Lecture Notes in Computer Science 4704

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Denilson Barbosa Angela Bonifati
Zohra Bellahsène Ela Hunt
Rainer Unland (Eds.)

Database and XML Technologies

5th International
XML Database Symposium, XSym 2007
Vienna, Austria, September 23-24, 2007
Proceedings

Springer

Volume Editors

Denilson Barbosa
University of Calgary, Canada
E-mail: denilson@ucalgary.ca

Angela Bonifati
Italian National Research Council (Icar-CNR), Rende CS, Italy
E-mail: bonifati@icar.cnr.it

Zohra Bellahsène
Université Montpellier, 2LIRMM - UMR 5506 CNRS, Montpellier, France
E-mail: bella@lirmm.fr

Ela Hunt
ETH-Zentrum, Institut für Informationssysteme, Zürich, Switzerland
E-mail: elahunt@inf.ethz.ch

Rainer Unland
University of Duisburg-Essen, Essen, Germany
E-mail: UnlandR@informatik.uni-essen.de

Library of Congress Control Number: 2007935597

CR Subject Classification (1998): H.2, H.3, H.4, D.2, C.2.4

LNCS Sublibrary: SL 3 – Information Systems and Application, incl. Internet/Web and HCI

ISSN 0302-9743
ISBN-10 3-540-75287-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-75287-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12167241 06/3180 5 4 3 2 1 0

Preface

Since its first edition in 2003, the XML Database Symposium series (XSym) has been a forum for academics, practitioners, users and vendors to discuss the use of and synergy between databases and XML. The previous symposia have provided opportunities for timely discussions on a broad range of topics pertaining to the theory and practice of XML data management and its applications. XSym 2007 continued this tradition with a program consisting of three research sessions, two keynote talks and a panel. We received 28 full paper submissions, out of which 8 were accepted for publication in these proceedings. Each submitted paper underwent a rigorous and careful review by a minimum of four independent referees.

The contributions in these proceedings are a fine sample of the very best current research in XPath and XQuery processing, XML Updates, Temporal XML and Constraints. This volume also contains two invited short papers by the XSym 2007 keynote speakers: Leonid Libkin (University of Edinburgh, UK) contributed a thorough discussion of Normalization Theory for XML, while Erhard Rahm (University of Leipzig, Germany) contributed an interesting overview of the work on Dynamic Fusion of Web Data carried out by his research group. This year, XSym will top its program off by an exciting panel with the title "The Generation Y of XML Schema Matching." It was proposed and organized by Avigdor Gal from Technion – Israel Institute of Technology.

The organizers would like to express their gratitude to the authors, for submitting their work to XSym 2007, and to the Program Committee, for providing very thorough evaluations of the submitted papers as well as for the discussions that followed under significant time constraints. We would also like to thank the invited speakers and the panel moderator for their efforts in contributing to XSym 2007. Finally, we are also grateful to Microsoft for their generous sponsorship, Andrei Voronkov for the EasyChair conference management system, and the local organizers for their effort in making XSym 2007 a pleasant and successful event.

July 2007

Denilson Barbosa
Angela Bonifati
Zohra Bellahsene
Ela Hunt
Rainer Unland

Organization

General Chair

Zohra Bellahsene, LIRMM (France)

Program Committee Chairs

Denilson Barbosa, University of Calgary (Canada)
Angela Bonifati, Icar CNR (Italy)

Proceedings

Rainer Unland, University of Duisburg-Essen (Germany)

Communications and Sponsorship

Ela Hunt, ETH Zurich (Switzerland)

Local Chairs

Bernhard Schandl, University of Vienna, Austria
Eugen Mühlvenzl, Austrian Computer Society, Austria
Robert Mosser, University of Vienna, Austria

International Program Committee

Ashraf Aboulnaga, University of Waterloo (Canada)
Sihem Amer Yahia, Yahoo! Research (USA)
Pablo Barceló, Universidad de Chile (Chile)
Zohra Bellahsene, LIRMM CNRS/ University of Montpellier 2 (France)
Veronique Benzaken, Universite Paris Sud 11 (France)
Vanessa Braganholo, Universidade Federal do Rio de Janeiro (Brazil)
Mariano Consens, University of Toronto (Canada)
Irini Fundulaki, University of Edinburgh (UK)
Avigdor Gal, Technion (Israel)
Giorgio Ghelli, Università di Pisa (Italy)
Carlos Heuser, Universidade Federal do Rio Grande do Sul (Brazil)
Ela Hunt, ETH Zurich (Switzerland)

Ihab F. Ilyas, University of Waterloo (Canada)
Anastasios Kementsietsidis, University of Edinburgh (UK)
Carl-Christian Kanne, University of Mannheim (Germany)
Mirella Moro, University of California at Riverside (USA)
Giansalvatore Mecca, University of Basilicata (Italy)
Jérôme Siméon, IBM T.J. Watson Research (USA)
Altigran Soares da Silva, Universidade Federal do Amazonas (Brazil)
Yannis Velegrakis, University of Trento (Italy)
Jeffrey Xu Yu, The Chinese University of Hong Kong (China)
Ning Zhang, Oracle Corp. (USA)

External Reviewers

Marcelo Arenas, Pontificia Universidad Católica de Chile (Chile)
Ladjel Bellatreche, Lab. d'Informatique Scientifique et Industrielle (France)
George Beskales, University of Waterloo (Canada)
Luc Bouganim, INRIA Rocquencourt (France)
Bogdan Cautis, INRIA Futurs (France)
James Cheney, University of Edinburgh (UK)
Nina Edelweiss, Universidade Federal do Rio Grande do Sul (Brazil)
Mirian Halfeld Ferrari, Université François-Rabelais de Tours (France)
Renata Galante, Universidade Federal do Rio Grande do Sul (Brazil)
Françoise Gire, Université de Paris 1 (France)
Greg Leighton, University of Calgary (Canada)
Chengfei Liu, Swinburne University of Technology (Australia)
Savvas Makalias, University of Edinburgh (UK)
Viviane Moreira Orengo, Universidade Federal do Rio Grande do Sul (Brazil)
Lu Qin, The Chinese University of Hong Kong (China)
Nikos Rizopoulos, Imperial College London (UK)
Flavio Rizzolo, University of Toronto (Canada)
Mark Roantree, Dublin City University (Ireland)
Kristoffer Rose, IBM T.J. Watson Research (USA)
Stefanie Scherzinger, Saarland University (Germany)
Mohamed A. Soliman, University of Waterloo (Canada)
Nan Tang, The Chinese University of Hong Kong (China)
Junhu Wang, Griffith University, Gold Coast (Australia)
Qiang Wang, University of Waterloo (Canada)

Table of Contents

Panel

Normalization Theory for XML

Leonid Libkin

School of Informatics, University of Edinburgh
libkin@inf.ed.ac.uk

Abstract. Specifications of XML documents typically consist of typing information (e.g., a DTD), and integrity constraints. Just like relational schema specifications, not all are good – some are prone to redundancies and update anomalies. In the relational world we have a well-developed theory of data design (also known as normalization). A few definitions of XML normal forms have been proposed, but the main question is *why* a particular design is good. In the XML world, we still lack universally accepted query languages such as relational algebra, or update languages that let us reason about storage redundancies, lossless decompositions, and update anomalies. A better approach, therefore, is to come up with notions of good design based on the intrinsic properties of the model itself. We present such an approach, based on Shannon's information theory, and show how it applies to relational normal forms as well as to XML design, for both native and relational storage.

1 Introduction

Data organization is one of the most fundamental topics in the study of databases. In fact, the concept of normalization was proposed by Codd [5] in 1971 – a mere year after he introduced the relational model. By 1974, the standard 2nd, 3rd, and Boyce-Codd [6] normal forms (2NF, 3NF, BCNF) had been developed. Bernstein's work on 3NF in the mid 1970s [3] is often viewed as the birth of database theory. It was understood very early by both database practitioners and theoreticians that having well-organized and well-designed databases is absolutely crucial for storing, querying, and updating data. Already in the 1980s, the standard normal forms such as 3NF and BCNF were covered by the majority of database texts.

After three decades of relational dominance, we have seen a new data format that is extremely widely used and can seriously challenge relational databases. Thanks to the proliferation of data on the web, much of it now appears in various markup language formats, of which XML is the most common one. Given the amount of data available in XML, it is natural to expect that some of XML designs will exhibit problems similar to those of relational designs, and indeed this is the case. As a simplest example, we can represent an arbitrary relational schema $R_1(A_1^1, \ldots, A_{n_1}^1), \ldots, R_1(A_1^k, \ldots, A_{n_k}^k)$ in XML by means of the following DTD D_1:

$$db \to R_1, \ldots, R_k$$
$$R_i \to \mathsf{tuple}_i^*, \quad i \le k$$

D. Barbosa et al. (Eds.): XSym 2007, LNCS 4704, pp. 1–13, 2007.

that declares $A^i_1, \ldots, A^i_{n_i}$ to be the attributes of tuple_i. This way, a bad relational design translates into a bad XML design, inheriting its problems such as redundancies and update anomalies. But there are other ways to have designs prone to update anomalies due to the *hierarchical* nature of XML. Consider, for example, the following DTD D_2 for storing information about conference publications:

$$\begin{gathered} db \to \mathit{conf}^* \\ \mathit{conf} \to \mathit{paper}^* \\ \mathit{paper} \to \mathit{author}^+, \mathit{title}, \mathit{year} \end{gathered}$$

with *author*, *title*, and *year* elements each carrying an attribute with its value. Now suppose we know – and this is a reasonable assumption – that all papers in a conference have the same publication year. Then the *year* information is redundant, as it is stored repeatedly for all papers in the conference. In addition it is likely to lead to update problems: if a year needs to be changed, one cannot do it just once; instead it needs to be changed for every paper in the conference.

To build foundations of good XML design, we need to answer the following two questions:

1. How do we recognize poor XML designs, and how do we convert them to good designs? In other words, we want to develop a theory of normalization for XML.
2. What constitutes a good XML design? In other words, we want to formulate criteria for good XML designs. In the relation case, one usually appeals to the intuitive notions of redundancy and update anomaly. But this approach is problematic in the case of XML for three reasons:
   - First, due to the complicated hierarchical structure of XML documents, it is harder to see when a schema contains redundancies.
   - Second, the notion of an update is not nearly as clean as the notion of relational updates, which makes it hard to say what constitutes an update anomaly (especially in the absence of a universally accepted notion of XML updates).
   - Third, there is no query language with the same yardstick status as relational algebra has for relational databases. The process of normalization needs to be lossless (meaning that the original data can be recovered from a differently designed schema). Thus, the notion of losslessness depends on a query language.

   Thus, we need an approach based on "standards-free" XML concepts, that is, concepts that will not change even if the W3C comes up with a new query or update language for XML tomorrow.

Several XML normal forms have been proposed recently, see, e.g., [1,18,17,19,8]. They differ in terms of schema and constraint description, but are based on essentially the same set of transformations, first proposed in [1]. As for the work on justification of XML normal forms, an approach was proposed in [2] based on information theory. The idea of the approach is that we measure the amount of redundancy in a design *regardless* of any query/update language for a data model.

This approach, when applied to relational design, confirms our intuitive view of which designs are good [2,12], and then it can be applied in the case of XML to reason about XML designs for both native [2] and relational storage [13].

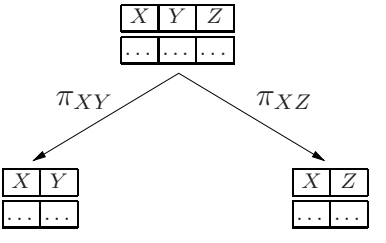
We give a brief survey of these developments. We start with a quick overview of relational normalization. After that, we introduce the main idea of XML normalization. We then present the intuition behind the information-theoretic approach to normalization, and show how it justifies commonly used relational normal forms. Finally, we analyze the implications of the information-theoretic approach to XML design.

2 Relational Normalization: A Brief Reminder

We give an overview of two normal forms based on functional dependencies (FDs): the third normal form 3NF, and the Boyce-Codd normal form BCNF.

If integrity constraints are specified as FDs, the main cause for problems in a design is a functional dependency $X \to Y$ in which the left-hand side X is not a key. For example, if we have three attributes A, B, C and an FD $A \to B$, then a given value a of A can be associated with an arbitrary number of values of the C-attribute, i.e. we can have tuples $(a, b, c_1), \ldots, (a, b, c_n)$ in a relation, but the value of the B-attribute must be the same in all of them, as it is determined by a. Hence, we store b unnecessarily many times. Besides, an attempt to update b leads to problems as it needs to be updated in all of the tuples – otherwise the database would be inconsistent.

A standard solution in this case is to split a schema into two; in this case, into AB with a key dependency $A \to B$, and AC. In general, if we have an FD $X \to Y$, and Z is the set of attributes that are not dependent on X, one splits the schema XYZ as follows:



Thus, a "bad" FD $X \to Y$ is translated into a key dependency $X \to Y$ in the relation with XY attributes, and generates a foreign key from the relation with XZ attributes.

If we have a relational schema given by a set of attributes U and a set of functional dependencies F, we write F^+ for the set of all FDs logically implied by F. We say that a schema is in *BCNF* if, for every nontrivial FD $X \to Y$ in F (i.e. $Y \not\subseteq X$), it is the case that X is a key; in other words, $X \to U \in F^+$.

Intuitively, BCNF completely eliminates all the redundancies, as it eliminates "bad" FDs, and replaces them by keys and foreign keys. It does suffer from one

problem, however. Consider a schema with three attributes A, B, C and two FDs $AB \to C$ and $C \to A$. This schema is not in BCNF, since C is not a key. If we split it into AC and BC, we lose the FD $AB \to C$. In other words, the decomposition is not *dependency-preserving*, and in fact no lossless decomposition of this schema is. Hence, it is impossible to achieve complete elimination of redundancies and dependency-preservation at the same time.

If dependency-preservation is important (and it usually is, since it is important to maintain consistency of the data), one may have to settle for less than BCNF. Recall that an attribute is called *prime* if it belongs to a candidate (minimal) key. Now assume that in a schema we allow two conditions for a nontrivial FD $X \to Y$: either X is a key (as in the case of BCNF), or every attribute in $Y - X$ is prime. This is the definition of the third normal form 3NF (actually, this is not the original definition but a reformulation by [20], which is most commonly used these days).

A good property of 3NF is that every relational schema admits a lossless dependency-preserving 3NF decomposition. However, such a decomposition is not guaranteed to eliminate all the redundancies, but it restricts them to values of prime attributes. 3NF is a very common database design used in practice [15].

3 Measuring the Amount of Redundancy

Our goal is to provide a way of reasoning about XML designs without appealing to the notions of queries and updates for XML documents. Before introducing such techniques for XML, we would like to test them in the well-understood relational case, where we know what constitutes a good design.

The main idea of our approach, which was first proposed in [2], is as follows. Given an instance R of a schema with attributes $A_1, \ldots, A_n$ and FDs F, we define a notion of *relative information content* of a position p in R with respect to F. It will be denoted by $\text{RIC}_R(p|F)$. Here a position is identified by a tuple and an attribute. We define it in such a way that

$$0 \;\leq\; \text{RIC}_R(p|F) \;\leq\; 1,$$

with $\text{RIC}_R(p|F) = 1$ saying that position p carries no redundancy whatsoever. In general, the less the value of $\text{RIC}_R(p|F)$ is, the more redundancy this position p carries.

This notion is defined using the concept of entropy, more precisely, a certain conditional entropy. The notion of entropy was used in the past to reason about database constraints [7,14], but it is a bit of challenge to make it relative to a set of constraints F. The general idea is as follows. We want to measure the amount of information in p with respect to an arbitrary set P of positions in the instance R. This way we account for all possible interactions between p and sets of positions in R, and then we take the average such amount of information as the value of $\text{RIC}_R(p|F)$.

To measure the amount of information in p with respect to a set of positions P, assume that we lose the value in position p, and that we have a set of k possible

values $v_1, \ldots, v_k$ to choose this value from. We shall assign a certain probability $\pi_i(P)$ of picking the right value to each of the v_i's – this is the probability that v_i is a possible value for position p, given the information provided by positions P. We then look at the entropy of this distribution:

$$\textsc{Ric}_R^k(p, P|F) \;=\; \sum_{i=1}^{k} \pi_i(P) \log \frac{1}{\pi_i(P)}.$$

Note that this value is dependent on k, the number of possible values to put in position p.

The entropy tells us how much information is provided by a certain random event. For example, if there is only one way to replace the missing value by some v_i, then $\textsc{Ric}_R^k(p, P|F) = 0$, meaning that the information content of position p is 0, and that the value is redundant as it can be inferred from the rest. The opposite case when all the values v_i's are possible with equal probabilities $\pi_i(P) = \frac{1}{k}$. This is the the least redundant case, when we can infer nothing about the value in position p. In this case, $\textsc{Ric}_R^k(p, P|F) = \log k$, the maximum value of an entropy of a discrete distribution on k elements.

Now our measure (almost) is the average value of $\textsc{Ric}_R^k(p, P|F)$ over all sets P of positions:

$$\textsc{Ric}_R^k(p|F) \;=\; \frac{1}{2^{N-1}} \sum_{P \subseteq Positions(R)} \textsc{Ric}_R^k(p, P|F),$$

where N is the total number of positions in R (i.e. the number of tuples in R multiplied by the number of attributes in R).

To compute the $\pi_i(P)$'s, we need the FDs in F. Now suppose that the value if position p is v_i, and we lose the values in positions P. We now look at the ratio of the number of value assignments to position in P (from the same set $v_1, \ldots, v_k$) that make the resulting instance satisfy all the FDs in F. These are, essentially, the $\pi_i(P)$'s (in addition one needs to normalize these values to ensure that $\sum_i \pi_i(P) = 1$).

We have almost defined our measure; the only problem is that $\textsc{Ric}_R^k(p|F)$ depends on k. Since the domain of values is assumed to be countably infinite, as the measure $\textsc{Ric}_R(p|F)$ we take the limit of the ratio of $\textsc{Ric}_R^k(p|F)$ and the maximum entropy for a discrete distribution on k values, i.e.

$$\textsc{Ric}_R(p|F) \;=\; \lim_{k \to \infty} \frac{\textsc{Ric}_R^k(p|F)}{\log k}.$$

It was proved in [2] that this limit always exists (in fact it exists for far more general classes of constraints than FDs), and thus can be taken to be the relative information content of a position in an instance with respect to a set of constraints F.

Some basic facts about the measure $\textsc{Ric}_R(p|F)$ from [2]:

- The value of $\textsc{Ric}_R(p|F)$ is independent of the syntactic representation of the FDs in F. That is, if F and G are two sets of FDs and $F^+ = G^+$, then $\textsc{Ric}_R(p|F) = \textsc{Ric}_R(p|G)$.

- $0 \leq \text{RIC}_R(p|F) \leq 1$.
- If $F = \emptyset$, then $\text{RIC}_R(p|F) = 1$ (if there are no constraints, there is nothing to tell us that the schema may not be well-designed).

4 Applying the Measure: Relational Designs

We now use the information-theoretic measure to define well-designed schemas.

Definition 1. *A relational schema given by a set of FDs F is* well-designed *if $\text{RIC}_R(p|F) = 1$ for every instance R of the schema and every position p in R.*

In other words, the schema is well-designed if no position in any instance of the schema admits any redundancy.

Theorem 1 (see [2]). *A schema given by FDs is well-designed if it is in BCNF.*

This confirms our intuition that BCNF completely eliminates redundancies. This result was further extended in [2] to deal with other classes of constraints such as multi-valued and join dependencies, and justify normal forms such as 4NF [9].

But what about the normal form most commonly used in practice, i.e. 3NF? The first result looks rather discouraging: 3NF schemas may admit an arbitrarily high amount of redundancy (demonstrated by arbitrarily low values of the measure).

Proposition 1 (see [11]). *For every $0 < \varepsilon < 1$, one can find a 3NF relational schema given by a set of FDs F, an instance R of that schema and a position p in R such that $\text{RIC}_R(p|F) < \varepsilon$.*

However, the situation is not as bad as it might seem. First, the result requires schemas with a large number of attributes. More importantly, it has been known for a long time [20] that not all 3NF designs are equally good: for some schemas already in 3NF, better 3NF designs can be obtained by applying the standard 3NF synthesis algorithm [3].

3NF designs guarantee the integrity of the database. One may ask whether 3NF is the best choice of a dependency-preserving normal form. That is, if we look at all normal forms that guarantee dependency-preservation (hence excluding BCNF), is it 3NF that has the least amount of redundancy?

The answer to this is positive. Assume that $\mathcal{NF}$ is some dependency-preserving normal form: i.e., every schema admits a lossless dependency-preserving decomposition into $\mathcal{NF}$. We define the *guaranteed information content* provided by $\mathcal{NF}$ as the largest number $c \in [0, 1]$ such that every schema may be decomposed into $\mathcal{NF}$ in such a way that in all instances R of the decomposed schema and all positions p, the information content $\text{RIC}_R(p|F)$ is at least c.

If $\text{RIC}_R(p|F) \geq c$, then $1 - c$ is the *price of dependency preservation*, denoted by $\text{PRICE}(\mathcal{NF})$: that is, the minimum amount of information content one *must* lose due to dependency preservation. The following theorem shows that among normal forms that guarantee dependency preservation, 3NF is the one with the least amount of redundancy.

Theorem 2 (see [12]). $\textsc{Price}(3\text{NF}) = 1/2$. *Furthermore, if $\mathcal{NF}$ is an arbitrary dependency-preserving normal form, then* $\textsc{Price}(\mathcal{NF}) \geq 1/2$.

Furthermore, we can analyze "good" 3NF schemas produced by the standard synthesis algorithm. We refer to them as 3NF^+ schemas (they can be syntactically characterized [20], but for our purposes it suffices to think of them as 3NF schemas that cannot be further decomposed using the standard synthesis algorithm of [3]).

