 5.2 and assume to project it according to the path $/child :: a/dos :: c$. Before matching the first node (actually the root element node) against the path, we perform a level alignment transformation over the path itself, by replacing the first step $/child :: a$ with $/self :: a$, thus obtaining $/self :: a/dos :: c$.We can, then, check that the $\mathrm{l}_1$ node matches the first step. As a side result of this phase, the path is rewritten into the *residual path* $/dos :: c$, in order to prepare the matching against the nodes of the next tree level. Then, before analyzing the $\mathrm{l}_2$ node, a new alignment operation is performed. This time, due to the presence of the recursive step $/dos :: c$, two paths are produced: $/self :: c$ and $/self :: node()/dos :: c$. These two paths are then compared with $\mathrm{l}_2$, which

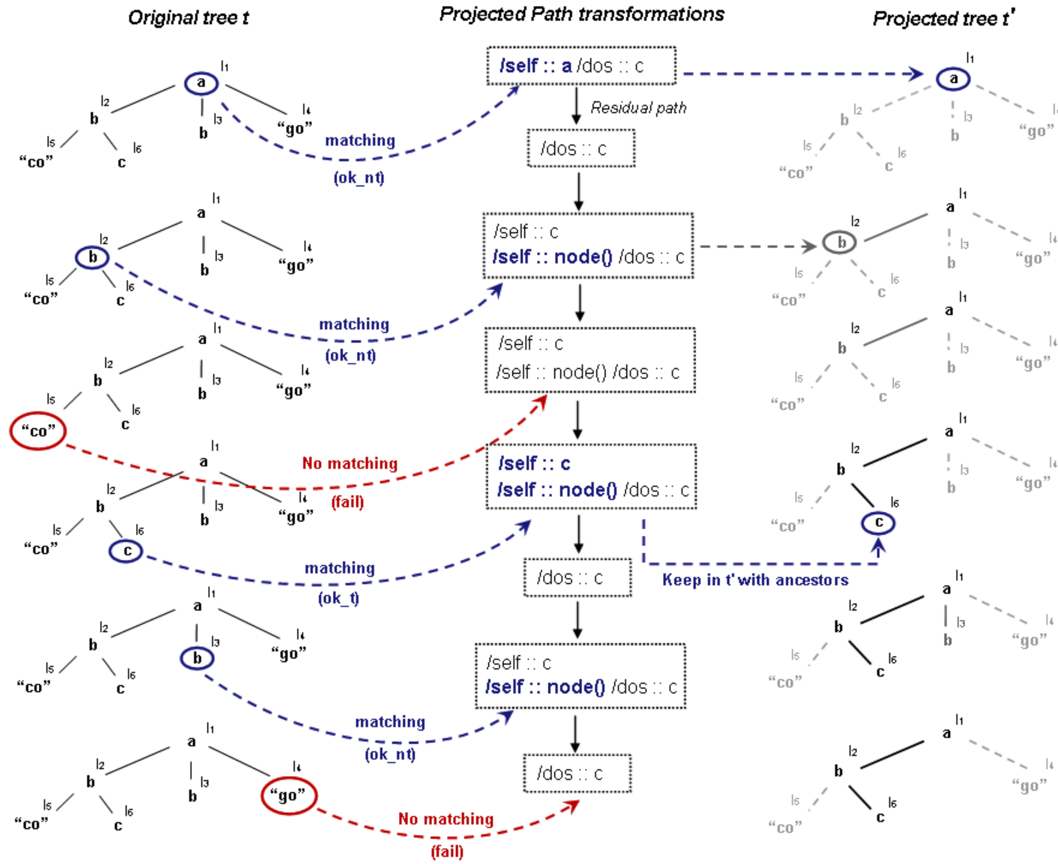



Figure 5.8: Path $/child :: a/dos :: c$ transformations.

actually matches the head $self :: node()$ axes of the second path, which is then rewritten into its residual $/dos :: c$. Path alignment then works as before and produces $/self :: c$ and $/self :: node()/dos :: c$ for the node $l_5$; in this case no path is matched. The next node considered is $l_6$, still matched against the paths $/self :: c$ and $/self :: node()/dos :: c$. Now we have a matching with $/self :: c$, and the node is added to the projection. This entails that the ancestors $l_2$ and $l_1$ are included in the projection as well. The process then goes on in a similar way with other nodes, which will not be included in the projection due to no matching with compared paths. Figure 5.8 illustrates the process above in details.

□

Before illustrating the projecting-partitioning process, we need a few preliminary definitions and notions. Hereafter a match for a path is called a *terminal* match, while an ancestor of a match is called a *non-terminal* match.

For instance, for the input tree in Figure 5.9, and the path $P = /dos :: c$, terminal matches are nodes $l_3$, $l_{10}$, $l_{15}$ and $l_{17}$, while non-terminal matches are ancestors of these nodes, i.e. $l_1$, $l_2$, $l_9$ and $l_{13}$.

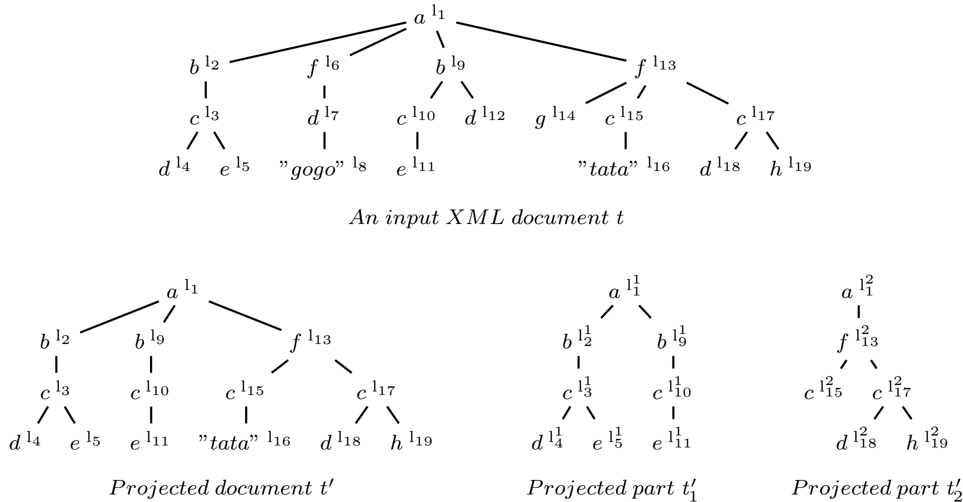



Figure 5.9: Partition plus projection.

Given a tree $t{=}(\sigma, \mathrm{l}_t)$ and a path $P$, we find terminal and non-terminal matches of $P$ by means of an iterative procedure that visits the tree $t$ in a top-down manner, and matches each node to a set of paths obtained from $P$ by means of two rewriting operations.

A first rewriting aims at *aligning* paths each time a deeper level is visited. For instance, in the previous example the root node is compared to the set of paths $\{/self :: c,\ /self :: node()/dos :: c\}$, obtained by the alignment of $P{=}/dos :: c$. The path $/self :: node()/dos :: c$ is an unfolding of the original one, and is needed to match $c$ nodes at deeper levels in subsequent steps of the process.

Projector alignment is performed by the function $Down(\tau)$, which aligns all paths in a set $\tau$, according to the following definition.

**Definition 5.4.2 (Path Alignment)** *The alignment* $Down(\tau)$ *of a projector* $\tau$ *is defined as* $\bigcup_{P\in\tau} Down(P)$ *where:*

1. $Down(/child :: NT/P) \quad = \quad \{/self :: NT/P\}$
2. $Down(/dos :: NT/P) \quad = \quad \{/self :: NT/P,\ /self :: node()/dos :: NT/P\}$
3. $Down(\varepsilon) \quad = \quad \{\varepsilon\}$

Paths obtained by alignment all start with a *self* step, which a terminal or non-terminal node has to necessarily match. After alignment, resulting paths may contain consecutive steps using the *self* axis (in particular, if the path already contained a *self* step before alignment). We assume that a path extracted from a query does not contain the self axis in the first step. This assumption is not restrictive as in practice the first step always perform a downward navigation. If consecutive *self* steps like in $/self :: b/self :: c$ occur in an aligned path, then the path is discarded from the process as it has empty semantics. Non-contradictory consecutive *self* steps (like $/self :: b/self :: node()$) are collapsed in a single *self*

step (like $/self :: b$) by means of a simple rewriting. These simple rewritings are routinely made after each alignment operation; obtained paths are then considered for matching with the current node, as discussed shortly.

We discuss now the second rewriting function. In the search of matches for a path $P$ in a tree $t$, given a node whose tag or text value is $\alpha \in \{\mathrm{a}, text[\mathrm{s}]\}$, and the corresponding set of aligned paths is $\tau$ (obtained from $P$), the residuation function $Res(\alpha; \tau)$ returns a path set $\tau'$ and a value $\mathrm{MATCH} \in \{\mathrm{ok_t}, \mathrm{ok_nt}, \mathrm{fail}\}$. The path set $\tau'$ will be recursively matched against children of the node after an alignment operation, while MATCH specifies whether the current node is a terminal match, a *possible* non-terminal match, or neither a terminal nor a non-terminal node. A possible non-terminal match is finally confirmed as such when one of its descendants happens to be deemed as a terminal match in subsequent steps.

Deriving the value MATCH produced by residuation relies on the following binary (commutative and associative) function $\uplus$, shown in Table 5.1, where the symbol $-$ indicates any value.

| $\mathrm{MATCH}_1$ | $\mathrm{MATCH}_2$ | $\mathrm{MATCH}_1 \uplus \mathrm{MATCH}_2$ |
|---|---|---|
| ok_t | – | ok_t |
| ok_nt | fail | ok_nt |
| ok_nt | ok_nt | ok_nt |
| fail | fail | fail |

Table 5.1: The function $\uplus$.

**Definition 5.4.3 (Path Residuation)** *The residual of a path $P$ is defined by distinguishing the following cases (recall that $\alpha \in \{\mathrm{a}, text[\mathrm{s}]\}$):*

$$
\begin{array}{llll}
Res(\mathrm{a}\ ; /self :: NT) & = & <\varepsilon\ ; \mathrm{ok_t}> & \textit{if } NT \in \{\mathrm{a}, node()\} \\
Res(\mathrm{a}\ ; /self :: NT/P) & = & </P\ ; \mathrm{ok_nt}> & \textit{if } P \neq \varepsilon \textit{ and } NT \in \{\mathrm{a}, node()\} \\
Res(text[\mathrm{s}]\ ; /self :: NT) & = & <\varepsilon\ ; \mathrm{ok_t}> & \textit{if } NT \in \{text(), node()\} \\
Res(\alpha\ ; /P) & = & <\varepsilon\ ; \mathrm{fail}> & \textit{otherwise}
\end{array}
$$

*The residual of a path set $\tau = \{P_1, P_2, \cdots, P_{\mathrm{n}}\}$ is then defined as follows:*

$$
Res(\alpha\ ; \tau) = <\bigcup_{\mathrm{i}=1}^{\mathrm{n}} \{P'_{\mathrm{i}}\};\ \biguplus_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{MATCH}_{\mathrm{i}}> \quad \textit{with} \quad Res(\alpha; P_{\mathrm{i}}) = <P'_{\mathrm{i}}; \mathrm{MATCH}_{\mathrm{i}}>
$$

As illustrated shortly, residuation is always applied after an alignment operation, and produces paths that are immediately aligned when descending to a new level of the tree. That said, going back to our observation concerning the handling of consecutive *self* steps, note that since each path extracted from a query never starts with a *self* step, and since multiple and consecutive *self* steps are eventually collapsed (otherwise the path is discarded) after alignment, residuation always takes as input paths starting with a *self* step, followed by a non-*self* step, and eventually

produces new paths by simply discarding the initial (matched) *self* step. This explains why the definition of alignment (Definition 5.4.2) does not include a case for a first *self*-step.

To illustrate how the just presented rewriting functions are used to select terminal and non-terminal matches of a path, consider again the input tree in Figure 5.9, and the path $P{=}/dos :: c$. Terminal and non-terminal matches of this path are determined as illustrated next, where for each node we indicate the corresponding aligned and residuated paths. We focus on the first 8 nodes, according to document order, as Table 5.2 illustrates.

| node | $\alpha$ | alignment | residuation |
|---|---|---|---|
| $l_1$ | a | $\tau_1{=}Down(\{P\})$ <br> $=\{/self :: c,\ /self :: node()/dos :: c\}$ | $Res(a; \tau_1){=}{<}\tau_2; \text{ok_nt}{>}$ <br> with $\tau_2{=}\{/dos :: c\}$ |
| $l_2$ | b | $\tau_3 = Down(\tau_2)$ <br> $=\{/self :: c,\ /self :: node()/dos :: c\}$ | $Res(b; \tau_3){=}{<}\tau_4; \text{ok_nt}{>}$ <br> with $\tau_4{=}\{/dos :: c\}$ |
| $l_3$ | c | $\tau_5{=}Down(\tau_4)$ <br> $=\{/self :: c,\ /self :: node()/dos :: c\}$ | $Res(c; \tau_5){=}{<}\tau_6; \text{ok_t}{>}$ <br> with $\tau_6{=}\{/dos :: c\}$ |
| ... | | | |
| $l_6$ | f | $\tau_7{=}Down(\tau_6)$ <br> $=\{/self :: c,\ /self :: node()/dos :: c\}$ | $Res(f; \tau_7){=}{<}\tau_8; \text{ok_nt}{>}$ <br> with $\tau_8{=}\{/dos :: c\}$ |
| $l_7$ | d | $\tau_9{=}Down(\tau_8)$ <br> $=\{/self :: c,\ /self :: node()/dos :: c\}$ | $Res(d; \tau_9){=}{<}\tau_{10}; \text{ok_nt}{>}$ <br> with $\tau_{10}{=}\{/dos :: c\}$ |
| $l_8$ | text[gogo] | $\tau_{11}{=}Down(\tau_{10})$ <br> $=\{/self :: c,\ /self :: node()/dos :: c\}$ | $Res(text[gogo]; \tau_{11}){=}{<}\varepsilon; \text{fail}{>}$ <br> with $\tau_{10}{=}\{/dos :: c\}$ |

Table 5.2: Rewriting functions $Down(\tau)$ and $Res(\alpha; \tau)$.

According to the residuation above indicated, $l_1$ and $l_2$ are deemed as non-terminal matches since both nodes have a descendant node $l_3$ being a terminal match. Observe that a terminal match is selected when a single-step path in the current set of aligned paths is matched by the current node: this means that the last step of the initial path is successfully matched. Concerning nodes $l_6$ and $l_7$, they have no descendant that residuation deems as a terminal match, hence these nodes are not deemed as non-terminal matches.

Algorithm 1 presents the code of the *Projection* algorithm. It takes as input a store $\sigma$, a current location l, and a projector $\tau$. It outputs a pair $(\sigma', Size)$ where $\sigma'$ is the projection of the tree rooted at l ($\sigma@l$) with respect to the projector $\tau$. The value *Size* is the size of the projected document and will be used when combining partitioning and projection.

This algorithm uses $Down(\tau)$ and $Res(\alpha; \tau)$ for alignment and residual rewriting. These both function are at the core of our technique. In order to compute *Size*, the algorithm uses the function $length(x)$ (see Example 4) returning the length of the string of $x$ (which can be either an element tag or a content of a textual node). Note

**Algorithm 1:** *Projection*

```
   Input: A store σ, a location l∈dom(σ), a projector τ;
   Output: A store σ', an integer Size;
 1 begin
       /* Case 1.  σ(l) = text[s]                                  */
 2     if Res(text[s]; τ) = <−; fail> then
 3         σ' := ∅;    Size := 0
 4     else
 5         σ' := {l ← text[s]};    Size := length(s)
       /* Case 2.  σ(l) = a[L]                                     */
 6     <τ'; MATCH> := Res(a; τ);
 7     if MATCH = fail then
 8         σ' := ∅;    Size := 0
 9     else if MATCH = ok_nt and L = () then
10         σ' := {l ← a[()]};
11     else
12         let L = (l_1, l_2, ··· , l_n)
13         for i = 1...n do
14             (σ_i, Size_i) := Projection(σ; l_i; Down(τ'))
15         π := {l_i∈L | σ_i ≠ ∅}
16         if (MATCH = ok_t) or (MATCH = ok_nt and π ≠ ∅) then
17             σ' := {l ← a[L_|π]} ∪ ⋃_{i=1}^{n} σ_i;    Size = 2.length(a) + Σ_{i=1}^{n} Size_i
18         else
19             σ' := ∅;    Size := 0
20     return (σ', Size)
```

that the size of an element includes the size of both the start and end tag.

**Example 4** Consider the tree $t$ = `<a><b>coco</b></a>`. We have that:

$$\begin{aligned} length(<a>) &= 1 \\ length(</a>) &= 1 \\ length(\text{"coco"}) &= 4 \\ length(<b>\text{"coco"}</b>) &= 1+ 4+ 1 = 6 \\ length(t) &= 1 + 6 + 1 = 8 \end{aligned}$$

□

Also, in the algorithm the notation $\mathtt{L}_{|\pi}$ indicates the location sequence obtained from L by retaining only locations in the set $\pi$, and preserving the sequence ordering (we have $\mathtt{L}_{|\pi} \preceq \mathtt{L}$).

Algorithm 1 consists of two main cases. When the current node location l contains a text node, if residuation does not fail, then for at least one path in the

projector the last step matches the node l (recall that only the final step in a path can use the text node condition).

When the current location, instead, contains an element node, then a more complex analysis is necessary. If residuation fails, then the empty store is output. If the current node is an intermediate match for the current projector, and the node has no child, then the node is added to the projection; this is necessary because this node can be later on matched as a terminal node after residuation of the projector, during the recursive process. For instance, consider a projector including $/a/b/self :: node()$ and a tree where the root $a$ has an empty $b$ element as child. Otherwise, projection is recursively propagated on child nodes. Then, if the current element node is a terminal match for the projector, this node is added to the projection together with its projected subtrees; if the current element matches an intermediate step of a path in the projector, then the node will be added to the projection if at least one of its descendant will match a final step in the projector. If none of the above conditions holds, the empty projection is output.

Differently from [MS03] we provide here a formal specification of the projection algorithm. Also, the algorithm described is DOM-oriented. We present it just to provide a clear and formal specification. In Section 5.6 we will provide some detail about our SAX-based streaming implementation, which has a negligible memory footprint.

**Lemma 5.4.4** *Let $Q$ be a* well-formed *query, $\tau$ its associated projector and $t$=$(\sigma, \mathrm{l}_t)$ a tree. Assuming that Projection$(\sigma;\ \mathrm{l};\ Down(\tau))$=$(\sigma'; Size)$ we have:*

*(i) $Q(t) \cong Q(t')$ where $t'$=$(\sigma', \mathrm{l}_t)$, and*

*(ii) $Size$=$size(t')$*

## 5.5 The partitioning algorithm

The partitioning algorithm takes as input an XML document $D$, an iterative query $Q$, and a threshold value *maxSize*. Through the static analysis technique described in the previous sections, the algorithm extracts the set of projection paths $\tau$ and the partitioning path set $PP$. These two sets of paths $\tau$ and $PP$ drive the projection-partitioning process, as the following example illustrates.

**Example 5** Consider the query $Q$ below and the XML document of Figure 5.9.

$$Q = \texttt{for}\ x\ \texttt{in}\ /dos :: c\ \texttt{return}\ (x/child :: d,\quad x/child :: e)$$

According to previous definitions, this query is iterative with partitioning path $PP = /dos :: c$. Also, the set of extracted paths $\tau$ is (for-variables are erased):

$$\tau = \{/dos :: c,\ /dos :: c/child :: d/dos :: node(),\ /dos :: c/child :: e/dos :: node()\}$$

Through $\tau$ we can prune out all nodes of the document that are not touched during query evaluation, and create after that the projected parts $t'_1$ and $t'_2$, containing the fragments that are sufficient for correctly evaluating $Q$.

This means that the store $\sigma'$ should contain only the following locations $\{l_1, l_2, l_3, l_4, l_5, l_9, l_{10}, l_{11}, l_{13}, l_{15}, l_{16}, l_{17}, l_{18}, l_{19}\}$ and neglect the others.

$$\sigma' = \begin{cases} l_1 & \leftarrow & a[l_2, l_9, l_{13}], & l_2 & \leftarrow & b[l_3], \\ l_3 & \leftarrow & c[l_4, l_5], & l_4 & \leftarrow & d[()], \\ l_5 & \leftarrow & e[()], & l_9 & \leftarrow & b[l_{10}], \\ l_{10} & \leftarrow & c[l_{11}], & l_{11} & \leftarrow & e[()], \\ l_{13} & \leftarrow & f[l_{15}, l_{17}], & l_{15} & \leftarrow & c[l_{16}], \\ l_{16} & \leftarrow & text[''tata''], & l_{17} & \leftarrow & c[l_{18}, l_{19}], \\ l_{18} & \leftarrow & d[()], & l_{19} & \leftarrow & h[()] \end{cases}$$

If, just to illustrate, we assume that the above projection cannot be processed, then partitioning is needed. According to Definition 5.3.3, the partitioning of the input tree in Figure 5.9 is made according to the partitioning path in $PP = \{/dos :: c\}$. The tree is traversed top-down according to document order and the first part is determined as follows. During the visit of the tree, non-terminal and terminal matches of the partitioning path are added to the part. Whenever a terminal match of $PP$ is met, its subtree is projected according to our projection (see Algorithm 1), in order to limit as much as possible the number of created parts.

Just after a projected sub-tree of a $PP$ terminal match has been added to the part, a check is made in order to verify whether the current size of the part has exceeded a given threshold *maxSize*. In the current example, we consider *maxSize*=12, which is exceeded when the subtree rooted at the second $PP$ terminal match is added to the part. Recall that each time an element is added to a part, the current size is incremented by twice the length of the element tag (both starting and ending tags have to be taken into account), while each time a text node is added to the part the current size is incremented by the length of the text content of the node. This causes the creation of a second part. With *maxSize*=12 we finally have the two parts indicated in Figure 5.9. Note that nodes that are neither non-terminal nor terminal matches of the partitioning path are pruned out during partitioning. These nodes can be safely pruned out because they are useless to the evaluation of the query $Q$. This is because $PP$ is a prefix of each path in $\tau$ (extracted from the query, Definition 5.3.1), and that a node is needed by $Q$ if it is (an ancestor of) a match of a path in $\tau$ (for the same reasons, in Figure 5.9, note that since subtrees rooted at terminal matches of $PP$ are projected according to $\tau$; for instance, the node $l_{16}$ is not in the second part).

Note that ancestors of $PP$ nodes may belong to more than one part, in particular this is the case for the document root node. At the same time, we need to create a store with unique locations, so we endow each $l_i$ with an identifier j indicating that $l_i$ belongs to the part j of the partition. The partition will be represented by a single store $\sigma^P$.[1] This store $\sigma^P$ will contain two parts using the following indexed

[1]While the definition of partitions rely on multiple trees (stores), we opt here for a single global store to easy the specification of the algorithms. As we will see, each single tree of the partition can be recovered straightforwardly.

locations:

$$\mathrm{dom}(\sigma^{\mathrm{P}}) = \{\mathrm{l}_1^1,\ \mathrm{l}_2^1,\ \mathrm{l}_3^1,\ \mathrm{l}_4^1,\ \mathrm{l}_5^1,\ \mathrm{l}_9^1,\ \mathrm{l}_{10}^1,\ \mathrm{l}_{11}^1,\quad \mathrm{l}_1^2,\ \mathrm{l}_{13}^2,\ \mathrm{l}_{15}^2,\ \mathrm{l}_{17}^2,\ \mathrm{l}_{18}^2,\ \mathrm{l}_{19}^2\}$$

□

Besides path alignment and residuation, the threshold value *maxSize* plays a key role in the whole partitioning process. The choice of *maxSize* depends on many factors, such as the input document, the query being processed, the specific query processor being used, the hardware configuration and the available main memory, the programming language used for implementing the query processor, the memory management technique adopted, and the operating system running on the hardware. *maxSize*, therefore, can be determined only through a *trial-and-error* process depending on the overall configuration, and cannot be formally predicted.

Note that if *maxSize* is too large, it can happen that one or more parts are too large to be loaded in main memory, hence undermining the whole approach. Surprisingly enough, as we will see later, our experimental evaluation showed that the actual value of *maxSize* does not influence either partitioning time or the total querying time on the partition.

### 5.5.1 The Algorithm

Algorithm 2 provides a formal presentation of our partitioning scheme. It is a recursive algorithm and takes as input a 5-tuple $<\mathrm{l}; \tau; PP; cSize; pId;>$ representing the current state of the recursive process: namely, this tuple indicates that the current node to be matched against the current aligned partitioning path-set $PP$ and projector $\tau$ is l, that the current size of the part under construction is *cSize*, and that the current number of created parts is $pId$. Of course, the algorithm is initially invoked with *cSize*=0 and $pId$=1, while the location l is the root of the input XML tree $t = (\sigma, \mathrm{l})$. Also, $PP$ is $Down(\{PP_Q\})$, the alignment of the initial partitioning path for the iterative query $Q$ to execute, while $\tau$ is $Down(\tau_Q)$, the alignment of the projector $\tau_Q$ of the query $Q$ (see Definition 5.4.1) . The store $\sigma$ is assumed to be a global parameter.

In the algorithm, the function $PartLabel(\sigma; pId)$ produces a new store obtained from $\sigma$ by renaming each location l to $\mathrm{l}^{pId}$. We will use $PartLabel^{-1}(\sigma')$ to undo the renaming in the store $\sigma'$.

The algorithm distinguishes among three main cases. In the first case (lines 3-10), the current node is an element node being a terminal match for the initial partitioning path $PP$. In this case, our projection algorithm is called to compute the projection of the subtree rooted at this node together with its size. If no projection algorithm is available, $Projection(\sigma; \mathrm{l}; \tau')$ just returns the input subtree and its size. Then (lines 7-10) the algorithm adds the resulting subtree to the current part, and checks whether the size of the projected subtree plus the current size does not exceed the maximal size: if the check is positive, then the current size is incremented with

**Algorithm 2:** *Partition*

```
   Input: A location l∈dom(σ), a partitioning path-set PP, a projector τ, a part size
          cSize, a part number pId ;
   Output: A store σ^P, a part size cSize' , part number pId' ;
 1 begin
 2   let σ(l) = a[L]
     /* Case 1. l is a PP target node                          */
 3   if Res(a; PP) = <−;ok_t> then
 4     τ':= Res(a;τ);
 5     (σ', Size):= Projection(σ; l; Down(τ'));     /* projection always keeps
       node l in σ' */
 6     σ^P:= PartLabel(σ'; pId);
 7     if cSize + Size ≤ maxSize then
 8       cSize':= cSize + Size;    pId':= pId
 9     else
10       cSize':= 0; pId':= pId + 1
     /* Case 2. l is not a PP target node                      */
11   if Res(a; PP) = <PP';ok_nt> then
12     pId_first:= pId;    σ_temp:= ∅;
13     cSize_temp:= cSize + 2.length(a);
14     τ':= Res(a;τ);
15     let L = (l_1, l_2, ··· , l_n);
16     for i = 1...n do
17       (σ_i^P; cSize_temp; pId):= PartProj(l_i, Down(PP'), Down(τ'), cSize_temp, pId);
18       σ_temp:= σ_temp ∪ σ_i^P;
19     if σ_temp = ∅ then
20       cSize':= cSize
               /* no descendant of the current node l is added in the
       partition */
21     else
22       cSize':= cSize_temp;
         /* Max-Pid returns the biggest part number used in the store
         */
23       pId_last:= Max-Pid(σ_temp);
24       D:= dom(σ_temp);
25       σ^P:= σ^P ∪ σ_temp;
26       for p = pId_first...pId_last do
27         σ^P:= σ^P ∪ {(l^p←a[rename−extr(L, p, D)])}
28     pId':= pId;
     /* Case 3. l does not match PP                            */
29   else if Res(a;τ) = <−;fail> then
30     σ^P:= ∅;   cSize':= cSize;   pId':= pId
31   return (σ^P, cSize',pId')
```

the projection size *Size*, otherwise the current size is reset to 0 and a new (empty) part is created (this empty part will be filled in subsequent steps of the processing).

In the second case (lines 11-28), the current l node is a *possible* non-terminal match for the partitioning path *PP*. A temporary current size variable $cSize_{\text{temp}}$ registers the current size plus twice the length of the current tag (both start and ending tags have to be taken into account). By considering $cSize_{\text{temp}}$ as the current size, the computation recursively goes on for each child $l_i$ of the l node (lines 16-20). For each $l_i$ partitioning is made according to paths obtained by alignment of paths resulted by residuation (line 17), and the resulting parts are kept in a temporary store $\sigma_{\text{temp}}$. Also, partitioning for each child node $l_i$ is made according to the current size and partition number produced by the partition process for the child $l_{i-1}$.

Once partitioning for all children is done, the resulting store $\sigma_{\text{temp}}$ is checked for emptiness (line 19). If the store is empty, then the current node l is not deemed as a non-terminal match as it has no descendant being a terminal match. Hence, the node does not contribute to the current part (it is pruned out), and the output current size is set to the input current size; note that in this case the input part identifier *pId* is unchanged. Otherwise (lines 21-27), the current partition and size have to be updated. The output current size is set to $cSize_{\text{temp}}$ (line 22), registering the current size of the current part or, eventually, the last part created while processing children $l_i$. After this (lines 23-25), the current partition $\sigma^P$ is enriched with $\sigma_{\text{temp}}$ and (lines 26-27) with elements for the current location l indexed by all new part numbers $pId_j$'s produced while processing $l_i$'s subtrees (recall that for a child $l_i$ more than one part could be created). In this case l has to be indexed accordingly. To this end, the algorithm uses a function $rename\!-\!extr(\mathtt{L}, p, D)$ which takes as input the sequence L of children locations, a part identifier $p$, and the domain $D{=}\text{dom}(\sigma^P)$ of the created sub-partition. The role of the function $rename\!-\!extr(\mathtt{L}, p, D)$ is to extract the sub-sequence of L used to create the part $p$ in $\sigma^P$, and to adorn with $p$ each location in this sub-sequence. Formally, we have:

$$rename\!-\!extr(\mathtt{L}, p, D) \quad = \quad \begin{cases} () & \text{if } \mathtt{L} = () \\ l_i^p, rename\!-\!extr(\mathtt{L}', p, D) & \text{if } \mathtt{L} = l_i, \mathtt{L}' \text{ and } l_i^p \in D \\ rename\!-\!extr(\mathtt{L}', p, D) & \text{if } \mathtt{L} = l_i, \mathtt{L}' \text{ and } l_i^p \notin D \end{cases}$$

For instance, if the current node of the case is $l \rightarrow a[l_1, l_2, l_3]$ and for subtrees rooted at $l_1, l_2$ data are put in part 3, while for the subtree rooted at $l_3$ data are put/split in two parts $4, 5$, then the renaming extraction produces $l_1^3, l_2^3$ and $l_3^4$ and $l_3^5$.

Finally, the third case (lines 29-30) applies when the current node does not match the partitioning path, hence the algorithm produces an empty part, and preserves the current part size and number.

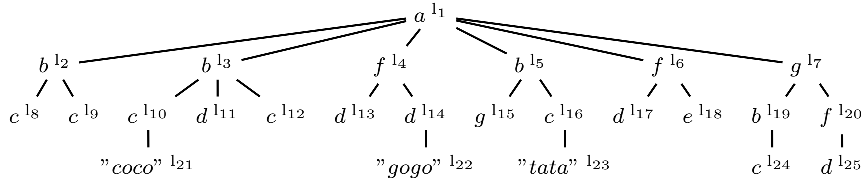



Figure 5.10: An input XML tree $t$.

### 5.5.2 Dealing with a Workload

A nice property of our *projecting-partitioning* system is that it can deal with a workload formed by queries $Q_1, \ldots, Q_\mathrm{n}$ in a natural way. To this end, it suffices to consider a global projector $\tau = \cup_1^n \tau_\mathrm{i}$ and set of partitioning paths $PP = \cup_1^n \{\ PP_\mathrm{i}\}$ where $\tau_\mathrm{i}$ and $PP_\mathrm{i}$ are, respectively, the projector and the partitioning path of $Q_\mathrm{i}$.

This follows from the fact that our system is already specified to deal with a set $PP$ (recall that $Down()$ produces set of paths in the presence of *dos* axis). So, with $PP = \cup_1^n \{\ PP_\mathrm{i}\}$ the partition is made in terms of nodes matching at least one of the paths $PP_\mathrm{i}$'s, and the corresponding subtrees are projected by means of the global projector $\tau$ keeping into account the data needs of the whole workload.

To illustrate the effectiveness of our *projecting-partitioning* algorithm with workload (described above). Example 6 explains, in details, how to deal with a workload formed by two iterative updates.