To compare 3NF and 3NF^+ designs, we use a new concept of a *gain of normalization* function. To define it, assume that we have a condition $\mathcal{C}$ on schemas consisting of FDs. We first define the set of possible values of $\textsc{Ric}_R(p|F)$ for m-attribute instances R of schemas satisfying $\mathcal{C}$:

$$\mathcal{POSS}_{\mathcal{C}}(m) = \{\textsc{Ric}_R(p|F) \mid R \text{ satisfies } F, \quad F \text{ satisfies } \mathcal{C}, \\ R \text{ has } m \text{ attributes}\}.$$

We are now interested in the lowest possible value in such a set, i.e. $\inf \mathcal{POSS}_{\mathcal{C}}(m)$ (typically these sets are dense subsets of intervals $(\varepsilon, 1]$, so the infimum – ε – is well-defined). For two normal forms $\mathcal{NF}_1$ and $\mathcal{NF}_2$, the gain of normalization function $\textsc{Gain}_{\mathcal{NF}_1/\mathcal{NF}_2} : \mathbb{N} \to \mathbb{R}$ is

$$\textsc{Gain}_{\mathcal{NF}_1/\mathcal{NF}_2}(m) \;=\; \frac{\inf \mathcal{POSS}_{\mathcal{NF}_1}(m)}{\inf \mathcal{POSS}_{\mathcal{NF}_2}(m)}.$$

In other words, we measure the ratio of the least amount of information in instances of $\mathcal{NF}_1$- and $\mathcal{NF}_2$-schemas, which tells us how much better $\mathcal{NF}_1$ can be compared to $\mathcal{NF}_2$.

Let $\textsc{All}$ be the set of all (unnormalized) schemas.

Theorem 3 (see [12]). *For every $m > 2$:*

- $\textsc{Gain}_{3\text{NF}/\textsc{All}}(m) = 2$;
- $\textsc{Gain}_{\text{BCNF}/3\text{NF}^+}(m) = 2$;
- $\textsc{Gain}_{3\text{NF}^+/\textsc{All}}(m) = 2^{m-2}$.

Stated informally, an arbitrary 3NF design is at least twice as good as not doing any normalization at all. Furthermore, good 3NF designs are much better – in the worst case, they are within a constant factor of two of the best BCNF designs in terms of the amount of redundancy, while guaranteeing dependency-preservation.

5 Applying the Measure: XML Designs

The information theoretic measure is very robust. It lets us justify relational normal forms and go beyond them – we saw how to use it for normal form comparison, for example. It was also shown in [2] how to reason about normalization algorithms using the information-theoretic measure.

We now switch our attention to XML, and show that the same information-theoretic techniques give us a notion of redundancy-eliminating normal form, which is an analog of BCNF, as well as a notion of "second best" form, similar to 3NF. Namely, we shall do the following.

1. We define functional dependencies for XML, for specifying constraints.
2. We then show that the information-theoretic measure applies to XML documents.
3. We present the notion of a redundancy-eliminating normal form, called XNF, and sketch an algorithm for converting XML designs into XNF.
4. Finally, we analyze good XML designs assuming XML documents are shredded into relations. Again, we show that XNF eliminates redundancies.

We start with the notion of functional dependency. An analog of a relational attribute in the case of XML is a ***path*** through a DTD, i.e. a sequence of labels consistent with the DTD of a document. For example, if we represent a relational schema with attributes ABC and FDs $AB \to C$ and $C \to A$ by a DTD D:

$$r \to \mathit{tuple}^* \tag{1}$$

with ***tuple*** having attributes $@A, @B, @C$, respectively, then the FDs will be represented as follows:

$$\begin{array}{rcl} \{r.\mathit{tuple}.@A,\ r.\mathit{tuple}.@B\} & \to & r.\mathit{tuple}.@C \\ r.\mathit{tuple}.@C & \to & r.\mathit{tuple}.@A \end{array} \tag{2}$$

If we look at the DTD D' for storing information about conferences and papers:

$$\begin{array}{c} db \to \mathit{conf}^* \\ \mathit{conf} \to \mathit{paper}^* \end{array} \tag{3}$$

where the element type ***paper*** comes with attributes $@\mathit{author}, @\mathit{title}, @\mathit{year}$, then we have an FD

$$\mathit{db.conf} \quad \to \quad \mathit{db.conf.paper}.@\mathit{year} \tag{4}$$

For example, it is natural to expect that all the papers appearing in XSym 2007 will have 2007 as the year of their publication.

The notion of satisfaction of FDs should be intuitively clear from these examples; the interested reader is referred to [1] for a formal definition that relies on a notion of tree tuples.

Now we have XML schemas which are given by DTDs D and sets of XML FDs F. If we have a document T that conforms to D and satisfies constraints in F, we can define a set of ***positions*** in that document as the set of all the places where an attribute value occurs (more precisely, as sets of pairs $(v, @l)$, where v is a node identifier in a tree, and $@l$ is an attribute associated with that node). Once we have the notion of positions, we can define the measure

$$\textsc{Ric}_T(p|F)$$

in exactly the same way as we defined $\text{RIC}_R(p|F)$, except that the we use the set of positions in a document T as opposed to the set of positions in relation R. It satisfies all the same basic properties as $\text{RIC}_R(p|F)$.

We then say that an XML specification (D, F) is *well-designed* if for every document T that conforms to D and satisfies F, and every position p in T, we have $\text{RIC}_T(p|F) = 1$.

How do we characterize the notion of being well-designed? To see what a reasonable normal form for XML might be, we analyze the examples shown above. Given the DTD (1) and FDs (2), let us look at the FD $r.\mathit{tuple}.@C \rightarrow r.\mathit{tuple}.@A$. The left-hand side implies $r.\mathit{tuple}.@A$, but since $@C$ is not a key, the left-hand side does *not* imply $r.\mathit{tuple}$ – indeed, $@C$ does not determine the tuple uniquely.

Looking at the DTD (3) and the FD (4), we see that the left-hand side of (4) does not imply $\mathit{db.conf.paper}$ (as this would mean that there was a single paper published in the proceedings!).

So what is common to these examples? In both cases we have:

- redundancy built into the specification – in the first case it is essentially a non-BCNF relational design, and in the second case the attribute $@\mathit{year}$ is stored multiple times;
- an FD of the form $X \rightarrow \mathit{path}.@l$ such that $X \rightarrow \mathit{path}$ is *not* implied by other FDs.

This was the motivation for the following definition given in [1].

Definition 2. *An XML specification (D, F) given by a DTD D and a set F of FDs is in the* XML Normal Form (XNF) *i for every FD $X \rightarrow \mathit{path}.@l$ implied by F, it is the case that $X \rightarrow \mathit{path}$ is also implied by F.*

This turns out to be the definition suggested by the information-theoretic approach.

Theorem 4 (see [2]). *An XML specification (D, F) given by a DTD D and a set F of FDs is well-designed i it is in XNF.*

Two specifications seen earlier are *not* in XNF. A natural question is then how to convert them into XNF. Looking at the first example, we have to apply the usual relational decomposition; for example, a new DTD will look like

$$r \rightarrow \mathit{tuple}^*, r_{new}$$
$$r_{new} \rightarrow \mathit{tuple}^*_{new}$$

where tuple has attributes $@A, @C$, and tuple_{new} has attributes $@B, @C$. We put in the FD $r.\mathit{tuple}.@C \rightarrow r.\mathit{tuple}$, since $@C$ is a key, if tuple contains only $@A$ and $@C$ as attributes.

In the second example, we do not make a relational split, but instead simply make $@\mathit{year}$ an attribute of conf, which eliminates the violation of XNF seen above.

An algorithm for converting a specification into XNF is essentially this:

keep applying the "relational split" and the "hierarchical attribute move" steps illustrated above.

One can prove [1] that such an application of two basic transformation rules results in an XNF design.

So far we made an assumption that XML documents are represented as tree-structures. Very often, however, documents are shredded into relations [16]. One of the most common techniques for storing XML in relational databases is *inlining* [16]. The idea is that separate relations are created for element types that appear under a Kleene star, and all other element types are inlined in the relations corresponding to their parents. Each relation for an element type has an id attribute that is a key for that relation, as well as a parent id attribute that is a foreign key pointing for the parent of that element. All the attributes of a given element type in the DTD become attributes in the relation corresponding to that element type.

For example, the relational schema for storing XML documents conforming to the DTD in (3) would be

$$\begin{array}{l} conf(\underline{\mathsf{confID}}, \mathsf{name}) \\ paper(\underline{\mathsf{paperID}}, \mathsf{confID}, \mathsf{title}, \mathsf{author}, \mathsf{year}), \end{array}$$

assuming that the *conf* element type has attribute *@name*. Keys are underlined.

It is known [13] that XML functional dependencies F are translated into more general constraints over the inlined relational representation, more precisely, into a set Σ_F of equality-generating dependencies. The inlining mapping also associates a position $\delta(p)$ of the relational representation with each position p in the XML document.

Let (S, Σ_F) be an inlining translation (S, Σ_F) of (D, F), where S is a relational schema, and Σ_F is a set of equality-generating dependencies. We say that (D, F) is *well-designed for relational storage* iff for every XML tree T conforming to D and satisfying F and every position p in T, we have $\textsc{Ric}_{R_T}(\delta(p)|\Sigma_F) = 1$, where R_T is the relational instance of S into which T is transformed.

The next result shows that XNF captures the notion of being well-designed for relational representation of XML documents as well.

Theorem 5 (see [13]). *An XML specification (D, F) is well-designed for relational storage i it is in XNF.*

Summing up, the following are equivalent for a specification consisting of a DTD D and a set of XML FDs F:

1. (D, F) is well-designed;
2. (D, F) is well-designed for relational storage;
3. (D, F) is in XNF.

We conclude with a simple condition that guarantees "reasonable" designs from the point of view of the information-theoretic measure. First, one can show that

for documents not in XNF, the values of both $\text{RIC}_T(p|F)$ and $\text{RIC}_{R_T}(\delta(p)|\Sigma_F)$ can be arbitrarily low [13].

One type of constraints often used for XML documents is *relative*: such constraints do not hold in the entire document, but only in a part of it restricted to descendants of some element type [4,10]. In the case of XML FDs, we say that an FD $\{q_1, \ldots, q_n\} \to q$ is *relative* if

- for some $i \in [1, n]$, the path q_i ends with an element type (rather than an attribute);
- for all $j \neq i$, the paths q_j extend q_i (in other words, q_i is a prefix of q_j); and
- for some path p which is a prefix of q_i and ends on an element type τ, there exists an element type τ' and a rule $\tau' \to e$ in the DTD such that τ occurs under the scope of a Kleene star in the regular expression e.

Theorem 6 (see [13]). *Let (D, F) be a specification in which every FD violating the XNF condition is relative. Then for every tree T conforming to D and satisfying F and every position p in T, we have*

$$\text{RIC}_{R_T}(\delta(p)|\Sigma_F) \; > \; \frac{1}{2},$$

where R_T is the relational instance into which T is transformed.

Thus, if we design an XML document that might violate XNF but the only violating FDs are relative, then the redundancy of each position in the relational storage of the XML document would not be worse than $\frac{1}{2}$. In other words, this would match the worst-case redundancy of 3NF.

6 Open Problems

The information-theoretic approach has completely clarified the situation with good relational designs, and best possible XML designs for both native and relational storage. However, it is not yet entirely clear how to handle non-perfect designs that do not eliminate all redundancies. We provided an example of a sufficient condition that matches the bounds on the measure given by 3NF. However, it is not known whether one can achieve effective normalization with respect to that condition, nor is it known whether the condition guarantees dependency-preservation.

The notion of dependency-preservation itself is much less understood for XML. It was shown in [11] that one can produce XML designs capturing 3NF relational designs that do not have dependency-preserving BCNF decompositions in a way that accounts for all the constraints. This needs to be explored further, as it opens a possibility of storing relations in XML in a way that eliminates redundancies and guarantees dependency-preservation, even if there is no such relational representation.

Finally, it would be nice to extend the idea of using the information-theoretic framework for reasoning about and comparing different shredding techniques for XML documents.

Acknowledgments. This invited talk presents results on the information-theoretic approach to database design that have been obtained jointly with Marcelo Arenas and Solmaz Kolahi. I am very greatful to Marcelo and Solmaz for collaborating with me, and for their comments on this short survey. I gratefully acknowledge the support of the European Commission Marie Curie Excellence grant MEXC-CT-2005-024502 and EPSRC grant E005039.

References

1. Arenas, M., Libkin, L.: A normal form for XML documents. ACM TODS 29, 195–232 (2004) Extended abstract in PODS'02
2. Arenas, M., Libkin, L.: An information-theoretic approach to normal forms for relational and XML data. J. ACM 52(2), 246–283 (2005) Extended abstract in PODS'03
3. Bernstein, P.A.: Synthesizing third normal form relations from functional dependencies. ACM TODS 1(4), 277–298 (1976)
4. Buneman, P., Davidson, S., Fan, W., Hara, C., Tan, W.C.: Keys for XML. In: WWW 2001, pp. 201–210 (2001)
5. Codd, E.F.: Further normalization of the data base relational model. IBM Research Report (1971)
6. Codd, E.F.: Recent Investigations in Relational Data Base Systems. IFIP Congress1974, pp. 1017–1021 (1974)
7. Dalkilic, M., Robertson, E.: Information dependencies. In: PODS'00, pp. 245–253 (2000)
8. Embley, D.W., Mok, W.Y.: Developing XML documents with guaranteed "good" properties. In: Kunii, H.S., Jajodia, S., Sølvberg, A. (eds.) ER 2001. LNCS, vol. 2224, pp. 426–441. Springer, Heidelberg (2001)
9. Fagin, R.: Multivalued dependencies and a new normal form for relational databases. ACM TODS 2(3), 262–278 (1977)
10. Fan, W., Libkin, L.: On XML integrity constraints in the presence of DTDs. J. ACM 49(3), 368–406 (2002)
11. Kolahi, S.: Dependency-preserving normalization of relational and XML data. J. Comput. Syst. Sci. 73(4), 636–647 (2007)
12. Kolahi, S., Libkin, L.: On redundancy vs dependency preservation in normalization: an information-theoretic study of 3NF. PODS 2006, 114–123 (2006)
13. Kolahi, S., Libkin, L.: XML design for relational storage. WWW 2007, pp 1083–1092 (2007)
14. Lee, T.T.: An information-theoretic analysis of relational databases - Part I: Data dependencies and information metric. IEEE Trans. on Software Engineering 13(10), 1049–1061 (1987)
15. Oracle's General Database Design FAQ. `http://www.orafaq.com/faqdesgn.htm`
16. Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., DeWitt, D.J., Naughton, J.F.: Relational databases for querying XML documents: Limitations and opportunities. In: VLDB, pp. 302–314 (1999)
17. Vincent, M., Liu, J.: Multivalued dependencies and a 4NF for XML. In: Eder, J., Missikoff, M. (eds.) CAiSE 2003. LNCS, vol. 2681, pp. 14–29. Springer, Heidelberg (2003)

18. Vincent, M., Liu, J., Liu, C.: Strong functional dependencies and their application to normal forms in XML. ACM TODS 29(3), 445–462 (2004)
19. Wang, J., Topor, R.: Removing XML data redundancies using functional and equality-generating dependencies. In: ADC 2005, pp. 65–74 (2005)
20. Zaniolo, C.: A new normal form for the design of relational database schemata. ACM TODS 7, 489–499 (1982)

Dynamic Fusion of Web Data

Erhard Rahm, Andreas Thor, and David Aumueller

University of Leipzig, Germany
http://dbs.uni-leipzig.de

Abstract. Mashups exemplify a workflow-like approach to dynamically integrate data and services from multiple web sources. Such integration workflows can build on existing services for web search, entity search, database querying, and information extraction and thus complement other data integration approaches. A key challenge is the efficient execution of integration workflows and their query and matching steps at runtime. We relate mashup data integration with other approaches, list major challenges, and outline features of a first prototype design.

1 Introduction

The need to fuse data from multiple web sources is rapidly increasing. This is demonstrated by the recent proliferation of mashup applications which combine content from multiple sources and services. Mashup applications are interactive and utilize flexible Web2.0 user interfaces. Content integration for such mashups is dynamic, i.e., it occurs at runtime (on demand) based on specific user input. Driving forces for the broad adoption of mashups are the availability of several development frameworks (e.g., Google Web Toolkit), as well as the proliferation of web APIs and information extraction tools for easy access to many websites, search engines or data feeds. Several tools (e.g., Yahoo Pipes, OpenKapow, Mashmaker [2]) also support visual interfaces to enable the construction of simple mashups without programming.

The potential for fast development makes mashups a highly attractive approach for integrating web data from several sources. This is because traditional, schema-focused data integration approaches (data warehouses, query mediators and – to a lesser degree – schema-oriented peer data management systems) suffer from a high upfront development effort for resolving semantic heterogeneity [3]. The effort needed to determine a global schema and/or precise schema mappings also limits scalability of schema-focused approaches to many sources. Web search engines, on the other hand, scale to many web sites but lack sufficient support for structured data sources of the "hidden web". Several approaches are being investigated to better provide integrated access to both unstructured and structured web sources with good scalability. For example, MetaQuerier provides unified entity search interfaces over many structured web sources of the hidden web [1]. PayGo aims at providing web-scale, domain-spanning access to structured sources [4]. It tries to cluster related schemas together and to improve search results by transforming keyword search queries into structured queries on relevant sources. One aspect missing from such search approaches is the post-processing of heterogeneous search results, in particular an online fusion of corresponding (matching) objects.

D. Barbosa et al. (Eds.): XSym 2007, LNCS 4704, pp. 14–16, 2007.

Mashups demonstrate a more programmatic, workflow-like integration approach, complementary to the query- and search-based data integration approaches. In fact, mashups are massively built on the idea of reusing and combining existing services so that they can also use existing search engines and query services. However, current mashups are mostly very simple and do not yet exploit the full potential of workflow-like data integration, e.g. as needed for enterprise applications or to analyze larger sets of web data. Hence, we see the need for a more powerful workflow-like data fusion approach which preserves mashup features like Web2.0 GUIs, support for reuse and fast development.

Providing such an approach incurs several challenges, including the definition of an architecture supporting mashups for integration at three levels, i.e. data, application, and presentation level. Furthermore, a powerful workflow and programming model is needed supporting the execution of existing web services and generic services (or operators) for information extraction, entity search, database queries, and object matching. The set of usable services and data sources should be listed and semantically described in a metadata repository similar as proposed in [3]. A limiting factor for interactive mashups is runtime. Hence, techniques are needed to solve more complex integration tasks, e.g. involving query, search and object matching of larger datasets, within a short time.

In the next section we discuss features of a first prototype for workflow-like dynamic data fusion.

2 Information Fusion with iFuice

We are currently extending our iFuice system for dynamic, mashup-like data fusion [6] [7]. In [7] we also report on a complex mashup implementation to generate on demand aggregated (Google Scholar) citation counts for (DBLP) publication lists of authors and venues. Here we summarize some key features of the iFuice design which we believe make it suitable for dynamic data fusion within mashup-like applications.

- *Workflow-like data integration* and *operator-based programming model.* iFuice provides a high-level script language to define integration workflows or mashups. The language consists of powerful generic operators which can be applied to different data sources and services. For example, a query operator takes as input the id of a query service (data source) and a query specification. Most operators are set-oriented, i.e. they can be applied on a arbitrary set of input objects and generate a set of result objects. Intermediate results can be stored in variables for use by other operators. There are several operators for set operations (e.g., union, intersection, and difference) and data transformation (e.g., fuse, aggregate) which can be used to post-process query results.
- *Utilization of instance-level mappings:* iFuice utilizes instance-level mappings describing relationships between instances of entity types. Such mappings can relate instances of different sources (e.g. corresponding authors or publications of different bibliographic sources) and often exist already as hyperlinks. In addition, for structured sources we support instance-level associations between objects of a given source, e.g. to interrelate an author with her publications. Such instance-level mappings can efficiently be used to fuse together corresponding objects, even in the

absence of schema mappings. Materializing such mappings supports their reuse for different integration workflows and use cases.

- *Support for structured and unstructured data sources*. By providing appropriate access services structured as well as unstructured web sources are supported. Each source may be accessed based on entity ids (e.g. URLs), or using structured queries or keyword search. Furthermore, we can leverage existing entity search engines [5] or general search engines to reuse their results aggregated from many other sources.
- *Metadata repository:* Usable data sources and services are recorded in a repository and are assigned to entity types (e.g. publication, author). Furthermore, all available mappings and their semantic mapping type (e.g. publications of authors) are maintained. Entity and mapping types are part of a so-called domain model which can be incrementally extended as needed. A domain model is at a higher abstraction (ontological) level than a global database schema and helps to locate semantically relevant sources and services.
- *Iterative query strategies:* The use of existing search engines may require several queries for more complex integration tasks to obtain a sufficient number of relevant result entities. iFuice therefore allows to iteratively refine query results, where the execution of subsequent queries may be interactively controlled by the user. The OCS application [7] uses refining queries to the entity search engine Google Scholar to obtain citations for a set of publications. Intermediate results are shown to the user while the system executes additional queries to complete the result. Using such query strategies allows the quick generation of approximate results which can be further improved as needed.
- *On-the-fly object matching*: Dynamic data fusion requires to match corresponding objects from different sources and fuse their attribute values at run time. Using the MOMA framework [8] we provide a large spectrum of match strategies from which one can choose. In particular, the reuse of existing mappings can help to achieve a fast object matching.

Our investigations on dynamic mashup-like data fusion have just begun and several difficult research problems still need to be addressed, e.g. the automatic generation of iterative query strategies and on-the-fly object matching approaches.

References

1. Chang, K.: Toward Large Scale Integration: Building a MetaQuerier over Databases on the Web. In: Proc. CIDR (2005)
2. Ennals, R., Garofalakis, M.: MashMaker: Mashups for the Masses. In: Proc. Sigmod (2007)
3. Franklin, M., Halevy, A., Maier, D.: From Databases to Dataspaces: a New Abstraction for Information Management. SIGMOD Record 34(4), 27–33 (2005)
4. Madhavan, J., Jeffery, S.R., Cohen, S., Dong, X., Ko, D., Yu, C., Halevy, A.: Web-scale Data Integration: You can only afford to Pay As You Go. In: Proc. CIDR (2007)
5. Nie, Z., Wen, J.-R., Ma, W.-Y.: Object-level Vertical Search. In: Proc. CIDR (2007)
6. Rahm, E., Thor, A., Aumueller, D.: Do, H.-H., Golovin, N., Kirsten, T.: iFuice - Information Fusion utilizing Instance Correspondences and Peer Mappings. WebDB (2005)
7. Thor, A., Aumueller, D., Rahm, E.: Data Integration Support for Mashups. IIWeb (2007)
8. Thor, A., Rahm, E.: MOMA: - A Mapping-based Object Matching System. CIDR (2007)

XPath Query Satisfiability is in PTIME for Real-World DTDs

Manizheh Montazerian[1], Peter T. Wood[1], and Seyed R. Mousavi[2]

[1] Birkbeck, University of London, London WC1E 7HX, UK
{gmont05,ptw}@dcs.bbk.ac.uk
[2] Isfahan University of Technology, Isfahan, Iran
srm@cc.iut.ac.ir

Abstract. The problem of XPath query satisfiability under DTDs (Document Type Definitions) is to decide, given an XPath query p and a DTD D, whether or not there is some document valid with respect to D on which p returns a nonempty result. Recent studies in the literature have shown the problem to be NP-hard or worse for most fragments of XPath. However, in this paper we show that the satisfiability problem is in PTIME for most DTDs used in real-world applications. Firstly, we report on the details of our investigation of real-world DTDs and define two properties that they typically satisfy: being *duplicate-free* and being *covering*. Then we concentrate on the satisfiability problem of XPath queries under such DTDs. We obtain a number of XPath fragments for which the complexity of the satisfiablity problem reduces to PTIME when such real-world DTDs are used.

Keywords: XPath, Satisfiability, Document Type Definitions.

1 Introduction

With XML becoming the standard for data exchange, substantial work has been done on XML query processing and optimization [1,5,7,10,11,12,14]. Much of the work on query optimization has focused on XPath, since XPath is widely used in XML-related applications to select sets of nodes from an XML document tree. Particularly when an XPath query is to be evaluated over documents known to be valid with respect to a DTD (Document Type Definition), it is possible that the query might be *unsatisfiable*, that is, the query always returns an empty result, no matter what document (valid with respect to the DTD) is queried. Relatively little work has been done on detecting whether a given XPath query is satisfiable [2,6,8,9]. However, it is potentially important to detect unsatisfiable XPath queries and optimize queries to remove expressions that will always return an empty result set. Indeed, Lakshmanan *et al.* show that checking satisfiability as a first step in query processing often yields substantial savings in overall query processing time [9].

XPath supports a wide variety of operators whose presence or absence affects the complexity of the satisfiability problem. This has led to the study of various XPath fragments that include only certain operators. For example, in this

D. Barbosa et al. (Eds.): XSym 2007, LNCS 4704, pp. 17–30, 2007.

paper we study the fragment with child axis (/), descendant axis (//), qualifiers ([ ]), wildcard (*) and union (∪). As in [10], we denote this fragment by $XP^{\{/,[],*,//,\cup\}}$, indicating the operators permitted. Larger fragments allow operators such as negation, additional axes such as parent, ancestor and sibling, as well as comparisons involving data values or node identities.

Example 1. The XMark benchmark project[1] is based on an online auction application. A fragment of the XMark DTD is given below:

```
site          (regions, categories, catgraph, people, open_auctions,
               closed_auctions)
categories    (category+)
category      (name, description)
description   (text | parlist)
open_auctions (open_auction*)
open_auction  (initial, reserve?, bidder*, current, privacy?, itemref,
               seller, annotation, quantity, type, interval)
```

with `site` being the document (top-level) element.The XPath query

```
/site/open_auctions/open_auction[bidder][reserve]/seller
```

selects `seller` nodes that are children of `open_auction` nodes that have both a `bidder` and `reserve` child. It is easy to see that this query is satisfiable on documents valid with respect to the above DTD fragment.

In the full DTD, a `description` can occur as a descendant of more than one element, so one might write

```
/site//description[text][parlist]
```

to retrieve all `description` nodes that have both `text` and `parlist` children. However, this query is unsatisfiable with respect to the DTD, because a `description` can have only one of `text` or `parlist` as a child, not both.

Hidders investigates the satisfiability of XPath 2.0 expressions [8]. Various XPath fragments are classified as either in PTIME or NP-hard. Deciding satisfiability of XPath expressions in the context of a DTD/schema is one of the open problems mentioned in that paper. This problem is dealt with in [9], which considers a tree pattern formalism with expressiveness incomparable to XPath. The language expresses positive tree pattern queries, with data value equality and inequality along with a node-equality test. It shows that the satisfiability problem is NP-complete for several restrictions of this pattern language in the absence of DTDs. It also identifies cases in which satisfiability of tree pattern queries together with additional constraints, with and without schema, can be solved in polynomial time and develops algorithms for this purpose.

The most relevant work to ours is [2]. The authors consider a variety of XPath fragments widely used in practice, and investigate the impact of different XPath operators on satisfiability analysis. They study the problem for negation-free

[1] `http://monetdb.cwi.nl/xml/`

XPath fragments with and without upward axes, recursion and data-value joins, identifying which factors lead to tractability and which to NP-completeness. They show that with negation the complexity ranges from PSPACE to EXPTIME. When both data values and negation are in place, they find that the complexity ranges from NEXPTIME to undecidable. These results are extended in [6], where the satisfiability problem for a variety of XPath fragments with sibling axes is investigated, in the presence and the absence of DTDs and under various restricted DTDs. In these settings they establish complexity bounds ranging from NLOGSPACE to undecidable. They show that there are XPath satisfiability problems that are in PTIME and PSPACE in the absence of sibling axes, but that become NP-hard and EXPTIME-hard, respectively, when sibling axes are used instead of the corresponding vertical modalities.

We concentrate on the satisfiability problem under two special classes of DTDs for a variety of XPath fragments with child axis (/), descendant axis (//), qualifiers ([ ]), wildcard (*) and union ($\cup$). The two classes of DTDs we consider are *duplicate-free* DTDs and *covering* DTDs. Informally, a duplicate-free DTD is one in which no regular expression uses the same symbol more than once. A covering DTD, on the other hand, is one in which each regular expression R is such that the language $L(R)$ it denotes contains a string in which all the symbols used in R appear. Formal definitions of these properties are provided in Section 2.

Example 2. All of the DTD rules for the XMark DTD fragment shown in Example 1 are covering, except the following

```
description   (text | parlist)
```

since the language denoted by `(text | parlist)` does not contain a sequence that includes both element names. In addition, all of the rules in the XMark fragment are duplicate-free. The following is an example of a rule with duplicates, taken from the XML Schema DTD[2] after replacing entity references and ignoring namespace prefixes

```
schema ((include | import | redefine | annotation)*,
        ((simpleType | complexType | element | attribute
          | attributeGroup | group | notation), (annotation)*)* )
```

where the element name `annotation` is repeated.

The notion of a duplicate-free DTD was introduced in [13] and also used in [14]. Other authors have also studied and classified DTDs according to various properties [3,4]; however, the properties investigated are incomparable with the notions of covering and duplicate-free. We show in this paper that the classes of covering and duplicate-free DTDs comprise most real-world DTDs, i.e. those used in real-world applications. We identify a number of XPath fragments for which the complexity of the satisfiability problem reduces to PTIME

[2] `http://www.w3.org/2001/XMLSchema.dtd`

when duplicate-free or covering DTDs are used. For example, the fact that satisfiability for the fragment $XP^{\{/,[]\}}$is NP-hard in general follows from a result in [13]. Here we show that it becomes decidable in PTIME for duplicate-free DTDs, although results from [2] imply that it remains NP-hard for the fragments $XP^{\{/,[],*\}}$and $XP^{\{[],//\}}$. More significantly, for covering DTDs we show that satisfiability for the fragment $XP^{\{/,[],*,//,\cup\}}$is in PTIME.

The next section contains the definitions of the various forms of DTD and fragments of XPath we consider in this paper, and reviews the problem of XPath satisfiability in the presence of DTDs. Section 3 provides the results of our investigation into the relative frequency of covering and duplicate-free real-world DTDs. Section 4 presents our complexity results for a number of XPath fragments under duplicate-free and covering DTDs. Finally, Section 5 summarizes our main results and explains our future plans.

2 Notation and Background Material

In this section, we define DTDs, along with various subclasses of DTDs, the XPath fragments studied in this paper, and the notion of XPath satisfiability.

Definition 1. *A DTD over a finite alphabet Σ is a tuple (D, S_0, Σ) where $S_0 \in \Sigma$ is the start symbol, and D is a mapping from Σ to a set of regular expressions over Σ. We say that R is the* content model *for symbol a and write $a \rightarrow R$ (which we also call a* rule*). From now on, we refer to a DTD by D rather than (D, S_0, Σ) and assume that Σ is the set of symbols appearing in D. In examples, we will usually drop the arrow symbol from rules.*

We will use the DTD syntax for regular expressions, namely, "," for concatenation, "|" for alternation (disjunction), "$*$" for reflexive transitive closure, "+" for transitive closure and "?" for optional.

Definition 2. *Let R be a regular expression and Σ be the set of symbols appearing in R. We say that R* covers *Σ, or simply that R is* covering*, if there is a string in $L(R)$ that contains every symbol in Σ. A DTD D is called* covering *if and only if each content model in D is covering.*

Note that a number of common content models used in DTDs are covering. For example, the content models one gets from the naive representation of relational data as XML are covering, as are the so-called mixed content models found in "document-oriented" XML. Examples of covering and non-covering content models were given in Example 2.

Definition 3. *Let R be a regular expression and Σ be the set of symbols appearing in R. We say that R is* duplicate-free *if each symbol in Σ occurs exactly once in R. A DTD D is called* duplicate-free *if and only if each content model in D is duplicate-free.*

Note that the above definition is syntactic. In other words, we can have two regular expressions which denote the same language such that one expression is duplicate-free while the other is not. For example, $a?, b$ and $(a, b)|b$ denote the same language, but only the former expression is duplicate-free. Other examples of duplicate-free and non-duplicate-free expressions were given in Example 2.

A number of other subclasses of DTDs have been defined in order to study the complexity of problems such as XPath satisfiability. For example, [2] considers disjunction-free DTDs, while [3] considers simple regular expressions defined as follows.

Definition 4. *A* base symbol *is a regular expression a, $a?$ or a^* where $a \in \Sigma$; a* factor *is of the form e, e^* or $e?$ where e is a disjunction of base symbols. A* simple regular expression *is ϵ, $\emptyset$ or a sequence of factors.*

Clearly, a simple regular expression need not be duplicate-free nor covering. On the other hand, $a|(b, c)$ is duplicate-free but not simple, and $(a, b)^*$ is covering but not simple. We conclude that the 3 subclasses of DTDs are pairwise incomparable.

Definition 5. *The syntax of XPath expressions used in this paper is given by the following grammar:*

$$q \rightarrow \text{`/'}\ p$$
$$p \rightarrow p\ \text{`/'}\ p \mid p\ \text{`//'}\ p \mid p\ \text{`}\cup\text{'}\ p \mid p\ \text{`[' }p\text{ `]'} \mid \text{`}*\text{'} \mid n \mid \text{`.'}$$

where q is the start symbol, n is an element name and '.' refers to the context node.

Examples of XPath expressions using the above syntax were given in Example 1. As mentioned earlier, fragments of XPath are denoted by indicating which operators are supported. So the above fragment is denoted by $\text{XP}^{\{/,[\,],*,//,\cup\}}$, since child axis (/), descendant axis (//), qualifiers ([ ]), wildcard (*) and union ($\cup$) are all permitted.

Papers such as [2] use an alternative syntax where $\downarrow$ denotes use of the child axis without specifying an element name. So $\downarrow$ in their syntax corresponds to $*$ in ours, with $\downarrow / a$ corresponding to a. One consequence of this is that $*$ is implicitly permitted in all the XPath fragments considered by [2] that include the child axis, whereas we distinguish explicitly whether or not $*$ is included.

Definition 6. *The following notation is adapted from [2]. An XPath expression p is* satisfiable *if there is an XML tree T such that the answer of p on T is not empty, denoted by $T \models p$. Given a DTD D, we denote the fact that an XML tree* satisfies *(or is* valid *with respect to) D by $T \models D$. Given a DTD D and a query p, an XML tree T* satisfies *p and D, denoted by $T \models (p, D)$, iff $T \models p$ and $T \models D$. For an XPath fragment $\mathcal{X}$, the* XPath satisfiability problem *$SAT(\mathcal{X})$ is, given a DTD D and a query p in $\mathcal{X}$, is there an XML tree T such that $T \models (p, D)$.*

3 Real-World DTDs

In this section, we report on our investigation of "real-world" DTDs, i.e. those frequently used in real applications. For the purpose of this paper, we were concerned with two features of such DTDs, namely whether or not they were duplicate-free or covering. We will see that most of the real-world DTDs we studied have at least one of these two properties.

Table 1. The DTDs and their application domains

DTD Name	Application Domain
Oagis	Open Applications Group Archives
ODML 1.0	Optimal Design Markup Language
LevelOne	HL7 Clinical Document Architecture
Ecoknowmics	Economic Knowledge Management
XML Schema	XML Schema
HP	HL7 Document Architecture
Meerkat	Storehouse of News about Technological Developments
OSD	Open Software Description
Opml	Outline Processing Markup Language
Rss-091	XML Vocabulary for Describing Metadata about Websites
TV-Schedule	TV Schedule
Xbel-1.0	XML Bookmarks Exchange Language
XHTML1-strict	Extensible HTML version 1.0 Strict
Newspaper	Newspaper
DBLP	Digital Bibliography Library Project
Music ML	Music Digital Library
XMark DTD	XML Benchmark
Yahoo	Yahoo Auction Data
Reed	Courses from Reed College
Nlm Medline	National Library of Medicine
SigmodRecord	Index of Articles from SIGMOD Record
Ubid	UBid Auction Data
Ebay	EBay Auction Data
News ML	News
PSD	Protein Sequence Database
Mondial 3.1	World Geographic Database
321gone	Auction

In order to examine the frequency of covering and duplicate-free DTD rules in real-world applications, we obtained 27 real-world DTDs, 13 using the Google search engine and 14 from the XML Data Repository[3]. The DTD names and a brief description of their application domains are given in Table 1.

We classified the content models into four separate groups as shown in Table 2. The first and the second columns of Table 2 show, respectively, the DTD names, and the number of rules in each DTD. The last four columns show, respectively, the

[3] `http://www.cs.washington.edu/research/xmldatasets`

Table 2. The classification of DTD rules

DTD Name	**Number of Rules**	**Covering**		**Non-covering**	
		Duplicate-free	Duplicates	Duplicate-free	Duplicates
Oagis	617	422	161	16	18
ODML 1.0	85	84	0	1	0
LevelOne	31	29	0	2	0
Ecoknowmics	224	221	1	2	0
XML Schema	26	19	1	6	0
HP	59	59	0	0	0
Meerkat	14	14	0	0	0
OSD	15	14	0	1	0
Opml	15	15	0	0	0
Rss-091	24	24	0	0	0
TV-Schedule	10	10	0	0	0
Xbel-1.0	9	9	0	0	0
XHTML1-strict	77	74	1	2	0
Newspaper	7	7	0	0	0
DBLP	37	37	0	0	0
Music ML	12	9	3	0	0
XMark DTD	77	76	0	1	0
Yahoo	32	32	0	0	0
Reed	16	16	0	0	0
Nlm Medline	41	41	0	0	0
SigmodRecord	11	11	0	0	0
Ubid	32	32	0	0	0
Ebay	32	32	0	0	0
News ML	116	112	0	4	0
PSD	66	64	0	2	0
Mondial 3.1	23	23	0	0	0
321gone	32	32	0	0	0
Total	1740	1518	167	37	18
Percentage	100%	87.3%	9.6%	2.1%	1.0%

Table 3. The number of DTDs (out of 100) in each of the four categories

	Duplicate-free	Duplicates
Covering	47	8
Non-covering	28	17

number of rules that are (i) covering and duplicate-free, (ii) covering with duplicates, (iii) non-covering but duplicate-free, and (iv) non-covering with duplicates.

A quick glance at Table 2 reveals that the majority of the rules (87.3%) in these applications possess both the covering and duplicate-free properties. Most of the rest (11.7%) have exactly one of these properties, and only 1.0% of the rules are neither covering nor duplicate-free.

It should be pointed out that the 3 rules from the Music ML DTD[4] that contain duplicates are as follows:

```
musicrow     ((entrysegment, segment+) | (entrysegment, segment+, text))
entrysegment ((entrypart) | (entrypart, entrypart))
segment      ((subsegment) | (subsegment, subsegment))
```

These rules are not even "unambiguous" as required by the XML specification. They can, however, easily be rewritten to be unambiguous as follows

```
musicrow     (entrysegment, segment+, text?)
entrysegment (entrypart, entrypart?)
segment      (subsegment, subsegment?)
```

resulting in the first rule becoming duplicate-free as well.

In further experiments, we obtained 73 more DTDs using Google and classified the content models into the same four groups. There were 3794 rules in total in these 73 DTDs and the majority of the rules (93.2%) possessed both the covering and duplicate-free properties. Most of the rest (6.1%) had exactly one of these properties, and only 0.7% of the rules were neither covering nor duplicate-free.

Because of our assumption that even a single rule that is non-covering (or contains duplicates) results in the DTD being classified as non-covering (or not duplicate-free), we need to determine which DTDs, as opposed to which rules, are covering (duplicate-free). Table 3 classifies the examined 100 DTDs into the four categories of covering and duplicate-free (top left), covering with duplicates (top right), non-covering but duplicate-free (bottom left) and non-covering with duplicates (bottom right). As shown in Table 3, the largest number (47%) of DTDs possess both properties, with only 17% possessing neither property.

These experiments suggest, as a rule of thumb, that most real-word DTDs should be covering or duplicate-free. In fact, about 55% of the DTDs examined in the experiments were covering, and about 62% of the remaining (non-covering) DTDs (i.e. 28% of the whole) were duplicate-free. That is, 83% of the examined DTDs possessed at least one of the properties of being covering or duplicate-free.

4 XPath Satisfiability Under Real-World DTDs

In this section, we concentrate on the satisfiablity problem of XPath queries under "real-world" DTDs, i.e. those which are either duplicate-free or covering. We will see that although the satisfiability problem is NP-complete or worse for many XPath fragments under general DTDs, it is in PTIME for certain XPath fragments when the underlying DTDs have the duplicate-free or covering properties.

4.1 XPath Satisfiability Under Duplicate-Free DTDs

Recall that a DTD is duplicate-free if each element name appears at most once in each content model. The fact that duplicate-free DTDs are easier to analyze was

[4] `http://xml.coverpages.org/musicML-DTD.txt`

previously noted in [14], where it is shown that deciding containment under a DTD even for $XP^{\{/,[]\}}$is coNP-complete, but that it reduces to PTIME when the DTD is duplicate-free. Below we show that the analysis of XPath satisfiability under duplicate-free DTDs is also simpler for certain fragments.

Before doing so, we state the following straightforward result.

Proposition 1. *Given a DTD D, deciding whether D is duplicate-free can be done in PTIME.*

Benedickt *et al.* show that, in general, $SAT(XP^{\{/,[],*\}})$ and $SAT(XP^{\{[],//\}})$ are both NP-hard [2]. In fact, a result in [13] implies that $SAT(XP^{\{/,[]\}})$ is also NP-hard. However, we have the following result for duplicate-free DTDs.

Theorem 1. *Under duplicate-free DTDs, SAT($XP^{\{/,[]\}}$) is in PTIME.*

Proof. To prove the theorem, we first present two lemmas:

Lemma 1. *Let R be a duplicate-free regular expression and C be a nonempty set of symbols appearing in R. Then there is a string w_c in $L(R)$ which covers all the symbols in C if and only if every subexpression $(R_1|R_2)$ of R, where both R_1 and R_2 contain a symbol in C, appears as a subexpression of $(R_3)^*$ or $(R_3)^+$ for some R_3.*

Proof. It is important to note firstly that R is duplicate-free and that R is assumed to use every symbol in C. Therefore, each symbol in C appears exactly once in R.

Assume R contains a subexpression $(R_1|R_2)$ to which no closure operator applies, such that R_1 contains $c_1 \in C$ and R_2 contains $c_2 \in C$. Then each string in $L(R)$ containing c_1 has to exclude c_2 and each string in $L(R)$ containing c_2 has to exclude c_1, which means that no string in $L(R)$ covers C.

Conversely, assume there is no string in $L(R)$ which covers C. Since all the symbols in C appear in R, there must be a pair of distinct symbols c_1 and c_2 in C such that there are strings w_1 and w_2 in $L(R)$ such that c_1 (but not c_2) appears in w_1 and c_2 (but not c_1) appears in w_2, but no string in $L(R)$ contains both c_1 and c_2. Hence there must be a subexpression $(R_1|R_2)$ in R such that c_1 appears in R_1 (or R_2) and c_2 appears in R_2 (respectively R_1). Furthermore, the expression $(R_1|R_2)$ cannot be subject to a closure operator; otherwise there would be a string in $L(R)$ containing both c_1 and c_2. □

Lemma 2. *Let p be a two level XPath query in the fragment $XP^{\{/,[]\}}$ such that the root v_{root} has $n \geq 0$ leaf children. Then the satisfiability of p under a duplicate-free DTD D can be decided in PTIME.*

Proof. Let R_{root} be the regular expression representing the content model in the DTD D for the root node v_{root} of p. In the case that more than one child of v_{root} has the same label, say b, either the symbol b is subject to some closure operator in R_{root} (i.e. "$*$" or "$+$" applies to b or to a term in which b is contained) or not. In the former case, v_{root} can have a b-child (while D-consistent) if, and only if,

it can have many of them, and in the latter case all such b-children must map to the same b-node in any document tree which satisfies D anyway. Therefore, we only need to check whether $L(R_{root})$ contains a word w_c which includes all of the labels in C (with an arbitrary ordering), where C is the set, as opposed to the multiset, of labels of the children of v_{root}.