**Example 6** Consider the following iterative queries on the XML document $t$ illustrated in Figure 5.10:

$Q_1 =$ `for` $\$x$ `in` $/child :: a/child :: b$ `return` $\$x/child :: c$
$Q_2 =$ `for` $\$y$ `in` $/child :: a/child :: f$ `return` $\$y/child :: d$

Since that we have two independent queries, two different stores $(\sigma_1, \sigma_2)$ will be created to specify a projection for each query during the process. According to Definition 5.4.1 and by using the function $E()$ to extract paths from $Q_1$ and $Q_2$, we have the following distinct projectors:

$$\tau_1 = \{/child :: a,\ \ /child :: a/child :: b,\ \ /child :: a/child :: b/child :: c/dos :: node()\}$$
$$\tau_2 = \{/child :: a,\ \ /child :: a/child :: f,\ \ /child :: a/child :: f/child :: d/dos :: node()\}$$

Depending on the above described projectors, two projected trees $t'_{Q_1}$ and $t'_{Q_2}$ can be created by using path information in the projectors, along the lines of standard path-based projection [MS03]. As it can be seen, each projected tree contains element nodes that are sufficient to evaluate its query, as illustrated in Figure 5.11. Both sets of $\mathrm{dom}(t'_{Q_1}), \mathrm{dom}(t'_{Q_2})$ will contain only the following locations :

$$\mathrm{dom}(t'_{Q_1}) = \{\mathrm{l}_1,\ \mathrm{l}_2,\ \mathrm{l}_3,\ \mathrm{l}_5,\ \mathrm{l}_8,\ \mathrm{l}_9,\ \mathrm{l}_{10},\ \mathrm{l}_{12},\ \mathrm{l}_{16},\ \mathrm{l}_{21},\ \mathrm{l}_{23}\}$$
$$\mathrm{dom}(t'_{Q_2}) = \{\mathrm{l}_1,\ \mathrm{l}_4,\ \mathrm{l}_6,\ \mathrm{l}_{13},\ \mathrm{l}_{14},\ \mathrm{l}_{17},\ \mathrm{l}_{22}\}$$

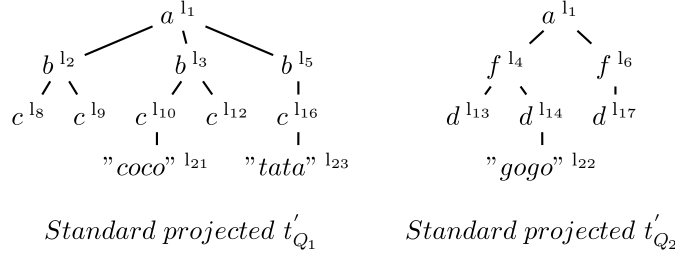



Figure 5.11: Standard projections $t^{'}_{Q_1}$, $t^{'}_{Q_2}$ XML trees created from the input $t$.

If, just to illustrate, we assume that trees whose size is bigger than 12 cannot be loaded for query processing (we assume that the size is in terms of characters) then we have that $Q_1$ cannot be evaluated on the projection $t^{'}_{Q_1}$ illustrated in Figure 5.11. Here, we need partitioning for query evaluation. According to Definition 5.3.3, we have that the partitioning path for $Q_1$ is $PP{=}/child :: a/child :: b$. A safe choice for the threshold value is $maxSize = 10$.

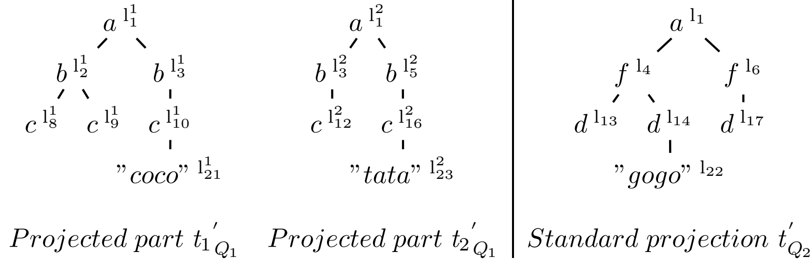



Figure 5.12: Partitioning scenario on $t$ for a given iterative query $Q_1$.

Figure 5.12 shows the two parts created by partitioning (and projection). In particular, the new store $\sigma^{\mathrm{P}}_{Q_1}$ contains the following indexed locations:

$$\mathrm{dom}(\sigma^{\mathrm{P}}_{Q_1}) = \{\mathrm{l}^1_1,\ \mathrm{l}^1_2,\ \mathrm{l}^1_8,\ \mathrm{l}^1_9,\ \mathrm{l}^1_3,\ \mathrm{l}^1_{10},\ \mathrm{l}^1_{21},\quad \mathrm{l}^2_1,\ \mathrm{l}^2_3,\ \mathrm{l}^2_{12},\ \mathrm{l}^2_5,\ \mathrm{l}^2_{16},\ \mathrm{l}^2_{23}\}$$

After finishing the partitioning process (described above), we can evaluate $Q_1$ on the two parts and obtain the final result by simply concatenating the two partial results in the obvious order.

Now, suppose that we want to evaluate a workload $W = \{Q_1,\ Q_2\}$ on the same XML tree $t$ presented in Figure 5.10. By using standard projection, [MS03] propose to consider a global projection for evaluating all queries. So we need to create a global projection of $t$ by considering the global projector $\tau_{\mathrm{W}}{=}\{\tau_1,\ \tau_2\}$, as illustrated in Figure 5.13:

$$\begin{aligned}\tau_{\mathrm{W}} = \ & \{/child :: a,\ /child :: a/child :: b,\ /child :: a/child :: b/child :: c/dos :: node(),\\ & /child :: a,\ /child :: a/child :: f,\ /child :: a/child :: f/child :: d/dos :: node()\}\end{aligned}$$

According to previous assumptions, we have once again that the global projection does not fit in the memory. Fortunately, our technique adapts gracefully to the case

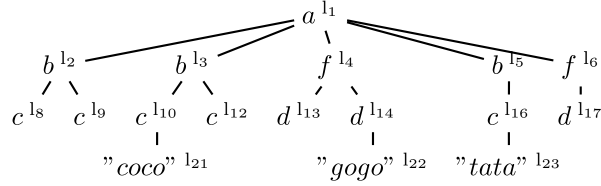



Figure 5.13: Global projection $t'$ for the workload $(Q_1, Q_2)$.

of a workload, and this allows us to overcome the problem, as follows. We use the following set of partitioning paths $\cup_1^n\{\ PP_{\rm i}\}$ extracted from $Q_{\rm i}$, for $i = 1, 2$. In particular, $PP_{\rm W}$=$\{PP_1,\ PP_2\}$ where

$$\begin{aligned} PP_1 &= /child :: a/child :: b \\ PP_2 &= /child :: a/child :: f \end{aligned}$$

Figure 5.14 illustrates the global partitioning which is capable to satisfy the query needs of the entire workload.

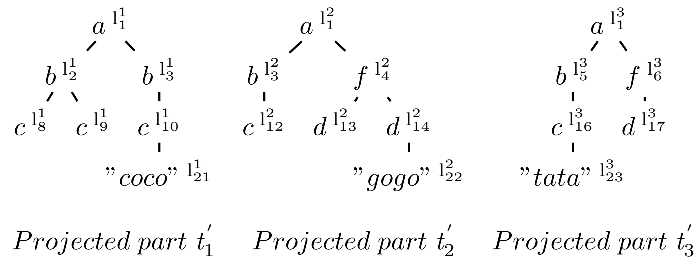



Figure 5.14: Partitioning scenario on the global projection $t'$ of workload $(Q_1, Q_2)$.

□

In the above examples, we have that each single created pat has a size which is less than *maxSize*. According to our partitioning algorithm, this is not always the case: it may happen a part creation ends as soon as its size exceeds the threshold.

Soundness of partitioning is stated by the following theorem, using the notation $Part_i(\sigma^{\rm P})$ to indicate the i-th part $\sigma_{\rm i}$ in the partition $\sigma^{\rm P}$: formally $\sigma_{\rm i} = \{{\rm l}^{\rm i} \leftarrow a[{\tt L}] \mid {\rm l}^{\rm i} \leftarrow {\rm a}[{\tt L}] \in \sigma^{\rm P}\}$.

**Theorem 5.5.1 (Soundness of Partition and Projection)** *Let maxSize be a size threshold value, let $Q_1, \ldots, Q_m$ be* well-formed *queries with their resp. projector $\tau_j$ and partitioning path $PP_j$. Let $t$=$(\sigma, {\rm l}_t)$ be an XML tree. Then:*
*Assuming*

- $\tau$= $\cup_1^m\ \tau_j$,
- $PP$= $\cup_1^m\ \{PP_j\}$ *and*

- $Part(\mathrm{l}_t; Down(\tau); Down(PP); 0; 1) = (\sigma^{\mathrm{P}}; cSize; pId)$.

*we have:*

$$Q_j(t) \cong Q_j(t_1) \cdot \ldots \cdot Q_j(t_{pId})$$

*where* $t_i$=$PartLabel^{-1}(Part_i(\sigma^P))$.

## 5.6 STREAMING IMPLEMENTATION

We implemented our partitioning algorithm in a streaming fashion on top of a SAX parser [ver00]. In our implementation, we considered the following SAX events:

| | | | |
|---|---|---|---|
| $SAXEvent$ | := | `startDocument` | called at the start of the input document |
| | \| | `startElement`($qName$) | called at the open-tag of the current qName |
| | \| | `endElement`($qName$) | called at the close-tag of the current qName |
| | \| | `Characters`($String$) | called to process the text-contents of the current qName |

In our SAX implementation of partitioning we used four main stack-based data structures (see lines 3-5 of Algorithm 3).

These stacks are used to record the current status of the algorithm when an open-tag is met, so that the status can be recovered when the corresponding close-tag is met.

- The first stack $stack_{tag}$ is used to record open tag-name of the node $qName$ being processed, the result of the residuation of $Res(qName; PP)$, the modality value (which is either *part* or *proj*, it will be explained later), a boolean flag *isStored*, which is set to *true* only when the open-tag has been written in the current part.
- The second stack $stack_\tau$ is used to record all alignment results of the projector path-set.
- The third stack $stack_{pp}$ records alignment results of the partitioning path-set.

The implementation also tracks some global values in the following variables:

- *cSize*, the size of the current part.
- *pId*, the current number of created parts.
- $\tau$, the projector.
- *PP*, the partitioning path set.
- *Size*, the size of the XML subtree nodes rooted at the node matching *PP*.

By using this status information, we can split the projection-partitioning algorithm in two distinct procedures, which are executed when `startElement` and `endElement` are invoked, respectively.

Before starting the processing, our algorithm takes the following inputs (see Algorithm 3):

- the input XML document $t$.
- the set of extracted paths $\tau$ and the set of partitioning path $PP$ extracted from an iterative query $Q$.
- the threshold integer value $maxSize$.

and it is initially invoked with $cSize$=0 and $pId$=1 (line 2 of Algorithm 3). Also, all data structures needed to perform the partitioning $stack_{tag}$, $stack_{\tau}$ and $stack_{pp}$ will be initialized.

**Algorithm 3:** Projection/Partition-Init-DataStructure

**Input**: An input XML document $t$, a pre-defined integer value $maxSize$; a set of paths $\tau$ extracted from a given query $Q$, a partitioning path $PP$;
**Output**: Initialize global variables $cSize$, $pId$, and three stack-based data structures $stack_{tag}$, $stack_{\tau}$ and $stack_{pp}$;

```
1 begin
2     cSize := 0;   pId := 1
3     stack_tag := ()
4     stack_τ := ()
5     stack_pp := ()
```

**Algorithm 4:** SAX-startDocument

**Input**: A projector $\tau$, a partitioning path $PP$, a flag $Modality$;
**Output**: Side effect on $\tau$, $PP$ and $Modality$;

```
1 begin
2     τ := Down(τ)
3     PP := Down(PP)
4     Modality := part
```

Both `startElement` and `endElement` algorithms work in two possible modalities, the partitioning modality ($part$) and the projection modality ($proj$). The first one concerns the case that the current node is either a (possible) non-terminal match or a terminal match of a partitioning path in $PP$. Under this modality the algorithms implement the specification reported in the DOM-based Algorithm 2). The second possible modality captures the case where the current node belongs to a subtree rooted at terminal node of a partitioning path. Under this modality, the two algorithms implement the projection as given by Algorithm 1.

**Algorithm 5:** SAX-characters

```
  Input: A string value str, current part size cSize
  Output: Side effect on the current part size cSize
1 begin
2     MATCH:= Res(str; τ)
3     if MATCH≠fail then
4         cSize:= cSize + length(str)
5         writeOutput(str)
```

In `startDocument` event (Algorithm 4), the algorithm performs the first alignment $Down(\tau)$ of the projector $\tau$ and the first alignment of the partitioning path $PP$ (see lines 2-4), then it initializes the *Modality* flag with *part*, which is the starting modality of our algorithm; projection starts when a target path of a partitioning path is met.

In `startElement` (see Algorithm 6), we put most of the logic of the DOM-based specification partitioning and projection algorithms (Algorithms 1 and 2): indeed, all partitioning and projection decisions are based on information that is available when an open tag is met. Concerning the partitioning modality (lines 2-26), we put here the updates of *Size* and *cSize*, as well as the residuation and alignment of the current partitioning path $PP$ (line 3), but we defer partitioning decisions to `endElement` calls. Concerning the projection modality (lines 27-49), we put here residuation of the current projector $\tau$ (line 28), the resulting case analysis to decide whether the current node has to be projected, and alignment of the path-set projector to the next level.

In `endElement` (see Algorithm 7), we first perform a pop operation on the stack $stack_\tau$ (line 2) and obtain information stored in the following variables (lines 3-5): MATCH is the current match value, $currModality$ is the current working modality, and $currStoredCase$ is the current storing status of the current $qName$ into the output. If $currStoredCase = false$ then the algorithm simply terminates the close-tag corresponds to the open-tag not stored in the partitioning.

If information got from the stack tell us that we are in the partitioning modality and the current storing case $currStoredCase$=*true*, then we make the following case analysis on the MATCH information relative to the current close-tag, and got from the $stack_{tag}$ (lines 6-30).

In the case the current closing-tag is for a non-terminal match of partitioning paths (MATCH=ok_nt) (lines 7-10), we increase the current part size *cSize* with *length*($qName$); write the current $qName$ into the current part; and finally pop the top element of $stack_{pp}$.

If the current closing-tag is for a terminal match of a partitioning path (MATCH=ok_t) (lines 11-30) then a projection phase comes to its end. So we change the *Modality* flag to *part* (line 12); compare the current part size *cSize* plus

**Algorithm 6:** SAX-startElement

```
   Input: Open-tag qName, a part number pId, a part size cSize;
   Output: Side effect on cSize and Size, Modality, τ and PP;
1  begin
       /* qName is in the partitioning modality case */
2      if Modality=part then
3          MATCH:= Res(qName; PP)
4          switch MATCH do
5              case ok_nt
                   /* Case 1. qName is non-terminal PP node */
6                  stack_tag.add(qName, ok_nt, part, false)
7                  τ:= stack_τ.add(Down(τ))
8                  PP:= stack_pp.add(Down(PP))
9              case ok_t
                   /* Case 2. qName is a terminal PP node */
10                 for i=[0..(stack_tag.size − 1)] do
11                     ancestTrNode_tagname:= stack_tag(i).get(0)
12                     ancestTrNode_isStored:= stack_tag(i).get(3)
13                     if ancestTrNode_isStored=false then
14                         stack_tag(i).set(3):= true
15                         writeOutput(ancestTrNode_tagname)
16                         cSize:= cSize + length(ancestTrNode_tagname)
17                     else if ancestTrNode_isStored=true then
18                         SkipElement_stack_tag(i)
19                 stack_tag.add(qName, ok_t, part, true)
20                 τ:= stack_τ.add(Down(τ))
21                 PP:= stack_pp.add(Down(PP))
22                 cSize:= cSize + length(qName)
23                 writeOutput(qName)
24                 Modality:= proj
25             case fail
                   /* Case 3. qName does not match PP */
26                 stack_tag.add(qName, fail, part, false)
       /* qName is in the projection modality */
27     else if Modality=proj then
28         MATCH_τ:= Res(qName; τ)
29         switch MATCH do
30             case ok_nt
                   /* Case 1. qName is non-terminal τ node */
31                 stack_tag.add(qName, ok_nt, proj, false)
32                 τ:= stack_τ.add(Down(τ))
33                 Size:= Size + length(qName)
34             case ok_t
                   /* Case 2. qName is terminal τ node */
35                 for i=[0..(stack_tag.size − 1)] do
36                     ancestTrNode_tagname:= stack_tag(i).get(0)
37                     ancestTrNode_isStored:= stack_tag(i).get(3)
38                     if ancestTrNode_isStored=false then
                           /* Switch flag isStored to true value to write the current
                              element stack into the current part t'_pId */
39                         stack_tag(i).set(3):= true
40                         writeOutput(ancestTrNode_tagname)
41                         cSize:= cSize + length(ancestTrNode_tagname)
42                     else if ancestTrNode_isStored=true then
43                         SkipElement_stack_tag(i)
44                 stack_tag.add(qName, ok_t, proj, true)
45                 τ:= stack_τ.add(Down(τ))
46                 Size:= Size + length(qName)
47                 writeOutput(qName)
48             case fail
                   /* Case 3. qName does not match τ */
49                 stack_tag.add(qName, fail, proj, false)
```

**Algorithm 7:** SAX-endElement

```
   Input: Close-tag qName, part size cSize, projection size Size, part number pId;
   Output: Side effect on cSize, pId, τ, PP Modality;
1  begin
       /* Pop the top element from stack_tag and keep match, currModality,
       currStoredCase values                                              */
2      τ:= stack_τ.pop
3      MATCH:= stack_tag.pop(stack_tag(top).get(1))
4      currModality:= stack_tag.pop(stack_tag(top).get(2))
5      currStoredCase:= stack_tag.pop(stack_tag(top).get(3))
6      if currModality=part  and  currStoredCase=true then
7          if MATCH=ok_nt then
8              cSize:= cSize + length(qName)
9              writeOutput(qName)
10             PP:= stack_pp.pop
11         else if MATCH=ok_t then
12             Modality:= part
13             if cSize + Size ≤ maxSize then
14                 cSize:= cSize + Size
15                 writeOutput(qName)
16             else
                   /* Close current part t'_pId                    */
17                 writeOutput(qName)
18                 for i=[(stack_tag.size − 1)..0] do
19                     currTagName:= stack_tag(i).get(0)
20                     currStored:= stack_tag(i).get(3)
21                     if currStored=true then
22                         writeOutput(currTagName)
23                 cSize:= 0;   pId:= pId + 1
                   /* Create new part t'_pId                       */
24                 for i=[0..(stack_tag.size − 1)] do
25                     currTagName:= stack_tag(i).get(0)
26                     currStored:= stack_tag(i).get(3)
27                     if currStored=true then
28                         writeOutput(currTagName)
29                 cSize:= cSize + length(qName)
30                 PP:= stack_pp.pop
31     else if currModality=proj  and  currStoredCase=true then
32         if MATCH=ok_nt then
33             cSize:= cSize + length(qName)
34             writeOutput(qName)
35         else if MATCH=ok_t then
36             cSize:= cSize + length(qName)
37             writeOutput(qName)
```

the projected subtree size *Size* with the maximal part size allowed *maxSize* (line 13), and create a new part if the size the current part has exceeded *maxSize*. The creation of a new part requires one to iterate on the stack $stack_{tag}$, close all the open tags (lines 17-22); in this case the algorithm also resets *cSize* to 0 and increase the part number *pId* by 1 (line 23); then reopen the same tags in reversal order in the new part and increase *cSize* with the length of each tag *length*(`currTagName`) for each reopened tag (lines 24-28). At the end of both cases, we increase the current part size with $length(qName)$ (line 29), then we pop the top element from the $stack_{pp}$ (line 30).

Going back to the case analysis on the modality got from the stack at the beginning of the algorithm, the remaining case is that of the projection modality. If $currStoredCase{=}false$ nothing happens. Otherwise, if $currStoredCase{=}true$ then we make a case analysis on the MATCH value (lines 31-37).

In the case of the current closing-tag being a non-terminal match for the projector path-set (MATCH=ok_nt), we increase the current size *cSize* with the length of close-tag qName $length(qName)$, and write it into the current part (lines 32-34).

If the current closing-tag is a terminal match $\tau$ (MATCH=ok_t), we increase the current size *cSize* with the length of this close-tag $length(qName)$, and write it into the current part (lines 35-37).

In `Characters` event (see Algorithm 5), we only increase the current part size *cSize* with the length of the text-content *str* of the current node $qName$ and write it into the current part.

To illustrate how the streaming projection-partitioning algorithms works, we will use the following example.

**Example 7** Consider the following iterative query $Q$:

$$Q = \texttt{for}\ \$x\ \texttt{in}\ /child :: doc/child :: a/child :: b\ \texttt{return}\ \$x/child :: c$$

and the input XML document $t$ reported in Figure 5.15 where we assume *maxSize*=12, and we have the following partitioning path $PP$ and the following projector $\tau$:

$$\begin{aligned} PP &= \{/child :: doc/child :: a/child :: b\} \\ \tau &= \{/child :: doc, /child :: doc/child :: a, /child :: doc/child :: a/child :: b, \\ &\quad /child :: doc/child :: a/child :: b/child :: c\} \end{aligned}$$

For our example, the processing starts with $Down(\tau)$, $Down(PP)$ and in partitioning modality. When the open tag `<doc>` (see Figure 5.16) is met, the algorithm performs a residuation on this tag and current partitioning path-set. We have MATCH=ok_nt meaning that the current $qName$ is a possible non-terminal match for the partitioning path-set (but it is not a target node). In this case, the algorithm adds the record `[doc,ok_nt,part,false]` at the top of $stack_{tag}$. The same process repeats for the next open-tag `<a>` which is non-terminal node for the partitioning

| Input document $t$ | Projected Part $t'_1$ | Projected Part $t'_2$ |
|---|---|---|
| <doc><br><a><b><c></c></b></a><br><a><f><c></c></f></a><br><a><b><c>to</c></b></a><br><a><f><d>go</d></f></a><br><a><b><c></c></b></a><br></doc> | <doc><br><a><b><c></c></b></a><br><a><b><br><c>co</c><br></b></a><br></doc> | <doc><br><a><br><b><br><c></c><br></b></a><br></doc> |

Figure 5.15: An input document $t$ and its projected parts $t'_1, t'_2$.

path-set. So the record [a,ok_nt,part,false] is added at the top of $stack_{tag}$ (see Figure 5.17).

| Input document $t$ | Projected Part $t'_1$ | $stack_{tag}$ [qName,MATCH, *Modality*,*isStored*] |
|---|---|---|
| <doc><br><a><b><c></c></b></a><br><a><f><c></c></f></a><br><a><b><c>to</c></b></a><br><a><f><d>go</d></f></a><br><a><b><c></c></b></a><br></doc> | <doc> | [doc,ok_nt,part,false] |

$$PP = \{/child :: a/child :: b\}$$
$$\tau = \{/child :: a, /child :: a/child :: b, /child :: a/child :: b/child :: c\}$$

Figure 5.16: Projection-partitioning processing: the current open-tag is <doc>.

| Input document $t$ | Projected Part $t'_1$ | $stack_{tag}$ [qName,MATCH, *Modality*,*isStored*] |
|---|---|---|
| <doc><br><a><b><c></c></b></a><br><a><f><c></c></f></a><br><a><b><c>to</c></b></a><br><a><f><d>go</d></f></a><br><a><b><c></c></b></a><br></doc> | <doc><br><a> | [a,ok_nt,part,false]<br>[doc,ok_nt,part,false] |

$$PP = \{/child :: b\}$$
$$\tau = \{/child :: b, /child :: b/child :: c\}$$

Figure 5.17: Projection-partitioning processing: the current open-tag is <a>.

The next event is for the open-tag <b> which residuation deems as a $PP$ terminal node. Here the algorithm visits the whole stack $stack_{tag}$ to write all ancestors open-tag relative to non-terminal matches and whose *isStored* value is $false$ into the current part $t'_1$. For each written open-tag the corresponding *isStored* value is set to *true*, and the whole record is kept into $stack_{tag}$. Also, the size of each stored open-tag is added to the current size *cSize*. Then we write the current open tag <b> into the current part $t'_1$ and add the following [b,ok_t,part,true] at the top of $stack_{tag}$. We then perform a new path alignment on both partitioning and projector

path-sets and put them in the corresponding stacks. Finally, we set *Modality=proj* to indicate that a projection phase begins for the subtree rooted at the current `<b>` node. Figure 5.18 illustrates some effects of previous steps.

| Input document $t$ | Projected Part $t'_1$ | $stack_{tag}$ [qName,MATCH, *Modality*,*isStored*] |
|---|---|---|
| `<doc>`<br>`<a><b><c></c></b></a>`<br>`<a><f><c></c></f></a>`<br>`<a><b><c>to</c></b></a>`<br>`<a><f><d>go</d></f></a>`<br>`<a><b><c></c></b></a>`<br>`</doc>` | `<doc>`<br>`<a><b>` | `[b,okt,part,true]`<br>`[a,ok_nt,part,true]`<br>`[doc,ok_nt,part,true]` |

$$PP = \{-\}\ ;\ cSize = 3 \qquad \tau = \{/child :: c\}$$

Figure 5.18: Projection-partitioning processing: the current open-tag is `<b>`.

The algorithm then goes to current *qName* which is `<c>` and deemed by residuation as a terminal $\tau$ node. Here the algorithm will keep the record `[c,ok_t,proj,true]` at the top of $stack_{tag}$; write the current tag into the current part $t'_1$; performs $Down(\tau)$ and keep the result in $stack_\tau$; increase the projection size *Size* with the $length(c)$. Effects are illustrated in Figure 5.19.

| Input document $t$ | Projected Part $t'_1$ | $stack_{tag}$ [qName,MATCH, *Modality*,*isStored*] |
|---|---|---|
| `<doc>`<br>`<a><b><c></c></b></a>`<br>`<a><f><c></c></f></a>`<br>`<a><b><c>to</c></b></a>`<br>`<a><f><d>go</d></f></a>`<br>`<a><b><c></c></b></a>`<br>`</doc>` | `<doc>`<br>`<a><b><c>` | `[c,okt,proj,true]`<br>`[b,okt,part,true]`<br>`[a,ok_nt,part,true]`<br>`[doc,ok_nt,part,true]` |

$$PP = \{-\}\ ;\ cSize = 4 \qquad \tau = \{-\}$$

Figure 5.19: Projection-partitioning processing: the current open-tag is `<c>`.

Now we have the close-tag `</c>`. Here the algorithm performs the following tasks: pop the top element of $stack_{tag}$ and keep `[c,okt,proj,true]` in the following variables $currTag$, MATCH, $currModality$, $currStoredCase$ values; pop the top element of $stack_\tau$. Then the algorithm checks values for $currModality$ and $currStoredCase$. In the current case, we have *proj* and *true*. The process is in projection modality so it increases the current part size *cSize* with *length*(`</c>`), and writes the close-tag `</c>` into the current part $t'_1$. Effects are illustrated in Figure 5.20.

Then the process goes to the next *qName* which is `</b>`. the algorithm pops the top element of $stack_{tag}$. Here we have a close-tag of a *PP* target node (MATCH=ok_t, *currModality=part* and *currStoredCase=true*). In this case the

| Input document $t$ | Projected Part $t'_1$ | $stack_{tag}$ [qName,MATCH, *Modality*,*isStored*] |
|---|---|---|
| <doc><br><a><b><c></c></b></a><br><a><f><c></c></f></a><br><a><b><c>to</c></b></a><br><a><f><d>go</d></f></a><br><a><b><c></c></b></a><br></doc> | <doc><br><a><b><c></c> | [b,okt,part,true]<br>[a,ok_nt,part,true]<br>[doc,ok_nt,part,true] |

$PP = \{-\}$ ; $cSize = 5$
$\tau = \{/child :: c\}$

Figure 5.20: Projection-partitioning processing: the current close-tag is `</c>`.

algorithm will perform the following tasks: increase the current part size *cSize* with $length(qName)$; pop the top element of $stack_{pp}$; put *Modality=part* to declare that the parsing of the current target node subtree is finished. Then the algorithm checks whether the current size *cSize* plus the projection size *Size* exceed the maximal size *maxSize*; this is not the case (current size is 6), so the algorithm add *Size* to the current *cSize*, and write the current close-tag `</b>` into the current part $t'_1$, then go to the next $qName$ `</a>`. Effects are illustrated in Figure 5.21.

| Input document $t$ | Projected Part $t'_1$ | $stack_{tag}$ [qName,MATCH, *Modality*,*isStored*] |
|---|---|---|
| <doc><br><a><b><c></c></b></a><br><a><f><c></c></f></a><br><a><b><c>to</c></b></a><br><a><f><d>go</d></f></a><br><a><b><c></c></b></a><br></doc> | <doc><br><a><b><c></c></b> | [a,ok_nt,part,true]<br>[doc,ok_nt,part,true] |

$PP = \{/child :: b\}$ ; $cSize = 6$
$\tau = \{/child :: b, /child :: b/child :: c\}$

Figure 5.21: Projection-partitioning processing: the current close-tag is `</b>`.

Now we have qName `</a>` and `[a,ok_nt,part,true]` the top element of $stack_{tag}$. Here we have a non-terminal *PP* node (MATCH=ok_nt and *currModality=part*), so the algorithm will increase *cSize* with $length(qName)$; write it in the current part $t'_1$; and finally pop the top element from $stack_{pp}$, and then goes to the next node (see Figure 5.22).

The process parses an open tag `<a>` and repeats the same previous treatment. It pushes `[a,ok_nt,part,false]` on the $stack_{tag}$, then goes to the next node, whose $qName$ is `<f>` which does not match *PP*. Here the algorithm prunes out this $qName$ and does not write it into the current part $t'_1$, also no path alignments will be done. It only keeps the following values `[f,fail,part,false]` at the top of $stack_{tag}$, and the current size *cSize* does not increase. The algorithm continues in the same way for the next node `<c>`, performs the same previous treatment and prune it out. The only

| Input document $t$ | Projected Part $t'_1$ | $stack_{tag}$ [qName,MATCH, *Modality*,*isStored*] |
|---|---|---|
| <doc><br><a><b><c></c></b></a><br><a><f><c></c></f></a><br><a><b><c>to</c></b></a><br><a><f><d>go</d></f></a><br><a><b><c></c></b></a><br></doc> | <doc><br><a><b><c></c></b></a> | [doc,ok_nt,part,true] |

$$PP = \{/child :: a/child :: b\} \; ; \; cSize = 7$$
$$\tau = \{/child :: a/child :: b, /child :: a/child :: b/child :: c\}$$

Figure 5.22: Projection-partitioning processing: the current close-tag is </a>.

thing that the algorithm will do is to add the following record [c,fail,part,false] at the top of $stack_{tag}$. Effects are illustrated in Figure 5.23.

| Input document $t$ | Projected Part $t'_1$ | $stack_{tag}$ [qName,MATCH, *Modality*,*isStored*] |
|---|---|---|
| <doc><br><a><b><c></c></b></a><br><a><f><c></c></f></a><br><a><b><c>to</c></b></a><br><a><f><d>go</d></f></a><br><a><b><c></c></b></a><br></doc> | <doc><br><a><b><c></c></b></a> | [c,fail,part,false]<br>[f,fail,part,false]<br>[a,ok_nt,part,false]<br>[doc,ok_nt,part,true] |

$$PP = \{/child :: b\}$$
$$\tau = \{/child :: b, /child :: b/child :: c\}$$

Figure 5.23: Projection-partitioning processing for parsing the subtree <a><f><c>.

As illustrated in Figure 5.24, for the following nodes </c></f></a>, the algorithm just delete their information from $stack_{tag}$ and ignore writing them in the current part $t'_1$, because their relative open-tags did not match $PP$ (their $\sigma$ value is $false$).

The process continues in the same way for the nodes <a><b><c>to</c> until reading the close-tag of the target node </b>, here the algorithm checks if the current size *cSize* plus the projection size *Size* is more than the maximal size *maxSize*=12. In our case this check is positive, so the algorithm here close all open tags $stack_{tag}$, in backward order, in the current part $t'_1$, reset *cSize* to the value 0, and increase *pId* with 1 to become 2 in our example. Then the algorithm starts a new part $t'_2$, flushes open-tags in $stack_{tag}$ in the new part, according to document order. Effects are shown in Figure 5.25).

Next steps of the process are similar. The process ends up with two different projected parts $t'_1$ and $t'_2$, each one contains only nodes that are sufficient to evaluate $Q$, as illustrated in Figure 5.26.

□

| Input document<br>$t$ | Projected Part<br>$t'_1$ | $stack_{tag}$<br>[qName,MATCH, *Modality*,*isStored*] |
|---|---|---|
| <doc><br><a><b><c></c></b></a><br><a><f><c></c></f></a><br><a><b><c>to</c></b></a><br><a><f><d>go</d></f></a><br><a><b><c></c></b></a><br></doc> | <doc><br><a><b><c></c></b></a> | [doc,ok_nt,part,true] |

$$PP = \{/child :: a/child :: b\}$$
$$\tau = \{/child :: a/child :: b, /child :: a/child :: b/child :: c\}$$

Figure 5.24: Projection-partitioning processing for parsing the following close-tags </c></f></a>.

| Input document<br>$t$ | Projected Part<br>$t'_1$ | Projected Part<br>$t'_2$ | $stack_{tag}$<br>[qName,MATCH, *Modality*,*isStored*] |
|---|---|---|---|
| <doc><br><a><b><c></c></b></a><br><a><f><c></c></f></a><br><a><b><c>to</c></b></a><br><a><f><d>go</d></f></a><br><a><b><c></c></b></a><br></doc> | <doc><br><a><b><br><c></c><br></b></a><br><a><b><br><c>to</c><br></b></a><br></doc> | <doc><br><a> | [a,ok_nt,part,false]<br>[doc,ok_nt,part,true] |

$$PP = \{/child :: b\}\ ;\ cSize = 14 \geq maxSize$$
$$\tau = \{/child :: b, /child :: b/child :: c\}$$

Figure 5.25: Parsing the subtree <a><b><c>to</c></b></a>, and create a new projected part $t'_2$.