For each symbol b in C, if there is no symbol b in R_{root}, which can be checked in PTIME, then the answer (to whether $L(R_{root})$ contains w_c) is false. Otherwise, Lemma 1 applies and we only need to check whether R_{root} contains some expression $(R_1|R_2)$ to which no closure operator applies and such that both R_1 and R_2 contain some symbol in C. The number of "|" operators in R_{root} is $O(|R_{root}|)$ and to obtain each expression $(R_1|R_2)$ in R_{root} requires $O(|R_{root}|)$ time. For each expression $(R_1|R_2)$ in R_{root}, to check whether R_i, $i = 1, 2$, covers some symbol in C requires $O(|R_i| \times |C|)$ time. □

We now prove the theorem:

Let p be a given XPath query in the fragment $XP^{\{/,[]\}}$. For each internal node v in p, if v has more than one child with the same label, say b, then there are two possibilites: (i) the symbol b is subject to some closure operators in the regular expression, say R_v, corresponding to the content model of v in the DTD (i.e. "$*$" or "$+$" applies to b or to a term in which b is contained), or (ii) the symbol b is not subject to a closure operator, hence all such b-children must map to the same b-node in any document tree which satisfies the DTD D (because D is duplicate-free). The latter case was previously called a *functional constraint* in[14] and was shown to be detectable in PTIME. In case(i), there is no restriction on the number of b-children of $label(v)$ (where $label(v)$ denotes the label of v) in any document tree which satisfies D, hence such b-children must not be merged. In case (ii), we merge all such b-children of v, which is done in PTIME. So we assume in the rest of the proof that case (ii) holds; that is, they must not be merged. Let $SubP(v)$ denote the two-level subtree of p rooted at v. This implies that $SubP(v)$ has all the children of v as leaves.

Based on the above terminology and the definition of satisfiablity, p is satisfiable if and only if the subtree $SubP(v)$ is satisfiable for each internal node v in p. Using Lemma 2, to decide whether such a subtree is satisfiable is in PTIME. On the other hand, there are altogether m subtrees, where m is the number of internal nodes, that is $m = O(|p|)$. Therefore, to decide whether p is satisfiable is in PTIME. □

Even if a DTD as a whole is not duplicate-free, the above positive results can be used. For example, given a DTD D and a query p in $XP^{\{/,[]\}}$, if every internal node in the tree representing p is labelled by an element name whose content model is duplicate-free, then the satisfiability of p can be determined in PTIME. This gives us the following.

Corollary 1. *Given a query p in $XP^{\{/,[]\}}$ and a DTD D, $SAT(XP^{\{/,[]\}})$ is in PTIME if each rule in D in which a symbol from p appears is duplicate-free.*

We now consider the satisfiability of some other fragments of XPath under duplicate-free DTDs. The fact that $SAT(XP^{\{/,//,*\}})$ is in PTIME under duplicate-free DTDs

follows trivially from a result in [2] showing that this fragment including union is in PTIME in general. However, we have the following negative results.

Theorem 2. *Under duplicate-free DTDs, the following problems are NP-hard:*

1. *SAT(XP$^{\{/,[],*\}}$)*
2. *SAT(XP$^{\{[],//\}}$)*
3. *SAT(XP$^{\{/,[],\cup\}}$)*

Proof. (1) Benedikt *et al.* show in [2] that SAT(XP$^{\{/,[],*\}}$) is NP-hard by reduction from the 3SAT problem. Given a well-formed Boolean formula $\phi = C_1 \wedge \cdots \wedge C_n$ over variables $x_1, \ldots, x_m$, they define a DTD D (with start symbol S) and the following rules:

$$\begin{aligned} S &\rightarrow X_1, \ldots, X_m \\ X_i &\rightarrow T_i | F_i, \text{for } i \in [1, m] \\ T_j &\rightarrow C_{j_1}, \ldots, C_{j_k} \ /^* \text{ all clauses } C_{j_i} \text{ in which } x_j \text{ appears } */ \\ F_j &\rightarrow C_{j_1}, \ldots, C_{j_k} /^* \text{ all clauses } C_{j_i} \text{ in which } \overline{x}_j \text{ appears } */ \end{aligned}$$

They then define a query $XP(\phi) = /S[*/*/C_1] \cdots [*/*/C_n]$ such that ϕ is satisfiable iff $(XP(\phi), D)$ is satisfiable. As the DTD rules are duplicate-free, we can deduce that SAT(XP$^{\{/,[],*\}}$) under duplicate-free DTDs is NP-hard.

(2) The proof is the same as the proof of (1), except that $XP(\phi)$ is defined as $XP(\phi) = /S[.//C_1] \cdots [.//C_n]$.

(3) Using the same approach as (1), Benedikt *et al.* show in [2] that SAT (XP$^{\{/,[],\cup\}}$) is NP-hard. The DTD rules used in the proof are the following:

$$\begin{aligned} S &\rightarrow X \\ X &\rightarrow (X?), (T \mid F) \end{aligned}$$

and $XP(\phi) = /S[XP(C_1)] \cdots [XP(C_n)]$, where $XP(C_i)$ is defined as follows:

- For each variable x_i in ϕ, $XP(x_i) = X^i/T$ and $XP(\bar{x}_i) = X^i/F$, where X^i is the chain $X/\cdots/X$ of length i.
- For each clause C_j, $XP(C_j)$ is C_j in which each x_i is replaced by $XP(x_i)$ and each $\bar{x}_i$ is replaced by $XP(\bar{x}_i)$.

As the DTD rules are duplicate-free, SAT(XP$^{\{/,[],\cup\}}$) under duplicate-free DTDs is NP-hard. □

4.2 XPath Satisfiability Under Covering DTDs

Recall that the majority of DTDs studied in Section 3 were classified as covering. In this section we prove that SAT(XP$^{\{/,[],*,//,\cup\}}$) is in PTIME under covering DTDs. We should first point out the following fact which follows directly from a result in [13].

Proposition 2. *Given a DTD D, deciding whether D is covering is NP-complete.*

However, since we expect query processors to have to deal with relatively few, known DTDs while answering large numbers of XPath queries, the cost of detecting the covering property will be a one-off cost for each DTD.

Theorem 3. *Under covering DTDs, SAT(*$XP^{\{/,[],*,//,\cup\}}$*) is in PTIME.*

Proof. We prove this by the same method as Benedikt *et al.* use in [2], where they show that SAT($XP^{\{/,[],*,//,\cup\}}$) under disjunction-free DTDs is in PTIME. Clearly, a disjunction-free DTD is a special case of a covering DTD. It turns out, however, that the same proof technique can be used, the only substantial difference being that $*$ is implicitly permitted in all the XPath fragments considered by [2] that include the child axis, whereas we distinguish explicitly whether or not $*$ is included. We also need to adapt the proof from [2] to account for the different syntax for XPath used in that paper.

Let p be a query in ($XP^{\{/,[],*,//,\cup\}}$) and D be a covering DTD. We first construct a DTD digraph $G(V, E)$, with the set V of element names in D. G is rooted at $S_0 \in V$, where S_0 is the start symbol of D. For simplicity and without loss of generality we assume that neither D nor p is empty. Let a and b be two distinct symbols in V. There is an edge in E from vertex a to vertex b if and only if b appears in the content model of a in D. We start by compiling the list L of all sub-queries of p, topologically ordered such that p_1 precedes p_2 in L if p_1 is a sub-query of p_2. For each $p' \in L$ and element name a in D we use a variable $reach(p', a)$ to hold the set of all element names reachable from a via p' in the DTD graph G. These variables are initially set to $\emptyset$, except that $reach(S_0, S_0) = \{S_0\}$. We also use a variable $sat(p', a)$ to hold the truth value indicating whether or not p' is satisfiable at a.

The decision algorithm is outlined as follows[5]:

1. For each $p' \in L$ (in the order of L) and element name a in D, we compute $reach(p', a)$ and $sat(p', a)$, based on the structure of p':
   (a) $p' = .$: then $reach(p', a) = \{a\}$;
   (b) $p' = l$: then $reach(p', a) = \{l\}$ if l appears in the content model of a in D;
   (c) $p' = *$: then $reach(p', a)$ is the set of element names which appear in the content model of a;
   (d) $p' = \epsilon$: then $reach(p', a)$ is the set of element names reachable from a in G (ϵ can only appear as the *empty* XPath step, as in //);
   (e) $p' = p_1 \cup p_2$: then $reach(p', a) = reach(p_1, a) \cup reach(p_2, a)$;
   (f) $p' = p_1/p_2$: then $reach(p', a) = \bigcup_{b \in reach(p_1, a)} reach(p_2, b)$;
   In all the cases above, $sat(p', a) = true$ iff $reach(p', a) \neq \emptyset$.
   (g) $p' = .[p_1]$: $sat(p', a) = sat(p_1, a)$, and $reach(p', a) = \{a\}$ if $sat(p_1, a) = true$;
2. Return $sat(p, S_0)$, where S_0 is the root of G.

[5] The ϵ and $\downarrow^*$ used in [2] correspond to our '.' and ϵ, respectively.

Since $p[p_1] = p/.[p_1]$, the inductive case for $p[p_1]$ is reduced to p_1/p_2 and $.[p_1]$.

The algorithm iterates over all sub-queries in L and all element names in D. Hence, the main loop in the algorithm is executed at most $O(|p||D|)$ times. Each step in the loop takes at most $O(|D|)$ time. Hence, the worst-case time complexity is $O(|p||D|^2)$.

The proof that the algorithm returns true iff (p, D) is satisfiable follows the method used in [2]. The same method works because of the fact that since D is covering, $.[q_1][q_2]\cdots[q_n]$, e.g., is satisfiable at an element labelled a if and only if each of $q_1, q_2, \ldots, q_n$ is satisfiable at an element labelled a. This is also true for duplicate-free DTDs [2], but is not true in general. □

Corollary 2. *Given a query p in $XP^{\{/,[],*,//,\cup\}}$ and a DTD D, $SAT(XP^{\{/,[],*,//\}})$ is in PTIME if each rule in D in which a symbol from p appears is covering.*

5 Conclusion and Future Work

This paper was concerned with discovering properties of real-world DTDs and their impact on the satisfiability problem. The motivation behind this was the authors' belief that although common XPath problems are of high complexity, e.g. NP-hard, in general, real-world applications usually provide simpler structures under which such otherwise hard problems could be performed in PTIME.

In particular, we examined several real-world DTDs and discovered a new property, called covering, which most of them preserved. We observed that even the minority of the examined real DTDs which did not possess the covering property were duplicate-free. We showed that the satisfiability problem of the XPath fragment $XP^{\{/,[],*,//,\cup\}}$reduces to PTIME when the underlying DTD has the covering property. We also showed that the satisfiability of the fragment $XP^{\{/,[]\}}$reduces to PTIME when the underlying DTD is duplicate-free. These problems were previously shown to be NP-hard under general DTDs.

The presented work is just a starting point in the direction of discovering features of real-world applications and deriving low-cost algorithms for problems such as query satisfiability, containment, and minimization. Among possible avenues for further research in this regard are the following:

- The experimental results in this paper showed that most of the DTDs classified as non-covering (respectively having duplicates) were done so because of only a few rules being non-covering (respectively containing duplicates). This suggests that one might introduce the concept of *locally*, vs. *globally*, covering (respectively duplicate-free) DTDs. The satisfiability problem under locally-covering (respectively locally duplicate-free) DTDs may still be in PTIME if the given queries preserve certain features.
- One could investigate PTIME algorithms for XPath *containment* under covering and duplicate-free DTDs. Some results for containment of queries in $XP^{\{/,[]\}}$under duplicate-free DTDs are given in [14].
- When examining the DTDs, we noticed that some of the DTD rules are classified as having duplicates because of patterns such as $(a, b)|(a, c, d)$ where

a is duplicated. However, such a pattern is equivalent to $a, (b|(c, d))$ which is duplicate-free. Considering such a semantic notion of duplicate-free would increase the percentage of real-world DTDs classified as duplicate-free.

– We believe that, by combining the methods for covering and duplicate-free DTDs, SAT($XP^{\{/,[]\}}$) can be decided in PTIME if each rule in a DTD is either covering or duplicate-free. This would then be applicable to 92 out of the 100 DTDs covered by our experiments.

References

1. Amer-Yahia, S., Cho, S., Lakshmanan, L.V.S., Srivastava, D.: Tree pattern query minimization. The VLDB Journal 11, 315–331 (2002)
2. Benedikt, M., Fan, W., Geerts, F.: XPath satisfiability in the presence of DTDs. Proc. Twenty-fourth ACM Symp. on Principles of Databases Systems (2005) (to appear in J. ACM)
3. Bex, G.J., Neven, F., Van den Bussche, J.: DTDs versus XML schema: A practical study. In: Proc. Seventh Int. Workshop on the Web and Databases, pp. 79–84 (2004)
4. Choi, B.: What are real DTDs like? In: Proc. Fifth Int. Workshop on the Web and Databases, pp. 43–48 (2002)
5. Flesca, S., Furfaro, F., Masciari, E.: On the minimization of XPath queries. In: Proc. 29th Int. Conf. on Very Large Data Bases, pp. 153–164 (2003)
6. Geerts, F., Fan, W.: Satisfiability of XPath queries with sibling axes. In: Proc. 10th Int. Workshop on Database Programming Languages, pp. 122–137 (2005)
7. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. ACM Trans. on Database Syst. 30(2), 444–491 (2005)
8. Hidders, J.: Satisfiability of XPath expressions. In: Proc. 9th Int. Workshop on Database Programming Languages (September 2003)
9. Lakshmanan, L., Ramesh, G., Wang, H., Zhao, Z.: On testing satisfiability of tree pattern queries. In: Proc. 30th Int. Conf. on Very Large Data Bases, pp. 120–131 (2004)
10. Miklau, G., Suciu, D.: Containment and equivalence for a fragment of XPath. J. ACM 51(1), 2–45 (2004)
11. Neven, F., Schwentick, T.: On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. Logical Methods in Computer Science 2(3) (2006)
12. Ramanan, P.: Efficient algorithms for minimizing tree pattern queries. In: Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 299–309. ACM Press, New York (2002)
13. Wood, P.T.: Minimising simple XPath expressions. In: Proc. Fourth Int. Workshop on the Web and Databases, pp. 13–18 (2001)
14. Wood, P.T.: Containment for XPath fragments under DTD constraints. In: Proc. 9th Int. Conf. on Database Theory, pp. 300–314 (2003)

Fast Answering of XPath Query Workloads on Web Collections

Mariano P. Consens and Flavio Rizzolo

University of Toronto
{consens,flavio}@cs.toronto.edu

Abstract. Several web applications (such as processing RSS feeds or web service messages) rely on XPath-based data manipulation tools. Web developers need to use XPath queries effectively on increasingly larger web collections containing hundreds of thousands of XML documents. Even when tasks only need to deal with a single document at a time, developers benefit from understanding the behaviour of XPath expressions across multiple documents (e.g., what will a query return when run over the thousands of hourly feeds collected during the last few months?). Dealing with the (highly variable) structure of such web collections poses additional challenges.

This paper introduces DescribeX, a powerful framework that is capable of *describing* arbitrarily complex XML summaries of web collections, enabling the efficient evaluation of XPath workloads (supporting all the axes and language constructs in XPath). Experiments validate that DescribeX enables existing document-at-a-time XPath tools to scale up to multi-gigabyte XML collections.

1 Introduction

Web applications rely heavily on XML tools to manipulate data encoded in XML. Data can be exchanged, as in web feeds (blogs, news feeds, podcasts) or via web service messages. Data can also be stored, as in hypertext collections like Wikipedia. Several XML manipulation tasks (and the tools used to implement them) process one document at a time, whether the document is an individual RSS file, a single SOAP message, or a Wikipedia article in XHTML. The vast majority of software tools utilized in this context rely on XPath as the core dialect for XML querying. Hence, web developers make extensive use of embedded XPath queries for processing XML collections.

A developer working with this type of collection faces several challenges. She must learn enough about the (semi)structure present in the XML collection to be able to write meaningful XPath queries. She must also develop an understanding of how the XPath expressions behave across different documents in the collection.

Understanding the actual structure of a web collection can be a significant barrier. Some collections (like Wikipedia or personal blogs) do not really have a schema, or the schema allows most elements to occur almost anywhere. Even when XML documents are validated against a proper schema, their actual structure can vary significantly across the collection. This can happen because the

D. Barbosa et al. (Eds.): XSym 2007, LNCS 4704, pp. 31–45, 2007.

schema is large and only small (possibly disjoint) subsets are actually used (as happens with industry standard schemas, like IXRetail[1]), or because schemas can be arbitrarily composed using open content models (e.g. RSS extensions like Yahoo Media, podcasts, etc.). In these scenarios, schemas alone are not that helpful for understanding (nor for optimizing) XPath evaluation.

This paper argues that DescribeX, a tool supporting powerful structural summaries, can help with understanding the (semi)structure of large collections of XML documents. In fact, DescribeX summaries contribute to significantly speed up (and scale up) XPath evaluation with existing file at a time tools, enabling fast exploration of the results of XPath workloads on large collections.

XML structural summaries are graphs representing relationships between sets in a partition of XML elements. DescribeX summaries have a unique capability: they are the first ones to *describe* precisely the structural commonality that determine each individual set in the partition. DescribeX introduces a language of *axis path regular expressions* (*AxPREs*, for short) to describe the sets.

Most of the existing summary proposals define all sets in the partition using the same criteria, hence creating *homogeneous* summaries. These summaries are based on common element paths (in some cases limited to length k), whether incoming paths [7,11,17], both incoming and outgoing paths [12,21], or sequence of outgoing paths (common subtrees) [3]. The few examples of *heterogeneous* (adaptive) summaries [5,23] have no capability for describing the partitions, which are defined according to very simple criteria (e.g., just the incoming paths).

In contrast, DescribeX supports constructing heterogenous summaries where each set in the partition can be created according to *explicit criteria* obtained from an expression in the complete XPath language (all the axes, document order, use of parenthesis, etc.). Given an arbitrary XPath query, DescribeX can create a partition defined by an AxPRE that captures exactly the structural commonality expressed by the query.

This paper presents experimental results that demonstrate that using a summary created from a given workload can produce query evaluation times orders of magnitude better than using existing summaries. The experiments also validate that DescribeX summaries allow file at a time XPath processors to be a competitive alternative (in terms of performance) to DB-like XML query engines – even on gigabyte sized collections.

Overview and Contributions. The next section walks through a concrete example to illustrate how DescribeX summaries can help developers understand the behaviour of XPath queries across large XML collections. The following two sections present the main technical contributions of the paper. Section 3 provides an overview of the rich framework for describing summaries underlying the DescribeX tool (based on the novel technique of applying bisimilarity to element neighborhoods described by an AxPRE). Section 4 gives a translation from XPath expressions into AxPREs, hence supporting the creation of summaries with nodes that answer complex XPath expressions. The system contributions

[1] `http://www.nrf-arts.org/`

are presented in Section 5, where the implementation of the DescribeX tool is outlined, and in Section 6, where experimental results on gigabyte collections provide evidence of the benefits and scalability of DescribeX. The highlights are the up to three orders of magnitude speed-ups obtained against variations of incoming and outgoing path summaries (capturing existing proposals like 1-index [17], APEX [5], A(k)-index [13], D(k)-index [23], and F+B-Index [12]). We emphasize that query evaluation times on collections the size of Wikipedia are rarely reported in the literature. In fact, XML query evaluation systems (and not just research prototypes) become challenged when working with collections at this scale. Related work is discussed in Section 7.

2 Motivating Example

Consider a developer who has to implement a web application that retrieves RSS feeds from several content providers to produce an aggregated meta feed. The feed may span several days or weeks, and there might be more than one item per day. Figure 1 shows the instances of two sample RSS feeds represented as *axis graphs*.

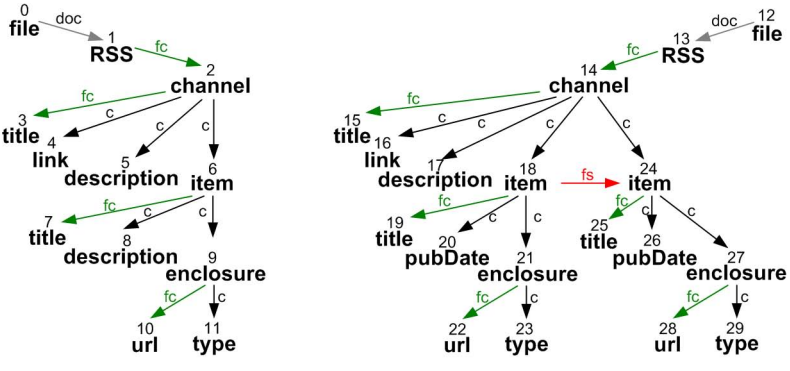


Fig. 1. Axis graphs of RSS feed samples

An axis graph can display selected binary relations between elements in an XML document tree, like *c*, *fs*, and *fc* shown in the figure (shorthands for XPath axes *child* and *following-sibling*, and for the derived axis *firstchild*, respectively). The semantics of these axes is straightforward: the edge from element 6 to 7 labeled *fc* means that 7 is the first child of 6 in document order, and the edge from element 18 to 24 labeled *fs* means that 24 is a following sibling of 18 in document order. Being binary relations, axes have inverses, e.g., the inverse of *c* is *p* (shorthand for *parent*) and the inverse of *fs* is *ps* (shorthand for *preceding-sibling*). These inverses are not shown in the figure.

Using DescribeX, the developer can create a *summary descriptor* (SD for short) like the one shown on Figure 2 (a). This *label SD*, created from the two

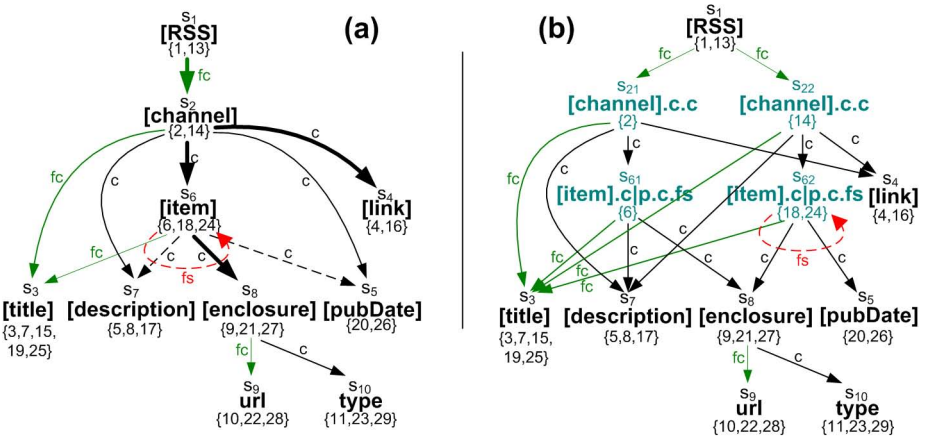


Fig. 2. (a) Label SD, and (b) heterogeneous SD of the RSS feed samples

feeds in Figure 1, partitions the elements in the feeds by element name. For example, SD node s_6 represents all the item elements in the two documents, $\{6, 18, 24\}$ (this set is called the *extent* of s_6).

An SD edge is labeled by the axis relation it represents. For instance, edge (s_6, s_5) is labeled by c, which means that there is a c axis relation between elements in the extent of s_6 and s_5. Figure 2 (a) shows three kinds of edges, depending on properties of the sets that participate in the axis relation: dashed, regular, and bold. Dashed edges, like (s_6, s_5) labeled c, mean that some element in the extent of s_6 has a child in the extent of s_5. Regular edges, like (s_6, s_3) labeled fc, mean that every element in the extent of s_6 has a first child in the extent of s_3. Finally, bold edges, like (s_6, s_8) labeled c, mean that every element in the extent of s_8 is a child of some element in the extent of s_6 and that every element in the extent of s_6 have some child in the extent of s_8.

From the label SD the developer learns that items in the collection always include title and enclosure elements, but they may contain any combination of description and pubDate elements. However, the label SD does not provide information on exactly which combinations actually appear. Since the developer knows that some items have a pubDate, she can write a query to retrieve those channels that contain such items

```
Q1 = /rss/channel[item[pubDate]]
```

The developer can decide to either run Q1 using the current SD or make DescribeX *adapt* the current SD to Q1. If she picks the second option, DescribeX changes the SD by partitioning the channel node s_2 in Figure 2 (a), which represents all channels in the collection, into two channel nodes: one that contains items with a pubDate and another one that contains items without it (s_{22} and s_{21} in Figure 2 (b), respectively).

Summaries in DescribeX are defined and manipulated via AxPREs. AxPREs describe the *neighbourhood* of the elements in a given extent. A neighbourhood of an element v for an AxPRE α is the subgraph local to v that matches α. For instance, the p^* AxPRE describes the neighbourhood of v containing all label paths from v to the root, c^* all label paths from v to the leaves, and $fc.ns^*$ the sequence of v's child labels. AxPREs can also be derived from a query in order to adapt an SD to it. For example, the $[channel].c.c$ AxPRE of node s_{21} in Figure 2 (b) was derived from Q1 and describes the neighbourhood of channel elements with common outgoing label paths of length 2 (more on this in Section 3).

The developer is also interested in items containing both title and enclosure elements, but she does not know whether such items exist in the collection and, if they do, how common they are. In addition, she wants those items to be part of a series (i.e., to belong to channels that contain more than one item element, as done in feeds for podcasts published daily). Therefore, the developer writes another query

```
Q2 = /rss/channel[item/following-sibling::item]
[not(pubDate=../item[1]/pubDate)]/item[title][enclosure]
```

Q2 contains structural (in black) and non-structural (in grey) XPath constructs. The expression that results from removing all non-structural constraints is called the *structural subquery* of Q2. A structural subquery provides insight into the behaviour of the entire query and can be used by DescribeX to change the SD. It is important to note that "structural" here is used in a broad sense since some predicates on values can also be considered structural by the user (see Example 1 in Section 3).

As with Q1, the developer can decide to either evaluate Q2 on the current SD (the label SD with the refined channel node) or to add Q2 to the workload and make DescribeX adapt the current SD. Assuming she chooses the second, the system partitions the item node s_6 from Figure 2(a) into the nodes s_{61} and s_{62} in Figure 2(b) that describe the structure of the collection w.r.t. the workload including Q2 and Q1. Note that the extent of node s_{62} is exactly the answer to the structural subquery of Q2, and thus a superset of the answer of Q2. The elements in this extent are called *candidate elements*. Hence, by adapting the SD to the structural subquery, DescribeX has considerably reduced the search space for computing the entire query.

In a document-at-a-time approach to query evaluation, adapting the SD to a workload can reduce the number of documents on which queries in the workload need to be evaluated, potentially yielding a significant speedup (see Section 6). That is, after adapting the SD to a given query Q, DescribeX can evaluate Q only on those documents (called *candidate documents*) that are guaranteed to provide a non-empty answer for the structural subquery of Q. Those candidate documents that do contain an answer for the entire query are called *answer documents*.

3 The DescribeX Framework

This section introduces the DescribeX framework that provides a powerful language based on *axis path regular expressions* (AxPREs) for describing the partitions in an SD. For representing an XML instance, DescribeX uses a labeled graph model called *axis graph*.

Definition 1 (Axis Graph). *An axis graph* $\mathcal{A} = (Inst, Axes, Label, \lambda)$ *is a structure where Inst is a set of nodes, Axes is a set of binary relations* $\{E_1^{\mathcal{A}}, \ldots, E_n^{\mathcal{A}}\}$ *in Inst* $\times$ *Inst and their inverses, Label is a finite set of node names, and* λ *is a function that assigns labels in Label to nodes in Inst. Edges are labeled by axis names and nodes are labeled by element or attribute names (including namespaces), or by new labels defined using XPath.*

An axis graph is an abstract representation of the XPath data model [2] extended with edges that represent XPath axis binary relations. Axis graphs can also include additional axes, such as *id-idrefs* or *firstchild* and *nextsibling* (abbreviated *ns*), that can be expressed in XPath (e.g., $fc := child :: *[1]$ and $ns := following\text{-}sibling :: *[1]$).

Example 1. A new node label [*mpeg*] can be defined in an axis graph by the XPath expression `enclosure[type="audio/mpeg"]`, representing enclosure elements with different types of media as separate nodes.

We introduce next the formal notion of AxPRE that will be used to describe each set in the partition of elements (i.e. extents) that define an SD.

Definition 2 (Axis Path Regular Expressions). *An* axis path regular expression *is an expression generated by the grammar*

$$E \longleftarrow axis \mid axis[B(l)] \mid (E \mid E) \mid (E)^* \mid E.E \mid \epsilon$$

where $axis \in Axes$ *and* ϵ *is the symbol representing the empty expression.*

Definition 2 describes the syntax of path regular expressions on the binary relations (labeled edges) of the axis graph including node label tests ($B(l)$ is a boolean function on a label $l \in Label$ that supports more elaborate tests on labels, beyond just matching). AxPREs can be written using XPath syntax as well, but the semantics of the constructs are interpreted differently (as in Definition 3). We refer the reader to [6] for an updated semantics of XPath and to [15] for conditional XPath.

Having defined the AxPRE language, we introduce next the notion of *AxPRE neighbourhood*, which provides a description of the subgraph local to a node in the axis graph. This AxPRE neighbourhood of a node is computed by intersecting the automaton of the AxPRE and the axis graph starting from the node (i.e. the node must intersect the initial state of the automaton). The intersection between an automaton and a graph is a construction described in [16] (note that in our case we do not require the expensive simple path semantics).

[2] `http://www.w3.org/TR/xpath20`

Definition 3 (AxPRE Neighbourhood of v). *Let $\mathcal{A}$ be an axis graph, v a node in $\mathcal{A}$, α an AxPRE, and $NFA(\alpha)$ the non-deterministic finite automaton of α accepting all prefixes. The* AxPRE neighbourhood *of v for α, denoted $\mathcal{N}_\alpha(v)$, is the subgraph of $\mathcal{A}$ product of the intersection between $\mathcal{A}$ and $NFA(\alpha)$ such that v intersects with the initial state of $NFA(\alpha)$.*

This approach for defining summaries is based on the intuition that nodes that have *similar* neighbourhoods should be grouped together in an extent. The notion of similarity we use is the familiar notion of *labeled bisimulation*.

Definition 4 (Labeled Bisimulation). *Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be two subgraphs of an axis graph $\mathcal{A}$, such that $Axes_{\mathcal{G}_1} \subseteq Axes$ and $Axes_{\mathcal{G}_2} \subseteq Axes$. A labeled bisimulation between $\mathcal{G}_1$ and $\mathcal{G}_2$ is a symmetric relation $\approx$ such that for all $v \in \mathcal{G}_1$, $w \in \mathcal{G}_2$, $E_i^{\mathcal{G}_1} \in Axes_{\mathcal{G}_1}$, and $E_i^{\mathcal{G}_2} \in Axes_{\mathcal{G}_2}$: if $v \approx w$, then $\lambda(v) = \lambda(w)$; if $v \approx w$, and $\langle v, v' \rangle \in E_i^{\mathcal{G}_1}$, then $\langle w, w' \rangle \in E_i^{\mathcal{G}_2}$ and $v' \approx w'$.*

Example 2. Consider neighbourhoods $\mathcal{N}_{[item].c|p.c.fs}(18)$ and $\mathcal{N}_{[item].c|p.c.fs}(24)$ in Figure 1, which is computed by the $NFA([item].c|p.c.fs)$ accepting all prefixes (Definition 3). Both neighbourhoods consist of the subgraph given by the c edges from 14, c edges from 18 and 24, and the fs edge between 18 and 24. They differ only in one edge: while $\mathcal{N}_{[item].c|p.c.fs}(18)$ contains a p edge from 18, $\mathcal{N}_{[item].c|p.c.fs}(24)$ contains a p edge from 24 (p edges are not shown in the figure). However, according to Definition 4, they are bisimilar and thus nodes 18 and 24 belong to the same extent (that of node s_{62} in Figure 2 (b)). In contrast, neighbourhood $\mathcal{N}_{[item].c|p.c.fs}(6)$ does not contain an fs edge and thus it is not bisimilar to either $\mathcal{N}_{[item].c|p.c.fs}(18)$ nor $\mathcal{N}_{[item].c|p.c.fs}(24)$. Consequently, node 6 is assigned to a different extent (that of node s_{61} in Figure 2 (b)).

A bisimulation provides a way of computing a double homomorphism between graphs. The widespread use of bisimulation in summaries is motivated by its relatively low computational complexity properties. The bisimulation reduction of a labelled graph can be done in time $O(m \log m)$ (where m is the number of edges in a labelled graph) as shown in [19], or even linearly for acyclic graphs, as shown in [10]. Using bisimulation also allows us to capture all the existing bisimulation-based proposals in the literature [8].

Definition 5 (AxPRE Partition). *Let $\mathcal{A}$ be an axis graph, $N \subseteq Inst$, and α an AxPRE. An* AxPRE partition *of N for α, denoted $\mathcal{P}_\alpha(N) = \{P_i | \bigcup_i P_i = N$ and $\bigcap_i P_i = \emptyset\}$, is a partition of the nodes in N defined as follows: two nodes $v, w \in N$ belong to the same class $P_i \in \mathcal{P}_\alpha(N)$ iff there exists a* labeled bisimulation $\approx$ *between $\mathcal{N}_\alpha(v)$ and $\mathcal{N}_\alpha(w)$ such that $v \approx w$.*

Definition 6 (Summary Descriptor (SD)). *Let $\mathcal{A}$ be an axis graph. A* summary descriptor *(SD for short) of $\mathcal{A}$ is an structure $\mathcal{D} = (P, G)$ that consists of a set of AxPRE partitions $P = \{\mathcal{P}_{\alpha_1}(N), \ldots, \mathcal{P}_{\alpha_n}(N)\}$, and a labeled graph G, called* SD graph*, representing axis relationships between elements in the equivalence classes of the AxPRE partitions. Each node s in the SD graph has associated one set in the partition called the* extent *of s (denoted extent(s)).*

In addition, each SD node is labeled by the AxPRE α_i that defines its extent. Like in the axis graph, SD graph edges are labeled by the axis relation they represent.

When the extents of all nodes in a SD $\mathcal{D}$ are defined with the same AxPRE α we have an *homogeneous* SD. In this case we say that $\mathcal{D}$ is an α SD. In contrast, if at least two different nodes are defined with different AxPREs we have an *heterogeneous* SD.

4 From XPath to AxPREs

We mentioned in Section 2 that DescribeX can adapt an SD node to an XPath query Q. This section formalizes how an AxPRE is obtained from Q by using two derivation functions L and P given in Figure 3.

$$P(Op(e_1,\ldots,e_m)) := \epsilon \quad (1)$$

$$P(axis::l[e_1]\ldots[e_m]/rlocpath) := Ax(axis).(P(e_1)|\ldots|P(e_m)|P(rlocpath)) \quad (2)$$

$$P((locpath)[e_1]\ldots[e_m]/rlocpath) := P(locpath).(P(e_1)|\ldots|P(e_m)|P(rlocpath)) \quad (3)$$

$$P(locpath_1|\ldots|locpath_m) := (P(locpath_1)|\ldots|P(locpath_m)) \quad (4)$$

$$L(rlocpath/axis::l[e_1]\ldots[e_m]) := Ax(axis^{-1}).L(rlocpath)|P(e_1)|\ldots|P(e_m) \quad (5)$$

$$L(rlocpath/(locpath)[e_1]\ldots[e_m]) := L(locpath).L(rlocpath)|P(e_1)|\ldots|P(e_m) \quad (6)$$

$$L(locpath_1|\ldots|locpath_m) := (L(locpath_1)|\ldots|L(locpath_m)) \quad (7)$$

Fig. 3. AxPRE derivation functions L and P

Example 3. Consider the following query

```
Q3 = (rss | RDF)/channel[item[pubDate][not(pubDate=../item[1]/pubDate)]]
```

Q3 returns all channels that have RDF or rss parents and item children with a pubDate different from the pubDate of the first item in the channel. Note that the structural subquery appears in black (the last predicate in grey is not part of the structural subquery) and that $Q3$ is in abbreviated syntax (*channel* and *item* for instance, mean *child* :: *channel* and *child* :: *item*, respectively).

The first rule of Figure 3 that applies is (5), resulting in

$$Ax(child^{-1}).L((rss|RDF))|P(e_1)$$

where $e_1 = item[pubDate][not(pubDate = ../item[1]/pubDate)]$ and Ax is a function that translates the XPath axis into its AxPRE axis counterpart. In particular, $Ax(axis^{-1})$ returns the actual AxPRE inverse (e.g., $child^{-1}$ is converted into p) and recursive axes are translated to an equivalent Kleene closure of non-recursive axes (e.g., *descendant* translates into c^*).

For expanding $P(e_1)$, the first rule invoked is (2) with $axis = child$, $l = item$, an empty $rlocpath$, and two predicates $[pubDate]$ and $[not(pubDate = ../item[1]/pubDate)]$. Since the second predicate is a function, it matches rule (1) and the result of $P(not(pubDate = ../item[1]/pubDate))$ is ϵ (Remember that this predicate is not part of the structural subquery). The application of rule (2) to the only remaining predicate $[pubDate]$ results in $P(e_1) = c.(c)$. Since $c.(c) = c.c$, then $P(e_1) = c.c$.

For expanding $L((rss|RDF))$, the rule that applies is (6) with no predicates and an empty $rlocpath$, which simply results in $L(rss|RDF)$. The expansion continues by invoking rule (7) with $locpath_1 = rss$ and $locpath_2 = RDF$. At this point, the partial expansion of $Q3$ is

$$p.(L(rss)|L(RDF))|c.c$$

Both $L(rss)$ and $L(RDF)$ match rule (5) with $axis = child$, no predicates and an empty $rlocpath$. Therefore, $L(rss) = L(RDF) = p$, being the resulting AxPRE $p.(p|p)|c.c$. Since $(p|p) = p$, we obtain the simplified AxPRE $p.p|c.c$. Finally, the node test of the step corresponding to the answer (*channel* in this case) is prefixed as a label predicate to the AxPRE. Therefore, the resulting AxPRE of query $Q3$ is

$$\alpha = [channel].p.p|c.c$$

Once the query AxPRE α of a given XPath query Q is computed, the next step in adapting the SD to Q is finding the SD node whose AxPRE α' contains α. (The problem of AxPRE containment is related to that of regular expression containment [14].) After finding the node, DescribeX proceeds to change α' to α, which in fact modifies the description of the node and thus the neighbourhood it summarizes. This entails performing a *refinement* of the extent of the node. For instance, in order to adapt the SD of Figure 2 (a) to query Q2 from Section 2, the extent of s_6 was refined into two sets (Figure 2 (b)). An in-depth discussion of refinements is beyond the scope of this paper and can be found in [8].

5 Document-at-a-time Evaluation Using SDs

In the previous section we have shown how to translate any XPath expression into an equivalent AxPRE. In this section we will discuss how this AxPRE can be used to find the SD nodes that contain candidate documents.

DescribeX is implemented in Java using Berkeley DB Java Edition to store and manage indexed collections (tables). The DescribeX tool can invoke an arbitrary XPath processor for the evaluation of XPath expressions. Saxon [3] was used for the experiments reported here.

The DescribeX architecture is tailored to process XML collections one file at a time, the prevalent data processing model for the Web. Each file is parsed and processed independently of the other files in the collection. The extent relation is

[3] `http://saxon.sourceforge.net/`

stored in an indexed table named `elemDB` that has schema `elemDB(SID, docID, endPos, startPos)`, where the underlined attributes are the key (also used for indexing). The `elemDB` table contains a tuple for each XML element in the collection. Each SD node is identified by a unique id called SID. Each element belongs to the extent of a unique SD node, whose SID is stored in the `SID` attribute. The attribute `docID` holds the identifier of the document in which the element appears. The `startPos` and `endPos` are the positions, in the document, where the element starts and ends, respectively.

Once DescribeX has computed the query AxPRE α of a given XPath query Q as described in the previous section, it needs to find the SD node whose AxPRE contains α in order to get the candidate documents for evaluating Q. If there is an SD node s with AxPRE α, then all docIDs from the ElemDB table that correspond to s are in fact candidate documents. In contrast, if s has an AxPRE α' containing α, DescribeX has two alternatives. One, it can adapt the SD by refining s from α' to α and then get the candidate documents as in the previous case. Two, it can get all docIDs from the ElemDB table that correspond to s and run the structural subquery of Q on them in order to get the candidates. Once the candidate documents are found, finding the answer documents entails running Q on all candidates.

6 Experimental Results

In this section we provide performance results for obtaining candidate and answer documents for several XPath queries using a variety of SDs. The experiments demonstrate that DescribeX easily scales up to gigabyte sized XML collections with response times that are (for the most part) in the order of seconds.

Our experiments were conducted over three collections of documents. The first two collections (Wiki5 and Wiki45) were created from the Wikipedia XML Corpus provided in INEX 2006 [9] (using one tenth of the corpus and the entire corpus, respectively). The third collection (RSS2) was obtained by collecting RSS feeds from thousands of different sites. The size, number of documents, and p^* SD load (creation) times of our test collections are summarized in Table 1.

For measuring document selection times, five separate runs for each query were conducted starting with a cold Java Virtual Machine (JVM). The best and worst times were ignored and the reported runtime is the average of the remaining three times. The experiments were carried out on a Windows XP Virtual Machine running on a 2.4GHz dual Opteron server, and the JVM was allocated 1 GB of RAM.

Table 1. Test Collections

Collection	MB	#docs	p^* Load (sec)
RSS2	210	9600	215
Wiki5	545	30000	567
Wiki45	4500	659388	9700

Table 2. RSS and Wikipedia Queries

Query	XPath Expression
R1	/rss/channel[item[position()>1]]/item[title][enclosure] [not(pubDate=../item[1]/pubDate)]
R2	/rss/channel/image[width][height][title][description][link][url] [width/following-sibling::height][width < height]
R3	/rss/channel/item[comments][title][category][description][guid] [pubDate][link][source][category/following-sibling::category] [category="EuroAmerica"]
W1	/article/body/template/template[figure/caption][figure/image] [figure][collectionlink][contains(.,'billion')]
W2	/article/body/figure[image][caption][caption/collectionlink] [caption/outsidelink][caption/unknownlink] [image/following-sibling::caption][contains(.,'February 25')]
W3	/article/body/section/section/section/section[title][p] [title/following-sibling::figure/following-sibling::p] [p/collectionlink][p/unknownlink][contains(.,'Mac OS')]

Table 2 shows the six queries in our benchmark (the structural subqueries appear in black). These queries were selected to show the use of different SDs and how the system scales w.r.t. the number of documents selected. Our benchmark queries focus on the navigational features of XPath, following the approach of the MemBeR XQuery Micro-Benchmark [2] (which provide some form of standardization for studying different aspects of XML data management systems).