After generating the projected parts, we evaluate our iterative query $Q$ on each part $t_i^{'}$'s to obtain the results $Q(t_i^{'})$'s, and we use a simple concatenation to merge all partial results, to produce the final result $Q(t)$, where: $Q(t)=Q(t'_1).Q(t'_2)$.

## 5.7 EXPERIMENTAL EVALUATION

In the previous sections, we described a novel XML data partitioning scheme that, given a query $Q$ and an input document $t$, partitions $t$ in a set of fragments $\{t_1, \ldots, t_\kappa\}$ so that $Q(t)$ is equivalent to the concatenation of $Q(t_1)$, ..., $Q(t_\kappa)$. When this partitioning scheme is applicable, it can improve the scalability of existing main-memory engines, as it allows the system to process one part at time.

In this section we present an experimental evaluation of the proposed approach. We will first show that the proposed algorithm significantly improves the scalability of a popular main-memory query engine. Then, we will show that partitioning, when combined with a projection algorithm, introduces little overhead with respect to the projection algorithm. Finally, we will experimentally analyze the relation between the overall performance of the system and the actual value of *maxSize* (the

| Input document $t$ | Projected Part $t'_1$ | Projected Part $t'_2$ |
|---|---|---|
| <doc> | <doc> | <doc> |
| <a><b><c></c></b></a> | <a><b><c></c></b></a> | <a><f><d>go</d></f></a> |
| <a><f><c></c></f></a> | <a><f><c></c></f></a> | <a> |
| <a><b><c>to</c></b></a> | <a><b> | <b> |
| <a><f><d>go</d></f></a> | <c>to</c> | <c></c> |
| <a><b><c></c></b></a> | </b></a> | </b></a> |
| </doc> | </doc> | </doc> |

Figure 5.26: Final projected parts $t'_1, t'_2$ produced by projection+partitioning algorithm.

maximum part size).

### 5.7.1 Experimental Setup

We implemented our partitioning algorithm, as well as a standard path-based projection algorithm, in Java 6 and tested their behavior on the XMark benchmark [SWK+02a]. In particular, we evaluated our system on XMark documents by relying on two widely used XQuery engines, Saxon [sax] and Qizx [qiz]. While Saxon is an engine supporting all main W3C standards for XML manipulation and schema validation, Qizx is specialized on querying and updating, and offers powerful optimization techniques. However, we will see that even with the use of standard path-based projection, these systems do not scale up in terms of document size (other powerful systems like BaseX [bas] have quite similar performances). Our test results show that our technique overcome this limitation for iterative queries, as it allows these engines to scale up to arbitrary document sizes.

All experiments were performed on a 2.53 Ghz Intel Core 2 Duo machine (4 GB main memory) running Mac OSX 10.6.8. All XML documents were loaded on an external USB2 7200 rpm 1 TB disk unit.

To avoid the perturbations introduced by system activity, we ran each experiment ten times, discarded the best and the worst performance, and computed the average of the remaining results.

### 5.7.2 Tests Results

We used documents whose size ranges from 1GB to 5GB for Saxon and from 1GB to 9GB for Qizx. Concerning the threshold value *maxSize*, we set ($\sim$ 25MB) for Saxon, and ($\sim$ 95.36 MB) for Qizx. These differences in terms of memory and part sizes are due to differences of performance between the two engines in terms of memory management. For both Saxon and Qizx we allocated 512MBs for main memory of the Java Virtual Machine.

Concerning queries, we considered XMark queries $Q_1$–$Q_5$, $Q_{10}$, $Q_{14}$–$Q_{20}$, (see Section A.1 of Appendix A) which form the iterative core of XMark [SWK+02a]. Also, we wrote the following three new XMark queries ($N_1, N_2$ and $N_3$):

```
N1 = let $auction := doc("xmark.xml") return
     for $i in $auction/site//item
     where $i/location/text() = "UnitedStates"
     return
            <itemInfo name ="$i/name/text()">
                      <paymentWay>$i/payment/text()</paymentWay>
                      <shippingWay>$i/shipping/text()</shippingWay>
                      <moreInfo>$i/description</moreInfo>
                      <mailboxInfo>$i/mailbox</mailboxInfo>
            </itemInfo>
```

```
N2 = let $auction := doc("xmark.xml") return
     for $i in $auction/site//description
     where contains(string(exactly-one($i)), "gold")
     return $i/node()
```

```
N3 = let $auction := doc("xmark.xml") return
     for $i in $auction/site//item
     where empty($i/payment/text())
     return
            <item id="$i/@id" name="$i/name/text()" location="$i/location/text()">
                  {$i/description, $i/mailbox}
            </item>
```

and two queries $(D_1, D_2)$ to be evaluated on a 800MB DBLP document [ver11]:

```
D1 = let $auction := doc("dblp.xml") return
     for $a in $auction/dblp//author
     return
            <AuthorName> {$a/text()} </AuthorName>
```

```
D2 = let $auction := doc("dblp.xml") return
     for $a in $auction/dblp/node()
     return
            <item>{$a/author, $a/title, $a/booktitle, $a/year}</item>
```

### 5.7.3 Experiments

In our first experiment we evaluate and compare scalability of both Saxon and Qizx. We consider a 1GB document and a 5GB document for Saxon, and 2GB and 9GB for Qizx test. For each document and for each query, we compare total execution time obtained with only projection with that obtained with partitioning (and projection). Total execution time includes the overall time required by the system to partition and/or project the input document, to evaluate the input query on the projection/partition, and (in the case of partitioning) to concatenate the final results.

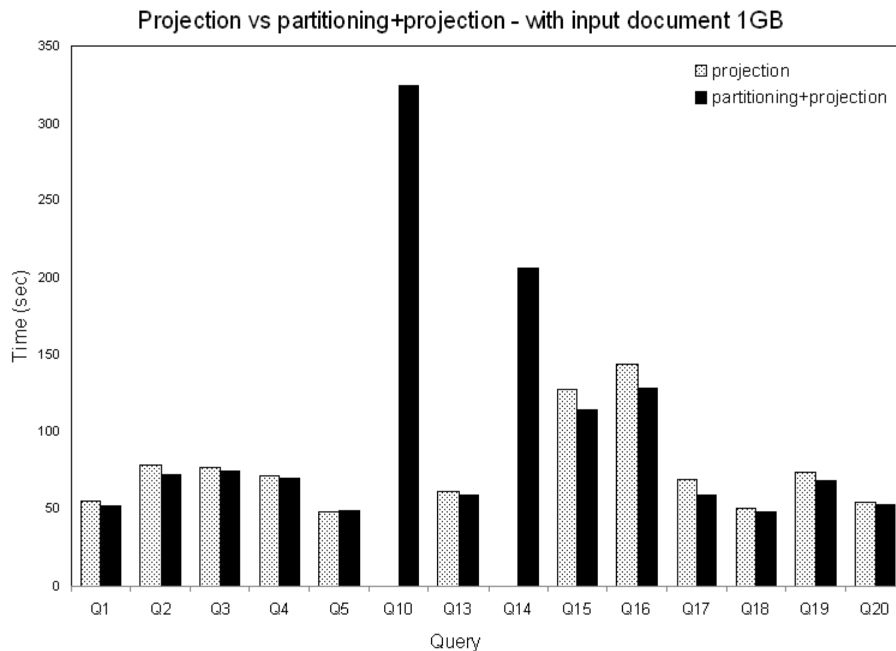



Figure 5.27: Projection vs partitioning+projection - with input document 1GB - using Saxon.

We first comment on results obtained by using Saxon. When projection only is used, this system starts showing limitations even for a 1GB document, for which queries Q10 and Q14 could not be executed due to memory failure. As shown in Figure 5.27, our partitioning technique enables execution of all XMark iterative queries, with no overhead (absence of overhead is due to the combination of projection and partitioning).

As illustrated in Figure 5.28, for the 5GB document, improvements of our partitioning technique are substantial: 8 queries could not be executed with only projection, while all queries are executed by means of partitioning.

Figure 5.29 reports execution times obtained with Saxon and partitioning, for all considered document size. As shown by the figure, our technique scales up and has a linear behavior.

| input in GB | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| proj in GB | 593.92 MB | 0.98 | 1.48 | 1.97 | 2.50 |

Table 5.3: Global projections size.

Concerning Saxon, we also compared projection vs partitioning for a workload comprising all XMark iterative queries. Actually we performed this experiment by using a global projection, containing all paths extracted from XMark iterative queries, and starting from 1GB until 5GB. By using only projection, already for a

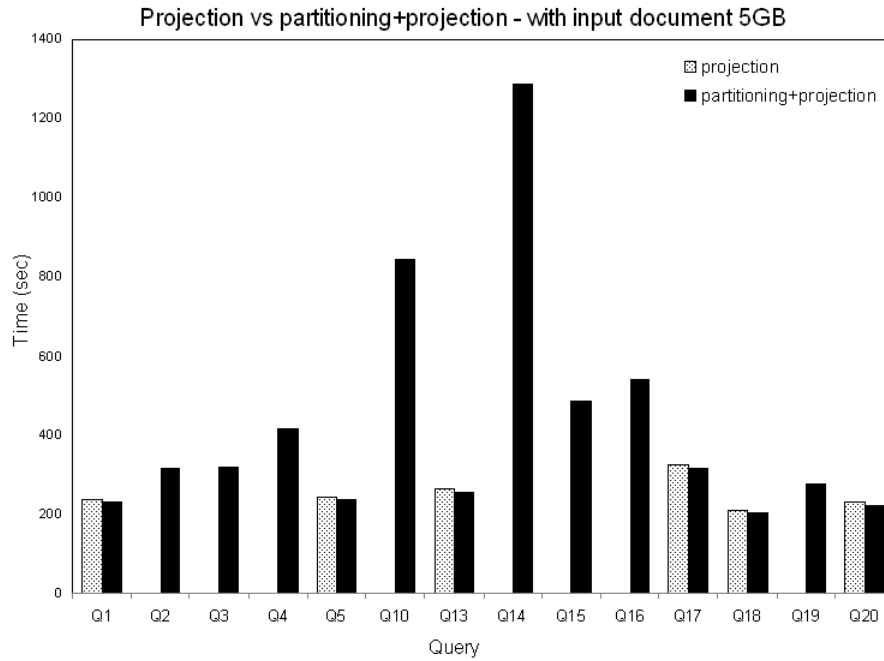



Figure 5.28: Projection vs partitioning+projection - with input document 5GB - using Saxon.

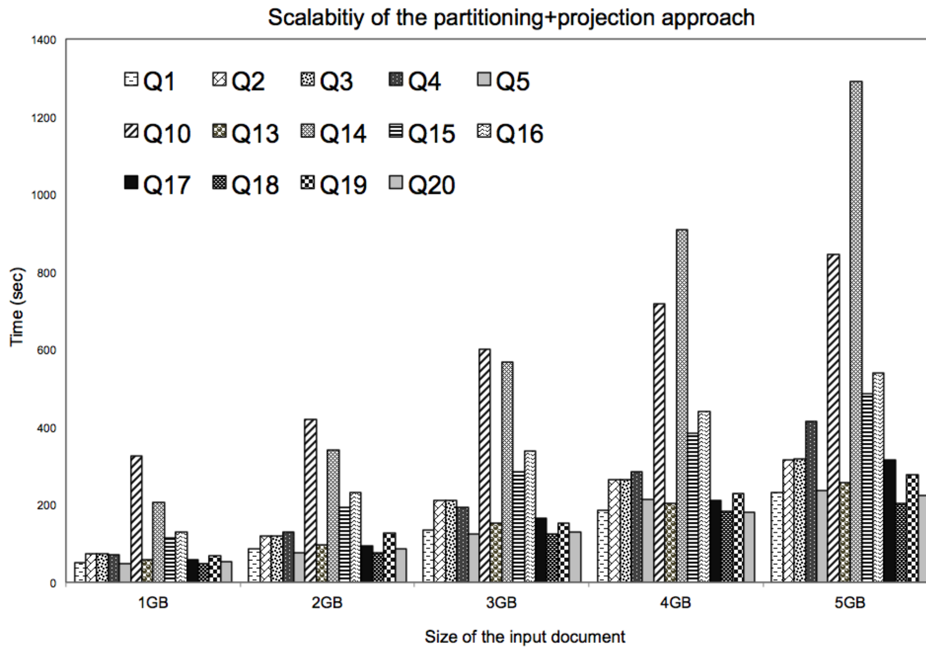



Figure 5.29: Scalability of the partitioning approach - using Saxon.

1GB document we could not run the workload as the projected document was too large for Saxon. Table 5.3 illustrates the size of these global projected documents.

Fortunately, by using our partitioning technique, we were able to run the workload for each size, as illustrated in Figure 5.30. Again, the technique features a linear behavior.

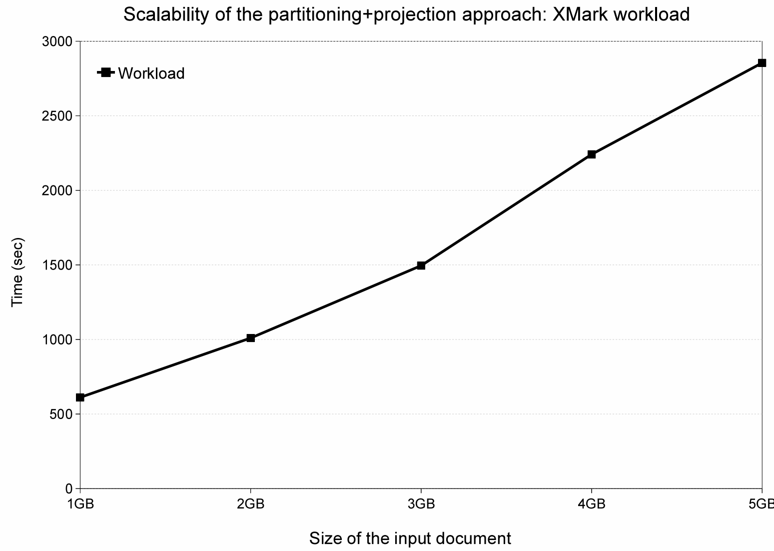



Figure 5.30: Scalability of the partitioning approach: workload - using Saxon.

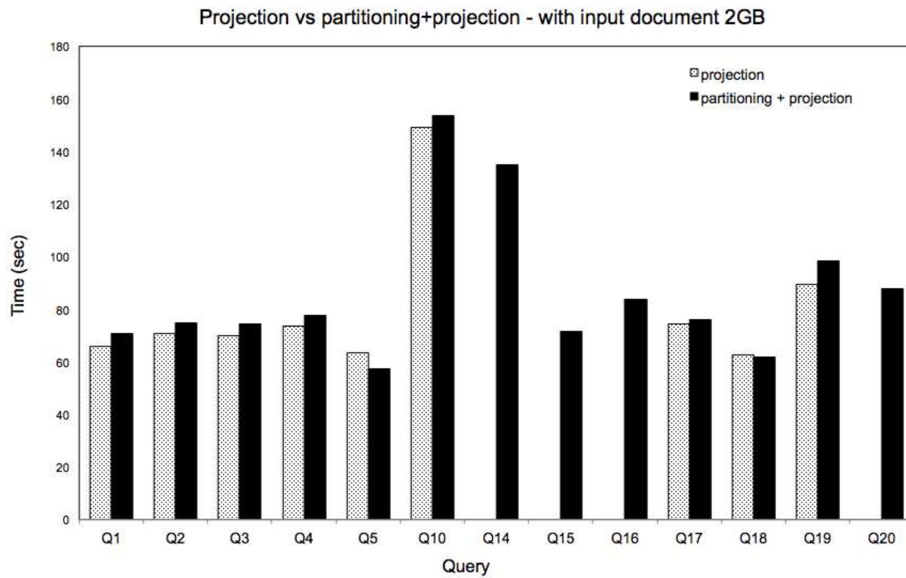



Figure 5.31: Projection vs partitioning - with input document 2GB - using Qizx.

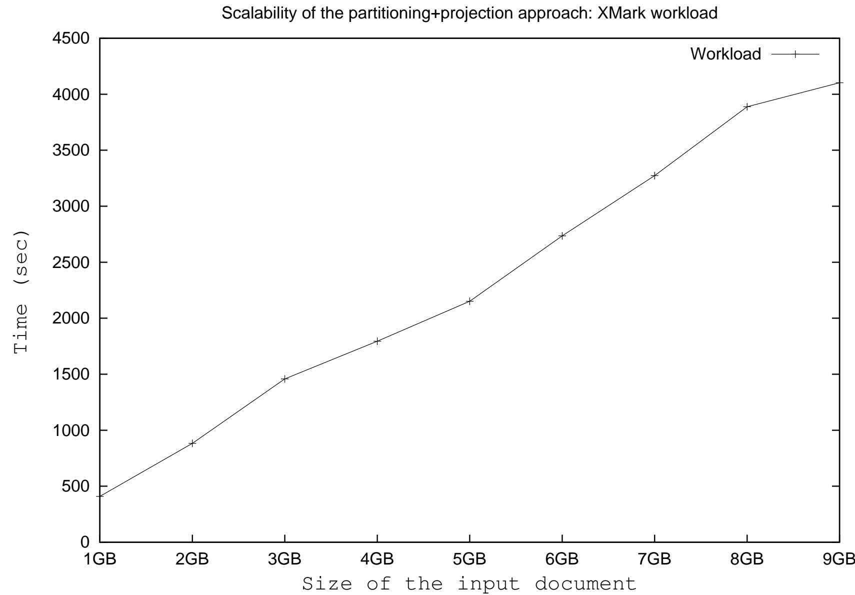



Figure 5.32: Scalability of the partitioning approach: workload - using Qizx.

Concerning Qizx, we performed the same kind of experiments. As already said, Qizx is specialized on querying and updating, and this has permitted the adoption of efficient document representation in main-memory. For a 2GB document, Qizx does not exhibit any limitation with the use of projection. As can be seen in Figure 5.31, again no overhead is exhibited by our partitioning technique.

For the 9GB document, things are different, see Figure 5.33. Five queries could not be executed with the sole use of projection. Instead, our partitioning technique enabled the processing of all queries.

Results about scalability by using Qizx are reported in Figure 5.39. Again test results show that our technique scales up with a linear behavior.

Concerning Qizx and scalability on the workload of XMark iterative queries, results are reported in Figure 5.32. As the figure illustrates, partitioning scales up without problems and still in a linear fashion. We repeated this experiment by using projection only; however, we got no experimental results, as, even in the case of the 1GB document, the projected documents were too big to be handled by the query engine.

### 5.7.4 Experiments on Queries $\{N_1,\ N_2,\ N_3\}$, and $\{D_1,\ D_2\}$

In the previous chapter, we presented queries $N_1, N_2$ and $N_3$ as examples of queries requiring large projections of XMark documents. With the same aim, we also presented queries $D_1$ and $D_2$ over DBLP data.

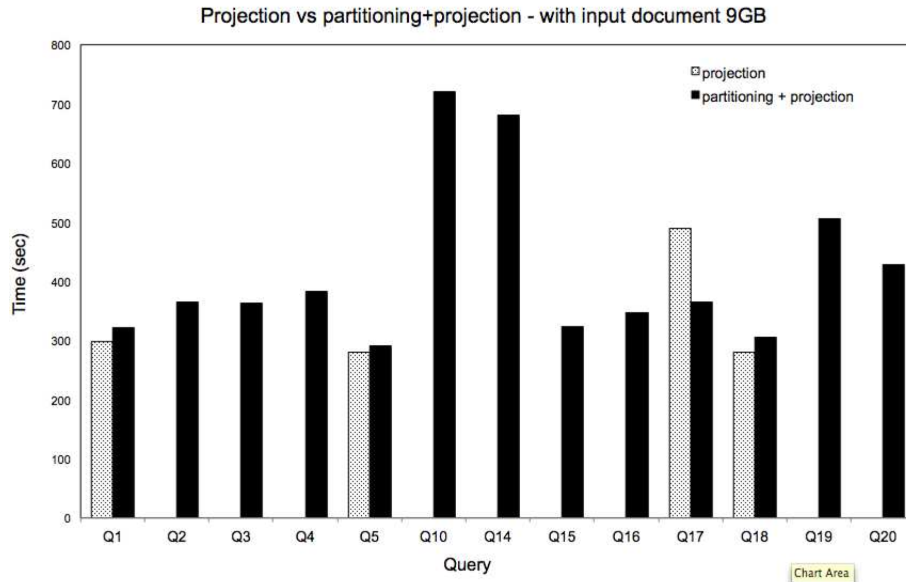



Figure 5.33: Projection vs partitioning - with input document 9GB - using Qizx.

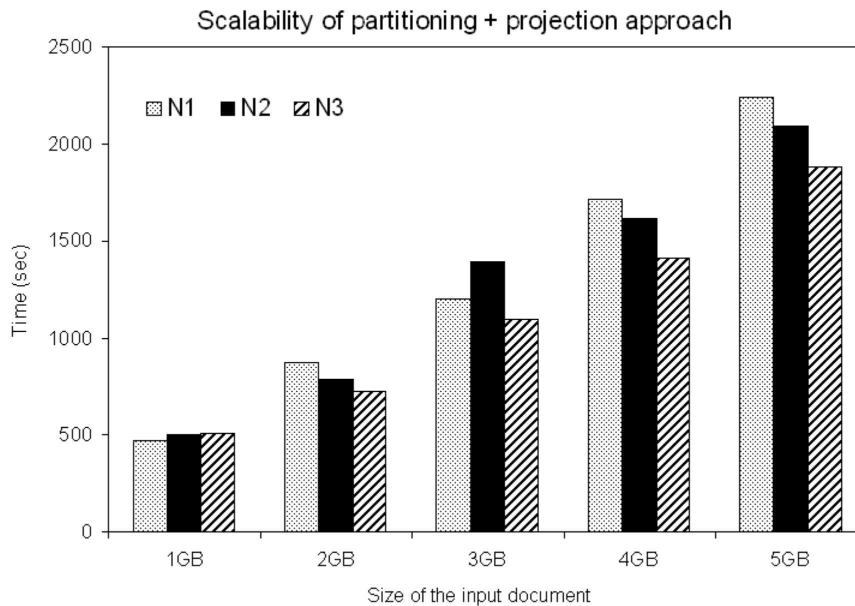



Figure 5.34: Scalability of the partitioning approach - using Saxon.

Actually, we evaluate our partitioning/projection technique on the queries $N_1, N_2$ and $N_3$. We consider a 1GB document until 5GB document for Saxon test. As illustrated in Figure 5.34, our partitioning technique enables executions of these three queries with no overhead. It is worth noticing that these queries could not be executed with only projection due to their large projected documents. As

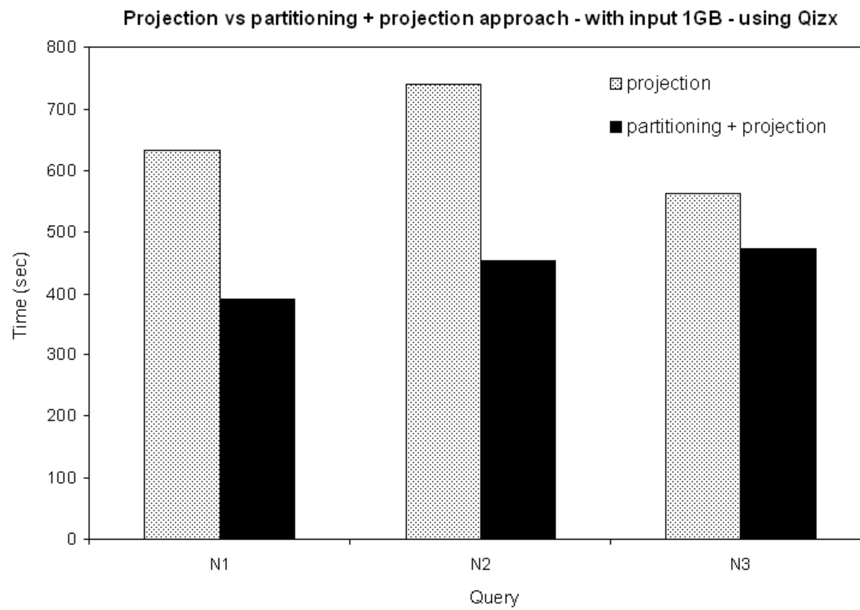



Figure 5.35: Projection vs partitioning - with input document 1GB - using Qizx.

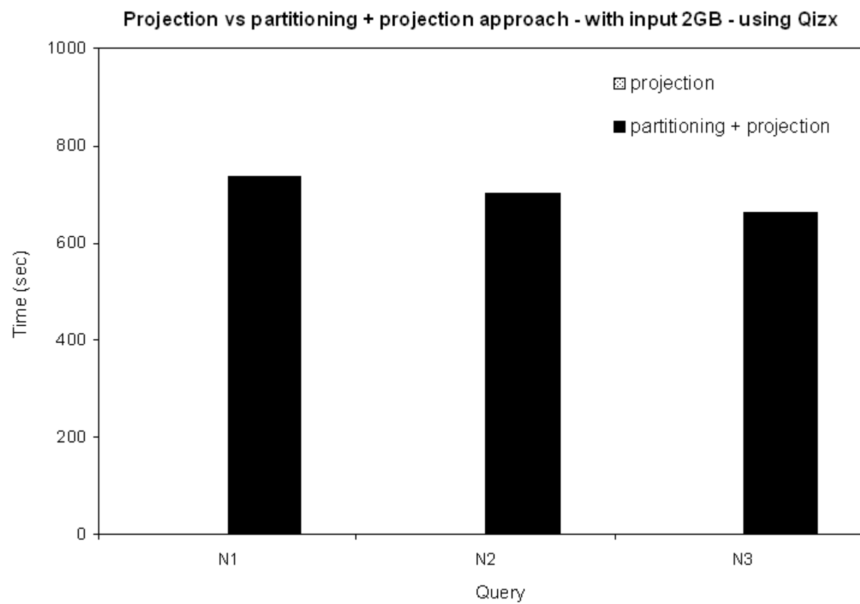



Figure 5.36: Projection vs partitioning - with input document 2GB - using Qizx.

shown by the figure, our technique scales up and has a linear behavior.

We repeat the same kind of previous experiments for our queries $N_1, N_2, N_3$ with Qizx. As illustrated in Figure 5.35, for a 1GB document Qizx does not exhibit any limitation with the use of only projection, but query evaluation with partitioning

resulted much faster. This can be explained by the fact that handling a big projection entails some overhead which disappears when handling small parts. For the 2GB document, the three queries could not be executed with the sole use of projection (see Figure 5.36). Instead, our technique enabled the processing of these three queries. Tests results on scalability from 1GB to 5GB are illustrated in Figure 5.37. The linear behavior previously observed is confirmed once again.

We then performed experiments on queries $D_1$ and $D_2$ on a 800MB DBPL document, by using on Saxon and Qizx, Table 5.4 reports the results for both queries by using projection only, and by using our partitioning/projection technique. In this cases Qizx was able to process both queries with only projection, but Saxons failed. With partitioning, Saxon was able to execute both queries.

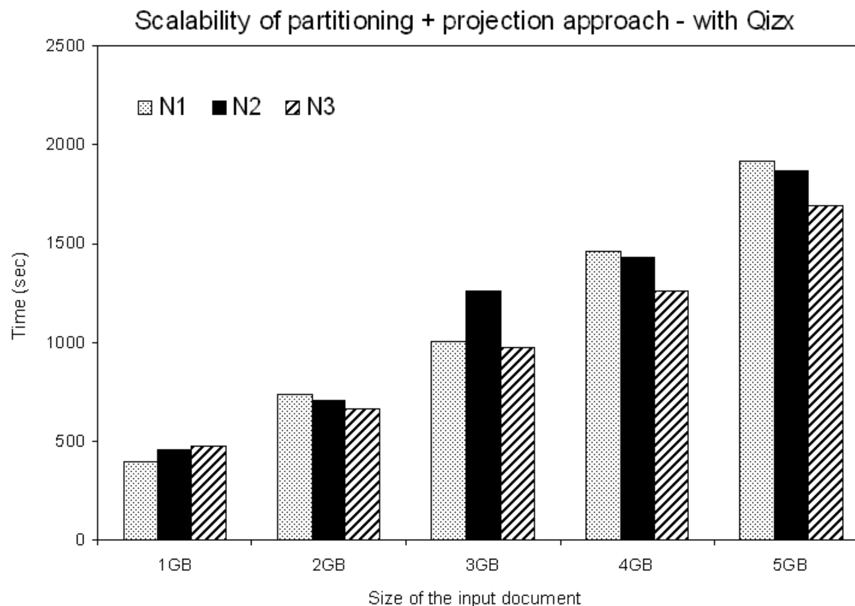



Figure 5.37: Scalability of the partitioning approach - using Qizx.

| **Performance of the partitioning approach on DBLP database** | | |
|---|---|---|
| Query | Total Time (sec) with Saxon | Total Time (sec) with Qizx |
| $D_1$ | 249.23 | 208.47 |
| $D_2$ | 409.62 | 358.17 |

Table 5.4: Qizx and Saxon performances with the partitioning approach - on DBLP database.

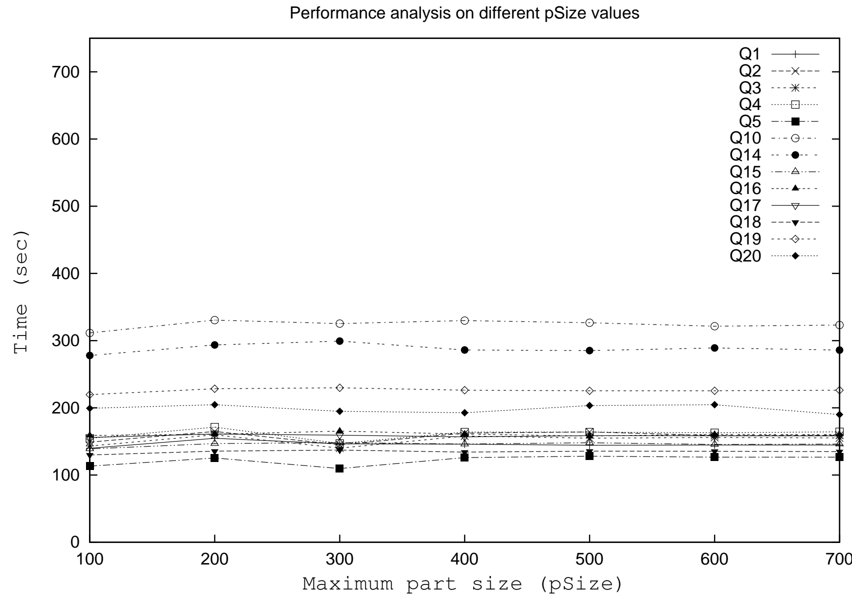



Figure 5.38: Relation between *maxSize* (`pSize`) and the performance of the partitioning approach.

In our final experiment we investigate the relationship between *maxSize* and the processing time, i.e., the impact of different values of *maxSize* on the overall performance of our approach. To this end, we evaluate all the queries in the iterative core on the 4GB document and vary *maxSize* from 100000000 bytes to 700000000 bytes ($\sim$ 668 MB). This kind of tests is quite time consuming, so we focused on Qizx, but we expect similar results for Saxon (by considering smaller part sizes).

The results are shown in Figure 5.38. Surprisingly enough, we can observe that the value of *maxSize* has no significant impact on the overall performance. This could seem counter-intuitive, as bigger values of *maxSize* should decrease the total number of bytes written to disk. Actually this test reveals that our technique can be used even in contexts of high limitations concerning available memory. For such a context, small *maxSize* values can be used without compromising performance.

### 5.7.5 Summing Up

To summarize, our experiments show that existing main-memory engines do not scale with respect to document size. It is worth observing that this remains true even for bigger sizes of the main-memory of the Java Virtual Machine. Bigger memory would only imply a shift of the maximal document size that can be handled.

Instead, our experiments prove that the partitioning approach scales beautifully and is only slightly slower than the projection approach. To make experiments feasible in a reasonable time we considered 5GB and 9GB as the maximal size of documents. However, since the *maxSize* can be tuned to fit in the available main memory, we have that partitioning scales for arbitrary sizes.

We also discovered that the actual value of *maxSize* has no significant impact on the overall performance; this suggests that *maxSize* can be tuned by looking only at available main-memory.

## 5.8 CONCLUSION

In this chapter we presented a novel projection-partitioning technique for XML document. This technique generalizes existing path-based approaches, and applies to a large class of queries.

The proposed approach analyzes an input query and, if the query is *iterative*, extracts all the relevant paths and uses them to project and partition the input document. As shown in our experimental evaluation, by executing the input query on each part and combining the partial results, existing main-memory query engines can process an iterative query on very large input documents.

As each part can be queried independently by a distinct instance of the query engine, we are currently investigating potential applications of the proposed approach to cloud computing environments.

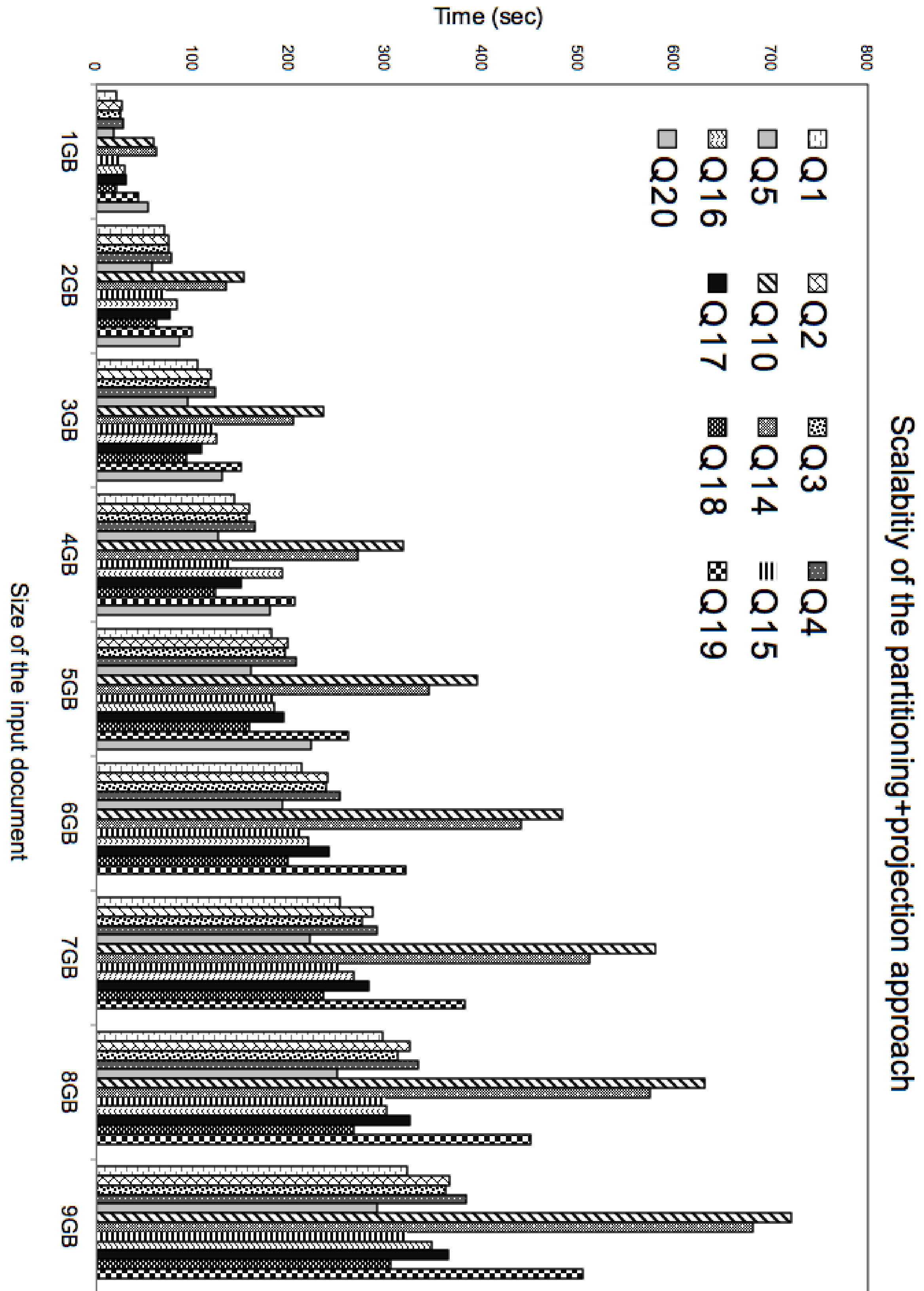



Figure 5.39: Scalability of the partitioning approach - using Qizx.

Chapter 6

# Partitioning for XQuery Updates

## Contents


**6.1 Overview** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **102**

**6.2 preliminaries** . . . . . . . . . . . . . . . . . . . . . . . . . . . **103**

6.2.1 Simple XQuery Update Facilities (SXUF) . . . . . . . . . . . 103

**6.3 iterative updates** . . . . . . . . . . . . . . . . . . . . . . . . . **105**

**6.4 partitioning for iterative updates** . . . . . . . . . . . . . **115**

6.4.1 Partitioning Algorithm . . . . . . . . . . . . . . . . . . . . . 116

6.4.2 Fusion Operation . . . . . . . . . . . . . . . . . . . . . . . . 119

**6.5 streaming implementation** . . . . . . . . . . . . . . . . . . . **121**

6.5.1 Partitioning . . . . . . . . . . . . . . . . . . . . . . . . . . . 121

6.5.2 Fusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132

**6.6 experimental evaluation** . . . . . . . . . . . . . . . . . . . . **135**

6.6.1 Experimental Setup . . . . . . . . . . . . . . . . . . . . . . . 136

6.6.2 Tests Results . . . . . . . . . . . . . . . . . . . . . . . . . . 136

6.6.3 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . 138

6.6.4 Summing Up . . . . . . . . . . . . . . . . . . . . . . . . . . . 141

**6.7 conclusion** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **143**


In this chapter, we present a partitioning technique for XQuery Update Facility (XUF). As for queries, partitioning enables the treatment of large documents, that could not be updated by using existing main-memory engines [qiz, exi, bas], even by using the existing projection-based technique [BBC+11].

In this chapter, we characterize a class of updates, called *iterative* updates, for which a partitioning-based evaluation is possible: first documents are partitioned, then parts are updated independently, and finally updated parts are merged by using a *fusion* operation in order to obtain the final updated document.

To recognize iterative updates we rely again on a path-based analysis. Extracted paths will be also used for partitioning. Differently from queries, partitioning will not rely on projection, and paths will be used to ensure that each part contains all that is needed for each single update operation. Projection is not used in order to avoid complex merge operations on updated parts, in order to recover pruned subtrees when constructing the global updated document. Effectiveness of the proposed approach is shown by means of extensive experiments comparing our partitioning-based

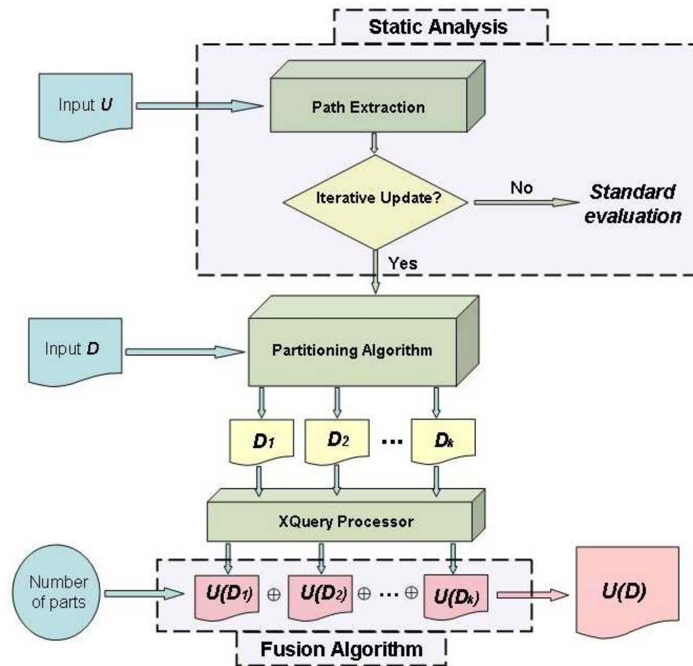



Figure 6.1: Partitioning update scenario.

approach with the projection-based one proposed in [BBC+11, MS03]. It is worth mentioning that this last one is type-based, and is the only available projection-based approach for updates.

The chapter is structured as follows. In Section 6.2, we introduce a few preliminary notations about the update query language used in this approach, then we provide our path extraction function. In Section 6.3, we formally describe *iterative* updates. Next, in Section 6.4, we present our *partitioning* technique for iterative updates, all formal definitions and DOM-based specifications of both partitioning and fusion. In Section 6.5 we provide all streaming algorithms (partitioning and fusion) used to perform our partitioning update scenario. The chapter ends with test results in Section 6.6 and some conclusive remarks in Section 6.7.

## 6.1 Overview

In order to simplify the presentation and the formal treatment of our static analysis, we focus on a particular class of *simple* XUF updates, SXUF for short. In a nutshell, restrictions posed on the XUF fragment are the following ones. Only downward XPath axes *self*, *child* and *dos* are allowed. Concerning update operations, source and target expressions use a simple class of queries. These restrictions are mild enough to capture a wide class of updates used in practice. More details will follow.

The main steps of our partitioning scenario for an SXUF update $U$ and an input XML document $t$ are the following ones:

- **Path Extraction:** We extract paths $\tau$ and target paths $\tau_{ap}$ from an SXUF update $U$.

- **Static analysis:** We use the sets of extracted paths $\tau, \tau_{ap}$ to check whether the update $U$ is iterative or not.

- **Partitioning and Updating:** If $U$ is iterative, we use the partitioning technique to create several parts $t_1, t_2, \cdots, t_\kappa$. As for queries, partitioning is so that each $t_i$ is a well-formed XML document. For optimization purposes, by using information coming from target paths in $U$, the partitioning process also flags those parts that do not need to be updated as they contain no target node. We then update each part that needs to, and obtain the documents $t'_1, t'_2, \cdots, t'_\kappa$, where either $t'_i = U(t_i)$ or $t'_i = t_i$ (if this part is not to be updated) for $i = 1 \ldots \kappa$. For simplicity, in the formal treatment made in the sequel we assume that each part is to be updated, while we will come back to this assumption in Section 6.5, when discussing implementation issues.

- **Fusion:** After producing the updated parts, we use a fusion operation $\oplus$ to concatenate them. During the fusion process, each $U(t_i)$ is processed in a streaming fashion, one at a time.

Figure 6.1 illustrates the whole mechanism of our partitioning update scenario.

It is worth noticing that one of the contributions of this approach is to provide streaming algorithms for performing partitioning and fusion. As already anticipated, the partitioning process is able to flag parts that do not need to be updated, thus saving time when updating only parts that need to.

## 6.2 PRELIMINARIES

### 6.2.1 Simple XQuery Update Facilities (SXUF)

The grammar of SXUF is illustrated in Figure 6.2. This language comprises `for`, `let` and `return` clauses as well as `if-then-else` conditional statement. Also, SXUF contains all *elementary* XUF update expressions (delete, insert, rename and replace).

The main restrictions behind SXUF are the following ones:

- All query paths $P$ and target paths $P_{\mathrm{tg}}$ used in the syntax of SXUF language obey the same grammars illustrated in Section 5.2 of Chapter 5, which we recall below:

| | | | | |
|---|---|---|---|---|
| Target Path | $P_{\text{tg}}$ | ::= | $/P \mid x/P$ | |
| Simple Query | $Q_{\text{s}}$ | ::= | $() \mid b \mid /P \mid x/P$ | |
| | | \| | <a>$Q_{\text{s}}$</a> \| $Q_{\text{s}}, Q_{\text{s}}$ | |
| Target Position | *Pos* | ::= | `as first into` \| `as last into` | |
| | | \| | `before` \| `after` | |
| Node Case | $N$ | ::= | `node` \| `nodes` | |
| Updates | $U$ | ::= | *delete* $N$ $P_{\text{tg}}$ | {deletion} |
| | | \| | *rename* $N$ $P_{\text{tg}}$ *as* a | {a is text-value} |
| | | \| | *replace* $N$ $P_{\text{tg}}$ *with* $Q_{\text{s}}$ | {replacement} |
| | | \| | *insert* $N$ $Q_{\text{s}}$ *Pos* $P_{\text{tg}}$ | {insertion} |
| | | \| | $U, U$ | {sequence} |
| | | \| | `if` $Q$ `then` $U$ `else` $U$ | {conditional} |
| | | \| | `for` $x$ `in` $Q$ `return` $U$ | {iteration} |
| | | \| | `let` $x := Q$ `return` $U$ | {let-binding} |

Figure 6.2: Syntax of SXUF.

$$
\begin{array}{rcl}
P & ::= & /\mathit{Step} \mid P/\mathit{Step} \\
\mathit{Step} & ::= & \mathit{Axis} :: \mathit{NT} \\
\mathit{Axis} & ::= & \mathit{self} \mid \mathit{child} \mid \mathit{dos} \\
\mathit{NT} & ::= & \mathrm{a} \mid \mathit{node}() \mid \mathit{text}()
\end{array}
$$

- Simple query expressions $Q_{\text{s}}$, used as source expression for in `replace/insert`, are only allowed to use element and sequence construction, plus path navigation to select nodes in the input document.
- Query expressions $Q$ used in `for/let` and conditional updates can be any query expression allowed by the query grammar presented in Figure 5.4 of Chapter 5.

As already said, restrictions behind SXUF have the purpose of ensuring a smooth formal characterization of iterative updates. At the same time, SXUF is expressive enough to cover most of needs in practical scenario.

For instance, several update expressions used in W3C XQuery Update Facilities 1.0 [Gro11b] strictly respect the syntax of the SXUF language, while other updates use function calls, conditions and arithmetic operations that are not supported by our simple grammar. However, as we will illustrate, our approach can be easily extended to deal with these mechanisms by means of simple query rewriting. As another example, all update expressions used in [BBC+11] and in Marina Sahakyan's Thesis [Sah11] are SXUF updates. The syntax of these update expressions are illustrated in Section A.3 of Appendix A.

Examples of SXUF expressions are below illustrated:

$U_1 =$ `delete nodes` $\$doc/child :: a/child :: f$

$U_2 =$ `insert node` $<n/>$ `as first into` $\$doc/child :: a/child :: b$

$U_3 =$ `rename node` $\$doc/child :: a/child :: f$ `as` $"new"$

$U_4 =$ `for` $\$x$ `in` $\$doc/child :: a/child :: b$
`return insert node` $<m>"toto"</m>$ `after` $\$x$

The following expressions are not SXUF updates:

$U_5 =$ `insert node` $< new/ >$ `after`
$\$doc/child :: a/child :: f[last()]$

$U_6 =$ `for` $\$x$ `in` $\$doc/child :: a/child :: f$ `return`
`replace value of node` $\$x/d$ `with` $\$x * 100$

In $U_5$, the target $P_{\text{tg}}$ makes use of the $last()$ function not allowed by SXUF, while in $U_6$ the source expression contains an arithmetical expression $\$x*100$, again not allowed by SXUF. However, these two updates can be easily rewritten into the following ones.

$U'_5 =$ `for` $\$x$ `in` $\$doc/child :: a/child :: f$
`return insert node` $< new/ >$ `after` $x

$U'_6 =$ `for` $\$x$ `in` $\$doc/child :: a/child :: f$ `return`
`replace value of node` $\$x/d$ `with` $\$x$

The rewriting is such that the iterative check and partitioning can be made in terms of the rewritten update, while the original one is used for update evaluation on the obtained partition. These simple rewritings can be easily lifted to the general case, thus enabling the application of our technique to a wide class of updates occurring in practice.

## 6.3 ITERATIVE UPDATES

As already indicated, our update scenario is based on the idea of partitioning an input document $D$ for an update $U$ into a collection of parts $\{D_1, D_2, \cdots, D_\kappa\}$, such that the final update result $U(D)$ on the document $D$ equals to the concatenation of all partial update results on each part $D_{\text{i}}$ produced by our partitioning strategy. This concatenation is performed by using a fusion operator $\oplus$, so that:

$$U(D) \cong U(D_1) \oplus U(D_2) \oplus \cdots \oplus U(D_\kappa) \qquad (6.1)$$

Essentially, the fusion operator concatenates updated parts, by taking care of considering only once nodes replicated in multiple parts by partitioning. We will give later on details about formalization of its semantics and streaming implementation.

In order to apply partitioning, we have to be sure that a partitioning for the input can be done so that Equation 6.1 can hold. This needs to be decided statically, before activating the partitioning scenario. If an update meets this property (*) it is called *iterative*.

Before providing a static analysis to recognize *iterative* updates, we see through some examples why our partitioning update scenario can be used in some cases of updates, while it is impossible to apply it in the other cases.

In the following, we are going to present three different kinds of updates: for the first one (e.g., $U_8$ and $U_9$) any kind of partition works; for the second kind of updates (e.g., $U_{10}$), only some partitions are good; for the last one (e.g., $U_{11}$ and $U_{12}$), no partition works.

We start the discussion with the first class. Figure 6.3 illustrates the XML document $t$ used as input for the following updates $U_8$ and $U_9$ used in examples.

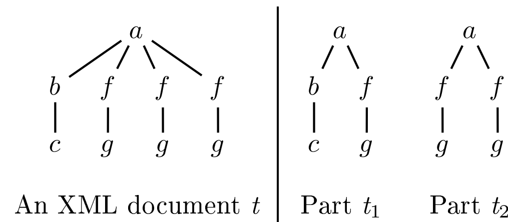



Figure 6.3: An XML document $t$ and a possible partition.

$U_8 =$ `delete nodes` $/child :: a/child :: f/child :: g$

$U_9 =$ `for` $\$x$ `in` $/child :: a/child :: f/child :: g$
`return insert node <n/> after` $\$x$

The first update $U_8$ deletes g-nodes selected by the target path $/child :: a/child :: f/child :: g$. By evaluating $U_8$ on the input $t$, we get the update result $U_8(t)$ which is reported in Figure 6.4.

Suppose that for the update $U_8$, we consider the possible partition $t_1, t_2$ illustrated in Figure 6.3.

In order to ensure the possibility of distributing the update $U_8$ on the partition $t_1$ and $t_2$, the update result $U_8(t)$ must be equal to the concatenation of all partial update results $U_8(t_i)$'s produced by evaluating $U_8$ on each part $t_i$. Actually this is the case as illustrated in Figure 6.4.

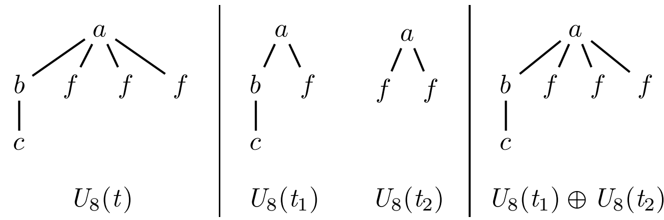



Figure 6.4: Equivalence between $U_8(t)$ and $U_8(t_1) \oplus U_8(t_2)$.

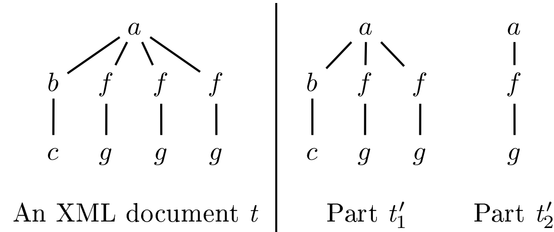



Figure 6.5: Another possible parts $t_1', t_2'$ of the XML document $t$.

Another partition is illustrated in Figure 6.5 that also works with $U_8$. Figure 6.6 illustrates the equivalence between the updated result $U_8(t)$ and the concatenation of partial update results $U_8(t_1') \oplus U_8(t_2')$. Actually, the update $U_8$ is such that its execution can be spread over any possible partition.

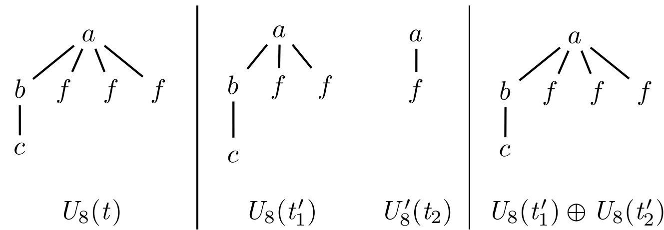



Figure 6.6: Equivalence between $U_8(t)$ and $U_8(t_1') \oplus U_8(t_2')$.

Now, let us consider the update $U_9$ which inserts an empty new node `<n/>` after each `g`-node (child of $/child :: a/child :: f$) in the same document $t$. By evaluating $U_9$ on $t$ and on its partition $t_1$ ,$t_2$ proposed in Figure 6.3, we have that the updated result $U_9(t)$ and the concatenation of the partial results $U_9(t_1) \oplus U_9(t_2)$ are equivalent, as Figure 6.7 illustrates.

Also for the other partition ($t_1'$, $t_2'$) proposed in Figure 6.5 for the same input document $t$, Equation 6.1 holds for the update $U_9$.

The update $U_8$ meets the property (*) that ensures that each modification performed by the update only depends on the current target node. The same property is met by update $U_9$.

The following update $U_{10}$ which uses the input document $t$ reported in Fig-

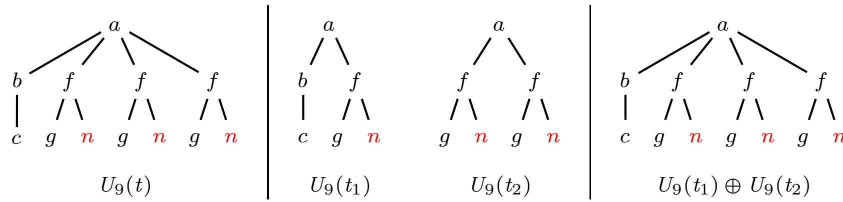



Figure 6.7: Equivalence between $U_9(t)$ and $U_9(t_1) \oplus U_9(t_2)$.

ure 6.8, illustrates that for some updates, one should be more careful in choosing a partition of the input document. This update inserts a new empty node `<n/>` as last into the target path $P_{\text{tg}}{=}/child :: a/child :: f$, as follows:

$U_{10}$ = **for** $\$x$ **in** $/child :: a/child :: f$
**return insert node <n/> as last into** $\$x$

This update is similar to the two previous ones in that each modification is focused on the current target node, but, differently, each update operation needs that the sub-tree rooted at the current target node has not been split during partitioning. This is because of the `as last into` clause. If the subtree is split, say, in two parts, then the `<n/>` would be inserted twice for a target node. This is illustrated in the sequel.

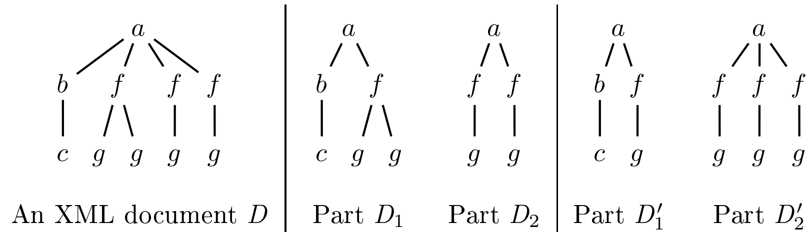



Figure 6.8: An XML document $D$ and two different kinds of partition.

By evaluating the update $U_{10}$ on the input document $D$ and the partition $D_1$, $D_2$ as Figure 6.9 illustrates, we observe that the update result $U_{10}(D)$ and the concatenation of partial update results $U_{10}(D_1) \oplus U_{10}(D_2)$ are equivalent, and thus we can say that this partition works with the update $U_{10}$.

Instead if we use the other partition $D'_1$, $D'_2$ (illustrated in Figure 6.8), we have that $U_{10}(D)$ and the concatenation $U_{10}(D'_1) \oplus U_{10}(D'_2)$ are not equivalent, as illustrated in Figure 6.10. This is because the update $U_{10}$ inserts a new node `n` as last of each subtree rooted at `f`-node on the document $D$ and its parts $D'_1$ and $D'_2$. This means that we will have two nodes `n` in the first subtree rooted at `f` of the concatenation result $D'_1 \oplus D'_2$.

The next examples illustrate the third kind of updates previously discussed, and for which no partition works.

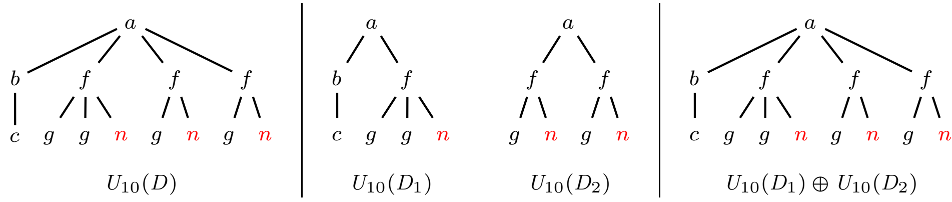



Figure 6.9: Equivalence between $U_{10}(D)$ and $U_{10}(D_1) \oplus U_{10}(D_2)$.

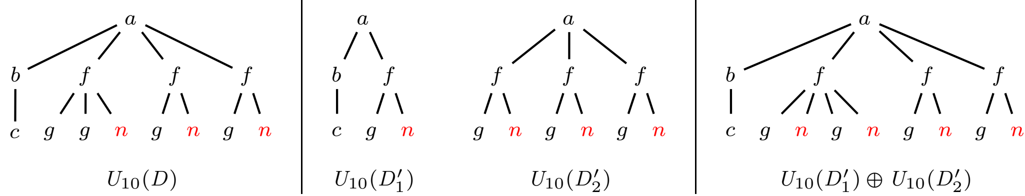



Figure 6.10: Non-equivalence between $U_{10}(D)$ and $U_{10}(D'_1) \oplus U_{10}(D'_2)$.

Consider the following update $U_{11}$ on the input document $t$ (illustrated in Figure 6.3) which replaces a target node `c` with a set of nodes labeled by `g`:

$$U_{11} = \texttt{replace node } /child :: a/child :: b/child :: c \ \texttt{ with} \\ /child :: a/child :: f/child :: g$$

and let us evaluate this update on both partitions $t_1, t_2$ (illustrated in Figure 6.3) and $t'_1, t'_2$ (illustrated in Figure 6.5) for the input document $t$.

Observe that the above update performs two main operations: it navigates through the *whole* document in order to evaluate the source expression $Q_s$=$/child :: a/child :: f/child :: g$, and use the obtained result to update target nodes found by evaluation of the target expression $P_{tg}$=$/child :: a/child :: b/child :: c$. This entails that distributing the update on any partition, would prevent the source expression from correctly building its result. This in turns prevents Equation 6.1 from holding, as exemplified next.

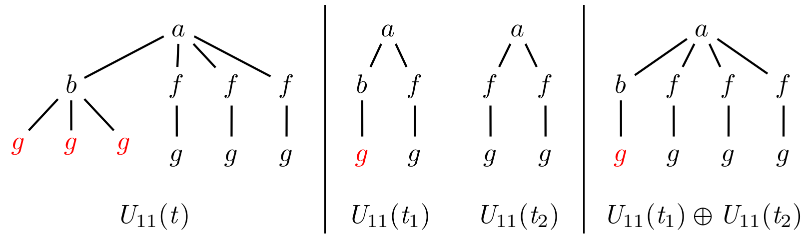



Figure 6.11: Non-equivalent case between $U_{11}(t)$ and $U_{11}(t_1) \oplus U_{11}(t_2)$.

Figure 6.11 illustrates that the update result $U_{11}(t)$ and the concatenation of partial update results $U_{11}(t_1) \oplus U_{11}(t_2)$ are not equivalent. The same happens if we use the other partition $t'_1, t'_2$, as Figure 6.12 illustrates, and any other partition.

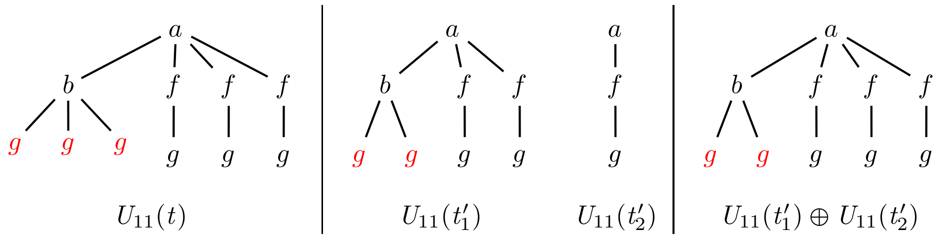



Figure 6.12: Non-equivalent case between $U_{11}(t)$ and $U_{11}(t'_1) \oplus U_{11}(t'_2)$.

Consider now the update $U_{12}$, evaluated on the input document $t$ and its proposed parts $t_1$ and $t_2$, illustrated in Figure 6.3. This update inserts the set of subtrees $/child :: a/child :: f/child :: g$ as last into the only b-node. As for the previous update, the source expression needs the *whole* input tree for its evaluation. Then, partitioning can not be applied, as illustrated in Figure 6.13.

$U_{12} =$ for $\$x$ in $/child :: a/child :: b$
return insert node $/child :: a/child :: f/child :: g$ as last into $\$x$

Concerning $U_{12}$, note that a slight variation would make partitioning applicable:

$U'_{12} =$ for $\$x$ in $/child :: a/child :: b$
return insert node $\$x/child :: c$ as last into $\$x$

Now the source expression $\$x/child :: c$ needs the current sub-tree selected by the outer iteration, and this makes partitioning applicable.

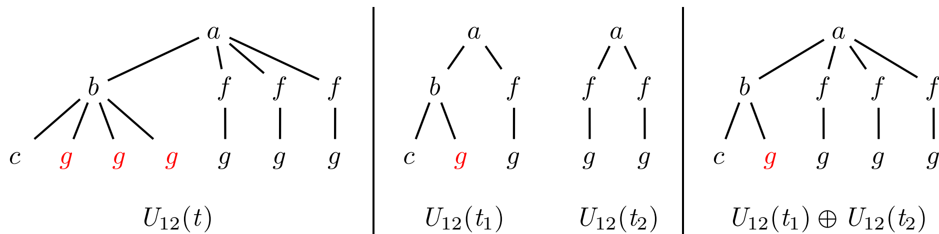



Figure 6.13: Non-equivalent case between $U_{12}(t)$ and $U_{12}(t_1) \oplus U_{12}(t_2)$.

From previous examples, we can conclude that in order to guarantee the realization of Equation 6.1 for a given update $U$, our partitioning update scenario can be applied only when the update $U$ performs many times the same operation on different subtrees, and each subtree contains all the information for the operation. Previous examples also illustrate that these subtree should be not split by partitioning (see update $U_{10}$). Updates satisfying this requirement are called *iterative updates*.

Informally, iterative updates are those ones described by the SXUF grammars and such that:

- if the update is elementary, then its target expression is a simple update expression, while its source expression $Q_s$ does not use XPath expressions (only element, sequence and text node construction are allowed).
- otherwise, the update first selects a sequence of nodes, and then perform update operations inside each subtree rooted at one of these nodes.

In order to formally characterize iterative updates and to performs data partitioning for them, we need to extract paths $P$ and target paths $P_{tg}$ from these updates, and then we need to analyze these paths. To this end, we define the function $E_{path}(U)$ for extracting path, and the function $E_{target}(U)$ to extract target paths from an update $U$. Both functions are defined along the same lines of the extraction function for queries, defined in Figure 5.5 of Chapter 5. The two functions are defined in Figure 6.14.

**Example 8** Consider the following update $U_{13}$ and the same XML document $t$ illustrated in Figure 6.3.

$$U_{13} = \texttt{for } \$x \texttt{ in } /child :: a/child :: f$$
$$\texttt{return rename node } \$x/child :: g \texttt{ as } "n"$$

By using the path extraction functions $E_{path}(U_{13})$ and $E_{target}(U_{13})$ illustrated in Figure 6.14, we show that the set of extracted paths is $\tau$=$\{P_1, P_2, P_3\}$, and the set of target paths $\tau_{ap}$=$\{P_3\}$, where

$$\begin{aligned} P_1 &= /child :: a \\ P_2 &= /child :: a/child :: f\{for\ x\} \\ P_3 &= /child :: a/child :: f\{for\ x\}/child :: g/dos :: node() \end{aligned}$$

□

As for queries, the variable information is not useful to perform the partition. Hence and in the rest of this chapter, we will rely on extracted paths once variable information has been eliminated. In Example 8, we will use the path $(/child :: a/child :: f)$ rather than $(/child :: a/child :: f\{for\ x\})$. We will do this by means of the function $ErVar(P)$ (already defined in Definition 5.2.1 of Chapter 5).

We are now ready to provide a formal characterization of iterative updates.