Table 3 shows the times for obtaining the candidate and answer documents for RRS2 (queries Rx) and Wiki45 (queries Wx). The **SD AxPRE** column contains the AxPRE of the SD node used to obtain the candidate documents. The **ED#** column reports the number of extent documents for each SD node. Columns **CD#** and **AD#** contain the number of candidate and answer documents respectively. The last row of each query corresponds to the most refined SD node for the query, which contains only candidate documents. For instance, for R1 two different refinements of the same SD node are used, the first one contains 6509 extent documents, and the second one 178. This last refinement contains only candidate documents. The times reported under column **CD(s)** correspond to selecting the candidate documents from the extent documents. This entails opening every extent document and evaluating the structural subquery. However, running the structural subquery is not necessary for the last row of each query (all extent documents are candidates), thus the reported times are just for retrieving the pointers to the documents. For instance, obtaining the candidate documents from query R1 took 45.3 s. using the p^* SD, and just 0.3 s. using a $p^*|c$ refinement. Finally, the times reported under the **AD(s)** column correspond to selecting the answer documents by evaluating the query on the candidate documents. For instance, selecting the 170 answer documents for R1

Table 3. Query Results and Times (RSS2 and Wiki45)

Query	SD AxPRE	ED#	CD#	AD#	CD(s)	AD(s)
R1	p^*	6509			45.3	
	$p^*\|c$	178	178	170	0.3	3.4
R2	p^*	3297			34.9	
	$p^*\|c^*$	386	352	8	4.7	4.1
	$p^*\|c^*\|c.fs$	352			0.2	
R3	p^*	6509			45.3	
	$p^*\|c^*$	9	3	1	0.8	0.3
	$p^*\|c^*\|c.fs$	3			0.1	
W1	p^*	82112			1332.0	
	$p^*\|c$	423	423	132	29.6	4.7
	$p^*\|c^*$	423			0.3	
W2	p^*	115575			1673.0	
	$p^*\|c^*$	18	18	2	2.2	0.3
	$p^*\|c^*\|c.fs$	18			0.2	
W3	p^*	736			23.2	
	$p^*\|c$	27	1	1	2.8	0.2
	$p^*\|c^*\|c.fs^*$	1			0.2	

from the 178 candidate documents took 3.4 s. It is easy to see from these results that the more precise (or refined) the SD node for a query, the smaller the extent document set and thus the faster DescribeX computes the candidates.

Comparison with summary proposals. The results in Table 3 also provide a comparison with the summary literature. Proposals like like 1-index [17], APEX [5], A(k)-index [13], and D(k)-index [23] can provide, at best, a description equivalent to the p^* SD and thus a similar performance to that reported on the first row of each query. The $p^*|c^*$ rows give an indication of the performance provided by the F+B-Index [12]. DescribeX can create SDs tailored to a workload that yield query evaluation times one to three orders of magnitude faster than these proposals (last row of each query). Using a precise SD can have a significant impact on both candidate and answer documents selection, and thus on overall query evaluation. Note that no summary in the literature (even recent proposals that cluster together nodes with the same subtree structure [3]) can capture AxPREs like $p^*|c^*|c.fs^*$.

Comparison with XPath evaluators. Table 4 reports the times for selecting answer documents using DescribeX, DB2 v9 [4], X-Hive/DB [5], XQuest DB [6], and Saxon (stand-alone, without summaries) on the RSS2 and Wiki5 collections. Comparative times for Wiki45 are not reported because neither XHive/DB nor

[4] http://www-306.ibm.com/software/data/db2/9/
[5] http://www.x-hive.com/products/db/
[6] http://www.axyana.com/xquest/

Table 4. Query Evaluation Comparative Times (RSS2 and Wiki5)

Query	DescribeX	DB2 v9	X-Hive	XQuest	Saxon
R1	3.7	58.1	8.7	(*)	95
R2	4.3	n/a	7.2	2.9	97
R3	0.4	n/a	8.0	0.9	92
W1	0.2	9.2	27.1	1.2(*)	345
W2	0.1	n/a	34.8	15.7	362
W3	0.1	n/a	37.4	2.5(*)	370

XQuest DB could load the entire collection. DB2 v9 does not support following-sibling or preceding-sibling XPath axes, so queries R2, R3, W2 and W3 could not be run on DB2. XQuest DB returned an incorrect answer for some of the queries, which are marked with an asterisk. DescribeX times span selecting the answer documents and evaluating the entire query using the most refined SD. These times are obtained by adding up the times for getting the candidate documents and the times for evaluating the entire query on them (using Saxon).

The comparative analysis uses two commercial systems, DB2 and X-Hive/DB, and an open source system, XQuest DB. X-Hive/DB and XQuest DB were selected because of their good performance in published XQuery benchmarks [1]. In addition, a comparison against Saxon stand-alone evaluation (without summaries) is provided. While DescribeX can invoke any XPath processing tool, Saxon was selected for being a popular processor that can also evaluate XQuery and XSLT in a file-at-a-time fashion. Keep in mind that the selected DB-like XML processors may have additional functionality (such as transaction processing capabilities). The comparison aims to show that the DescribeX architecture with the default implementation (combining summaries with Saxon) can achieve results competitive with that of XML indexing engines, even with gigabyte sized collections. In addition, comparing against Saxon provides a performance base line for a file-at-a-time evaluation when the collection is stored as XML text files in the file system and no summary structures are available. The results confirm that, without summaries, Saxon loses by several orders of magnitude.

7 Related Work

The large number of summaries that have been proposed in recent years clearly establishes the value and usefulness of these structures for describing semistructured data, assisting with query evaluation, helping to index XML data, and providing statistics useful in XML query optimization. A more exhaustive comparison with related work can be found in [8], including the specific AxPREs that can be used in DescribeX to express previously proposed summaries.

Most summary proposals in the literature define synopses of predefined subsets of paths in the data. Examples of such summaries are region inclusion graphs (RIGs) [7], representative objects (ROs)[18], dataguides [11], 1-index, 2-index and T-index [17], ToXin [24], A(k)-index [13], F+B-Index and F&B-Index [12].

A few *adaptive* summaries, like APEX [5] and D(k)-index [23], use dynamic query workloads to determine the subset of incoming paths to be summarized. APEX uses an ad-hoc construction mechanism to summarize paths that appear frequently in a query workload. The workload APEX considers are expressions containing a number of child axis composition that may be preceded by a descendant axis, without any predicate. However, APEX is tailored to incoming paths (i.e. SDs defined by the p^* AxPRE) and does not provide an explicit description of the extents, whereas DescribeX supports arbitrary AxPRE's. Regarding summaries that capture document order, the only proposals we are aware of are the earlier region order graphs (ROGs) [7] and the Skeleton summary [4,3]. Even though Skeleton uses an entirely different construction approach, its essence can be captured by the $(fc.ns^*)^*$ AxPRE.

Other summaries are augmented with *statistical information* of the instance for selectivity estimation, including path/branching distribution (XSketch [21]), value distributions [20], and additional statistical information for approximate query processing [22].

8 Conclusion and Future Work

The paper introduces DescribeX, a novel framework for describing structural summaries of XML collections. Summary partitions are defined by AxPRE's created from arbitrary XPath queries, supporting fast evaluation of complex XPath workloads over large web document collections.

Experimental results demonstrate that DescribeX's powerful mechanism for adapting summaries to a workload can provide speedups of one to three orders of magnitude compared to other proposals. The experiments also show that DescribeX's file-at-a-time XPath evaluation architecture can be a competitive alternative (in terms of query response times) to DB-like XML query engines, even on gigabyte sized collections.

Since this XPath-to-AxPRE syntactic translation can be applied to any XPath query, it can also be used to translate XPlainer queries [6] to AxPREs. XPlainer expressions have the same syntax as XPath but a different semantics which provide an explanation in the form of the *intermediate nodes*, a kind of data provenance of the answer. Future work includes creating AxPREs for the XPlainer expressions of a query, so that DescribeX can adapt SDs to accelerate the retrieval of intermediate nodes. In addition, we plan to study the impact of adjusting the workload (e.g, by finding frequent patterns), and also how to optimize SD selection given budget constraints.

References

1. Afanasiev, L., Franceschet, M., Marx, M.: XCheck: a platform for benchmarking XQuery engines. In: VLDB, pp. 1247–1250 (2006)
2. Afanasiev, L., Manolescu, I., Michiels, P.: MemBeR: A micro-benchmark repository for XQuery. In: XSym, pp. 144–161 (2005), `http://ilps.science.uva.nl/Resources/MemBeR/`

3. Buneman, P., Choi, B., Fan, W., Hutchison, R., Mann, R., Viglas, S.: Vectorizing and querying large XML repositories. In: ICDE, pp. 261–272 (2005)
4. Buneman, P., Grohe, M., Koch, C.: Path queries on compressed XML. In: VLDB, pp. 141–152 (2003)
5. Chung, C.-W., Min, J.-K., Shim, K.: APEX: An adaptive path index for XML data. In: SIGMOD, pp. 121–132 (2002)
6. Consens, M.P., Liu, J.W., Rizzolo, F.: XPlainer: Visual explanations of XPath queries. In: ICDE (2007)
7. Consens, M.P., Milo, T.: Optimizing queries on files. In: SIGMOD, pp. 301–312 (1994)
8. Consens, M.P., Rizzolo, F., Vaisman, A.A.: Exploring the (semi-)structure of XML web collections. Technical report, University of Toronto - DCS (2007), `http://www.cs.toronto.edu/~consens/describex/`
9. Denoyer, L., Gallinari, P.: The Wikipedia XML Corpus. SIGIR Forum (2006)
10. Dovier, A., Piazza, C., Policriti, A.: An efficient algorithm for computing bisimulation equivalence. Theoretical Computer Science 311(1-3), 221–256 (2004)
11. Goldman, R., Widom, J.: Dataguides: Enabling query formulation and optimization in semistructured databases. In: VLDB, pp. 436–445 (1997)
12. Kaushik, R., Bohannon, P., Naughton, J.F., Korth, H.F.: Covering indexes for branching path queries. In: SIGMOD, pp. 133–144 (2002)
13. Kaushik, R., Shenoy, P., Bohannon, P., Gudes, E.: Exploiting local similarity for indexing paths in graph-structured data. In: ICDE, pp. 129–140 (2002)
14. Martens, W., Neven, F., Schwentick, T.: Complexity of decision problems for simple regular expressions. In: 29th International Symposium on Mathematical Foundations of Computer Science, MFCS, pp. 889–900 (2004)
15. Marx, M.: XPath with conditional axis relations. In: EDBT, pp. 477–494 (2004)
16. Mendelzon, A.O., Wood, P.T.: Finding regular simple paths in graph databases. SIAM Journal on Computing 24(6), 1235–1258 (1995)
17. Milo, T., Suciu, D.: Index structures for path expressions. In: ICDT, pp. 277–295 (1999)
18. Nestorov, S., Ullman, J.D., Wiener, J.L., Chawathe, S.S.: Representative objects: Concise representations of semistructured, hierarchial data. In: ICDE, pp. 79–90 (1997)
19. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM Journal on Computing 16(6), 973–989 (1987)
20. Polyzotis, N., Garofalakis, M.N.: XCLUSTER synopses for structured XML content. In: ICDE (2006)
21. Polyzotis, N., Garofalakis, M.N.: XSKETCH synopses for XML data graphs. ACM Transactions on Database Systems (TODS) 31(3), 1014–1063 (2006)
22. Polyzotis, N., Garofalakis, M.N., Ioannidis, Y.E.: Approximate XML query answers. In: SIGMOD, pp. 263–274 (2004)
23. Qun, C., Lim, A., Ong, K.W.: D(k)-index: An adaptive structural summary for graph-structured data. In: SIGMOD, pp. 134–144 (2003)
24. Rizzolo, F., Mendelzon, A.O.: Indexing XML data with ToXin. In: WebDB, pp. 49–54 (2001)

Let a Single FLWOR Bloom
(To Improve XQuery Plan Generation)*

Matthias Brantner, Carl-Christian Kanne, and Guido Moerkotte

University of Mannheim
{brantner,kanne,moerkotte}@informatik.uni-mannheim.de

Abstract. To globally optimize execution plans for XQuery expressions, a plan generator must generate and compare plan alternatives. In proven compiler architectures, the unit of plan generation is the query block. Fewer query blocks mean a larger search space for the plan generator and lead to a generally higher quality of the execution plans. The goal of this paper is to provide a toolkit for developers of XQuery evaluators to transform XQuery expressions into expressions with as few query blocks as possible.

Our toolkit takes the form of rewrite rules merging the inner and outer FLWOR expressions into single FLWORs. We focus on previously unpublished rewrite rules and on inner FLWORs occurring in the `for`, `let`, and `return` clauses in the outer FLWOR.

1 Introduction

XQuery evaluators become more and more mature in terms of features and performance, and XQuery is being integrated into mainstream DBMS products as a native language. However, XQuery processing research is still missing some fundamental tools to facilitate the development of industrial-strength XQuery optimizers. The goal of this paper is to fill one of these gaps: We provide a rewrite toolkit that allows to reduce the number of query blocks in a query expression. This widens the search space for plan generators by making more information visible to a single run of the plan generation algorithm. Let us begin by stressing the importance of our goal:

Industrial-strength query optimizers proceed in a two-phase manner. In a first phase, the query is translated into an internal representation, and heuristical rewrite rules are applied to simplify and normalize the query. In a second phase, a plan generator enumerates alternative execution plans, determines their costs, and chooses the optimal plan. Alternative plans can differ in the access paths used for the basic input sets (e.g. whether to use an index or not), in the order in which the basic input sets are joined, and in the position of other operators, such as grouping or sorting.

However, efficient plan generation algorithms cannot take arbitrary query structures as input. Instead, the unit of plan generation is the *query block*. Depending on the design of the query compiler, a query block can be represented in a variety of ways, for example as a source language construct (SELECT FROM WHERE in SQL, or FLWOR in XQuery), as a node in an internal graph representation (such as the Query Graph

* This work was supported by the Deutsche Forschungsgemeinschaft under grant MO 507/10-1.

D. Barbosa et al. (Eds.): XSym 2007, LNCS 4704, pp. 46–61, 2007.

Model QGM [17]), or as an algebraic expression. Some queries exhibit a nested structure, where a query block references subquery blocks. In such cases, the plan generator is called in a bottom-up fashion, generating plans for all subquery blocks before the surrounding query block is processed. It is easy to see that in such cases, the search space examined by the plan generator is limited, because only locally good solutions are computed. For globally optimal plans, it is desirable to reduce the number of query blocks to have more information available in a single run of the plan generator, creating a larger search space of alternative plans. For this reason, in the first phase of optimization, queries are rewritten by merging as many query blocks as possible. This is state-of-the-art for SQL query processing (e.g. [4,10,18]), but not highly developed for XQuery.

For an industrial-strength approach to XQuery optimization, such a rewriting step to merge query blocks is particularly necessary:

- In XQuery expressions in real applications, a nested query structure is the norm rather than an exception. This is due to a number of reasons, including the construction of hierarchical XML results, the absence of a grouping construct, the generation of queries using visual editors, and, last but not least, the inlining of (non-recursive) XQuery functions that contain FLWOR expressions.
- XML query processing can benefit from holistic n-way joins [3] which perform single-pass tree-pattern matching instead of constructing results just using binary joins. The detection of tree patterns and the decision when to use regular joins and when to use pattern matching is a global decision during plan generation that requires access to as much of the query as possible.

An example for a highly nested query (inspired by XMark Query 3) is shown here:

```
let $auction := doc("auction.xml") return
  let $euro:=for $o in $auction/site/open_auctions/open_auction
             for $i in $auction/site/regions/europe/item/@id
             where $o/itemref/@item eq $i
             return $o
  for $a in $euro
  where zero-or-one($a/bidder[1]/increase/text()) * 2
        <= $a/bidder[last()]/increase/text()
  return
    for $p in $auction/site/people/person[profile/@income > 5000]
    for $w in $p/watches/watch
    where $a/@id = $w/@open_auction
    return <auction id="{$a/@id}">
             <increase first="{$a/bidder[1]/increase/text()}"
                       last="{$a/bidder[last()]/increase/text()}"/>
               <watched_by id="{$p/@id}"/>
           </auction>
```

The query body is constructed of four FLWOR expressions, three of which are nested inside other FLWORs. However, these are only the explicit FLWOR blocks. Depending on the compiler design, the number of nested query blocks may be even deeper. For example, with a plan generator that focuses on purely structural tree-pattern matching, nested value-based predicates such as `profile/@income > 5000` may be separate query blocks.

Without further processing, such a query is optimized using several runs of the plan generation algorithm, where each plan for a FLWOR expression is used in the plan for

the surrounding FLWOR. This separate optimization of subqueries impedes the discovery of good overall execution plans. This is demonstrated by our example, in which there are two value-based joins, one joining the Open Auctions to the European Items, and one joining the Open Auctions to the Persons with an income higher than 5000. However, the join conditions in the `where` clauses are in different FLWORs, prohibiting the plan generator to see both of the joins and optimize their order. Join order optimization is a cornerstone of efficient relational query processing and just as important in XQuery processing [6].

As in many other cases, the nested structure of the query is not required to obtain the query result, but is used because this way, the query is simpler to write. In fact, the whole query above can be formulated using a single FLWOR block. One alternative to do so is shown below, with the results of each processing step bound to a separate variable:

```
let $auction := doc("auction.xml"), $x32 := $auction/site
for $o in $x32, $x13 in $o/open_auctions, $a in $x13/open_auction
for $i in $x32, $x15 in $i/regions, $x16 in $x15/europe
for $x17 in $x16/item, $x18 in $x17/@id
let $x4 := $a/itemref, $x19 := $x4/@item
let $x33 := $a/bidder[1], $x34 := $x33/increase, $x35 := $x34/text()
let $x36 := $a/bidder[last()], $x37 := $x36/increase, $x38 := $x37/text()
let $x39 := $a/@id
for $p in $x32, $x20 in $p/people, $x21 in $x20/person
for $w in $x21/watches, $x22 in $w/watch
let $x8 := $x21/@id, $x10 := $x22/@open_auction
let $x13 := $x21/profile, $x27 := $x13/@income
for $x1 in  <auction id="{$x39}">
              <increase first="{$x35}" last="{$x38}"/>
              <watched_by id="{$x8}"/>
            </auction>
where zero-or-one($x35) * 2 <= $x38 and $x39 = $x10
      and $x27 > 5000 and $x19 eq $x18
return $x1
```

While this form of the query is less readable and more difficult to write, it is easier to optimize because all the basic operations, intermediate results, input sets, and their dependencies are uniformly represented in a single, top-level FLWOR construct.

The goal of this paper is to provide a toolkit for developers of XQuery evaluators to transform XQuery expressions into expressions with as few query blocks as possible. This toolkit takes the form of rewrite rules merging the inner and outer FLWOR expressions into single FLWORs. These unnesting rules are supplemented by some helpful normalization rewrites. We have chosen to present our rules using regular XQuery syntax because other representations (such as QGM or algebraic expressions) are less universal and would be more difficult to adapt to different evaluators. We do not use the XQuery Core sublanguage because it does not have a query block construct suitable for plan generation. It is, instead, inherently nested, even for quite simple XQuery expressions. Due to space constraints, we limit our presentation to previously unpublished rewrite rules and to inner FLWORs occurring in the `for`, `let`, and `return` clauses in the outer FLWOR. More rewrite rules, including rules for `order by` clauses and positional variables, can be found in the extended version of this paper [2].

The remainder of this paper is structured as follows: We begin with a discussion of related work in Section 2 and give an overview of the rewrite toolkit in Section 3. The main Sections 4 and 5 present normalization and FLWOR merging rules, respectively.

We finish with a short evaluation (see Sec. 6), demonstrating the effect of our rules when generating execution plans.

2 Related Work

Michiels et al. [16] discuss rewrite rules on two levels. Starting from expressions in XQuery Core, they propose to first rewrite them into normal forms (still in XQuery Core) that make the subsequent stages robust against different syntactic formulations of the same query, and to support tree-pattern detection. They also simplify the query by removing unnecessary constructs introduced by Core normalization. Some of these simplification rewrites could be incorporated into our toolkit. The rewritten query is then translated into an algebra that includes a tree-pattern matching operator. These algebraic expressions are then rewritten using algebraic equivalences in order to merge simple path-navigation operators into holistic tree-pattern matching operators. The rewrite rules on the algebraic level are orthogonal to the ones presented in our toolkit and can be used by a plan generator to create execution plans based on tree-pattern matching.

The very thorough paper by Hidders et al. [5] has a similar aim, but directly translates a fragment of XQuery into tree patterns without an intermediate algebraic phase. In a first phase, the queries are annotated with properties such as result cardinality, ordering, and occurrence of duplicates. These properties are then used to control a rewriting of the query into the Tree Pattern Normal Form (TPNF), which is always possible for the language fragment under consideration. For TPNF, a direct mapping onto tree patterns is then described. Unfortunately, the language fragment does not cover important XQuery constructs, such as value-based predicates. Another problem is that the rewrite rules are based on XQuery Core, which is unsuitable as a plan generator input, for example because the absence of a `where` clause makes it difficult to identify applicable join conditions. However, the property annotations are not only useful for TPNF rewriting and can be used when implementing our rewrite toolkit. Further, the TPNF technique may be used by plan generators to identify parts of the query that can be evaluated using pattern matching.

May et al. [15] have presented unnesting strategies for XQuery. Their approach is based on algebraic equivalences to be applied after translation of XQuery into the NAL algebra of the Natix system. The main focus of that work is unnesting of selection predicates which correspond to `where` clauses on the source level. The paper also discusses unnesting the subscripts of map operators, which on source level corresponds to `let` clauses. However, the rules are exclusively for the conversion of implicit grouping into explicit grouping operators, and not for the general unnesting of `let`. Translated into the source form, the presented rewrite rules are complementary to the rules discussed in this paper.