**Definition 6.3.1 (Iterative Update)** *Iterative updates are defined according the following case analysis.*

- *if $U$ is an elementary update, then it is iterative if and only if one of the following holds.*

  1. $U = delete\ N\ \ P_{tg}$
  2. $U = rename\ N\ \ P_{tg}\ as\ \ \mathrm{a}$
  3. $U = replace\ N\ \ P_{tg}\ with\ Q_s$
  4. $U = insert\ N\ \ Q_s\ Pos\ P_{tg}$

| $E_{\text{path}}(U\ , \Gamma, m)$ | | $\tau$ |
|---|---|---|
| $E_{\text{path}}(()\ , \Gamma, m)$ | = | $()$ |
| $E_{\text{path}}((Q_{\text{s}1}, Q_{\text{s}2})\ , \Gamma, m)$ | = | $E_{\text{path}}(Q_{\text{s}1}\ , \Gamma, m) \cup E_{\text{path}}(Q_{\text{s}2}\ , \Gamma, m)$ |
| $E_{\text{path}}(< \text{a} > Q_{\text{s}} < /\text{a} >\ , \Gamma, m)$ | = | $E_{\text{path}}(Q_{\text{s}}\ , \Gamma, 1)$ |
| $E_{\text{path}}(/P\ , \Gamma, 0)$ | = | $\{/P\}$ |
| $E_{\text{path}}(/P\ , \Gamma, 1)$ | = | $\{/P/dos :: node()\}$ |
| $E_{\text{path}}(x/P\ , \Gamma, 0)$ | = | $\{P'\{for\ x\}/P \mid P'\{for\ x\} \in \Gamma\}$ |
| $E_{\text{path}}(x/P\ , \Gamma, 1)$ | = | $\{P'\{for\ x\}/P/dos :: node() \mid P'\{for\ x\} \in \Gamma\}$ |
| $E_{\text{path}}(\textit{delete N}\ \ P_{\text{tg}}\ , \Gamma, 1)$ | = | $E_{\text{path}}(P_{\text{tg}}\ , \Gamma, 1)$ |
| $E_{\text{path}}(\textit{rename N}\ \ P_{\text{tg}}\ \textit{as}\ \ \text{a}\ , \Gamma, 1)$ | = | $E_{\text{path}}(P_{\text{tg}}\ , \Gamma, 1)$ |
| $E_{\text{path}}(\textit{replace N}\ \ P_{\text{tg}}\ \textit{with}\ Q_{\text{s}}\ , \Gamma, 1)$ | = | $E_{\text{path}}(P_{\text{tg}}\ , \Gamma, 1)\ \cup\ E_{\text{path}}(Q_{\text{s}}\ , \Gamma, 1)$ |
| $E_{\text{path}}(\textit{insert N}\ \ Q_{\text{s}}\ \textit{Pos}\ P_{\text{tg}}\ , \Gamma, 1)$ | = | $E_{\text{path}}(Q_{\text{s}}\ , \Gamma, 1)\ \cup\ E_{\text{path}}(P_{\text{tg}}\ , \Gamma, 1)$ |
| $E_{\text{path}}((U_1, U_2)\ , \Gamma, m)$ | = | $E_{\text{path}}(U_1\ , \Gamma, m) \cup E_{\text{path}}(U_2\ , \Gamma, m)$ |
| $E_{\text{path}}(\texttt{if}\ Q\ \texttt{then}\ U_1\ \texttt{else}\ U_2\ , \Gamma, m)$ | = | $E(Q, \Gamma, 0) \cup E_{\text{path}}(U_1\ , \Gamma, 1) \cup E_{\text{path}}(U_2\ , \Gamma, 1)$ |
| $E_{\text{path}}(\texttt{for}\ x\ \texttt{in}\ Q\ \texttt{return}\ U\ , \Gamma, m)$ | = | $\Gamma' \cup E_{\text{path}}(U, \Gamma \cup \Gamma', m)$<br>$where\ \Gamma' = \{P\{for\ x\} \mid P \in E(Q, \Gamma, 0)\}$ |
| $E_{\text{path}}(\texttt{let}\ x := Q\ \texttt{return}\ U\ , \Gamma, m)$ | = | $\Gamma' \cup E_{\text{path}}(U, \Gamma \cup \Gamma', m)$<br>$where\ \Gamma' = E(Q, \Gamma, 0)$ |

| $E_{\text{target}}(U\ , \Gamma, m)$ | | $\tau_{ap}$ |
|---|---|---|
| $E_{\text{target}}(()\ , \Gamma, m)$ | = | $()$ |
| $E_{\text{target}}((Q_{\text{s}1}, Q_{\text{s}2})\ , \Gamma, m)$ | = | $()$ |
| $E_{\text{target}}(< \text{a} > Q_{\text{s}} < /\text{a} >\ , \Gamma, m)$ | = | $()$ |
| $E_{\text{target}}(/P\ , \Gamma, 0)$ | = | $\{/P\}$ |
| $E_{\text{target}}(/P\ , \Gamma, 1)$ | = | $\{/P/dos :: node()\}$ |
| $E_{\text{target}}(x/P\ , \Gamma, 0)$ | = | $\{P'\{for\ x\}/P \mid P'\{for\ x\} \in \Gamma\}$ |
| $E_{\text{target}}(x/P\ , \Gamma, 1)$ | = | $\{P'\{for\ x\}/P/dos :: node() \mid P'\{for\ x\} \in \Gamma\}$ |
| $E_{\text{target}}(\textit{delete N}\ \ P_{\text{tg}}\ , \Gamma, 1)$ | = | $E_{\text{target}}(P_{\text{tg}}\ , \Gamma, 1)$ |
| $E_{\text{target}}(\textit{rename N}\ \ P_{\text{tg}}\ \textit{as}\ \ \text{a}\ , \Gamma, 1)$ | = | $E_{\text{target}}(P_{\text{tg}}\ , \Gamma, 1)$ |
| $E_{\text{target}}(\textit{replace N}\ \ P_{\text{tg}}\ \textit{with}\ Q_{\text{s}}\ , \Gamma, 1)$ | = | $E_{\text{target}}(P_{\text{tg}}\ , \Gamma, 1)$ |
| $E_{\text{target}}(\textit{insert N}\ \ Q_{\text{s}}\ \textit{Pos}\ P_{\text{tg}}\ , \Gamma, 1)$ | = | $E_{\text{target}}(P_{\text{tg}}\ , \Gamma, 1)$ |
| $E_{\text{target}}((U_1, U_2)\ , \Gamma, m)$ | = | $E_{\text{target}}(U_1\ , \Gamma, m) \cup E_{\text{target}}(U_2\ , \Gamma, m)$ |
| $E_{\text{target}}(\texttt{if}\ Q\ \texttt{then}\ U_1\ \texttt{else}\ U_2\ , \Gamma, m)$ | = | $E_{\text{target}}(U_1, \Gamma \cup \Gamma', 1) \cup E_{\text{target}}(U_2, \Gamma \cup \Gamma', 1)$<br>$where\ \Gamma' = \{P \mid P \in E(Q, \Gamma, 0)\}$ |
| $E_{\text{target}}(\texttt{for}\ x\ \texttt{in}\ Q\ \texttt{return}\ U\ , \Gamma, m)$ | = | $E_{\text{target}}(U, \Gamma \cup \Gamma', 1)$<br>$where\ \Gamma' = \{P\{for\ x\} \mid P \in E(Q, \Gamma, 0)\}$ |
| $E_{\text{target}}(\texttt{let}\ x := Q\ \texttt{return}\ U\ , \Gamma, m)$ | = | $E_{\text{target}}(U, \Gamma \cup \Gamma', 1)$<br>$where\ \Gamma' = E(Q, \Gamma, 0)$ |

Figure 6.14: Path extraction function for updates.

- *if $U$ is either a let-update or a for-update expression, then it is iterative if and only if it satisfies the properties required by Definition 5.3.2 in Chapter 5 by considering $E_{\text{path}}(U)$ as the set of extracted paths.*

- *If $U{=}U_1, U_2, \cdots, U_n$, then it is iterative if each $U_i$ is.*

In the above definition, the first case has been already motivated by means of examples. The second case relies on Definition 5.3.2 which presents iterative queries. It is worth noticing that when this case applies, the iterative update $U$ may contain elementary update sub-expressions not meeting properties 1-4, as in the following examples.

**Example 9** Consider the following update $U$:

$U =$ `for` $\$x$ `in` $/child :: a/child :: b$
`return insert nodes` $\$x/child :: f/child :: g$ `as last into` $\$x$

According to Definition 6.3.1, we have that the inner insert-update is not iterative, but the whole update is. As we will see, partitioning will be made in such a way that a subtree selected by the partitioning path $/child :: a/child :: b$ is never split into two distinct parts. This ensures the possibility of correctly distribute the update evaluation on subtrees selected by the partitioning path.

□

Still concerning the second case, it is worth noticing that `let-updates` are iterative only if the `let` binding does not use paths. For instance, the following update is not iterative.

$U =$ `let` $\$x$ `:=` $/child :: a/child :: b$ `return`
`if` $\$x/child :: c$ `then`
`delete node` $\$x$

This is because the `let` binding performs a global visit of the document before evaluating the inner update. For reasons already explained, this global visit prevents any possible partitioning based evaluation.

Instead, the following update is iterative:

$U =$ `let` $\$x$ `:= <c/> return`
`for` $\$y$ `in` $/child :: a/child :: b$ `return`
`insert` $\$x$ `after` $\$y$

Also note that in the second item of the definition of iterative updates, if-expressions are not considered. Actually these expressions may occur as inner sub-expressions of iterative updates, like in the following variant of the above example.

$U =$ `let` $\$x$ `:= <c/> return`
`for` $\$y$ `in` $/child :: a/child :: b$ `return`
`if` $\$y/child :: d$ `then insert` $\$x$ `after` $\$y$

The reason why if-expressions have been excluded as top-level expressions, is that in the general case the query defining the if-condition may require a global visit of the input document, and as already seen this makes partitioning impossible.

The third item of the characterization of iterative updates captures sequence updates. Partitioning can be applied for such updates, if it can be applied for each single update. This is quite intuitive. Shortly an example will be discussed.

As seen in previous examples, the crucial issue while partitioning for updates is to avoid splitting some particular subtrees. In order to specify a partitioning algorithm, we need to know how to recognize such subtrees. To this end, we use the set of target paths in the case the update is iterative according to conditions 1-4, or the partitioning path (Definition 5.3.3) otherwise. We call such a path *atomic*, since subtrees they point to cannot be split. Since an update can be a sequence of different updates, actually partitioning has to consider a set of atomic paths during the construction of a partition. The following example illustrates this.

**Example 10** Consider the following update $U$ and the input XML document $t$ illustrated in Figure 6.3:

$U =$ `(for` $\$x$ `in` $/child :: a/child :: b$ `return delete node` $\$x$ `),`
`(for` $\$x$ `in` $/child :: a/child :: f$ `return rename node` $\$x$ `as` $"n"$ `)`

Here, the set of atomic paths of $U$, denoted $AP(U)$, is $\{P_1, P_2\}$ with

$$\begin{array}{rcl} P_1 & = & /child :: a/child :: b \\ P_2 & = & /child :: a/child :: f \end{array}$$

□

From the above discussion the following atomic-paths extraction definition follows. It faithfully reflects the characterization of iterative updates. We denote with $AP(U)$ the set of atomic paths of the iterative update $U$.

**Definition 6.3.2 (Atomic Paths)** *Assume $U$ is an iterative update.*

- *If one of the following holds*

  1. $U = delete\ N\ \ P_{\mathrm{tg}}$
  2. $U = rename\ N\ \ P_{\mathrm{tg}}\ as\ \ \mathrm{a}$
  3. $U = replace\ N\ \ P_{\mathrm{tg}}\ with\ Q_{\mathrm{s}}$
  4. $U = insert\ N\ \ Q_{\mathrm{s}}\ Pos\ P_{\mathrm{tg}}$

  *then* $AP(U) = \{P_{\mathrm{tg}}\}$

- *if $U$ is either a let-update or a for-update expression, $AP(U) = \{PP\}$ where $PP$ is the partitioning path of $U$ according to Definition 5.3.3.*
- *If $U{=}U_1, U_2, \ldots, U_n$, then*

$$AP(u) = \bigcup_{i=1}^{n} AP(U_i)$$

Note that the above two definitions directly give conditions to deal with a workload of $n$ iterative updates $U_1, U_2, \ldots, U_n$. In this case the entire workload is iterative, and atomic paths can be extracted just as indicated above for the sequence case.

## 6.4 PARTITIONING FOR ITERATIVE UPDATES

As already said in the introduction, our partitioning technique for updates does not perform projection. The main motivation for this is to avoid complex merge operations (like the ones used in [BBC+11]) for recovering subtrees pruned out by projection. Actually, this is not a limitation since partitioning alone is already sufficient to ensure that each part is small enough to be processed by any main-memory XQuery engine. This is because, as for queries, the size of each part can be controlled by stopping its generation as soon as its size exceeds the threshold value *maxSize*. This value can be fixed along the same principles indicated for queries in the previous chapter, in particular by keeping into account main-memory features of the particular given used engine.

Our partitioning algorithm takes as input an XML document $D$, an iterative update $U$ and a threshold *maxSize* value. Through the static analysis technique described in the previous sections, our technique extracts the set of atomic paths $\tau_{ap} = AP(U)$ from the iterative $U$. These paths guide the partitioning process so that, a said before, subtrees they select are not split.

To illustrate how the partitioning algorithm works, let us consider the input document $t$ in Figure 6.15 and the following iterative update $U$:

$U = \texttt{for}\ \$x\ \texttt{in}\ /child :: a/child :: f\ \texttt{return}$
$\quad\texttt{insert node}\ {<}n/{>}\ \texttt{as last into}\ \$x$

for which we have $AP(U){=}/child :: a/child :: f$. Let us assume that *maxSize*=8.

During partitioning, similarly to the case of queries, and for the same reasons, both path alignment and residuation are performed on atomics paths.

We start the partitioning process from the root element $\mathrm{l}_1$ (see in Figure 6.15) which is a $P_{\mathrm{ap}}$ non-terminal node. Here a path alignment $Down(P_{\mathrm{ap}})$ is performed, the current size *cSize* is increased with the length of the current node $2.length(a)$ and the current $\mathrm{l}_1$ is added to the first part $t_1$. The next node considered is $\mathrm{l}_2$. In this case, the current node is a terminal $P_{\mathrm{ap}}$ node. In this case we perform the following steps:

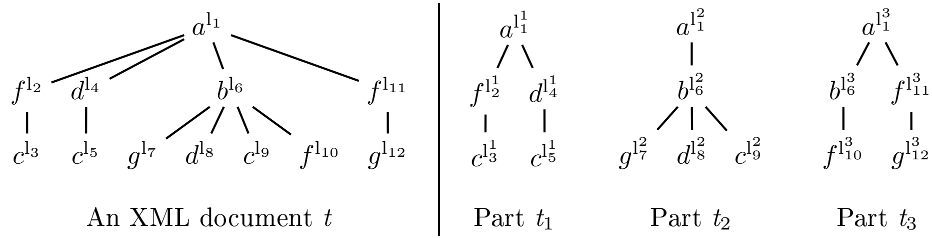



Figure 6.15: An XML document $t$ and its parts $t_1, t_2, t_3$.

- we parse the whole subtree of the current node and write it in the current part; while doing this we also calculate the size *Size* of this subtree.
- we increase *cSize* with the length of the current node, and *Size*.
- we add $\mathrm{l}_2$ (with its subtree $\mathrm{l}_3$) to the current part $t_1$.

The process goes on in a similar way with $\mathrm{l}_4$ and $\mathrm{l}_5$. After having parsed the second one, the current size happens to exceed the *maxSize*. This implies that the current part has to be ended and a new one has to be started. To this end, the algorithm resets $cSize'$ to 0 value, increases the number of parts $pId'$ with 1, and creates new part $t_{pId'}$ (we have $pId'$=2).

Then, the process goes to the next location $\mathrm{l}_6$ which does not match $P_{\mathrm{ap}}$, and continues the parsing following locations of its subtree $\mathrm{l}_7, \mathrm{l}_8, \mathrm{l}_9$, and stop this parsing at the location $\mathrm{l}_{10}$, due to the fact that the current size of the part exceeds *maxSize*. So the algorithm will close the current part, and create another part which will contain the rest of the input document locations $\{\mathrm{l}_{10}, \mathrm{l}_{11}, \mathrm{l}_{12}\}$. The process ends up with three different parts as illustrated in Figure 6.15.

As for queries (Chapter 5), we adopt a unique store for the partitioning resulted by our algorithm. Again, some nodes may belong to more than one part; this happens for the root node in particular. The resulting partitioning store contains three different parts formed by the following indexed locations:

$$\mathrm{dom}(\sigma^{\mathrm{P}}) = \{\mathrm{l}_1^1,\ \mathrm{l}_2^1,\ \mathrm{l}_3^1,\ \mathrm{l}_4^1,\ \mathrm{l}_5^1,\quad \mathrm{l}_1^2,\ \mathrm{l}_6^2,\ \mathrm{l}_7^2,\ \mathrm{l}_8^2,\ \mathrm{l}_9^2,\quad \mathrm{l}_1^3,\ \mathrm{l}_6^3,\ \mathrm{l}_{10}^3,\ \mathrm{l}_{11}^3,\ \mathrm{l}_{12}^3\}$$

We now provide a formal presentation of our partitioning algorithm and its auxiliary functions.

### 6.4.1 Partitioning Algorithm

Algorithm 9 provides a formal presentation of the partitioning process. This algorithm is recursive and takes as inputs the following 5-tuples $<\mathrm{l}; P_{\mathrm{ap}}; cSize; pId; List_{pId}>$ representing the current state of the recursive process. Namely, this tuple indicates that the current node to be matched against the current target path $P_{\mathrm{ap}}$ is l; that the current size of the part whose creation is in progress is

**Algorithm 8:** *Parse*

```
   Input: A store σ, a location l∈dom(σ);
   Output: A store σ′, an integer Size;
1  begin
2      if σ(l) = text[s] then
3          σ′:= {l ← text[s]};    Size:= length(s)
4      if σ(l) = a[()] then
5          σ′:= {l ← a[()]};    Size:= 2.length(a)
6      else
7          let L = (l_1, l_2, ··· , l_n)
8          for i = 1...n do
9              (σ_i, Size_i):= Parse(σ; l_i)
10         σ′:= {l ← a[L]} ∪ ⋃_{i=1}^{n} σ_i;
11             Size = 2.length(a) + Σ_{i=1}^{n} Size_i
12     return (σ′, Size)
```

*cSize*; that the current number of created parts is *pId*; and finally that the current indexed list $List_{pId}$ included parts *pId*'s. Of course, the algorithm is initially invoked with *cSize*=0 and *pId*=1, while the location l is the root of the input XML document $(\sigma, \mathrm{l})$, and $P_{\mathrm{ap}}$ is the set of atomic paths extracted from the iterative update $U$ according to Definition 6.3.2.

In this algorithm, we still use the function $PartLabel(\sigma; pId)$ which produces a new store obtained from $\sigma$ by renaming each location l to $\mathrm{l}^{pId}$.

The algorithm distinguishes two main cases.

- In the first case (lines 3-11) the current node is a terminal match for the atomic paths. In this case, the function $Parse(\sigma;\mathrm{l})$ parses the subtree rooted at the current node and results the corresponding store $\sigma'$ plus the size of the subtree *Size* (line 4). The function $Parse(\sigma;\mathrm{l})$ is illustrated in details in Algorithm 8, it performs a simple parse of the tree and updates the tree size each time a new node is encountered. After this parsing, the resulting subtree store $\sigma'$ is labeled by means of $PartLabel(\sigma'; pId)$. Then (lines 6-11), the algorithm adds the resulting subtree to the current part, and checks whether the *Size* size of the subtree plus the current size *cSize* exceeds the maximal size *maxSize*: If the check is negative, then current size *cSize* is increased with *Size*, otherwise the current size *cSize* is reset to 0, a new (empty) part is created, and the current *pId* is increased with 1.

  In this case, in order to optimize the time consumed for updating parts, the algorithm uses an integer list $List_{pId}$ (lines 9-10) which contains a list of identifiers *pId* of the parts that needs to be updated. In this case we have a node which is a possible target node of the updates, so the current part is added to the list.

**Algorithm 9:** *Partition*

**Input**: A location $l \in dom(\sigma)$, a set of atomic paths $P_{ap}$, a part size $cSize$, a part number $pId$, an empty list of part $pId$'s $List_{pId}$ ;
**Output**: A store $\sigma^P$, a part size $cSize'$, part number $pId'$, list of $pId$'s $List_{pId}'$ ;

1 **begin**
2   **let** $\sigma(l) = a[L]$
  /* *Case 1.* the current l is a $P_{ap}$ terminal node */
3   **if** $Res(a; P_{ap}) = <-; ok_t>$ **then**
4     $(\sigma', Size) := Parse(\sigma; l)$
5     $\sigma^P := PartLabel(\sigma'; pId)$
6     **if** $cSize + Size \leq maxSize$ **then**
7       $cSize' := cSize + Size; \quad pId' := pId$
8     **else**
9       **if** $pId' \notin List_{pId}$ **then**
10         $List_{pId}' := List_{pId}, \ pId'$ /* Current closed part will be updated */
11       $cSize' := 0; \quad pId' := pId + 1$
  /* *Case 2.* the current l is a $P_{ap}$ non-terminal node or does not match $P_{ap}$ */
12   **else**
13     $pId_{first} := pId; \quad \sigma^P := \emptyset;$
14     **let** $L = (l_1, l_2, \cdots, l_n)$
15     **for** $i = 1...n$ **do**
16       $(\sigma^P_i; cSize; pId; List_{pId}) := Partition(l_i; Down(P_{ap}); cSize; pId; List_{pId});$
17       $\sigma^P := \sigma^P \cup \sigma^P_i;$
18     $pId_{last} := Max\text{-}Pid(\sigma^P); \quad D := dom(\sigma^P);$
    /* *Max-Pid()* returns the biggest part number used in the store */
19     **for** $p = pId_{first}...pId_{last}$ **do**
20       $\sigma^P := \sigma^P \cup \{(l^p \leftarrow a[rename\!-\!extr(L, p, D)])\}$
21     $cSize' := cSize + 2.length(a)$
22     **if** $cSize' \leq maxSize$ **then**
23       $pId' := pId$
24     **else**
25       $cSize' := 0; \quad pId' := pId + 1$
26   **return** $(\sigma^P, cSize', pId', List_{pId}')$

- In the second case (lines 12-25), the current node l either is a possible non-terminal match of atomic paths, or does not match them. In both cases, the computation recursively goes on for each child $l_i$ of the l node, after having aligned atomic paths to the new tree level (line 15). After this partitioning proceeds in a way which is similar to that of the partitioning algorithm for queries (Algorithm 2). When the recursive calls on children of $l_i$ has termi-

nated, and the partitioning store updated (lines 18-20), the current part size is updated and the check for eventually creating a new part is made (lines 21-25).

Going back to the iterative update $U$ used in our previous example, thanks to the use of the $List_{pId}$ list, at the end of the partitioning process we know that the second part does not need to be updated because it does not contain any target node. Figure 6.16 illustrates the input three and updated parts.

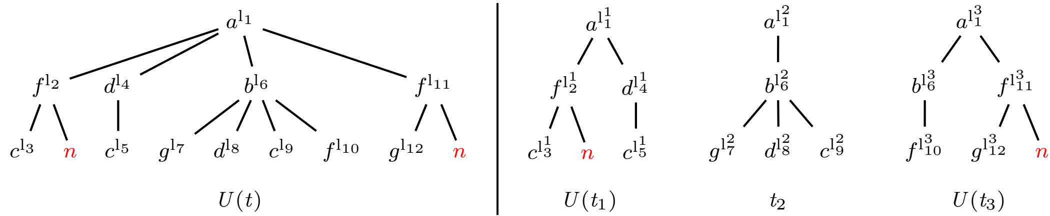


Figure 6.16: Partitioning update scenario on the input document $t$ and its parts, for a given iterative update $U$.

In the following section, we will present our fusion operation and its formal definitions, then we will provide details about the streaming implementation of our partitioning and fusion algorithms.

### 6.4.2 Fusion Operation

As illustrated before, the last step in our partitioning update scenario is the fusion operation. The main idea behind this operation is to concatenate all partial update results $U(D_i)$'s in a streaming way, to produce the final update result $U(D)$. The parts $D_i$'s are already created by the partitioning algorithm 9, and the partial updated results $U(D_i)$'s are performed by using a particular XQuery engine.

The fusion operation takes as input the set of updated parts $U(D_i)$ and returns $U(D)$. A particular issue in the fusion process concerns the presence of repeated locations is distinct parts. For our example, repeated locations are:

$$\mathrm{l}_1^1,\ \mathrm{l}_1^2,\ \mathrm{l}_1^3,\ \mathrm{l}_6^2,\ \mathrm{l}_6^3\ \in\ \mathrm{dom}(\sigma^{\mathrm{P}})$$

The fusion process has to be carefully specified in order to ensure that these locations are re-collapsed to a unique location, as illustrated in Figure 6.17. In this figure, the final update result $U(t_1) \oplus t_2 \oplus U(t_3)$ contains only one root element $\mathrm{l}_1$ and $\mathrm{l}_6$, while the repeated nodes appeared in distinct parts will be eliminated.

The fusion operation $\oplus$ is defined via the following definitions.

**Definition 6.4.1** (*ErIndex*($\mathrm{l}_\mathrm{i}^\mathrm{j}$)) *Given an indexed location* $\mathrm{l}_\mathrm{i}^\mathrm{j}$*, the function* *ErIndex*($\mathrm{l}_\mathrm{i}^\mathrm{j}$) *removes the index* $\mathrm{j}$ *from* $\mathrm{l}_\mathrm{i}^\mathrm{j}$*:*

$$ErIndex(\mathrm{l}_\mathrm{i}^\mathrm{j}) \ = \ \mathrm{l}_\mathrm{i}$$

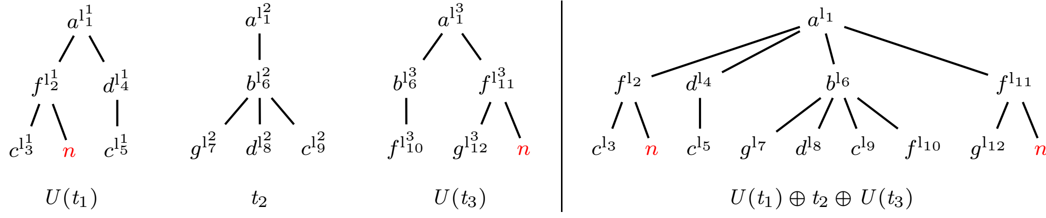



Figure 6.17: Fusion scenario on distinct (updated and non-updated) parts.

**Definition 6.4.2 (Fusion of locations $F(\mathrm{l_i}, C)$)** *Given a collection of trees $C = \{t_1, t_2, \cdots, t_\kappa\}$, we have*

$$F(\mathrm{l_i}, C) = \mathrm{l_i} \leftarrow \mathrm{a}[\mathtt{L}]$$

*with*

$$\mathtt{L} = \mathit{ErIndex}(\mathtt{L}_i) \cdot \mathit{ErIndex}(\mathtt{L}_{i+1}) \cdot \ldots \cdot \mathit{ErIndex}(\mathtt{L}_m)$$

*and* $\mathrm{l_i^j} \leftarrow \mathrm{a}[\mathtt{L_j}] \in t_\mathrm{j}$ *for* $j = i \ldots m$*, and for some* $i$ *and* $m$ *with* $1 \leq i \leq m \leq \kappa$*.*

**Definition 6.4.3 (Fusion $\oplus$)** *The collection of trees $C = \{t_1, t_2, \cdots, t_\kappa\}$ represents the set of parts created by partitioning the input tree $t = (\sigma_t, \mathrm{l})$ for an iterative update. For each tree $t_\mathrm{j} = (\sigma^\mathrm{j}, \mathrm{l^j})$ the root node $\mathrm{l^j}$ is the same for all parts in $C$, but with different index* $\mathrm{j}$.

*The updated parts are noted as $t_1^{'}, t_2^{'}, \cdots, t_\kappa^{'}$. The fusion operation $\oplus$ concatenates all trees $t_1^{'}, t_2^{'}, \cdots, t_\kappa^{'}$ to produce the final update result of the input tree $t$, as follows:*

$$\begin{aligned} t_1^{'} \oplus t_2^{'} \oplus \cdots \oplus t_\kappa^{'} &= (\sigma', \mathrm{l}) \\ \textit{where } \sigma' &= \{\textstyle\bigcup_{\mathrm{l_i} \in \sigma_t} F(\mathrm{l_i}, C)\} \cup \{\mathrm{l} \leftarrow \mathrm{a}[\mathit{ErIndex}(\mathtt{L})] \mid \exists\ \mathrm{i}\ .\ \mathrm{l}^{\perp} \leftarrow \mathrm{a}[\mathtt{L}] \in t_\mathrm{i}\} \end{aligned}$$

*Above, locations $\mathrm{l}^{\perp}$ are those newly created by the update.*

Soundness of our partitioning scenario is stated below, for the general case of an update workload.

**Theorem 6.4.4 (Soundness of Partition and Fusion)** *Let maxSize be a size threshold value, let $U_1, \ldots, U_m$ be* well-formed *iterative updates with their respecting atomic path sets $P_{\mathrm{ap}_j}$. Let $t$=$(\sigma, \mathrm{l}_t)$ be an XML tree. Then:*
*Assuming*

- $P_{\mathrm{ap}} = \cup_1^m \{P_{\mathrm{ap}_j}\}$ *and*
- $\mathit{Partition}(\mathrm{l}_t; \mathit{Down}(P_{\mathrm{ap}}); 0; 1; \mathit{pId}) = (\sigma^{\mathrm{P}}; \mathit{cSize}; \mathit{pId})$.

*we have:*

$$U_j(t) \cong U_j(t_1) \oplus \ldots \oplus U_j(t_{pId})$$

*where* $t_i$=$\mathit{PartLabel}^{-1}(\sigma_i^P; \mathit{pId})$.

In the following section, we will provide a streaming representation of our partitioning and fusion algorithms.

## 6.5 STREAMING IMPLEMENTATION

Previous formalizations of partitioning and fusion algorithms are not amenable to handle big files, as they are DOM-oriented: they assume that the whole input stores are available. As already said, this DOM-based formulation is presented to give a formal specification of the algorithm.

To handle arbitrary large files, we implemented previous partitioning and fusion algorithms in a streaming fashion on top of a SAX parser [ver00]. In our implementation, we consider the SAX events already considered for the case of queries (see Chapter 5):

$$\begin{array}{rcl} SAXEvent & := & \texttt{startDocument} \\ & | & \texttt{startElement}(qName) \\ & | & \texttt{endElement}(qName) \\ & | & \texttt{Characters}(String) \end{array}$$

Our SAX implementations has two essential tasks: the first one is to perform the partitioning (see Section 6.5.1), and the second one is to apply the fusion operation over the updated partitioning (see Section 6.5.2).

### 6.5.1 Partitioning

The SAX implementation of partitioning is similar to that for queries (Section 5.6 of Chapter 4). It uses two main stack-based data structures. These stacks are used to record the current status of the algorithm when an open-tag is met, so that the status can be recovered when the corresponding close-tag is met. The first stack $stack_{tag}$ is used to record open-tag name of the current node being processed $qName$, the result of the residuation of $Res(qName; \tau_{ap})$, an identifier *tagId* of the current open-tag node $qName$. The second stack $stack_{\tau_{ap}}$ is used to record all alignment results $Down(\tau_{ap})$ of the atomic paths $\tau_{ap}$.

Also, the partitioning algorithm uses and maintains two text-files during the processing: the first one $File_{art}$ which contains all artificial tags and their *tagId*'s. These artificial tags are those ones created during partitioning to preserve the well-formedness of generated parts, but that do not belong to the original files. These tags are closed and reopened when the creation of a part ends, and when the creation of the following part begins. The second text file $File_{pId}$ contains identifiers (natural numbers) corresponding to parts that need to be updated. As already said, filter partitioning only parts mentioned in this file will be updated, thus saving processing time as some parts will be not processed.

The implementation also records some values in the following global variables:

- *cSize* the current size (nodes and text-values) of the current part.
- *pId* the current number of created parts.
- *tagId* the current identifier node.

- $\tau_{ap}$ the set of atomic paths.
- *containsAtomicNode* the number of terminal $\tau_{ap}$ nodes in the current part.

**Algorithm 10:** Partition-Init-DataStructures

**Input**: An input XML document $t$, a pre-defined integer value *maxSize*, a set of atomic paths $\tau_{ap}$ extracted from a given update $U$;
**Output**: Initialize flag *containsAtomicNode*, stacks $stack_{tag}$ and $stack_{\tau_{ap}}$, text files $File_{art}$ and $File_{pId}$;

```
1 begin
2    cSize:= 0; pId:= 1; tagId:= 0; containsAtomicNode:= 0
3    stack_tag:= ()
4    stack_τap:= ()
5    create File_art
6    create File_pId
```

**Algorithm 11:** SAX-startDocument

**Input**: A set of atomic paths $\tau_{ap}$;
**Output**: Side effect on $\tau_{ap}$ and *Modality*;

```
1 begin
2    τap:= Down(τap)
3    Modality:= part
```

**Algorithm 12:** SAX-characters

**Input**: A string-value *str*, current part size *cSize*;
**Output**: Side effect on the current part size *cSize*;

```
1 begin
2    writeOutput(str)
3    cSize:= cSize + length(str)
```

**Algorithm 13:** SAX-endDocument

**Input**: Flag *containsAtomicNode*, part number *pId*;
**Output**: Side effect on the text-file $File_{pId}$;

```
1 begin
2    if containsAtomicNode=1 then
         /* Set the current part t_pId to be updated */
3        writeFile_pId(t_pId::to-be-updated)
```

By using this status information, we can split the partitioning algorithm in two distinct parts, which are executed when `startElement` and `endElement` are invoked, respectively.

**Algorithm 14:** SAX-startElement

**Input**: Open-tag qName, part number *pId*, part size *cSize*, node id *tagId*;
**Output**: Side effect on *cSize*, *tagId*, $\tau_{ap}$ and *containsAtomicNode*;

1 **begin**
2 MATCH:= $Res(qName; \tau_{ap})$
3 *cSize*:= *cSize* + *length*(*qName*)
4 *tagId*:= *tagId* + 1
5 **if** *Modality*=*part* **then**
6 **switch** MATCH **do**
7 **case** ok_nt
`/* qName is a non-terminal match` $\tau_{ap}$ `*/`
8 $stack_{tag}.add(qName, \text{MATCH}, tagId, part)$
9 $stack_{\tau_{ap}}.add(Down(\tau_{ap}))$
10 $\tau_{ap}$:= $stack_{\tau_{ap}}.peek$
11 **case** ok_t
`/* qName is a terminal match` $\tau_{ap}$ `*/`
12 $stack_{tag}.add(qName, \text{MATCH}, tagId, part)$
13 $stack_{\tau_{ap}}.add(Down(\tau_{ap}))$
14 $\tau_{ap}$:= $stack_{\tau_{ap}}.top()$
15 *containsAtomicNode*:= 1
16 *Modality*:= *parse*
17 **case** fail
`/* qName does not match` $\tau_{ap}$ `*/`
18 $stack_{tag}.add(qName, \text{MATCH}, tagId, part)$
19 *writeNodeAttribute*(*qName*, *tagId*)
20 **else if** *Modality*=*parse* **then**
21 $stack_{tag}.add(qName, -, -, parse)$
22 *writeOutput*(*qName*)

Before starting the processing, our partitioning algorithm takes the following inputs (see Algorithm 10):

- the input XML document $t$.
- the set of atomic paths $\tau_{ap}$ extracted from the iterative updates (recall that the case of a workload is considered too).
- the threshold integer value *maxSize* for the part sizes.

and it is initially invoked with *cSize*=0, *tagId*=0 and *pId*=1 (line 2 of Algorithm 10). Also, all data structures needed to perform the partitioning $stack_{tag}$, $stack_{\tau_{ap}}$, $File_{art}$ and $File_{pId}$ will be defined (lines 3-6 of Algorithm 10).

During partitioning we associate a unique identifier *tagId* with each element we put in the partition. This identifier is needed in order to distinguish among

**Algorithm 15:** SAX-endElement

```
   Input: Close-tag qName, part number pId, part size cSize
   Output: Side effect on cSize, pId, τ_ap and containsAtomicNode
1  begin
2      MATCH:= stack_tag.pop(stack_tag(top).get(1))
3      currTagId:= stack_tag.pop(stack_tag(top).get(2))
4      currModality:= stack_tag.pop(stack_tag(top).get(3))
5      if currModality=part then
6          Size:= length(qName)
7          switch MATCH do
8              case ok_nt
9                  τ_ap:= stack_τap.pop
10             case ok_t
11                 τ_ap:= stack_τap.pop
12                 Modality:= part
13         if cSize + Size ≤ maxSize then
14             cSize:= cSize + Size
15             writeOutput(qName)
16         else
               /* Close current part t_pId                              */
17             for i=[(stack_tag.size − 1)...0] do
18                 currTagName:= stack_tag(i).get(0)
19                 currTagId:= stack_tag(i).get(2)
20                 writeOutput(currTagName)
21                 writeFile_art(currTagName | currTagId | pId | close)
22             if containsAtomicNode=1 then
                   /* Set the current part t_pId to be updated          */
23                 writeFile_pId(t_pId::to-be-updated)
24                 containsAtomicNode:= 0
               /* Reset cSize to 0 value and increase pId with 1        */
25             cSize:= 0;    pId:= pId + 1
               /* Create new part t_pId                                 */
26             for i=[0..(stack_tag.size − 1)] do
27                 currTagName:= stack_tag(i).get(0)
28                 currTagId:= stack_tag(i).get(2)
29                 writeNodeAttribute(currTagName, currTagId)
30                 writeFile_art(currTagName|currTagId|pId|open)
31                 cSize:= cSize + length(currTagName)
32             cSize:= cSize + length(qName)
33             writeOutput(qName)
34     else if currModality=parse then
35         cSize:= cSize + length(qName)
36         writeOutput(qName)
```

original and artificial tags, and will be erased during fusion. This identifiers is a positive integer, whose value starts from 1 and which is incremented each time a new open-tag is met. Later on we will illustrate details of this aspect.

In `startDocument` event (see Algorithm 11), the algorithm performs the first alignment $Down(\tau_{ap})$ (line 2) and initializes the modality flag *Modality* with *part* value (line 3). We will explain the functionality of this flag later.

Both `startElement` and `endElement` algorithms work in two possible modalities, the partitioning modality (*part*) and the parsing modality (*parse*). The first one concerns the case that the algorithm is in the search of a terminal-match for $P_{\text{ap}}$ and the terminal match is either not-found or the the current node is one. Under this modality the two algorithms implement the specification reported in the DOM-based Algorithm 9). The second possible modality is for the case that the current node is inside a subtree rooted at a terminal $P_{\text{ap}}$ node. Under this modality, the two algorithms implement the specification given in the parsing DOM-based Algorithm 8; under this modality, the a new part can not be created; the entire subtree has to be added to the current part.

In `startElement` event (see Algorithm 14), we put most of the logic of the DOM-based specification partitioning and parsing algorithms (Algorithms 8 and 9). Actually, all partitioning decisions are based on information that are available when an open-tag is met. Also, we put the updates of *cSize*, *tagId* and the residuation of the atomic paths set $Res(qName; \tau_{ap})$ (lines 2-4), but we defer partitioning decision to `endElement` calls.

Concerning the partitioning modality (lines 5-20) and if the MATCH value is either ok_nt or ok_t, we put the current status ($qName$, MATCH, *tagId*, *part*) of the algorithm into the $stack_{tag}$, we also perform a path alignment of the current $\tau_{ap}$ and put the result into the $stack_{\tau_{ap}}$ (lines 7-10 and 11-16). In addition to these tasks, and in case of MATCH=ok_t we do the following: we set $containsAtomicNode$ to 1 to indicate that the current part contains a terminal $\tau_{ap}$ node (line 15) and as such it has to be updated; finally we set the *Modality* flag with *parse* value during the ok_t matching case (line 16), as for the following subtree no new part has to be created. If MATCH value is fail, we only keep the current information ($qName$, fail, *tagId*, *part*) into the $stack_{tag}$ (line 18). Finally we write the current open-tag $qName$ into the current part. Note that in the partitioning modality, we add a new attribute *tId* which contains the current *tagId* value for each open-tag $qName$ (line 20).

Concerning the parsing modality (lines 21-23), we only keep the following information of the current $qName$ ($qName$, -, -, *parse*) into the $stack_{tag}$ (line 22), and write open-tag $qName$ into the current part (line 23). Note that we do not consider a *tagId* for each $qName$ manipulated in the parsing modality.

In `endElement` (see Algorithm 15), we first perform a pop operation on the $stack_{tag}$ and keep the information in the following variables: MATCH is the cur-

rent match value; $currTagId$ is the current tag identifier; and $currModality$ is the current working modality (lines 2-4). This pop operation permit to recover status information at the moment the corresponding open-tag was met.

If the information got from the $stack_{tag}$ tell us that we have a *part* current modality (line 5), we calculate the length of the current $qName$ and keep it in *Size* variable (line 6), then we make the following case analysis on the MATCH information relative with the current close-tag $qName$, and got from the $stack_{tag}$ (lines 5-34). While if the information tell us that we have a $parse$ current modality (line 35), we only increase the current size *cSize* with the *length*($qName$) (line 36), and then write the current close-tag $qName$ in the current part (line 37).

If the current close-tag is for a non-terminal $P_{\text{ap}}$ node (lines 8-9), we pop the top element of the $stack_{\tau_{ap}}$. While If the current close-tag is for a terminal $P_{\text{ap}}$ node (lines 10-12), we pop the top element of the $stack_{\tau_{ap}}$, and change the *Modality* flag to *part* (line 12).

Since the parsing of the atomic subtree has ended, we compare the current size part with the maximal part size allowed *maxSize* (line 13). If the creation of a new part has to be done, then we iterate on the stack $stack_{tag}$, close all the open tags (lines 17-21), and keep these closed-tags with their information tag-name, *tagId*, *pId* and tag-case which is either open or close into the text file $File_{art}$. Then we check if the current close part will be updated or not. To this end, we check if the *containsAtomicNode* value equals 1 (line 22), we keep the current part name ($t_{pId}$ :: to-be-update) into another text file $File_{pId}$ (line 23), and reset *containsAtomicNode* value to 0 (line 24). Then the algorithm resets *cSize* to 0 and increases the part number *pId* by 1 (line 25). After that the new part is created, by reopening all tags kept into the $stack_{tag}$ into the new created part, in reversal order (lines 26-30). During this process, we add respective records [tag-name,*tagId*,*pId*,tag-case open] into the text file $File_{art}$ (line 30), and increase *cSize* with the length of each re-opened tag *length*(`currTagName`) (line 31). At the end, we increase the current part size with *length*($qName$) (line 32), and finally we write the current close-tag $qName$ in the current part (line 33).

In `Characters` event (see Algorithm 12), we only write the text-content *str* of the current $qName$ into the current part $t_{pId}$ (line 2), then add the length of *str* to the current part size *cSize* (line 3).

In `endDocument` (see Algorithm 13), we need to verify if the last created part $t_{pId}$ will be updated or not, we do this by checking the value of flag *containsAtomicNode*. If it equals 1, this means that the current part will be updated, otherwise it is considered as non-updated part, and as we did before, we keep the checking result into the $File_{pId}$.

To illustrate how the streaming partitioning algorithm works, we will use the following iterative update.

**Example 11** Consider the following iterative update $U$:

$U$ = `for` $\$x$ `in` $/child :: a/child :: b/child :: f$
`return rename node` $\$x$ `as` $"n"$

and the input XML document $t$ illustrated in Figure 6.18. This update renames each child `f`-node of the `b`-node as `"n"`. We have $\tau_{ap}$=$/child :: a/child :: b/child :: f$ and assume $maxSize = 9$.

| Input document $t$ | Part $t_1$ | Part $t_2$ | Part $t_3$ |
|---|---|---|---|
| `<a>`<br>`<d><c></c></d>`<br>`<b>`<br>`<f><c>go</c></f>`<br>`<f><g>to</g></f>`<br>`</b>`<br>`<c><f><d></d></f></c>`<br>`</a>` | `<a tId="1">`<br>`<d tId="2">`<br>`<c tId="3"></c>`<br>`</d>`<br>`<b tId="4">`<br>`<f tId="5">`<br>`<c>go</c>`<br>`</f></b>`<br>`</a>` | `<a tId="1">`<br>`<b tId="4">`<br>`<f tId="6">`<br>`<g>to</g>`<br>`</f>`<br>`</b>`<br>`</a>` | `<a tId="1">`<br>`<c tId="7">`<br>`<f tId="8">`<br>`<d></d>`<br>`</f>`<br>`</c>`<br>`</a>` |

Figure 6.18: An input document $t$ and its created parts $t_1, t_2, t_3$.

The partitioning process starts when the document is opened; at this moment atomic paths are aligned, and modality is set to partitioning. Then the root element `<a>` is met. Algorithm 14 performs a residuation $Res(a; \tau_{ap})$; increases the current size $cSize$ with the length of the current node $length(qName)$; increases the tag identifier $tagId$ with 1. Since we are in partitioning modality, the algorithm checks the MATCH value, which is in our case ok_nt. This means that we have a possible non-terminal $\tau_{ap}$ node, so we add the following values `[a,ok_nt,1,part]` at the top of the $stack_{tag}$, perform a path alignment $Down(\tau_{ap})$ and add it to the top of the $stack_{\tau_{ap}}$, and finally write the open-tag of the current $qName$ with the current $tagId$ as attribute (we write `<a tId="1">`) into the current part (line 8 of Algorithm 14). Then the process goes to the next node (see Figure 6.19).

| Input document $t$ | Part $t_1$ | $stack_{tag}$ [qName,MATCH, $tagId$, $Modality$] |
|---|---|---|
| `<a>`<br>`<d><c></c></d>`<br>`<b>`<br>`<f><c>go</c></f>`<br>`<f><g>to</g></f>`<br>`</b>`<br>`<c><f><d></d></f></c>`<br>`</a>` | `<a tId="1">`<br>`<d>`<br>`<c></c>`<br>`</d>`<br>`<b><f>`<br>`<c>go</c>`<br>`</f></b>`<br>`</a>` | `[a,ok_nt,1,part]` |

| | | |
|---|---|---|
| $\tau_{ap}$ | = | $\{/child :: a/child :: b/child :: f\}$ |
| $cSize$ | = | 1 |

Figure 6.19: Partitioning scenario: the current open-tag is `<a>`.

The next event is for an open-tag `<d>` which does not match the current set of atomic paths, so we only increase $cSize$ with the length of the tag and $tagId$ with 1, add the following information `[d,fail,2,part]` at the top of the $stack_{tag}$, and

write the current open-tag with the respective *tagId* attribute (`<d tId="2">`) into the current part. We repeat the same treatment with the following open-tag `<c>` which does not match $\tau_{ap}$ as well, and add the tuple `[c,fail,3,part]` at the top of the $stack_{tag}$, and write this node with its *tagId* attribute `<c tId="3">` into the current part (see Figure 6.20).

| Input document<br>$t$ | Part<br>$t_1$ | $stack_{tag}$<br>[qName,MATCH, *tagId*, *Modality*] |
|---|---|---|
| <a><br><d><c></c></d><br><b><br><f><c>go</c></f><br><f><g>to</g></f><br></b><br><c><f><d></d></f></c><br></a> | <a tId="1"><br><d tId="2"><br><c tId="3"></c><br></d><br><b><f><br><c>go</c><br></f></b><br></a> | [c,fail,3,part]<br>[d,fail,2,part]<br>[a,ok_nt,1,part] |

$$\tau_{ap} = \{/child :: b/child :: f\}$$
$$cSize = 3$$

Figure 6.20: Partitioning scenario: parsing the open-tags `<d><c>`.

Now we have the close-tag `</c>`. Here the algorithm performs the following tasks: pop the top element of the $stack_{tag}$ and keep the pop values in the following variables MATCH, $currTagId$ and $currModality$. Then the algorithm checks the $currModality$ value which is *part* in the current case, so the process will update *Size*, then add it to the current part size *cSize*, which now equals to 5 and does not exceed the *maxSize*. So we finally write the current close-tag in the current part. We repeat the same process with the close-tag `</d>`, as illustrated in Figure 6.21

| Input document<br>$t$ | Part<br>$t_1$ | $stack_{tag}$<br>[qName,MATCH, *tagId*, *Modality*] |
|---|---|---|
| <a><br><d><c></c></d><br><b><br><f><c>go</c></f><br><f><g>to</g></f><br></b><br><c><f><d></d></f></c><br></a> | <a tId="1"><br><d tId="2"><br><c tId="3"></c><br></d><br><b><f><br><c>go</c><br></f></b><br></a> | [a,ok_nt,1,part] |

$$\tau_{ap} = \{/child :: b/child :: f\}$$
$$cSize = 5$$

Figure 6.21: Partitioning scenario: parsing the close-tags `</c></d>`.

The process continues in the same scenario for the next open-tag `<b>` which is a non-terminal match for atomic paths; the current *cSize* and *tagId* are increased; a path alignment $Down(\tau_{ap})$ is performed and the new $\tau_{ap}$ is added to the $stack_{\tau_{ap}}$; also, the following record `[b,ok_nt,4,part]` is added at the top of the $stack_{tag}$; and finally the current open-tag with its `tId` attribute is written into the current part file.

Then the process goes to the next node `<f>` which is a terminal $P_{\text{ap}}$ node. The current *cSize* and *tagId* are increased. Then, we perform the following tasks: we add the current information [f,ok_t,5,part] at the top of the $stack_{tag}$; we perform a path alignment $Down(\tau_{ap})$ and push the new $\tau_{ap}$ on the $stack_{\tau_{ap}}$; we increase the *containsAtomicNode* by 1 and change the partitioning modality to the parsing one *Modality=parse*, to start the parsing of the subtree rooted at our current terminal match of atomic paths. Then we write the current open-tag with its tId attribute on the current part. Note that in the parsing modality, we only keep the tag-name and the current modality *parse* for each open-tag encountered in the current subtree. So for the following open tag `<c>` we add records of the form [c,-,-,parse] in the stack. The tag size is added to the current part size *cSize* , and write the encountered open- tag into the current part. Figure 6.22 illustrates all tasks performed above.

| Input document<br>$t$ | Part<br>$t_1$ | $stack_{tag}$<br>[qName,MATCH, *tagId*, *Modality*] |
|---|---|---|
| <a><br><d><c></c></d><br><b><br><f><c>go</c></f><br><f><g>to</g></f><br></b><br><c><f><d></d></f></c><br></a> | <a tId="1"><br><d tId="2"><br><c tId="3"></c><br></d><br><b tId="4"><f tId="5"><br><c>go</c><br></f></b><br></a> | [c,-,-,parse]<br>[f,okt,5,part]<br>[b,ok_nt,4,part]<br>[a,ok_nt,1,part] |

| | | |
|---|---|---|
| $\tau_{ap}$ | = | $\{-\}$ |
| *cSize* | = | 10 |

Figure 6.22: Partitioning scenario: parsing the subtree `<b><f><c>go`.

Now we have the current event is for the close-tag `</c>`. We are in parsing modality. Here Algorithm 15 recovers information from the $stack_{tag}$, and verifies that the corresponding open-tag does not match atomic paths. So it increases the current size *cSize* with the length of the current close-tag `</c>`, and writes the close-tag into the current part. The next close-tag `</f>` occurs still in a parsing modality. The algorithm performs the following tasks: it recovers information from the $stack_{tag}$ and realizes that the tag is relative to a terminal match of atomic paths. So it pops path information from $stack_{\tau_{ap}}$, then changes the *Modality* flag to *part* mode; then it checks whether the current part size *cSize* plus the *Size* value exceeds *maxSize* or not. It is positive in the current case, so we will close all the open tags, and keep the information status [tag-name,*tagId*,1,close] of these new closed-tags into the $File_{art}$. Then we reset *cSize* to 0 and increase the part number *pId* by 1, then reopen the same tags in reversal order in the new created part, increase *cSize* with the length of each new open-tag $length(currTagName)$, and keep them with their information [tag-name,*tagId*,1,open] into the $File_{art}$. Finally we increase the current part size with $length(qName)$, then we write it in the new created part $t_2$. Figure 6.23 shows us all previous tasks.

The process continues parsing the subtree starting from the previous open-tag

| Input document $t$ | Part $t_1$ | Part $t_2$ | $stack_{tag}$ [qName,MATCH, $tagId$,$Modality$] |
|---|---|---|---|
| `<a>`<br>`<d><c></c></d>`<br>`<b>`<br>`<f><c>go</c></f>`<br>`<f><g>to</g></f>`<br>`</b>`<br>`<c><f><d></d></f></c>`<br>`</a>` | `<a tId="1">`<br>`<d tId="2">`<br>`<c tId="3"></c>`<br>`</d>`<br>`<b tId="4"><f tId="5">`<br>`<c>go</c>`<br>`</f></b>`<br>`</a>` | `<a tId="1">`<br>`<b tId="4">`<br>`<f>`<br>`<g>to</g>`<br>`</f>`<br>`</b>`<br>`</a>` | `[b,ok_nt,4,part]`<br>`[a,ok_nt,1,part]` |

$$\tau_{ap} = \{/child :: f\}$$
$$cSize = 14 \geq maxSize$$

| $File_{art}$ | $File_{pId}$ |
|---|---|
| `[b,4,1,close]`<br>`[a,1,1,close]`<br>`[a,1,2,open]`<br>`[b,4,2,open]` | $t_1$`::to-be-updated` |

Figure 6.23: Partitioning scenario: parsing close-tags `</c></f>`, and create a new part $t_2$

`<b>` and arrives to the current `<f>` which is again a terminal node for atomic paths. Then the current *cSize* and *tagId* will be updated. Then the same treatment with terminal nodes illustrated before will be repeated: we add the current information of this node `[f,ok_nt,6,part]` at the top of the $stack_{tag}$; a path alignment on atomic paths is performed and the new set is added to the $stack_{\tau_{ap}}$; the *containsAtomicNode* is set to 1; and then the flag *Modality* is changed to *parse*, to start the parsing of the subtree rooted at the current terminal match for atomic paths; finally we write the current *tagId* tag into the current part $t_2$.

Since the current modality is *parse*, we parse the subtree rooted at the current terminal match, and copy it to the current part, as done before. In particular, this subtree contains the following fragment `<g>to`, so the information status `[g,-,-,parse]` is put into $stack_{tag}$. Then the process goes to the next close-tag node `</g>`. Figure 6.24 illustrates the effect of these steps.

Both close-tags `</g>` and `</f>` will be written in the current part $t_2$ by calling `endElement` event, because the checking of exceeding the maximal size *maxSize* is still negative (*cSize*=8). While when we arrive to the close-tag `</b>`, the checking size will be positive. So we will close the current part $t_2$ as we did before by creating a new close-tag `</a>`, and add its information status `[a,1,2,close]` to the $File_{art}$. We check then the updating status for the closed part which is *to-be-updated* part, and keep the result $t_2$`::to-be-updated` into the $File_{pId}$; finally create a new part $t_3$ which starts with a new open-tag `<a tId="1">`. Then the process goes for the next node `<b>`. Figure 6.25 illustrates all previous tasks.

For the rest of the document, the process goes according the lines of previously illustrated steps, and ends up with three different parts. Only parts $t_1, t_2$ will be

| Input document<br>$t$ | Part<br>$t_1$ | Part<br>$t_2$ | $stack_{tag}$<br>[qName,MATCH,<br>$tagId$,$Modality$] |
|---|---|---|---|
| <a><br><d><c></c></d><br><b><br><f><c>go</c></f><br><f><g>to</g></f><br></b><br><c><f><d></d></f></c><br></a> | <a tId="1"><br><d tId="2"><br><c tId="3"></c><br></d><br><b tId="4"><f tId="5"><br><c>go</c><br></f></b><br></a> | <a tId="1"><br><b tId="4"><br><f tId="6"><br><g>to</g><br></f><br></b><br></a> | [g,-,-,parse]<br>[f,okt,6,part]<br>[b,ok_nt,4,part]<br>[a,ok_nt,1,part] |

$$\tau_{ap} = \{-\}$$
$$cSize = 6$$

| $File_{art}$ | $File_{pId}$ |
|---|---|
| [b,4,1,close]<br>[a,1,1,close]<br>[a,1,2,open]<br>[b,4,2,open] | $t_1$::to-be-updated |

Figure 6.24: Partitioning scenario: parsing open-tags <f><g>to

| Input document<br>$t$ | Part<br>$t_1$ | Part<br>$t_2$ | Part<br>$t_3$ |
|---|---|---|---|
| <a><br><d><c></c></d><br><b><br><f><c>go</c></f><br><f><g>to</g></f><br></b><br><c><f><d></d></f></c><br></a> | <a tId="1"><br><d tId="2"><br><c tId="3"></c><br></d><br><b tId="4"><f tId="5"><br><c>go</c><br></f></b><br></a> | <a tId="1"><br><b tId="4"><br><f tId="6"><br><g>to</g><br></f><br></b><br></a> | <a tId="1"><br><b><br><f><br><d></d><br></f><br></b><br></a> |

$$\tau_{ap} = \{/child :: b/child :: f\}$$
$$cSize = 10 \geq maxSize$$

| $stack_{tag}$<br>[$qName$, MATCH, $tagId$, $Modality$] | $File_{art}$ | $File_{pId}$ |
|---|---|---|
| [a,ok_nt,1,part] | [b,4,1,close]<br>[a,1,1,close]<br>[a,1,2,open]<br>[b,4,2,open]<br>[a,1,2,close]<br>[a,1,3,open] | $t_2$::to-be-updated<br>$t_1$::to-be-updated |

Figure 6.25: Parsing close-tags </g></f></b>, and create a new part $t_3$.

flagged as parts that will be updated. The third one will be not flagged, because it does not contain any terminal $P_{\text{ap}}$ node, as illustrated in Figure 6.26.

□

After generating three parts $t_1$, $t_2$ and $t_3$, and updating the parts $t_1$ and $t_2$, the next step is to concatenate the two updated parts with the non-updated third one $t_3$. To this end, we rely on the fusion algorithm which is illustrated next.

| Input document $t$ | Part $t_1$ | Part $t_2$ | Part $t_3$ |
|---|---|---|---|
| `<a>`<br>`<d><c></c></d>`<br>`<b>`<br>`<f><c>go</c></f>`<br>`<f><g>to</g></f>`<br>`</b>`<br>`<c><f><d></d></f></c>`<br>`</a>` | `<a tId="1">`<br>`<d tId="2">`<br>`<c tId="3"></c>`<br>`</d>`<br>`<b tId="4"><f tId="5">`<br>`<c>go</c>`<br>`</f></b>`<br>`</a>` | `<a tId="1">`<br>`<b tId="4">`<br>`<f tId="6">`<br>`<g>to</g>`<br>`</f>`<br>`</b>`<br>`</a>` | `<a>`<br>`<c tId="7">`<br>`<f tId="8">`<br>`<d></d>`<br>`</f>`<br>`</c>`<br>`</a>` |

Figure 6.26: Final parts $t_1, t_2, t_3$ produced by the partitioning technique.

### 6.5.2 Fusion

This section presents the SAX algorithms for the fusion process. The fusion algorithm takes as input the following values:

- the number of created parts *pId*;
- the text file $File_{art}$ which is already created during the partitioning process, and contains all information about artificial open/close tags.

Since the parts will be parsed sequentially, one after the other, a dynamic SAX parser is initialized for parsing each created (updated/non-updated) part alone. When the parsing of the current part is finished, the dynamic parser automatically goes to the next part to start the parsing process (see Algorithm 16).

Once the file $File_{art}$ is available, the fusion process has to perform very simple operations. Essentially, for each open/close-tag, using the $File_{art}$ file to check whether the current tag has to be put to the resulting document. Again, two essential SAX event handlers are used, one for the event `startElement` (see Algorithm 17), and one for the event `endElement` (see Algorithm 18).

In order to accelerate lookup operation on $File_{art}$, we first load all its content and store it into an array we call $array_{art}$. During this process, each line in $File_{art}$ is split, by using the delimiter "|" into four different values (*tagName, tagId, partId, tagCase*), and added i to the $array_{art}$. Once created this array will be not changed, and will be only used for lookup operations.

Also we use the $stack_{sync}$ to synchronize the writing of open/close tags of the current tag in the final result. In particular, we push in this stack the current open-tag with its *tagId* attribute when the `startElement` event occurs (see Algorithm 17), and pop the top element of this stack when `endElement` event occurs (see Algorithm 18).

In `startElement` event, Algorithm 17 first checks whether the current tag contains an attribute `tId`:

- if the check is positive, the algorithm will keep the `tId` value in the *currTagId* variable, otherwise put the value 0 in this variable (lines 2-5). Next, the algorithm verifies whether the current tag is will be put in the output tree

**Algorithm 16:** Fusion-mainProgram

```
    Input: A text file File_art, a number of parts pId
    Output: Create the final update result t_final
 1  begin
 2      openFile(File_art)                          /* Open the text-file File_art */
 3      strLine:= ReadTextFile(File_art)             /* Open File_art and read it
        line-by-line */
 4      while (File_art) is not finished do
 5          split(strLine,"|")        /* Split current string line by using the
            delimiter "|", and keep the 4 different values obtained from
            this line into stack_art */
 6          array_art.add(tagName, tagId, partId, tagCase)
 7      closeFile(File_art)                         /* Close the text-file File_art */
        /* Initialize a dynamic SAX parser for each Part        */
 8      for i=[1..pId] do
 9          StartParser(t_i)              /* Start parsing the current part t_i */
10          currPId:= i          /* Keep the index i of the current part t_i */
11      return (t_final)
```

**Algorithm 17:** Fusion-startElement

```
    Input: open-tag qName, array_art, stack_sync
    Output: Side effect on array_art and stack_sync
 1  begin
 2      if qName.containAttribute(tId) then
 3          currTagId:= getTagIdAttribute(qName)
 4      else
 5          currTagId:= 0
        /* Check all open-tags in array_art                    */
 6      for i=[0..(array_art.size - 1)] do
 7          tempTagName:= array_art(i).get(0)
 8          tempTagId:= array_art(i).get(1)
 9          tempPId:= array_art(i).get(2)
10          tempCase:= array_art(i).get(3)
11          if qName=tempTagName and currTagId=tempTagId and
            currPId=tempPId and tempCase="open" then
12              Skip(qName)  /* Do not write the current open-tag qName into
                t_final */
13              break
14          else
15              writeOutput(qName)   /* Write the current open-tag qName into
                t_final */
16          stack_sync.add(qName, currTagId)
```

**Algorithm 18:** Fusion-endElement

```
Input: close-tag qName, array_art, stack_sync
Output: Side effect on array_art and stack_sync
1  begin
       /* Pop the top element from stack_sync and keep (tagname, tagid)
       values                                                        */
2      tagname:= stack_sync.pop[top].get(0)
3      tagid:= stack_sync.pop[top].get(1)
       /* Compare the current close-tag qName with the content of array_art
       */
4      for i=[0..(array_art.size − 1)] do
5          tempTagName:= array_art(i).get(0)
6          tempTagId:= array_art(i).get(1)
7          tempCase:= array_art(i).get(3)
8          if qName=tempTagName and qName=tagname and
           tagid=tempTagId and tempCase="close" then
9              Skip(qName)      /* Do not write the current close-tag qName
               into t_final */
10             array_art.remove(i)
11             break;
12         else
13             writeOutput(qName) /* Write the current close-tag qName into
               t_final */
```

$t_{final}$ or not. To this end, it verifies whether $array_{art}$ contains a line matching the current tag and the current Id attribute, and whose tagCase is 'open' (line 11). If the check is positive, this means that the current open-tag is insignificant and we do not write into the output tree $t_{final}$, so it is dropped (lines 12-13).

- if the check is negative (line 15), the algorithm simply writes the open-tag to the output tree $t_{final}$ (in this case either the node has been added by the update, or it is a node belonging to a subtree selected by an atomic path).

In all the above cases, the algorithm add the tuple $(qName, tId)$ into $stack_{sync}$ (line 16).

In the `endElement` event, the Algorithm 18 performs similar steps as above, with the difference that at the beginning the top of the $stack_{sync}$ is popped. The popped ID attribute is used for checking whether the current close-tag should be written into the output tree $t_{final}$ or not (lines 4-13).

Going back to our example, we see now how the fusion operation works to concatenate three generated parts $U(t_1)$, $U(t_2)$ and $t_3$ which are illustrated in Figure 6.27. Note that $t_3$ is not updated because it does not contains any terminal $P_{\text{ap}}$ node.

| $U(t_1)$ | $U(t_2)$ | non-updated $t_3$ | $U(t_1) \oplus U(t_2) \oplus t_3$ |
|---|---|---|---|
| <a tId="1"><br><d tId="2"><br><c tId="3"></c><br></d><br><b tId="4"><n tId="5"><br><c>go</c><br></n></b><br></a> | <a tId="1"><br><b tId="4"><br><n tId="6"><br><g>to</g><br></n><br></b><br></a> | <a><br><c tId="7"><br><f tId="8"><br><d></d><br></f><br></c><br></a> | <a><br><d><c></c></d><br><b><br><n><c>go</c></n><br><n><g>to</g></n><br></b><br><c><f><d></d></f></c><br></a> |

Figure 6.27: Updated parts $U(t_1)$, $U(t_2)$, non-updated $t_3$, and the fusion final result.

The fusion process starts parsing the first part, and read the first open-tag `<a tId="1">`. By checking this tag, we show that this one is significant tag, so we keep it in the output tree $t_{final}$. The pair $(a, 1)$ is pushed into the $stack_{sync}$. The process goes on in a similar way for the following open-tags `<d tId="2">` and `<c tId="3">`, and pairs $(d, 2)$ and $(c, 3)$ are pushed into the $stack_{sync}$.

When the process arrives to the closed-tag `</c>`, the algorithm pops the top element from $stack_{sync}$ which is (c,3). Then, by also considering the retrieved ID attribute and $array_{art}$, it checks whether the current tag is needed to write into the output tree $t_{final}$ or not. This is not the case, so it writes `</c>` in $t_{final}$. The same happens for `</d>` .

The fusion process continues in the same scenario for the rest of the current part $t_1$, and write the following fragment `<b><n><c>go</c></n>` into the final result $t_{final}$. When the process arrives to the close-tag `</b>`, here we have that the current tag is insignificant close-tag, so the algorithm does not write it. This happens for the following `</a>` as well. At this moment, the parsing of the current part finishes, and the process goes to the updated second part, and then the third non-updated part, in a similar way. The process ends up with the final updated result $U(t_1) \oplus U(t_2) \oplus t_3$ as illustrated in Figure 6.27.

## 6.6 EXPERIMENTAL EVALUATION

In the previous sections, we presented our XML data partitioning scheme that, given an iterative update $U$ and an input document $D$, partitions $D$ in a set of parts $\{D_1, \ldots, D_\kappa\}$ so that $U(D)$ is equivalent to the concatenation of $U(D_1) \oplus \ldots \oplus U(D_\kappa)$, where $\oplus$ is our fusion operation. When this partitioning scheme is applicable, it can improve the scalability of existing main-memory engines, as it allows the system to process one part per time.