3 Overview

The overall goal of this paper is to flatten an XQuery expression, i.e. merge as many query blocks (i.e. FLWOR expressions) as possible. To achieve this goal, we basically

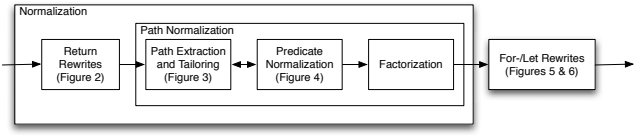


Fig. 1. Processing model

proceed in two phases: (1) Normalization and (2) FLWOR Merging. Fig. 1 gives an overview of our processing model.

In both phases, we apply a set of rules based on XQuery syntax on a query. A separate figure presents one set of rules for each normalization and rewriting step. The Overview Figure 1 contains references to each of them.

Normalization. Comprises two major subtasks:

1. All `ExprSingle` expressions[1] from the `return` clause are moved to the expression creating the binding sequence of a new `for` expression.
2. Path expressions are normalized (as far as possible). In particular, (1) all path expressions not directly associated with a `for` clause are bound to variables using `let`, (2) path expressions are taken to single steps, (3) predicates are moved into the `where` clause, and (4) common location steps are factorized.

FLWOR Merging. Starting from this normalized form, we remove as many query blocks (FLWOR expressions) as possible. Specifically, we present rewrite rules that eliminate or merge inner FLWORs occurring in the `for` or `let` clause, respectively.

Notation. Our rewrite rules are formulated using XQuery syntax [8]. However, to simplify the presentation, we use the following abbreviations for frequently used clauses:

```
ForOrLetClause  :=  ForClause | LetClause
ForOrLetClauses :=  ForOrLetClause*
```

Moreover, we assume that all variable names are unambiguous. Since we sometimes introduce new variables or change the bindings of existing ones, we introduce a notation for variable substitution: `Expr`$[\$x2 \leftarrow \$x1]$ denotes `Expr` with all free occurrences of $\$x2$ replaced by $\$x1$.

Running Example. We illustrate the application of our rules on the query from the introduction. Applying our rules to the example query yields a query which has a single FLWOR block.

In practice, this query could well be the result of an inlined XQuery function. XQuery functions are often used as *views* to increase data independence, or simply to make queries more readable, similar to views in SQL. In our case, the sequence bound to `$euro` could be an inlined function to retrieve European Auctions, whereas the bottommost FLWOR expression could be a function to retrieve all watchers for a given

[1] Note that `ExprSingle` is the expression produced by the grammar rules from [8].

auction. The results of these functions are joined using the surrounding FLWOR block. In such a context, the application of our rewrite rules can also be described as *view merging*, allowing the plan generator to optimize join orders beyond view borders.

4 Normalization

Normalization does not decrease the FLWOR nesting level of a query. Instead, it transforms the query such that the unnesting rewrite rules can still be applied in case of minor syntactical variations. In addition to this preparatory character, normalization also directly helps to achieve our ultimate goal of preparing queries for plan generation: XQuery allows several different ways of formulating predicates (e.g. the `where` clause and XPath predicates). However, the plan generator requires a single unified formulation of all the constraints on all the variables in the currently considered query block to systematically explore the search space of alternative plans. Execution plan alternatives for value-based predicates include, but are not limited to, the placement of selection operators, the use of joins, and index selection. Which of these alternatives is used, and in which order the different predicates of a query are evaluated, should not depend on the nesting level or the placement of the predicates. This robustness is achieved by our normalization phase.

Normalization proceeds in several consecutive steps, as shown in in Fig. 1. We first enforce a simple form for all `return` clauses before we break down complex location paths into primitives, with an emphasis on predicate normalization. Finally, we eliminate common subexpressions.

4.1 Return Normalization

In order to allow a uniform treatment of nested expressions in `return` and `let` clauses, we move all `ExprSingle` expressions from `return` clauses to `let` clauses (see Rewrite 1). This way, we can treat the unnesting of `return` and `let` uniformly and can always assume a `return` clause that consists of a single variable reference.

Other than normalizing the `return` clause, we can further prepare optimization by converting the new `let` clause into a `for` clause (see Rewrite 2). This is possible

	ForOrLetClauses WhereClause? OrderByClause? return $\texttt{ExprSingle}_1$	→ ForOrLetClauses let $\$x1 := \texttt{ExprSingle}_1$ WhereClause? OrderByClause? return $\$x1$	(1)
	$\texttt{ForOrLetClauses}_1$ let $\$x1 := \texttt{ExprSingle}_1$ $\texttt{ForOrLetClauses}_2$ WhereClause? OrderByClause? return $\$x1$	→ $\texttt{ForOrLetClauses}_1$ for $\$x1$ in $\texttt{ExprSingle}_1$ $\texttt{ForOrLetClauses}_2$ WhereClause? OrderByClause? return $\$x1$	(2)
Condition: There are no other occurrences of $\$x1$.			

Fig. 2. Return rewrites

because on the right-hand side, the concatenation semantics of FLWOR blocks reestablishes the same result sequence as on the left-hand side of the rewrite, as long as $x1 is used nowhere but in the `return` clause.

Turning `let` into `for` expressions allows a significantly larger range of alternatives for plan generation. Evaluation of `for` clauses can be done in an iterative manner, generating the items of the binding sequence one by one, instead of computing and materializing the whole sequence at once. This allows efficient techniques such as pipelining and is the preferred style of implementation in database runtime engines [11].

Running Example. Applying the `return` elimination and `let` transformation rewrites (1 and 2) to the `return` expressions of our example query results in the following:

```
let $auction := doc("auction.xml")
for $x1 in let $euro:= for $o in $auction/site/open_auctions/open_auction
                       for $i in $auction/site/regions/europe/item/@id
                       where $o/itemref/@item eq $i
                       return $o
           for $a in $euro
           for $x2 in for $p in $auction/site/people/person[profile/@income > 5000]
                      for $w in $p/watches/watch
                      for $x3 in <auction id="{$a/@id}">
                                   <increase first="{$a/bidder[1]/increase/text()}"
                                             last="{$a/bidder[last()]/increase/text()}"/>
                                     <watched_by id="{$p/@id}"/>
                                   </auction>
                      where $a/@id = $w/@open_auction
                      return $x3
           where zero-or-one($a/bidder[1]/increase/text()) * 2
                 <= $a/bidder[last()]/increase/text()
           return $x2
return $x1
```

4.2 Path Normalization

Path expressions are a crucial performance factor for the evaluation of almost every XQuery query. For efficiently evaluating path expressions, the plan generator makes cost-based decisions on algorithms that should be used to evaluate them. For example, an optimizer decides whether a holistic approach (e.g. [3,14]) for evaluating multiple path expressions is superior to a fine granular approach that evaluates single steps individually (e.g. [7,12]), probably with the help of an index. The plan generator requires a canonical form of the path expressions to make such decisions. Besides separating each processing step for plan generation, cutting path expressions involves two other advantages:

- It allows to move location step predicates from the middle of location paths into the `where` clause.
- Common subexpression elimination (see below) can be done on the granularity of steps.

Path Tailoring. In order to separate each processing step, we first extract all path expressions from the query which are not already binding expressions of `for` or `let`,

`for` $x `in` `StepExpr/PathExpr`	→	`for` $x1 `in` `StepExpr` `for` $x2 `in` $x1`/PathExpr` (3)
Condition: `StepExpr` must not produce duplicates.		
`for` $x `in` `StepExpr/PathExpr`	→	`let` $x1 := `StepExpr` `for` $x2 `in` $x1`/PathExpr` (4)
`let` $x := `StepExpr/PathExpr`	→	`let` $x1 := `StepExpr` `let` $x2 := $x1`/PathExpr` (5)

Fig. 3. Path tailoring rewrites

and bind them to new `let` variables. We keep path expressions in `for` clauses because they need a different treatment in our predicate rewrites below.

Having extracted all path expressions, we cut them up into single location steps (see Fig. 3 for rewriting rules). Again to facilitate iterator-based evaluation, we attempt to avoid `let` clauses when possible (3 and 4) while breaking up path expressions in `for` clauses. Without further refinements, we can only cut those steps that do not produce duplicates (see [5,13]). Of course, location steps assigned to a `let` variable remain in a `let` binding (5).

Predicate Normalization. The plan generator not only decides on the path evaluation algorithms and the order of joins based on structural predicates, but also on the order of regular, value-based joins and selections. Moving all non-structural predicates into the `where` clause makes such join and selection predicates explicitly available in a uniform manner. This allows a search space of plans that is robust against the syntactical placement of the predicate. Further, a unified `where` also allows predicate processing, which includes, but is not limited to, the inference of new predicates and the elimination of redundant ones.

In Fig. 4, we present rules that get predicate expressions of location steps and move them into the `where` clause of the surrounding FLWOR block. For each extracted predicate expression, we have to set the context to the context defined by the according step. For example, if we move $\texttt{Expr}_1$ from a location step predicate into a `where` clause (see Rule 6), we have to guarantee that all context accesses are performed with respect to $x1, which is why we prepend $x1 to the predicate expression. Similarly, we can get comparison expressions that contain calls to the context position of a location step by creating a positional variable using the `for VarRef at VarRef` syntax and replacing accesses to the context position with the variable (see Rule 9). This is not strictly possible in XQuery syntax, but easily implemented in most evaluators because the context position is modeled as a special variable anyway. Our choice of variable name ($fs : position) follows the XQuery Formal Semantics, which also replaces context position by a special variable. Further, reverse axis steps cannot be handled this way, because the context position numbering is different from the order of the result sequence[2].

Note that for the sake of brevity, we assume that there is always a `where` clause in the outer expression. We treat outer FLWORs without a `where` clause as if there was a `where true` clause.

[2] If the rewrite is not done on source level, the internal representation may have a suitable special variable to bind for reverse axis numbering, making our rewrite possible again.

$$\begin{array}{l}\texttt{ForOrLetClauses}_1\\ \texttt{for \$x1 in StepExpr}[\texttt{Expr}_1]\\ \texttt{ForOrLetClauses}_2\\ \texttt{where Expr}_2\\ \texttt{OrderByClause?}\\ \texttt{return ExprSingle}_1\end{array} \rightarrow \begin{array}{l}\texttt{ForOrLetClauses}_1\\ \texttt{for \$x1 in StepExpr}\\ \texttt{ForOrLetClauses}_2\\ \texttt{where fn : boolean}(\$x1/(\texttt{Expr}_1))\ \texttt{and Expr}_2\\ \texttt{OrderByClause?}\\ \texttt{return ExprSingle}_1\end{array} \quad (6)$$

Condition: The value of $\texttt{Expr}_1$ must not depend on the context position or context size.

$$\begin{array}{l}\texttt{ForOrLetClauses}_1\\ \texttt{let \$x1 := StepExpr}[\texttt{Expr}_1]\\ \texttt{ForOrLetClauses}_2\\ \texttt{where Expr}_2\\ \texttt{OrderByClause?}\\ \texttt{return ExprSingle}_1\end{array} \rightarrow \begin{array}{l}\texttt{ForOrLetClauses}_1\\ \texttt{let \$x1 := StepExpr}\\ \texttt{ForOrLetClauses}_2\\ \texttt{where fn : boolean}(\texttt{Expr}_1)\ \texttt{and Expr}_2\\ \texttt{OrderByClause?}\\ \texttt{return ExprSingle}_1\end{array} \quad (7)$$

Condition: The value of $\texttt{Expr}_1$ must not depend on the focus (context item, context position, or context size).

$$\begin{array}{l}\texttt{ForOrLetClauses}_1\\ \texttt{for \$x1 in StepExpr}[\texttt{Expr}_1\ \texttt{and Expr}_2]\\ \texttt{ForOrLetClauses}_2\\ \texttt{where Expr}_3\\ \texttt{OrderByClause?}\\ \texttt{return ExprSingle}_1\end{array} \rightarrow \begin{array}{l}\texttt{ForOrLetClauses}_1\\ \texttt{for \$x1 in StepExpr}[\texttt{Expr}_2]\\ \texttt{ForOrLetClauses}_2\\ \texttt{where fn : boolean}(\$x1/(\texttt{Expr}_1))\ \texttt{and Expr}_3\\ \texttt{OrderByClause?}\\ \texttt{return ExprSingle}_1\end{array} \quad (8)$$

Condition: The value of $\texttt{Expr}_1$ must not depend on the context position or context size.

$$\begin{array}{l}\texttt{ForOrLetClauses}_1\\ \texttt{for \$x1 in StepExpr}[\texttt{Expr}_1\ \texttt{and Expr}_2]\\ \texttt{ForOrLetClauses}_2\\ \texttt{where Expr}_3\\ \texttt{OrderByClause?}\\ \texttt{return ExprSingle}_1\end{array} \rightarrow \begin{array}{l}\texttt{ForOrLetClauses}_1\\ \texttt{for \$x1 at \$y1 in StepExpr}[\texttt{Expr}_2]\\ \texttt{ForOrLetClauses}_2\\ \texttt{where Expr}'_1\ \texttt{and Expr}_3\\ \texttt{OrderByClause?}\\ \texttt{return ExprSingle}_1\end{array} \quad (9)$$

Conditions: The value of $\texttt{Expr}_1$ depends on the context position, but not the context size. $\texttt{Expr}'_1 := \texttt{Expr}_1[\$\text{fs} : \text{position} \leftarrow \$y1]$ and StepExpr must not consist of a reverse axis step (see text).

Fig. 4. Predicate normalization rewrites

Common Path Elimination. To avoid redundant evaluation, we eliminate common paths, binding them to new `for` or `let` variables as needed. For space reasons, we do not present rules for eliminating common paths here but refer to our technical report [2]. Moreover, we refer to [1] for algorithms on subexpression elimination.

Running Example. In the following, we present the query that is obtained by applying normalization, i.e. path extraction, path tailoring, predicate normalization, and common path elimination, to our example query.

```
let $auction := doc("auction.xml")
let $x32 := $auction/site
for $x1 in let $euro:= for $o in $x32, $x13 in $o/open_auctions
                       for $x14 in $x13/open_auction, $i in $x32, $x15 in $i/regions
                       for $x16 in $x15/europe, $x17 in $x16/item, $x18 in $x17/@id
                       let $x4 := $x14/itemref, $x19 := $x4/@item
                       where $x19 eq $x18
                       return $x14
          for $a in $euro
          let $x33 := $a/bidder[1], $x34 := $x33/increase. $x35 := $x34/text()
          let $x36 := $a/bidder[last()], $x37 := $x36/increase, $x38 := $x37/text()
          let $x39 := $a/@id
          for $x2 in for $p in $x32, $x20 in $p/people, $x21 in $x20/person
                     for $w in $x21/watches, $x22 in $w/watch
                     let $x8 := $x21/@id
```

```
                    let $x10 := $x22/@open_auction
                    let $x13 := $x21/profile, $x27 := $x13/@income
                    for $x3 in <auction id="{$x39}">
                                 <increase first="{$x35}" last="{$x38}"/>
                                 <watched_by id="{$x8}"/>
                               </auction>
                    where $x39 = $x10 and  $x27 > 5000
                    return $x3
          where zero-or-one($x35) * 2 <= $x38
          return $x2
return $x1
```

In this expression, for example, the XPath predicate `profile/@income > 5000` is removed from the location step and added to the `where` clause of the according FLWOR block. Moreover, we replaced the common path expressions from within the element construction and the `where` clauses (e.g. the path selecting the increases of the first and last bid) by single variables. Note that it is not possible to move the positional predicates into the `where` clause, as they occur in a `let` binding. Also note that for presentation purposes, we abbreviated consecutive occurrences of `for` and `let` expressions using commas. In the full representation of this query, `for` and `let` expressions that bind multiple variables are split into separate expressions.

5 Merging FLWOR Blocks

After finishing the normalization phase, the query is prepared for the core rules of our toolkit, the `for` and `let` merging rewrites. The ultimate goal of the rewrites presented in this section is to reduce the number of query blocks as much as possible.

Reconsider our normalized example query shown above. This formulation of the query contains several nested FLWOR expressions. The FLWOR nesting depth in line 3 is three. The `for`-clause binding $o is nested in a `let` clause which, in turn, is nested in the outer most `for`-clause binding $x1. Moreover, the query contains a `for` clause defining $x2 whose binding sequence is generated by another `for` clause.

In the following, we introduce rewrite rules that remove such nested expressions. Applying them to our example query eliminates all nested FLWORs.

We start with rewrites that remove FLWORs nested in `for` clauses (see Fig. 5) and then proceed to `let` clauses (see Fig. 6).

5.1 For Rewrites

The semantics of a `for` clause is to iterate over items of the binding sequence, binding the `for` variable to every item in this sequence. The remaining FLWOR expression is evaluated for each such binding, and the individual result sequences are concatenated. We are interested in a `for` clause if its binding sequence is created by a nested FLWOR expression. In some cases, we can lift the inner FLWOR to the outer level. This rewrite opportunity results from the fact that sequences in the XQuery data model are never nested. Hence, it often does not matter on how many levels an implicit concatenation of `return` sequences occurs because the result is always a flat sequence.

```
ForOrLetClauses1
for $x1 in (ForOrLetClauses2                ForOrLetClauses1
            for $x2 in ExprSingle1          ForOrLetClauses2
            ForOrLetClauses3                for $x1 in ExprSingle1
            where ExprSingle2        →      ForOrLetClauses'3                    (10)
            return $x2)                     ForOrLetClauses4
ForOrLetClauses4                            where ExprSingle3 and ExprSingle'2
where ExprSingle3                           return VarRef1
return VarRef1
```

Conditions: $\texttt{ForOrLetClauses}'_3 := \texttt{ForOrLetClauses}_3[\$x2 \leftarrow \$x1]$ and $\texttt{ExprSingle}'_2 := \texttt{ExprSingle}_2[\$x2 \leftarrow \$x1]$

```
ForOrLetClauses1
for $x1 in (ForOrLetClauses2                ForOrLetClauses1
            let $x2 := ExprSingle1          ForOrLetClauses2
            ForOrLetClauses3                let $x2 := ExprSingle1
            where ExprSingle2        →      ForOrLetClauses3                     (11)
            return $x2)                     for $x1 in $x2
ForOrLetClauses4                            ForOrLetClauses4
where ExprSingle3                           where ExprSingle3 and ExprSingle2
return VarRef1                              return VarRef1
```

Fig. 5. For rewrites

For example, consider the left-hand side of the first `for` Rewrite 10. In this rewrite, the variable $x1 is iteratively bound to each item returned by the inner FLWOR. The result of the inner FLWOR is generated by the `return` clause. Note that in our case, the `return` clause consists only of a variable reference, i.e. variable $x2. To merge the two blocks, we have to guarantee that the outer `for` variable $x1, after merging, is still bound to the same items, i.e. those generated by variable $x2. To this end, we replace the nested FLWOR with the expression responsible for binding $x2. In the rewrite, this expression is called $\texttt{ExprSingle}_1$ and bound by a `for` clause. The remaining (optional) clauses are moved into the outer FLWOR block. Specifically, $\texttt{ForOrLetClauses}_2$ and $\texttt{ForOrLetClauses}_3$ are pulled up one level. $\texttt{ExprSingle}_2$ from the inner `where` clause is conjunctively connected to the expression in the outer `where` clause [3]. After relocating the inner expressions, we have to replace free occurrences of the previous inner variable $x2 with $x1.

Similarly, we merge two query blocks if the binding sequence is created by a nested `let` variable (see our Rewrite Rule 11). Note that the right-hand side of Rule 11 may still contain a FLWOR nested in a `let` clause. This case is unnested by Rule 12, which is presented in the next section.

Other rules for FLWORs nested within `for` clauses are discussed in the extended version of this paper [2], including cases with positional variables and `order by` clauses.

Running Example. On our example query, we can apply Rewrite Rule 10 twice. First, to eliminate the inner `for`-clause binding $x2, as this variable is returned to create the binding sequence for $x1. Second, we apply this rule to eliminate the `for` expression binding $x3. This results in the following expression:

[3] As before, expressions without `where` are treated as if a `where true` clause was added.

```
let $auction := doc("auction.xml")
let $x32 := $auction/site
let $euro:= for $o in $x32, $x13 in $o/open_auctions, $x14 in $x13/open_auction
            for $i in $x32, $x15 in $i/regions, $x16 in $x15/europe
            for $x17 in $x16/item, $x18 in $x17/@id
            let $x4 := $x14/itemref, $x19 := $x4/@item
             where $x19 eq $x18
             return $x14
for $a in $euro
let $x33 := $a/bidder[1], $x34 := $x33/increase, $x35 := $x34/text()
let $x36 := $a/bidder[last()], $x37 := $x36/increase, $x38 := $x37/text()
let $x39 := $a/@id
for $p in $x32, $x20 in $p/people, $x21 in $x20/person
for $w in $x21/watches, $x22 in $w/watch
let $x8 := $x21/@id, $x10 := $x22/@open_auction
let $x13 := $x21/profile, $x27 := $x13/@income
for $x1 in <auction id="{$x39}">
            <increase first="{$x35}" last="{$x38}"/>
            <watched_by id="{$x8}"/>
           </auction>
where zero-or-one($x35) * 2 <= $x38 and $x39 = $x10 and $x27 > 5000
return $x1
```

5.2 Let Rewrites

`let` clauses require separate rewrites because they bind a variable to the result of its associated expression, i.e. without iterating over this result. Fig. 6 presents three rewrite rules to eliminate FLWORs nested in `let` clauses.