In this section we present an experimental evaluation of the partitioning update technique. We will first show that the proposed algorithm significantly improves the scalability of a popular main-memory query engine (particularly Saxon and Qizx Query engines). Then, we will show that partitioning, when combined with a fusion algorithm. Finally, we will experimentally analyze the relation between the

overall performance of the system and the actual value of *maxSize* (the maximum part size).

### 6.6.1 Experimental Setup

We implemented our partitioning algorithm, as well as our fusion algorithm, in Java 6 and tested their behavior on the XMark benchmark [SWK+02a]. In particular, we evaluated our system on XMark documents by relying on two widely used XQuery engines, Saxon [sax] and Qizx [qiz]. Saxon is an engine supporting all main W3C standards for XML manipulation and schema validation, while Qizx is specialized on querying and updating, and offers powerful optimization techniques. However, we will see that even with the use of standard path-based projection, these systems do not scale up in terms of document size (other powerful systems like BaseX [bas] have quite similar performances). Our test results show that our technique overcome this limitation for iterative updates, as it allows these engines to scale up to arbitrary document sizes.

All experiments were performed on a 2.53 Ghz Intel Core 2 Duo machine (4GB main memory) running Mac OSX 10.6.8. All XML documents were loaded on an external USB2 7200 rpm 1 TB disk unit.

To avoid the perturbations introduced by system activity, we ran each experiment ten times, discarded the best and the worst performance, and computed the average of the remaining times.

### 6.6.2 Tests Results

We used documents whose size ranges from 1GB to 5GB for Saxon and from 1GB to 15GB for Qizx. Concerning the threshold value *maxSize*, we set ($\sim$ 25MB) for Saxon, and ($\sim$ 95.36 MB) for Qizx. These differences in terms of memory and part sizes are due differences of performances between the two engines in terms of memory management. For both Saxon and Qizx we allocated 512MBs for main memory of the Java Virtual Machine.

Concerning updates, we used the following updates proposed by the PhD thesis of Marina Sahakyan [Sah11], which form the iterative core of XMark [SWK+02a]:

```
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

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"

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

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

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>
```

### 6.6.3 Experiments

In our first experiment we evaluate and compare scalability of Saxon. We consider a 1GB document and a 5GB document for Saxon test. For each document and for each update, we compare total execution time obtained with only standard projection with that obtained from the partitioning+fusion approach. Total execution time includes the overall time required by the system to partition the input document, to evaluate the input update on the parts, and to concatenate the partial results to produce the final result.

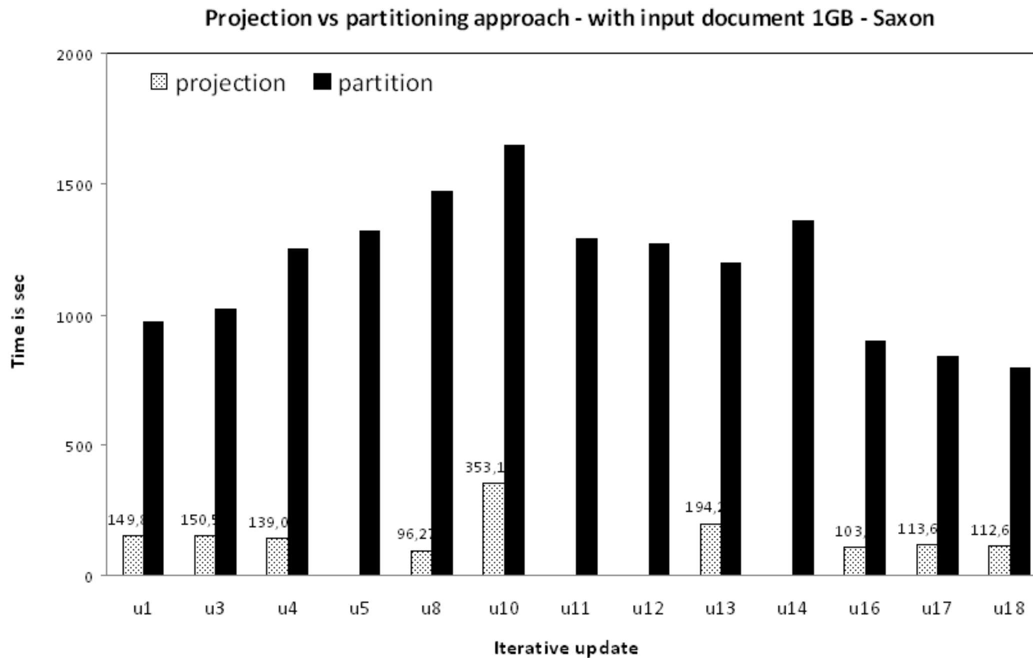



Figure 6.28: Projection vs partitioning - with input document 1GB - using Saxon.

Concerning results obtained by using Saxon. When projection only is used, this system starts showing limitations even for a 1GB document, for which updates U5, U11, U12 and U14 could not be executed due to memory failure. As shown in Figure 6.28. While our partitioning technique enables execution of all iterative updates.

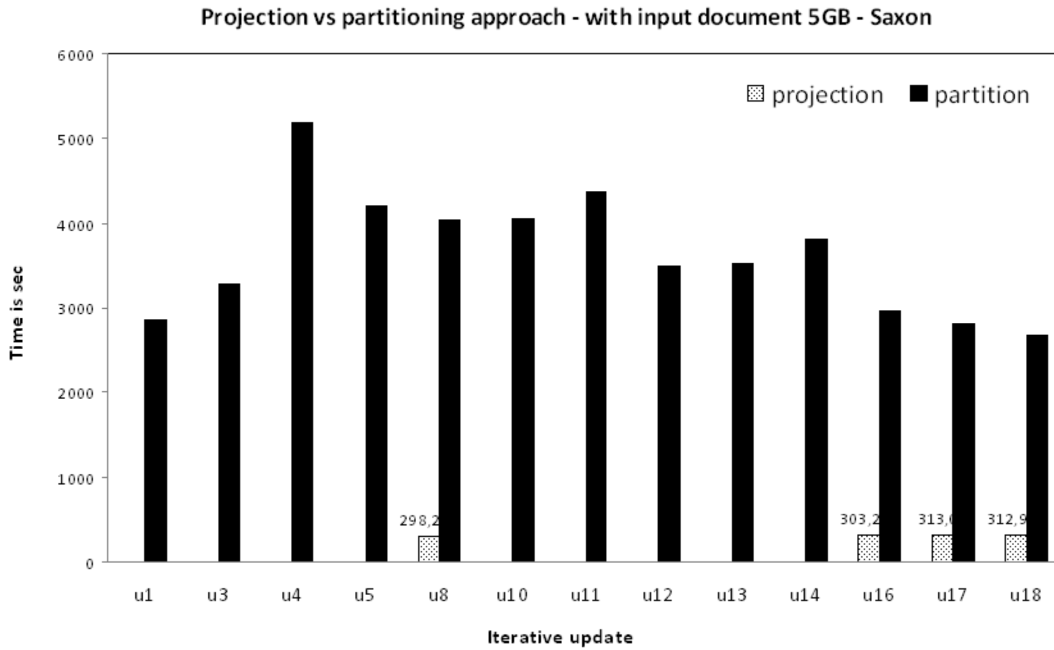



Figure 6.29: Projection vs partitioning - with input document 5GB - using Saxon.

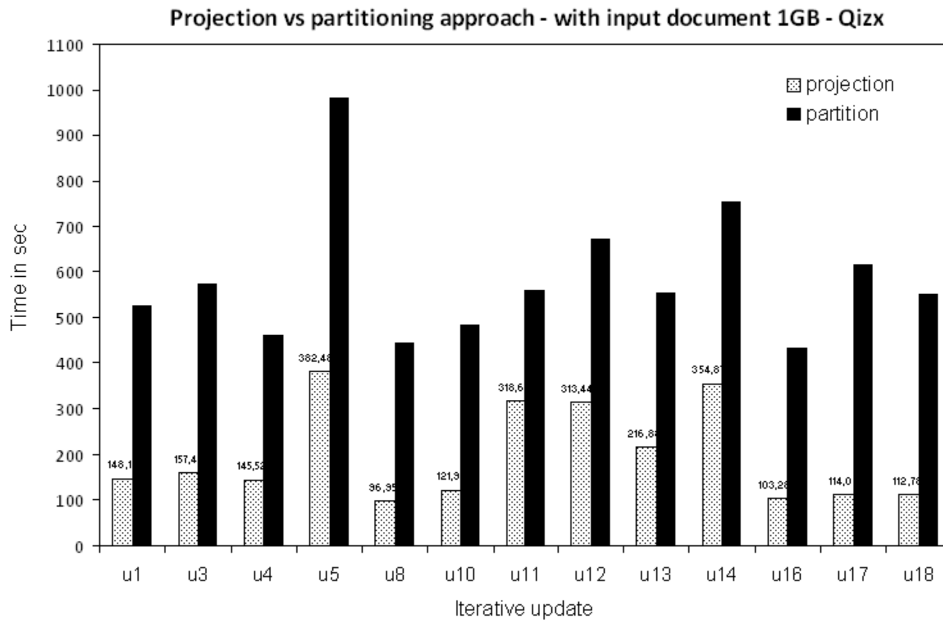



Figure 6.30: Projection vs partitioning - with input document 1GB - using Qizx.

As illustrated in Figure 6.29, for the 5GB document, improvements of our partitioning technique are substantial: 9 updates could not be executed with only

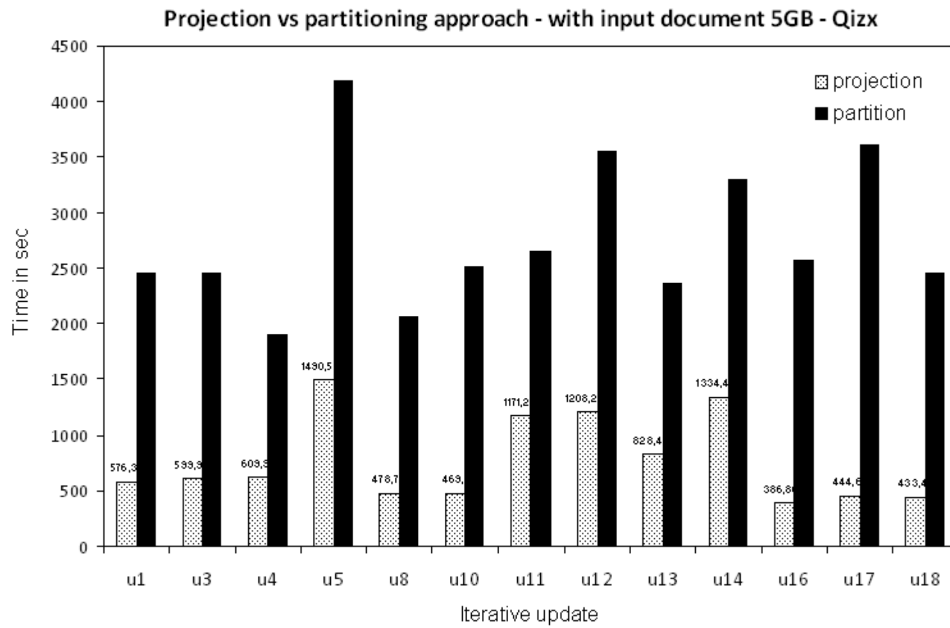



Figure 6.31: Projection vs partitioning - with input document 5GB - using Qizx.

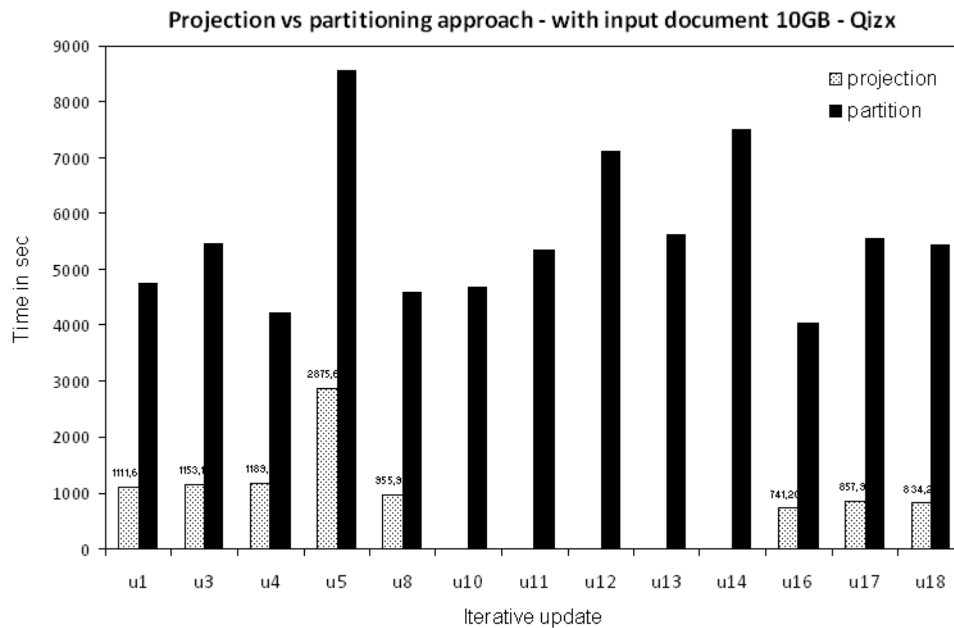



Figure 6.32: Projection vs partitioning - with input document 10GB - using Qizx.

projection, while all updates are executed by means of partitioning.

Figure 6.34 reports execution times obtained with Saxon and partitioning, for all considered documents size. As shown by the figure, our technique scales up and has a linear behavior.

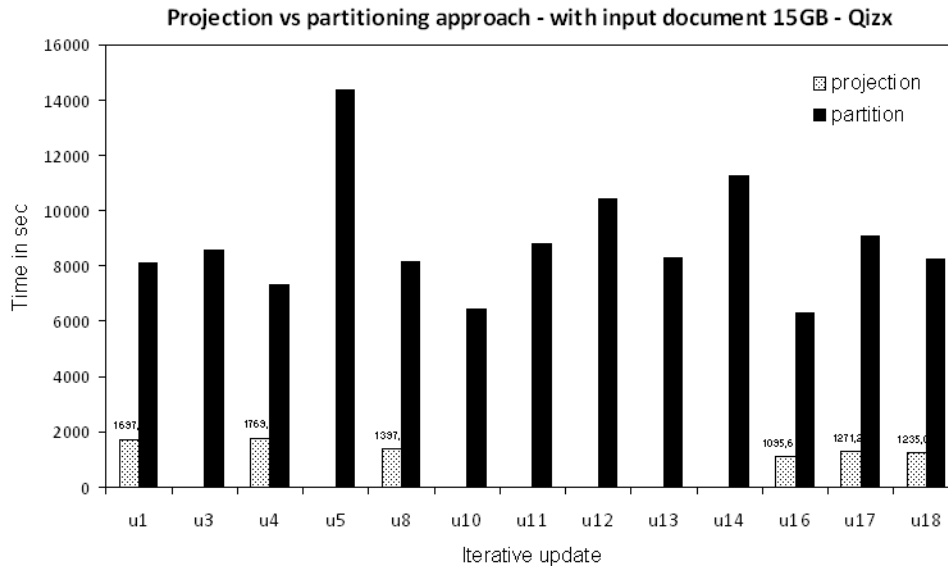



Figure 6.33: Projection vs partitioning - with input document 15GB - using Qizx.

Concerning Qizx, we consider 1GB, 5GB, 10GB and 15GB documents. For the 1GB input document (see Figure 6.30) and for the 5GB document (see Figure 6.31), all 13 iterative updates could be executed with the sole use of projection.

For the 10GB input document (see Figure 6.32), the standard projection technique starts showing limitations, and the updates U10, U11, U12, U13 and U14 could not be executed due to memory failure. As shown in Figure 6.32. While our partitioning technique enabled to process all 13 iterative updates.

Also for the 15GB input document (see Figure 6.33). Seven updates could not be executed with the sole use of projection. Instead, our partitioning technique enabled the processing of all 13 iterative updates.

Figure 6.35 reports execution times obtained with Qizx and partitioning, for all considered documents size. As shown by this figure, our technique scales up and has a linear behavior.

### 6.6.4 Summing Up

To summarize, our experiments prove that the partitioning approach scales beautifully and is only slightly slower than the projection approach with updates. To make experiments feasible in a reasonable time we considered 5GB for Saxon and 15GB for Qizx as the maximal size of documents. However, since the *maxSize* can be tuned to fit in the available main memory, we have that partitioning scales for arbitrary sizes.

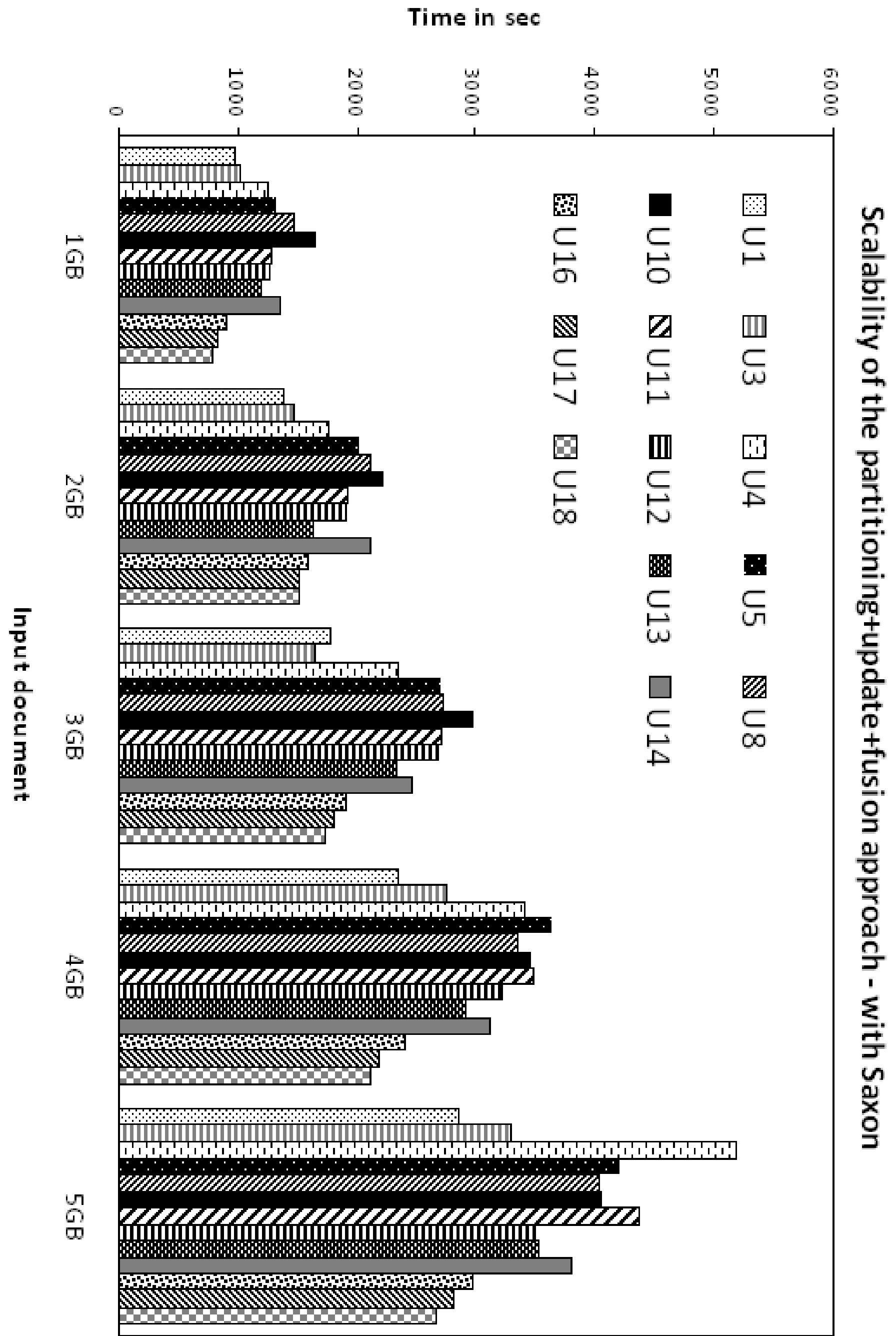



Figure 6.34: Scalability of the partitioning+update+fusion approach - using Saxon.

## 6.7 CONCLUSION

In this chapter, we presented out partitioning based technique for XML updates. As we have seen, the techniques differs from that for queries in many aspect. First of all in what concern the characterization of iterative updates, and secondly in the partitioning and fusion algorithms. Some preliminary results on experimental evaluation, showed that the technique succeeds in its main purpose: overcoming scalability limitations of main memory systems. We believe that similar experimental results could be obtained by using other engines, like the BaseX [bas] for instance, whose performances are close to that of Qizx. As future works we plan to perform more extensive tests, and to improve efficiency of the fusion algorithm in order to reduce the overhead in terms of time.

Another interesting future direction would be to combine projection with partitioning. This would require deep changes in the fusion algorithm, but probably permit to further lower the time overhead.

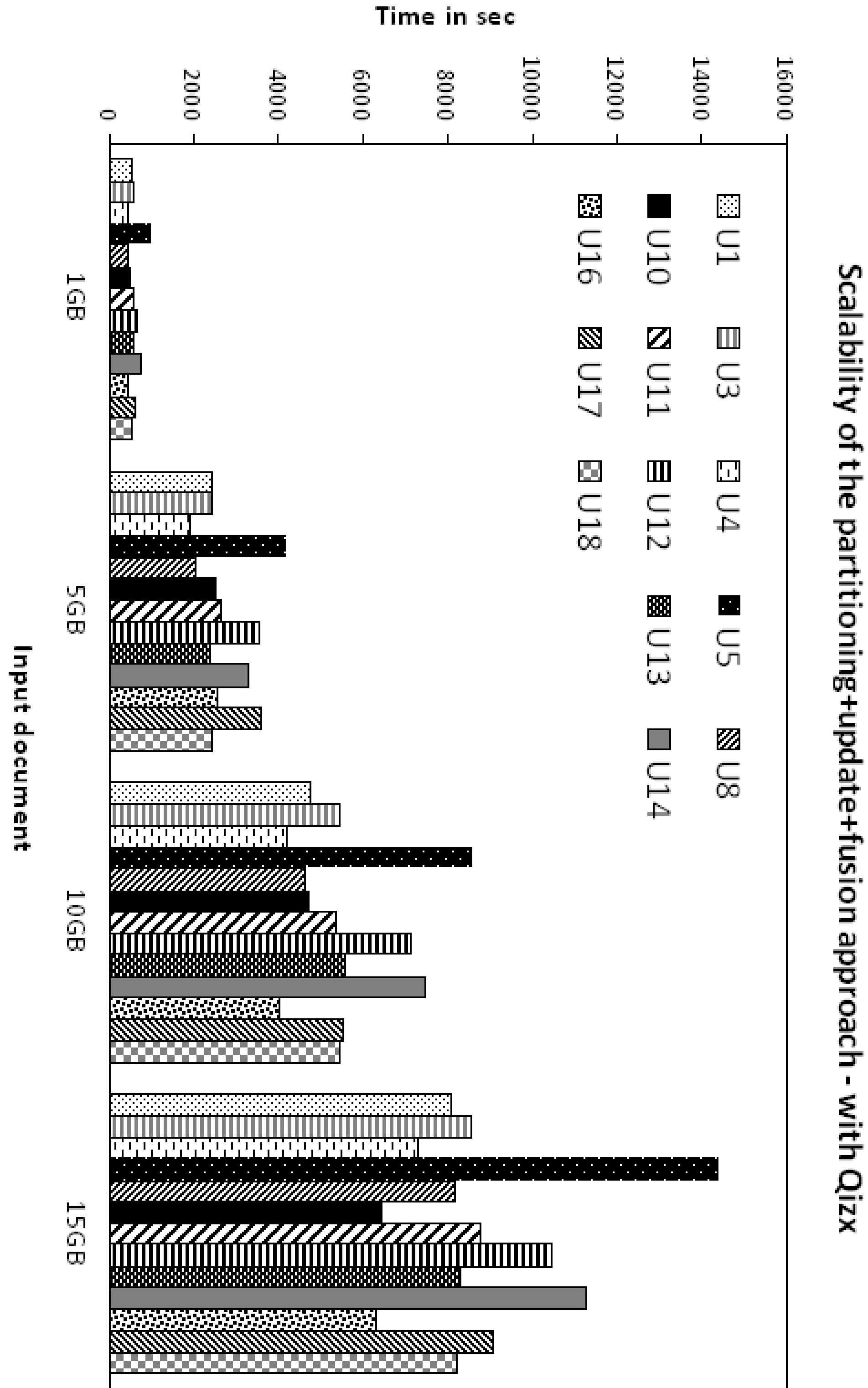



Figure 6.35: Scalability of the partitioning+update+fusion approach - using Qizx.

Chapter 7

# Parallel Query and Update Evaluation

**Contents**


**7.1 MapReduce . . . . . 145**

7.1.1 Logical View . . . . . 147

7.1.2 Execution Overview . . . . . 148

**7.2 parallel evaluation of iterative queries and updates via MapReduce . . . . . 150**

**7.3 conclusive remarks . . . . . 152**


Besides ensuring scalability, our partitioning technique illustrated in previous chapters also has the advantage that it naturally paves the way to parallel processing. This is a consequence of the fact that iterative queries and updates are such that evaluation on a part does not depend on evaluation on another part. As a consequence, parts in a partition can be queried/updated in parallel.

In this chapter, we discuss the main lines of a possible parallel implementation of our partitioning technique by means of the MapReduce programming model [DG08]. We would like to outline that the architecture we propose is the results of a collaboration with Carlo Sartiani (Assistant Professor at Università della Basilicata, Italy) and Maurizio Nole (Master student at Università della Basilicata, Italy).

We first introduce the basics of the MapReduce paradigm in Section 7.1, and then illustrate how our technique can be implemented into a MapReduce platform in Section 7.2. Finally, we draw our conclusion in Section 7.3.

## 7.1 MapReduce

When the first computers were adopted, programs were executed in a sequential manner and by means of a unique processor. Parallelism was introduced after in order to improve performances of some particular tasks, by executing them in parallel on several processors and on different chunks of data. These processors run either on a single computer or on multiple computers via a network. In order to aid programming in this context, parallel programming paradigms have been introduced.

In order to build a parallel program, we need to specify a set of tasks that can be executed concurrently over the same input data, or create several parts of input data on which our tasks are concurrently executed. A typical scenario that is more and more recurring in the context of large data collection generated over the Web, is that where the data collection is split into several parts and some predefined tasks are executed on these parts in parallel. To this end, we have several parallel implementation techniques. The most popular one is called *Master/Worker*.

Typically the *Master* initializes the parallel process, splits it into sub-tasks and assigns one of them to each *Worker*. Once the *Worker* has terminated it return results to the *Master*, which will opportunely combine them with other *Worker* results.

The MapReduce paradigm is based on these principles, and is currently adopted in many contexts where queries have to be executed on large amount of data, and the size of these data is such that a sequential evaluation would require an unacceptable amount of time.

Following [DG08], MapReduce is a parallel framework for processing or distributing large data sets, which often uses a large number of computers (nodes), either referred to a *cluster*, if these nodes are located in the same local network and use similar hardware, or a *grid*, if the nodes are shared across distributed systems, and use different hardware. MapReduce has been first introduced and adopted by Google [DG08].

MapReduce is successfully used in a wide range of applications including: distributed pattern-based searching, distributed sort, web access log states, inverted index construction, document clustering, machine learning [CKL+07], and statistical machine translation. Moreover, this framework has been adapted to several computing environments like multi-core systems [RRP+07], desktop grids [TMC+10], dynamic cloud environments [MTT10] and mobile environments [DKG+10].

MapReduce libraries have been written in many programming languages. The most popular free implementation is Apache Hadoop [had]. The Apache Hadoop offers a framework that allows to perform the distributed processing of large data sets across clusters of computers using the MapReduce model. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. One of the main functionalities it provides is high robustness. The library itself is designed to detect and handle failures at the application layer. Each time a task is detected to have failed, it is restarted on another processing unit. In order to ensure high robustness, Hadoop requires that each task stores results on the distributed file system (HDFS), so that in the case of a single task fails, only its results have to be regenerated, and there is no need to restart all the tasks. Hadoop requires the Java Runtime Environment JRE 1.6 or higher.

### 7.1.1 Logical View

The main goal of the MapReduce paradigm is to provide a model that can be easily adopted by programmers, even if they have no experience with parallel and distributed programming. The possibility of rapid development of parallel programs has been one the main reasons of the success of this paradigm.

The main idea behind MapReduce is to avoid the user to deal with operations that routinely occurs in parallel management of large data repositories. To define a MapReduce *job*, the programmer has to specify two functions, the *Map* function and the *Reduce* function. These functions are assumed to work on a data model consisting of collections of (key, value) pairs. The key component is generally a scalar value, while the value component can also be a complex value like a record coming from a relational database, a textual document (an XML document in particular), or some other complex value.

The semantics of Map and Reduce functions is described below.

- The function *Map*, written by the user, takes one pair of the input dataset $(k_1, v_1)$, and returns a list of pairs $\texttt{list}(k_2, v_2)$. The *Map* function is applied in parallel to every pair in the input dataset. This will produce a list of pairs for each call.

$$Map \quad (k_1, v_1) \quad \rightarrow \quad list(k_2, v_2)$$

- The MapReduce framework collects all pairs with the same key from all lists and groups them together, thus creating one group for each one of the different generated keys.

- The function *Reduce* which is written by the user and applied in parallel to each group, accepts the intermediate key $k_i$ and the set of values $v_i$ for that key. It merges together these values to form a possibly smaller set of values. The intermediate values are supplied to the *Reduce* function via an iterator. This allows the user to handle lists of values that are too big to fit in memory.

$$Reduce \quad (k_2, list(v_2)) \quad \rightarrow \quad list(v3)$$

The following example explains the mechanism of both *Map* and *Reduce* functions. One of the typical problems for which MapReduce can be successfully adopted is that of counting the number of occurrences of each word in large collection of documents. The *Map* and *Reduce* function the programmer has to specify can be as follows:

**Example 12** Consider the following *Map* and *Reduce* functions:

```
map(String key, String value):
// key: document name
// value: document contents
```

```
for each word w in value:
EmitIntermediate(w, "1");

reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int result = 0;
for each v in values:
result += ParseInt(v);
Emit(AsString(result));
```

□

Once these two functions have been specified, their execution happens as follows. The MapReduce framework automatically splits the input key-value collection into several splits (whose size is generally from 16MB to 64MB). Then a number of Map and Reduce Workers are started on several processing units. The framework assigns a split to each Map workers. In the above example, each Map Worker produces a list of key-value pairs, where for each pair the key is a word encountered in one of the documents, and the value is simply 1, to indicate that one occurrence of the word has been encountered.

Outputs of Map workers are processed by the framework so that key-value pairs that Mappers (Map Workers) have produced are partitioned in such a way that all pairs sharing the same key are in the same part. Then the framework assigns a number of such parts to each Reduce Worker.

In the above example, each Reduce worker is guaranteed to have all occurrences of a given word. Once these occurrences are counter, the results is made persistent on the file system. The final result is the concatenation of all Reduce results.

As the example illustrates, operations like initial partitioning of the key-value collections is done by the framework, as well as grouping operations before passing Mappers results to Reducers. This is of particular importance for rapid and safe development of parallel intensive data processing tasks, as the programmer has to concentrate on the pure logic of query he/she needs to execute.

### 7.1.2 Execution Overview

To explain the execution model in more detail we rely on [DG08]. Figure 7.1 illustrates the overall flow of a MapReduce job in the implementation proposed in [DG08]. When the user program calls the MapReduce functions, the following sequence of actions occurs. Note that the numbered labels in Figure 7.1 correspond to the numbers in the list below.

- The MapReduce library in the user program first shreds the input documents into $m$ pieces of typically 16 megabytes to 64 megabytes (MB) per piece. Then it starts up many copies of the program on a cluster of machines.

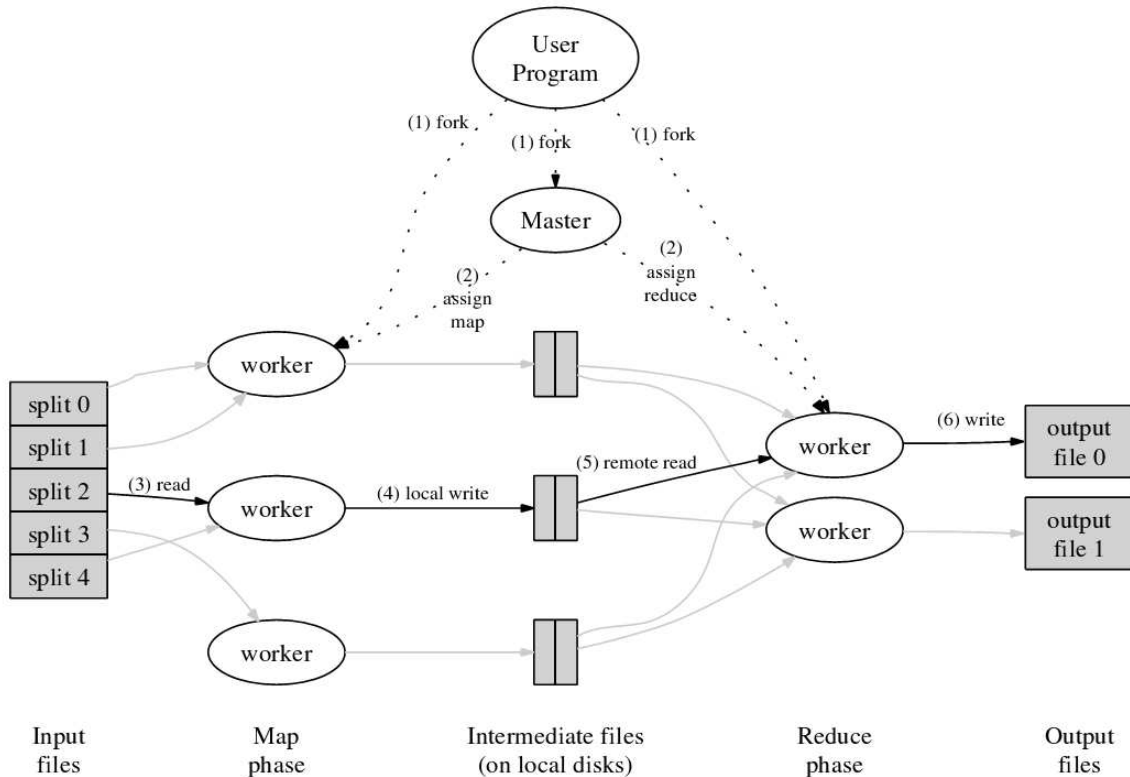



Figure 7.1: Execution overview.

- Only one part of the program is considered as *Master*. While the rest are *Workers* that are assigned work by the *Master*. There are $m$ `Map` tasks and $r$ `Reduce` tasks to assign. The *Master* picks inactive Workers and assigns each one a `Map` task or a `Reduce` task.

- A *Worker* who is assigned a `Map` task reads the contents of the corresponding input shard. It parses key/value pairs out of the input data and passes each pair to the user-defined `Map` function. The intermediate key/value pairs produced by the `Map` function are buffered in memory.

- Periodically, the buffered pairs are written to local disk, partitioned into $r$ regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the *Master*, who is responsible for forwarding these locations to the reduce *Workers*.

- When a `Reduce` *Worker* is notified by the *Master* about these locations, it uses remote procedure calls to read the buffered data from the local disks of the `Map` *Workers*. When a `Reduce` *Worker* has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key

are grouped together. If the amount of intermediate data is too large to fit in memory, an external sort is used.

- The reduce *Worker* iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's `Reduce` function. The output of the `Reduce` function is appended to a final output file for this reduce partition.
- When all `Map` tasks and `Reduce` tasks have been completed, the *Master* wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

After successful completion, the output of the MapReduce execution is available in the $r$ output files. Typically, users do not need to combine these $r$ output files into one file, they often pass these files as input to another MapReduce call, or use them from another distributed application that is able to deal with input that is partitioned into multiple files.

## 7.2 PARALLEL EVALUATION OF ITERATIVE QUERIES AND UPDATES VIA MAPREDUCE

As said before a MapReduce platform can be realized by means of several machines on which the Apache Hadoop open library runs. Hadoop makes all MapReduce functionalities available, and is widely used. So we will refer to it in illustrating how our approach can be transposed into a MapReduce framework.

We first focus on XML partitioning for queries. As seen in Chapter 4, projection can be profitably combined with partitioning so as to lower time overhead in the global query execution. Some features of our technique pose some constraints on the possible resulting MapReduce architecture. We still assume that one document is processed.

First of all, partitioning must be executed by the *Master* (recall the schema given in Figure 7.1) since this operation can not be performed in parallel. As soon as parts are generated, parallel evaluation can be started. In order to accelerate part generation, its better to decouple projection from partitioning.

The resulting execution schema is illustrated in Figure 7.1. The illustrated schema is of the kind Master-Map. This a particular modality under which MapReduce can work according to Hadoop, and is characterized by the fact that only a Master and Map Workers are adopted. In the figure, the local file system is where the input and output queries are stored, and is distinguished from the distributed Hadoop file system (HDFS) which is used to store input, output and intermediary data of a MapReduce job.

As already said, the Master takes the input document and performs the partitioning (without performing projection). As soon as a part is generated, a file

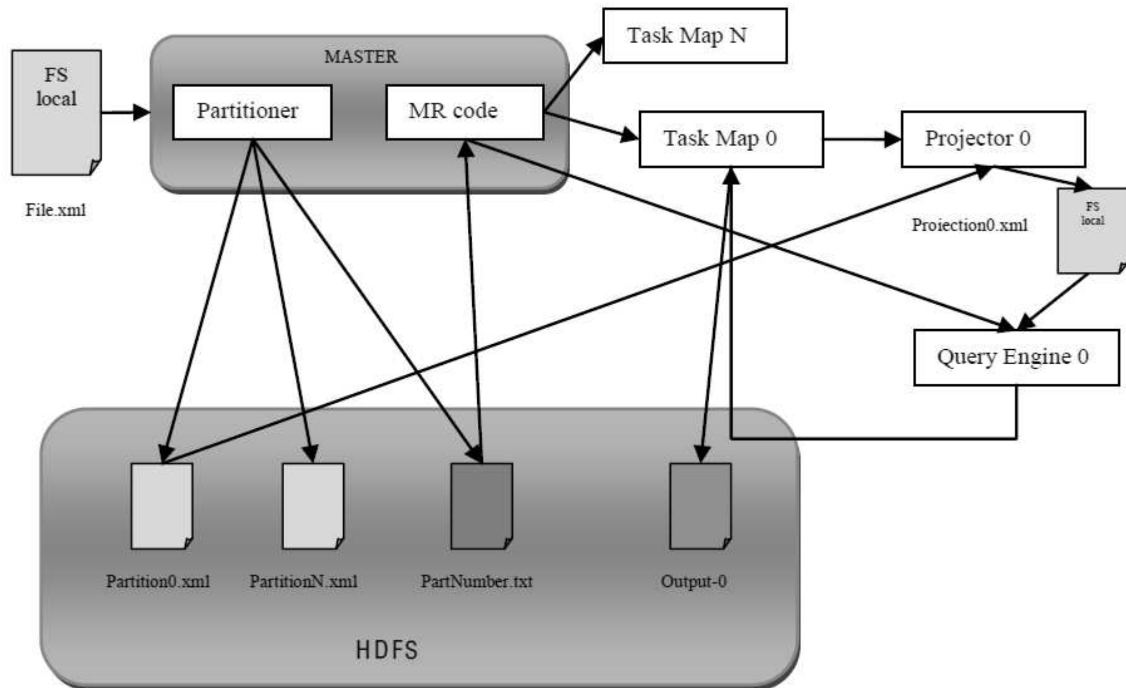



Figure 7.2: Graphical representation of the Master-Map schema.

partition N.xml is stored via HDFS (N indicates the number of the part, and is assumed to start from 0). Another file which is maintained by the *Master* is PartNumber.txt which contains, for each part in the partition, the number of the part and corresponding HDFS URI; this information is needed by a Map Worker in order to recover and process a part.

Actually, the PartNumber.txt file contains the key-value collection which is passed to the MapReduce job. As anticipated, the job only activates Map workers.

Rather than formally specifying the Map function, we describe the tasks it performs. It receives as input a number of pairs (part-number, part-URI) coming from the PartNumber.txt file. For each such pair, then Map worker retrieves the XML file corresponding to the part-number, and executes the projection algorithm on it (in a SAX fashion). The projected part is stored *locally* in order to avoid the overhead implied by HDFS. The Map worker then makes a call to a query engine locally installed, in order to execute the query on the locally stored projected part. The query engine can be any existing query engine, e.g., Saxon or Qizx. Once the query result is available, it is communicated to the output-collector of the Map worker. This collector writes the result on the HDFS support. If the result is relative to the part PartitionJ.xml, then the file including the results is stored into a file Output-J. Once all Map Workers have terminated Output-I.xml files are available on HDFS for being concatenated.

An alternative schema could be of the kind Master-Map-Reduce, where Map Workers perform projections, and Reduce Workers deals with query evaluation on each projected part. This schema seems to ensure an higher parallelism degree, but actually has the drawback that each projected file in order to be passed to a Reduce Worker has to be stored on the HDFS. This operation can be much slower than writing the projection on the local file system of the Map Worker. In particular, this is due to the fact that HDFS handle duplicated versions of stored files, distributed on several nodes connected via the network.

We believe that, when compared to the centralized framework presented in previous chapters, the above Master-Map schema could improve execution time for very large documents especially in those cases where the query performs time consuming operations on each part. Otherwise, the overhead implied by the MapReduce framework could entail higher total execution time.

Concerning updates, a possible schema is totally similar to the above described Master-Map schema. With the difference that Map Workers are activated only to update those parts that really need to. Besides partitioning, also fusion operation should be executed in a sequential fashion.

## 7.3 CONCLUSIVE REMARKS

In this short chapter we have described possible schemes of a MapReduce implementation of our-partitioning based frameworks. The main purpose of the chapter was to highlight another strength of our approach, that is the possibility of parallel query and update evaluation by relying on the MapReduce model.

CHAPTER 8

# Related Works and Conclusion

## Contents


8.1 RELATED WORKS . . . . . . . . . . . . . . . . . . . . . . . . . . . 153

8.2 CONCLUSIVE REMARKS AND FUTURE DIRECTIONS . . . . . . 154


## 8.1 RELATED WORKS

The main aim of our technique is to allow main-memory systems to scale up with respect to document size when querying and updating XML data. We have already commented on main traditional projection based approaches [BBC+11, BCCN06, BCMS09a, BCMS09b, MS03], and seen that these techniques have limitations in terms of scalability. On the positive side, these techniques do not pose restrictions on queries and updates. Differently, we have focused on a fragment of XQuery query and update languages, and proposed a partitioning based techniques that enable main-memory engines to scales up.

Concerning queries, techniques for partitioning XML documents have already been explored. Our technique resembles that of [BLS09] where an horizontal partitioning technique has been proposed in order to ensure parallel execution of single XPath queries. The partitioning technique proposed in this work can be performed on the main-memory representation of the XML document. As a consequence, very large XML documents cannot be managed. In [KÖD10], a *vertical* partitioning technique has been proposed still with the aim of parallel and distributed execution of XPath queries. The technique can handle very large documents, but requires the use of schema information on the input document. Both techniques proposed in [BLS09, KÖD10] require strong interventions inside a query engine. A recent work [GCL12] proposes new efficient algorithms for the distributed evaluation of XPath queries. This work uses horizontal-vertical partitioning, and assumes data have been statically partitioned according to existing techniques.

Differently from the above mentioned works, we address a wide class of XQuery queries, we do not require schema information, and we are able to deal with a workload executed on very large documents. Also, our technique does not require to modify the internal components of a query engine.

Concerning updates, to the best of our knowledge, we are not aware of any exiting partitioning-based techniques. Techniques exist in order to optimize memory

consumptions. The type-based projection technique has already been discussed and performed tests have illustrated improvements of our approach. Another effective technique that can be used to ensure scalability when updating XML files with main-memory engines has been presented in [CGM11]. This technique allows to apply updates in a streaming fashion, so to minimize the memory usage. In most cases the technique has high scalability abilities. At the same time, differently from our approach, this technique requires interventions into query-engine, in order to recover the list of update operations to apply, and apply them in streaming to the input. Also, in the case that a workload of distinct updates has to be applied to a document, this technique requires parsing the input document as many times as the number of updates. In our case, if the workload is iterative, we can perform a unique partitioning, then evaluate the workload on the partition, and finally perform a unique fusion operation.

## 8.2 CONCLUSIVE REMARKS AND FUTURE DIRECTIONS

In this Thesis, we presented a novel partitioning technique for XML document. This technique generalizes existing path-based approaches, and applies to a large class of queries and updates.

A distinctive feature of our approach is that it is schema-less. It uses path information coming from the query/update in order to perform the static analysis needed to recognize the iterative nature of the query/update, and use path information to perform partitioning. Another distinctive feature, is that the approach can be easily plugged on any main-memory system, as no intervention in the internal machinery of the system is required. Finally, we have seen that our approach is amenable to an easy transposition in a MapReduce like processing framework, thus allowing parallel querying and updating of parts in the partition. For huge document sets, and in the presence of a reasonable big cluster of machines, this could entail consistent time reduction with respect to the sequential approach we proposed here (parts are queried/updated sequentially).

We see several possible future directions. First of all, we would like to extend the approach to larger fragments of XQuery, and in particular to queries performing group-by operations and aggregations. Also, we would like to extend the technique in the case where queries performs joins. Especially in this second case, some performed tests have revealed that execution time can be huge with the use of main-memory system. To enable partitioning query/update evaluation would need to be split in several subtasks, some of which use partitioning. Then partial results of each task should be recombined. In our opinion, in this scenario a MapReduce approach could help in reducing execution time.

As a second future work, we would like to explore possibilities of handling workloads formed by both queries and updates. Once the path analysis is available to recognize the iterative nature of the workload, and to perform partitioning, this last

one could be performed once and reused many times, until the workload changes. An advantage would also come to the reduction of fusion operations, which would become useless as long as the workload is stable.

Finally, we plan to further investigate MapReduce implementations of our approach, along the lines of schemes illustrated in Chapter 7. In particular, we will focus on implementation issues, and in adapting our code to the MapReduce framework. In this context, we will also focus on experimental tests in order to realize for which kind of queries/updates a MapReduce execution is faster than a traditional centralized execution.

Appendix A

# XQuery Expressions and XQuery Updates

## A.1 XMark Queries proposed in [SWK+02a]

- Return the name of the person with ID person0.

```
Q1 = for $b in doc("xmark.xml")/site/people/person[@id="person0"]
     return $b/name/text()
```

- Return the initial increases of all open auctions.

```
Q2 = for $b in doc("xmark.xml")/site/open_auctions/open_auction
     return
     <increase>
          {$b/bidder[1]/increase/text()}
     </increase>
```

- Return the IDs of all open auctions whose current increase is at least twice as high as the initial increase.

```
Q3 = for $b in doc("xmark.xml")/site/open_auctions/open_auction
     where
     zero-or-one($b/bidder[1]/increase/text()) * 2 <=
     $b/bidder[last()]/increase/text()
     return
       <increase
           first="{$b/bidder[1]/increase/text()}"
           last="{$b/bidder[last()]/increase/text()}"/>
```

- List the reserves of those open auctions where a certain person issued a bid before another person.

```
Q4 = for $b in doc("xmark.xml")/site/open_auctions/open_auction
     where
      some $pr1 in $b/bidder/personref[@person = "person20"],
       $pr2 in $b/bidder/personref[@person = "person51"]
  satisfies $pr1 << $pr2
return <history>{$b/reserve/text()}</history>
```

- How many sold items cost more than 40?

```
Q5 = let $auction := doc("xmark.xml") return
     count(
       for $i in $auction/site/closed_auctions/closed_auction
       where $i/price/text() >= 40
       return $i/price)
```

- How many items are listed on all continents?

```
Q6 = for $b in doc("xmark.xml")//site/regions
     return count($b//item)
```

- List all persons according to their interest; use French markup in the result.

```
Q10 = let $auction := doc("xmark.xml") return
for $i in
  distinct-values($auction/site/people/person/profile/
  interest/@category)
let $p :=
  for $t in $auction/site/people/person
  where $t/profile/interest/@category = $i
  return
    <personne>
      <statistiques>
        <sexe>{$t/profile/gender/text()}</sexe>
        <age>{$t/profile/age/text()}</age>
        <education>{$t/profile/education/text()}</education>
        <revenu>{fn:data($t/profile/@income)}</revenu>
      </statistiques>
      <coordonnees>
        <nom>{$t/name/text()}</nom>
        <rue>{$t/address/street/text()}</rue>
        <ville>{$t/address/city/text()}</ville>
        <pays>{$t/address/country/text()}</pays>
        <reseau>
          <courrier>{$t/emailaddress/text()}</courrier>
          <pagePerso>{$t/homepage/text()}</pagePerso>
        </reseau>
      </coordonnees>
      <cartePaiement>{$t/creditcard/text()}</cartePaiement>
    </personne>
return
   <categorie>{<id>{$i}</id>, $p}</categorie>
```

- List the names of items registered in Australia along with their descriptions.

```
Q13 = for $i in doc("xmark.xml")/site/regions/australia/item
      return
        <item name="{$i/name/text()}">{$i/description}</item>
```

- Return the names of all items whose description contains the word 'gold'.

```
Q14 = for $i in doc("auction.xml")/site//item
      where
         contains(string(exactly-one($i/description)), "gold")
      return $i/name/text()
```

- Print the keywords in emphasis in annotations of closed auctions.

```
Q15 = for $a in
      doc("auction.xml")/site/closed_auctions/closed_auction/
      annotation/
      description/
      parlist/
      listitem/
      parlist/
      listitem/
      text/
      emph/
      keyword/
      text()
   return <text>{$a}</text>
```

- Return the IDs of those auctions that have one or more keywords in emphasis.

```
Q16 = for $a in doc("xmark.xml")/site/closed_auctions/closed_auction
      where
       not(
         empty(
           $a/annotation/description/parlist/listitem/parlist/
           listitem/
           text/
           emph/
           keyword/
           text()
         )
       )
       return
          <person id="{$a/seller/@person}"/>
```

- Which persons don't have a homepage?

```
Q17 = for $p in doc("xmark.xml")/site/people/person
         where empty($p/homepage/text())
      return <person name="{$p/name/text()}"/>
```

- Convert the currency of the reserve of all open auctions to another currency.

```
declare namespace local = "http://www.foobar.org";
declare function local:convert($v as xs:decimal?) as xs:decimal?
{ 2.20371 * $v (: convert Dfl to Euro :)
};
Q18 = let $auction := doc("auction.xml") return
      for $i in $auction/site/open_auctions/open_auction
      return local:convert(zero-or-one($i/reserve))
```

- Give an alphabetically ordered list of all items along with their location.

```
Q19 = for $b in doc("auction.xml")/site/regions//item
      let $k := $b/name/text()
      order by zero-or-one($b/location)ascending empty greatest
      return
        <item name="{$k}">
            {$b/location/text()}
        </item>
```

- Group customers by their income and output the cardinality of each group.

```
Q20 = let $auction := doc("auction.xml") return
<result>
  <preferred>
    {count(
       $auction/site/people/person/profile[@income >= 100000]
     )}
  </preferred>
  <standard>
    {
      count(
        $auction/site/people/person/
         profile[@income < 100000 and @income >= 30000]
      )
    }
  </standard>
```

```
    <challenge>
      {count($auction/site/people/person/profile[@income < 30000])}
    </challenge>
    <na>
      {
        count(
          for $p in $auction/site/people/person
          where empty($p/profile/@income)
          return $p
        )
      }
    </na>
  </result>
```

## A.2 Update Expressions used in [BBC+11]

U1. Insert a new node `<annotation>Empty Annotation</annotation>` as last of each `closed_auction` node.

```
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. Replace `address` of each element which its country is `United States` with another address.

```
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. Replace each `United States` location with the value `USA`.

```
for $x in $doc/site/regions//item/location
```

```
    where $x/text()="United States"
  return (replace value of node $x with "USA")
```

U4. Delete all subtrees rooted at `mail` from each item node.

```
delete nodes $doc/site/regions//item/mailbox/mail
```

U5. Rename each `bold` node with `emph`.

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

U6. Insert new `homepage` node for each `person` which does not have a `homepage`.

```
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. Insert ....

```
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/country="Malaysia"
return insert node $x/address
after $y/emailaddress
```

## A.3 XQuery Update expressions in [Sah11]

```
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

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
```

## A.4 XQuery Update Facilities 1.0 Use Cases

1- Add a new user (with no rating) to the users.xml view.

```
insert nodes
  <user_tuple>
    <userid>U07</userid>
    <name>Annabel Lee</name>
  </user_tuple>
into doc("users.xml")/users
```

2- Enter a bid for user Annabel Lee on February 1st, 1999 for 60 dollars on item 1001.

```
let $uid :=
doc("users.xml")/users/user_tuple[name="Annabel Lee"]/userid
return
  insert nodes
    <bid_tuple>
      <userid>{data($uid)}</userid>
      <itemno>1001</itemno>
      <bid>60</bid>
      <bid_date>1999-02-01</bid_date>
    </bid_tuple>
  into doc("bids.xml")/bids
```

3- Insert a new bid for Annabel Lee on item 1002, adding 10% to the best bid received so far for this item.

```
let $uid := doc("users.xml")/users/user_tuple[name="Annabel Lee"]/userid
let $topbid := max(doc("bids.xml")/bids/bid_tuple[itemno=1002]/bid)
return
  insert nodes
    <bid_tuple>
      <userid>{data($uid)}</userid>
      <itemno>1002</itemno>
      <bid>{$topbid*1.1}</bid>
      <bid_date>1999-02-01</bid_date>
    </bid_tuple>
  into doc("bids.xml")/bids
```

4- Set Annabel Lee's rating to B.

```
let $user := doc("users.xml")/users/user_tuple[name="Annabel Lee"]
return
  if ($user/rating)
```

```
    then replace value of node $user/rating with "B"
    else insert node <rating>B</rating> into $user
```

5- Place a bid for Annabel Lee on item 1007, adding 10% to the best bid received so far on that item, but only if the bid amount does not exceed a given limit. The first query illustrates the desired behavior if the limit is exceeded.

```
let $uid := doc("users.xml")/users/user_tuple[name="Annabel Lee"]/userid
let $topbid := max(doc("bids.xml")/bids/bid_tuple[itemno=1007]/bid)
where $topbid*1.1 <= 200
return
  insert nodes
    <bid_tuple>
      <userid>{data($uid)}</userid>
      <itemno>1007</itemno>
      <bid>{$topbid*1.1}</bid>
      <bid_date>1999-02-01</bid_date>
    </bid_tuple>
  into doc("bids.xml")/bids
```

6- Place a bid for Annabel Lee on item 1007, adding 10% to the best bid received so far on that item, but only if the bid amount does not exceed 500. This illustrates the behavior when the resulting value is within the limit.

```
let $uid := doc("users.xml")/users/user_tuple[name="Annabel Lee"]/userid
let $topbid := max(doc("bids.xml")/bids/bid_tuple[itemno=1007]/bid)
where $topbid*1.1 <= 500
return
  insert nodes
    <bid_tuple>
      <userid>{data($uid)}</userid>
      <itemno>1007</itemno>
      <bid>{$topbid*1.1}</bid>
      <bid_date>1999-02-01</bid_date>
    </bid_tuple>
  into doc("bids.xml")/bids
```

7- Erase user Dee Linquent and the corresponding associated items and bids.

```
let $user := doc("users.xml")/users/user_tuple[name="Dee Linquent"]
let $items := doc("items.xml")/items/item_tuple[offered_by=$user/userid]
let $bids := doc("bids.xml")/bids/bid_tuple[userid=$user/userid]
return (
  delete nodes $user,
  delete nodes $items,
```

```
  delete nodes $bids
)
```

8- Erase user Dee Linquent and the corresponding associated items and bids.

```
let $user := doc("users.xml")/users/user_tuple[name="Dee Linquent"]
let $items := doc("items.xml")/items/item_tuple[offered_by=$user/userid]
let $bids := doc("bids.xml")/bids/bid_tuple[userid=$user/userid]
return
  delete nodes $user, $items, $bids
```

9- Add the element <comment>This is a bargain !</comment> as the last child of the <item> element describing item 1002.

```
insert nodes
  <comment>This is a bargain !</comment>
as last into doc("items.xml")/items/item_tuple[itemno=1002]
```

10- Place a bid for Annabel Lee on item 1010, which does not exist in "items.xml". In this query, we assume that a referential integrity constraint in the underlying database system requires that no bid can be placed on an item unless it exists in the database.

```
let $uid := doc("users.xml")/users/user_tuple[name="Annabel Lee"]/userid
return
  insert nodes
    <bid_tuple>
      <userid>{data($uid)}</userid>
      <itemno>1010</itemno>
      <bid>60</bid>
      <bid_date>2006-04-23</bid_date>
    </bid_tuple>
  into doc("bids.xml")/bids
```

11- Add a bid for Annabel Lee on item 1002, at a price 5 dollars below the current highest bid. A trigger in the underlying database ensures that a bid cannot be made at a lower price than the highest bid made so far on that item.

```
let $uid := doc("users.xml")/users/user_tuple[name="Annabel Lee"]/userid
let $topbid := max(doc("bids.xml")//bid_tuple[itemno=1002]/bid)
return
  insert nodes
    <bid_tuple>
      <userid>{data($uid)}</userid>
      <itemno>1002</itemno>
      <bid>{$topbid - 5.00}</bid>
```

```
      <bid_date>2006-04-23</bid_date>
    </bid_tuple>
  into doc("bids.xml")/bids
```

12- Delete all parts in "part-tree.xml".

```
delete nodes doc("part-tree.xml")//part
```

13- Delete all parts belonging to a car in "part-tree.xml", leaving the car itself.

```
delete nodes doc("part-tree.xml")//part[@name="car"]//part
```

14- Delete all parts belonging to a car in "part-list.xml", leaving the car itself.

```
for $pt in doc("part-tree.xml")//part[@name="car"]//part,
    $pl in doc("part-list.xml")//part
where $pt/@partid eq $pl/@partid
return
  delete nodes $pl
```

15- Add a radio to the car in "part-tree.xml", using a part number that hasn't been taken.

```
let $next := max(doc("part-tree.xml")//@partid) + 1
  return
    insert nodes <part partid="{$next}" name="radio"/>
       into
       doc("part-tree.xml")//part[@partid=0 and @name="car"]
```

16- The head office has adopted a new numbering scheme. In "part-tree.xml", add 1000 to all part numbers for cars, 2000 to all part numbers for skateboards, and 3000 to all part numbers for canoes.

```
for $keyword at $i in ("car", "skateboard", "canoe"),
    $parent in doc("part-tree.xml")//part[@name=$keyword]
let $descendants := $parent//part
for $p in ($parent, $descendants)
return
  replace value of node $p/@partid with $i*1000+$p/@partid
```

# Bibliography

[Baa12] M.A Baazizi, *Analyse statique pour l'optimisation des mises à Analyse statique pour l'optimisation des mises à jour de documents XML temporels*, Ph.D. thesis, University of Paris Sud, 2012. 40

[bas] `http://basex.org/`. 9, 89, 101, 136, 143

[BBC+10] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon, *XML Path Language (XPath) 2.0 (Second Edition)*, Tech. report, World Wide Web Consortium, December 2010, W3C Recommendation. 56

[BBC+11] Mohamed-Amine Baazizi, Nicole Bidoit, Dario Colazzo, Noor Malla, and Marina Sahakyan, *Projection for XML Update Optimization*, Proceedings of the 14th International Conference on Extending Database Technology (New York, NY, USA), EDBT/ICDT, ACM, 2011, pp. 307–318. iv, 5, 7, 8, 9, 10, 15, 16, 21, 31, 32, 39, 40, 45, 101, 102, 104, 115, 153, 161

[BC09] M. Benedikt and J. Cheney, *Semantics, Types and Effects for XML Updates*, Database Programming Languages, Springer, 2009, pp. 1–17. 54, 57

[BCCN06] Véronique Benzaken, Giuseppe Castagna, Dario Colazzo, and Kim Nguyen, *Type-Based XML Projection*, VLDB (Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, eds.), ACM, 2006, pp. 271–282. 4, 8, 14, 31, 40, 41, 45, 58, 153

[BCMS09a] N. Bidoit, D. Colazzo, N. Malla, and M. Sahakyan, *Optimisation de Mises à jour XML par typage et projection*, BDA, Bases de Données Avancées Conférence, 2009. 8, 31, 153

[BCMS09b] ———, *Projection based optimization for xml updates*, 1st International Workshop on Schema Languages for XML, X-Schemas, 2009. 8, 31, 153

[BLS09] R. Bordawekar, L. Lim, and O. Shmueli, *Parallelization of XPath queries using multi-core processors: challenges and experiences*, Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, 2009, pp. 180–191. 153

[BPMM08] T. Bray, J. Paoli, M. McQueen, and E. Maler, *Extensible Markup Language(XML)1.0(Fifth Edition)*, 2008. 19

[CGM11] Federico Cavalieri, Giovanna Guerrini, and Marco Mesiti, *Dynamic reasoning on XML updates*, EDBT, International Conference on Extending Database Technology, 2011, pp. 165–176. 154

[CKL+07] C.T. Chu, S.K. Kim, Y.A. Lin, Y.Y. Yu, G. Bradski, A.Y. Ng, and K. Olukotun, *Map-reduce for machine learning on multicore*, Advances in neural information processing systems (2007), 281. 146

[DG08] J. Dean and S. Ghemawat, *MapReduce: Simplified data processing on large clusters*, Communications of the ACM **51** (2008), no. 1, 107–113. 6, 8, 10, 15, 17, 145, 146, 148

[DKG+10] A. Dou, V. Kalogeraki, D. Gunopulos, T. Mielikainen, and V.H. Tuulos, *Misco: A MapReduce framework for mobile systems*, Proceedings of the 3rd International Conference on PErvasive Technologies Related to Assistive Environments, ACM, 2010, p. 32. 146

[Dra02] W3C Working Draft, *XPath 2.0*, November 2002. 32

[exi] `http://exist.sourceforge.net/`. 3, 9, 13, 101

[gal] `http://galax.sourceforge.net/about.html`. 3, 13

[GCL12] Anastasios Kementsietsidis Jianzhong Li Gao Cong, Wenfei Fan and Xianmin Liu, *Partial Evaluation for Distributed XPath Query Processing and Beyond*, ACM Trans. Database System (2012), To appear. 4, 13, 153

[Gro03] W3C Working Group, *XML Path Language (XPath) 2.0*, May 2003. 8, 19, 22

[Gro11a] ———, *XQuery Update Facility 1.0*, March 2011. 3, 8, 13, 19, 27

[Gro11b] ———, *XQuery Update Facility 1.0 use cases*, January 2011. 3, 13, 104

[had] `http://hadoop.apache.org/`. 146

[KCD+03] H. Katz, D. Chamberlin, D. Draper, M. Fernandez, M. kay, J. Robie, M. Rys, J. Siméon, J. Tivyand, and P. Wadler, *XQuery from the Experts*, Etats-Unis, 2003. 22

[KÖD10] Patrick Kling, M. Tamer Özsu, and Khuzaima Daudjee, *Generating efficient execution plans for vertically partitioned xml databases*, PVLDB **4** (2010), no. 1, 1–11. 153

[KSS08] C. Koch, S. Scherzinger, and M. Schmidt, *XML prefiltering as a string matching problem*, Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on, IEEE, 2008, pp. 626–635. 4, 14

[MS03] A. Marian and J. Siméon, *Projecting XML documents*, Proceedings of the 29th international conference on Very Large Data Bases, vol. 29, VLDB Endowment, 2003, pp. 213–224. v, 4, 8, 9, 10, 14, 31, 32, 33, 34, 35, 58, 70, 75, 76, 102, 153

[MTT10] F. Marozzo, D. Talia, and P. Trunfio, *A peer-to-peer framework for supporting mapreduce applications in dynamic cloud environments*, Cloud Computing (2010), 113–125. 146

[Nic12] Nicole Bidoit and Dario Colazzo and Noor Malla and Carlo Sartiani, *Partitioning XML Documents for Iterative Queries*, IDEAS, 16th International Database Engineering & Applications Symposium, 2012. 6, 15

[qiz] `http://www.xmlmind.com/qizx/`. 3, 7, 9, 13, 17, 38, 89, 101, 136

[RRP+07] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, *Evaluating MapReduce for multi-core and multiprocessor systems*, High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on, Ieee, 2007, pp. 13–24. 146

[Sah11] M. Sahakyan, *Main Memory XML Update Optimization: algorithms and experiments*, Ph.D. thesis, DB Team - LRI - UNIVERSITY PARIS SUD 11- France, November 2011. iv, xi, 5, 15, 40, 45, 46, 104, 136, 162, 163

[sax] `http://saxon.sourceforge.net/`. 3, 7, 13, 17, 38, 89, 136

[SWK+02a] A. Schmidt, F. Waas, M. Kersten, M.J. Carey, I. Manolescu, and R. Busse, *XMark: A Benchmark for XML data management*, Proceedings of the 28th international conference on Very Large Data Bases, VLDB Endowment, 2002, p. 985. iv, 32, 89, 136, 157, 159

[SWK+02b] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse, *XMark: A Benchmark for XML Data Management*, VLDB, Morgan Kaufmann, 2002, pp. 974–985. 5

[TMC+10] B. Tang, M. Moca, S. Chevalier, H. He, and G. Fedak, *Towards mapreduce for desktop grid computing*, P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2010 International Conference on, IEEE, 2010, pp. 193–200. 146

[ver00] SAX 2.0.1 version, *The official website for SAX Parser*, 2000. 33, 78, 121

[ver11] XML version, *The DBLP Computer Science Bibliography*, Avril 2011. 37, 90

[W3S10] W3Schools, *XQuery 1.0: An XML Query Language (Second Edition)*, December 2010. 3, 8, 13, 19