Rewrite Rule 12 tackles a frequently used case. There, a `for` iteration is used to enumerate all items contained in a `let` variable. This technique is used in our example query and may, for example, result from inlining an XQuery function as explained at the beginning of this section. The rules suggest to eliminate the `let` variable if it is used

$$\begin{array}{l} \texttt{ForOrLetClauses}_1 \\ \texttt{let \$x1} := \texttt{ExprSingle}_1 \\ \texttt{ForOrLetClauses}_2 \\ \texttt{for \$x2 in \$x1} \\ \texttt{ForOrLetClauses}_3 \\ \texttt{where ExprSingle}_2 \\ \texttt{return VarRef} \end{array} \rightarrow \begin{array}{l} \texttt{ForOrLetClauses}_1 \\ \texttt{ForOrLetClauses}_2 \\ \texttt{for \$x2 in ExprSingle}_1 \\ \texttt{ForOrLetClauses}_3 \\ \texttt{where ExprSingle}_2 \\ \texttt{return VarRef} \end{array} \quad (12)$$

Condition: There are no other occurrences of $x1.

$$\begin{array}{l} \texttt{ForOrLetClauses}_1 \\ \texttt{let \$x1} := (\texttt{ForOrLetClauses}_2 \\ \quad \texttt{for \$x2 in ExprSingle}_1 \\ \quad \texttt{where ExprSingle}_2 \\ \quad \texttt{return \$x2}) \\ \texttt{where ExprSingle}_3 \\ \texttt{return \$x1} \end{array} \rightarrow \begin{array}{l} \texttt{ForOrLetClauses}_1 \\ \texttt{ForOrLetClauses}_2 \\ \texttt{for \$x2 in ExprSingle}_1 \\ \texttt{where ExprSingle}_3 \texttt{ and ExprSingle}_2 \\ \texttt{return \$x2} \end{array} \quad (13)$$

Condition: There are no other occurrences of $x1.

$$\begin{array}{l} \texttt{ForOrLetClauses}_1 \\ \texttt{let \$x1} := (\texttt{ForOrLetClauses}_2 \\ \quad \texttt{let \$x2} := \texttt{ExprSingle}_1 \\ \quad \texttt{where ExprSingle}_2 \\ \quad \texttt{return \$x2}) \\ \texttt{where ExprSingle}_3 \\ \texttt{return \$x1} \end{array} \rightarrow \begin{array}{l} \texttt{ForOrLetClauses}_1 \\ \texttt{ForOrLetClauses}_2 \\ \texttt{let \$x2 in ExprSingle}_1 \\ \texttt{where ExprSingle}_3 \texttt{ and ExprSingle}_2 \\ \texttt{return \$x2} \end{array} \quad (14)$$

Condition: There are no other occurrences of $x1.

Fig. 6. Let rewrites

only once and inline the associated expression (i.e. $\texttt{ExprSingle}_1$). On this result, the rewrites of the previous section (see Fig. 5) can be applied and eliminate the nesting.

Fig. 6 also contains two rewrites that remove nested `for` (see Rule 13) and `let` (see Rule 14) expressions, respectively. Without loss of generality, the outer `let` clause in both rules is immediately followed by the `where` clause. If there was another `for` or `let` clause, it would not contain occurrences of x1 and, hence, could be moved above the `let` clause binding x1.

Running Example. The result of applying the `let` Rewrite Rule 12 and the `for` Rewrite Rule 10 to our example is the following query finally consisting of a single query block.

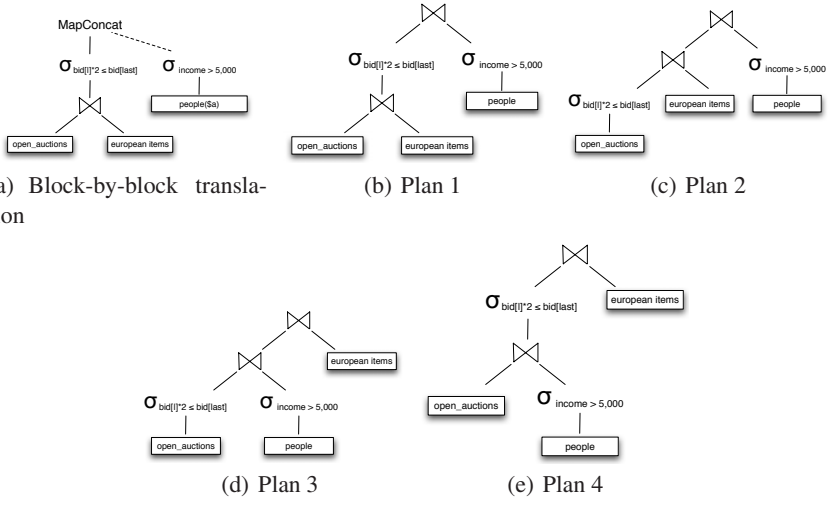
```
let $auction := doc("auction.xml"), $x32 := $auction/site
for $o in $x32, $x13 in $o/open_auctions, $a in $x13/open_auction
for $i in $x32, $x15 in $i/regions, $x16 in $x15/europe
for $x17 in $x16/item, $x18 in $x17/@id
let $x4 := $a/itemref, $x19 := $x4/@item
let $x33 := $a/bidder[1], $x34 := $x33/increase, $x35 := $x34/text()
let $x36 := $a/bidder[last()], $x37 := $x36/increase, $x38 := $x37/text()
let $x39 := $a/@id
for $p in $x32, $x20 in $p/people, $x21 in $x20/person
for $w in $x21/watches, $x22 in $w/watch
let $x8 := $x21/@id, $x10 := $x22/@open_auction
let $x13 := $x21/profile, $x27 := $x13/@income
for $x1 in  <auction id="{$x39}">
              <increase first="{$x35}" last="{$x38}"/>
              <watched_by id="{$x8}"/>
            </auction>
where zero-or-one($x35) * 2 <= $x38 and $x39 = $x10
      and $x27 > 5000 and $x19 eq $x18
return $x1
```

Note how in this form, all value-based join and selection predicates are available in a unified `where` clause. This allows a plan generator to decide on index access and join orders.

6 Evaluation

A goal of this paper is to show how to rewrite a query into a form that consists of a single query block to give a single run of the plan generator as much uniformly structured information about the query as possible. We now elaborate on the importance of this goal by discussing the optimization of our example query during plan generation. We will see how more efficient plans can be generated only when the query has been reduced to a single block.

Due to space constraints, we do not explore the whole search space available, but focus on join ordering. We assume that the optimizer has decided on subplans to produce the sequences for Open Auctions, European Items, and Persons. The subplans may be based on pattern matching algorithms. Further, we assume that the predicate selecting the auctions according to their bids has been converted into a single predicate subplan. This predicate is, however, more expensive to evaluate than a simple value comparison, and its placement in the overall plan does affect performance significantly. Thus,



(a) Block-by-block translation (b) Plan 1 (c) Plan 2

(d) Plan 3 (e) Plan 4

Fig. 7. Alternative execution plans

finding an optimal plan includes finding an optimal position for this predicate. We now discuss execution plans for our example query in the form of algebraic expressions on an abstract level (see Fig. 7).

A straightforward translation of the original, nested, multi-block query looks like Fig. 7(a). Here, the FLWOR blocks are translated directly into separate subplans, and no global optimization takes place. For simplicity, we disregard the first line of the example query (the initial `let` clause for the document root). The top-level MapConcat operator represents the main FLWOR expression. Its operand generates the tuple stream and contains subplans for the European Auctions query block. The subplan connected to MapConcat by the dashed line represents the query block in the `return` clause (the last eight lines of the query). It has a free variable $a in the subplan for the `people` sequence, and, hence, has to be reevaluated for every tuple of the MapConcat operand, as dictated by XQuery FLWOR semantics.

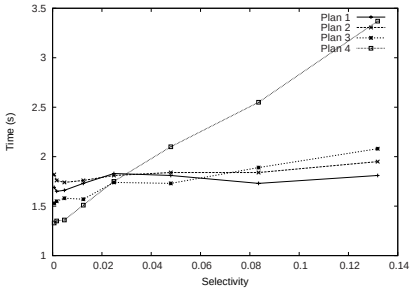


Fig. 8. Performance results

Fig. 7 shows four other execution plans based on the rewritten, single-block form of our example query. They can be enumerated by the plan generator because it has access to all value-based predicates of the query in a single `where` clause and can detect joins and determine an optimal order for them and the residual selections. We executed all five plans from Fig. 7 in our hybrid relational and XML DBMS Natix [9] on an XMark document with scaling factor one.

The experimental setup consisted of a PC with an Intel Pentium D CPU having 3.40GHz and 1GB of main memory, running

on openSUSE 10.2 with Linux Kernel 2.6.18 SMP. To investigate the relative performance of the execution plans, we varied the selectivity of the predicate restricting the people by their income between 0.14 and 0. This corresponds to incomes between 60,000$ to 130,000$ instead of 5,000$ in the original query. Fig. 8 shows the result of this small performance study (execution time in seconds) for four plans from Fig. 7.

The experiment makes obvious why careful global plan generation based on single-block queries is crucial for efficient execution. The results of the nested-loop strategy of the straightforward translation are orders of magnitude slower (well beyond 100s) and have been left out of the graph. The join-based plans made possible by our rewritten single-block query show that an enumeration of alternatives is as important as in relational query processing: Depending on selectivity, the overall best plan varies. The plan according to Fig. 7(e) performs best with a very low selectivity, whereas the plan belonging to Fig. 7(b) outperforms the others with an increasing selectivity.

Acknowledgments. We would like to thank Simone Seeger and the anonymous reviewers for their helpful comments.

References

1. Aho, A., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Wesley Longman Publishing, Boston, MA, USA (1986)
2. Brantner, M., Kanne, C-C., Moerkotte, G.: Let a single FLWOR bloom. Technical report, University of Mannheim, TR-2007-007 (2007)
3. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: SIGMOD, pp. 310–321 (2002)
4. Dayal, U.: Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In: Proc. VLDB, pp. 197–208 (1987)
5. Hidders, J., et al.: How to recognise different kinds of tree patterns from quite a long way away. In: Proc. PLAN-X (2007)
6. May, N., et al.: XQuery processing in Natix with an emphasis on join ordering. In: First International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P 2004) (2004)
7. Al-Khalifa, S., et al.: Structural Joins: A primitive for efficient XML query pattern matching. In: ICDE, pages 141- (2002)
8. Boag, S., et al.: XQuery 1.0: An XML query language. Technical report, World Wide Web Consortium, W3C Recommendation (January 2007)
9. Fiebig, T., et al.: Anatomy of a native XML base management system. j-VLDB-J 11(4), 292–314 (2002)
10. Ganski, R.A., Wong, H.K.T.: Optimization of nested SQL queries revisited. In: SIGMOD, pp. 23–33 (1987)
11. Graefe, G.: Query evaluation techniques for large databases. ACM Computing Surveys 25(2), 73–170 (1993)
12. Grust, T., Keulen, M.v.: Tree awareness for relational DBMS kernels: Staircase Join. In: Intelligent Search on XML Data, pp. 231–245 (2003)
13. Hidders, J., Michiels, P.: Avoiding unnecessary ordering operations in XPath. In: Database Programming Languages, pp. 54–70 (2003)

14. Josifovski, V., Fontoura, M., Barta, A.: Querying XML streams. j-VLDB-J 14(2), 197–210 (2005)
15. May, N., Helmer, S., Moerkotte, G.: Strategies for query unnesting in XML databases. ACM Transactions on Database Systems 31(3), 968–1013 (2006)
16. Michiels, P., Mihaila, G., Siméon, J.: Put a tree pattern in your algebra. In: Proc. ICDE (2007)
17. Pirahesh, H., Hellerstein, J.M., Hasan, W.: Extensible/rule based query rewrite optimization in starburst. In: SIGMOD, pp. 39–48 (1992)
18. Seshadri, P., Pirahesh, H., Leung, T.Y.C.: Complex query decorrelation. In: Proc. ICDE, pp. 450–458 (1996)

Efficient XQuery Evaluation of Grouping Conditions with Duplicate Removals

Norman May and Guido Moerkotte

University of Mannheim
B6, 29
68131 Mannheim, Germany
{norman,moer}@db.informatik.uni-mannheim.de

Abstract. Currently, grouping in XQuery must be expressed implicitly with nested FLWOR expressions. With XQuery 1.1, an explicit **group by** clause will be part of this query language. As users integrate this new construct into their applications, it becomes important to have efficient evaluation techniques available to process even complex grouping conditions. Among them, the removal of distinct values or distinct nodes in the partitions defined by the **group by** clause is not well-supported yet. The evaluation technique proposed in this paper is able to handle duplicate removal in the partitions efficiently. Experiments show the superiority of our solution compared to state-of-the-art query processing.

1 Motivation

XML gains importance as a storage format for business or scientific data. As more and more data is stored in this format, analytical query processing, i.e. XOLAP, becomes an important requirement. XQuery is the query language standardized for this purpose. While XQuery already includes a rich set of features, it lacks functions to support analytical query processing efficiently. Most importantly, grouping must be formulated implicitly with nested queries. While this challenge has already been addressed by techniques that unnest nested queries, end users and database implementors have identified the need for an explicit grouping construct that allows for efficient processing.

Consequently, a value-based grouping construct is part of the core requirements for XQuery 1.1, the next version of XQuery. Since the work on this version has just started, we will use the proposal for a **group by** construct by Beyer et. al. [1]. An efficient XQuery execution engine should include a powerful implementation of the grouping operator. With minor extensions of the relational grouping operators, it is possible to support several cases of grouping. We focus on a case that is neither well-supported for XQuery nor for SQL: We investigate efficient evaluation techniques for **group by** where duplicates are removed on different attributes of tuples that are in the same partition.

D. Barbosa et al. (Eds.): XSym 2007, LNCS 4704, pp. 62–76, 2007.

1.1 Motivating Example

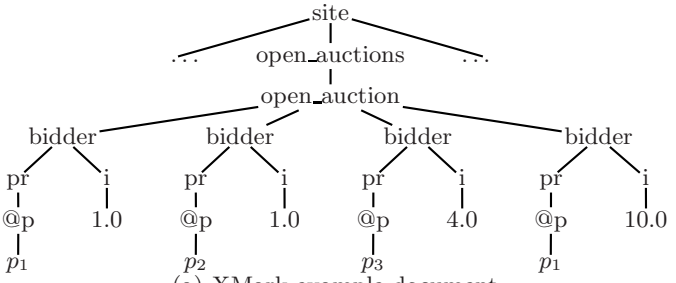
Consider the following example query on the XMark document instance depicted in Fig. 1(a). It counts for every open auction the total number of bidders, the number of distinct bidders, the maximum increase, and the number of different increases (We abbreviate some element or attribute names as follows: personref – pr, @person – @p, increase – i).

```
for $auction in $doc/site/open_auctions/open_auction,
    $bidder  in $auction/bidder
let $person   := $bidder/personref/@person,
    $increase := $bidder/increase
group by $auction into $a using fn:deep−equal
      nest $person   into $p,
           $increase into $i
      let $pd := distinct−values($p),
          $pi := distinct−values($i)
return
  <status>
       { $a/seller }
       <bid−count>       { count($p)  } </bid−count>
       <distinct−bidders> { count($pd) } </distinct−bidders>
       <max−increase>    { max($i)    } </max−increase>
       <distinct−steps>  { count($id) } </distinct−steps>
  </status>
```

In the example query, the keyword **group by** is directly followed by the *grouping expression* (a reference to variable $auction), the keyword **into**, and the *grouping variable* ($a). The function mentioned after the keyword **using** is used to partition the input tuples into groups. The *nesting expression* after the keyword **nest** is applied to every tuple that is assigned to some group. The result of this computation is appended to the current group referenced by the *nesting variable*. In the example query, we use the optional **let** clause to remove duplicates from the sequences bound to the nesting variables. The result of the **group by** is a sequence of tuples that contains bindings for every grouping variable and every nesting variable. Notice that sequence order is not meaningful in the context of this operator because there is no immediate relationship between input tuples and output tuples any more. We denote with *aggregation variable* all variables in the scope of the **return** clause that are the argument of an aggregate function, i.e. $p, $i, $pd, $pi in our example query.

Fig. 1(b) shows the tuples that serve as input to the **group by**. We trace how the group defined by a single open auction, a_1, is processed. In Fig. 1(c), we have identified groups – in this example, there is only a single group. To compute the result of the aggregate functions in the **return** clause of this query, we need to remove duplicate values from every sequence bound to the nesting variables. We have highlighted them in the two sequences.

Most systems try to avoid copying and, hence, would filter duplicates by discarding complete input tuples. In general, this only works when duplicates are removed from at most one aggregation variable. In our example, however,



(a) XMark example document

$auction	$person	$increase
a_1	p_1	1.0
a_1	p_2	1.0
a_1	p_3	4.0
a_1	p_1	10.0

(b) Input of **group by**

$a	$p	$i
a_1	< **p₁**,	< **1.0**,
	p_2,	**1.0**,
	p_3,	4.0,
	p₁ >	10.0 >

(c) Groups and duplicates detected

$a	$p	$pd	$i	$id
a_1	< p_1,	< p_1,	< 1.0,	< 1.0,
	p_2,	p_2,	1.0,	4.0,
	p_3,	p_3 >	4.0,	10.0 >
	p_1 >		10.0 >	

(d) Duplicates removed

Fig. 1. A complex grouping query

the second and the fourth tuple contain duplicates. However, we have to keep the second tuple because bidder p_2 is a unique value. We also have to keep the fourth tuple because it contains a unique value for the increase. Clearly, every aggregation variable with duplicate removal needs to be processed seperately. The result after duplicate removal in the **let** clause is shown in Fig. 1(d). It is the input to the aggregate functions in the **return** clause.

1.2 State-of-the-Art Processing

As we are not aware of any publically available system that supports the **group by** operator in XQuery, we have looked at similar queries in SQL.[1] We could distill two basic strategies to process this type of queries.

Sort-Based Strategy: Replicate the input for every aggregation variable that requires duplicate elimination. Sort the sequences to aggregate and use a sort-based implementation to evaluate the aggregate function with and without duplicate removal in one pass over the data.

[1] In SQL, the keyword `distinct` may occur only inside one single aggregate function. Nevertheless, actual database systems support the occurance of this keyword in several aggregate functions over distinct attributes.

Hash-Based Strategy: Perform one scan and compute the aggregate function for all attributes without duplicate removal, and perform another scan for every aggregation variable with duplicate removal. This alternative may also use hash-based implementations for grouping and duplicate removal.

In our experiments we show that these two strategies do not scale well with an increasing number of duplicate removals. For every expression that demands a duplicate removal, either strategy introduces a new scan over the input data. Consequently, query performance suffers as the number of such expressions grows.

1.3 Our Contributions

The contribution of our work is to avoid the repeated scan of the input data mentioned above. We propose to process a single group at a time. In many cases, the whole group will fit into main memory and, thus, expensive I/O is avoided. Moreover, we can handle groups of arbitrary size – even larger than available main memory. In our evaluation strategy, available main-memory is a tunable parameter. Thus, we can trade increased memory consumption for faster processing in main memory. Unlike other proposals for grouping of XML data, our processing strategy fits well into current database architectures. In particular, we need to extend the query execution engine only with two new algebraic operators. We have implemented all three alternative processing strategies in Natix, our native XML database system [7]. Our experiments show that our approach performs favourably compared to the state of the art.

Outline of the Paper. Next, in Sec. 2 we discuss related work. The core of this paper is Sec. 3, in which we discuss the three alternative strategies to process grouping with duplicate removal in the nesting expression. In experiments presented in Sec. 4, we compare the performance of these plan alternatives. In Sec. 5, we summarize the results of this paper.

2 Related Work

The recently published W3C recommendation of XQuery does not contain an explicit **group by** construct [2]. Consequently, grouping must be expressed with nested queries, and optimizers need to detect that grouping is formulated implicitly. Proposals to detect *implicit grouping* by unnesting nested query blocks in XQuery include [13,15,16].

As motivated in [1], it can be quite cumbersome to formulate grouping queries correctly. Moreover, if the query optimizer cannot detect implicit grouping in a nested query, evaluating the nested query usually results in poor performance. Hence, both users and database implementors seem to agree that XQuery should include an explicit **group by** syntax and, thus, value-based grouping as we discuss it in this paper is a core requirement for the next version of XQuery [6]. Proposals for a syntax for grouping in XQuery have appeared in [1,3,12]. In this paper, we use the syntax and semantics presented in [1].

Recently, implementations for **group by** and the cube operator for analytical XML query processing were developed. The grouping operator proposed in [9] computes arbitrary aggregates over a single XML document by merging XML nodes belonging to the same group. In this proposal, retrieval of the XML data, grouping, and aggregation are tightly integrated into a single processing strategy.

Another extension to XQuery, a cube operator for XML, is presented in [17]. The paper investigates computational and semantic challenges associated with the aggregation of XML. Since implementations of the cube operator can benefit from efficient algorithms of the grouping operator, our work is also relevant when the arguments of aggregate functions are subject to duplicate removal.

Both proposals above do not explicitly address the problem of duplicate elimination. In fact, we are not aware of any proposal to handle duplicate elimination in grouping operations. The two processing strategies, which we use to benchmark our implementation, are derived from the explain utilities of two commercial database products.

Our solution is closely related to the XML result construction operators presented by Fiebig et al. [8]. Based on this framework, we process one group at a time. This allows us to optimize the processing of every aggregate individually and, thus, improve the performance of query evaluation.

Standard implementation techniques for the grouping operator are discussed by Graefe [10]. This survey also discusses data flow and control flow beyond the standard iterator model, as we use them in this paper. We plan to extend our distinct processing strategy to the binary grouping operator [14] and to window-based aggregation. Of course, efficient implementations for the grouping operator need to be complemented with optimizations, as they were presented in [5,11,18]. These optimizations are still valid for our implementation.

3 The Grouping Algorithms

In this section, we develop the grouping algorithm that can handle duplicate elimination in arguments of aggregate functions efficiently. First, we introduce some notation necessary to understand the plan alternatives we present in this section. This notation is borrowed from [8]. Then, we discuss three alternatives to evaluate grouping with duplicate removal.

3.1 Notation

The algebraic operators in the query execution engine of Natix are implemented as iterators [10]. They consume and produce sequences of tuples. Every tuple contains a set of attributes which are either bound to base types such as strings, numbers, or node references, or again contain ordered sequences of tuples. Thus, our operators conform to the basic iterator interface `open`, `getNext`, and `close`; details can be found in [7]. The operators we use in this paper are shown in Fig. 2. In the upper part of an operator, we give its name, whereas the lower part contains information about the subscript, e.g. the sort key of the `Sort` operator. They include:

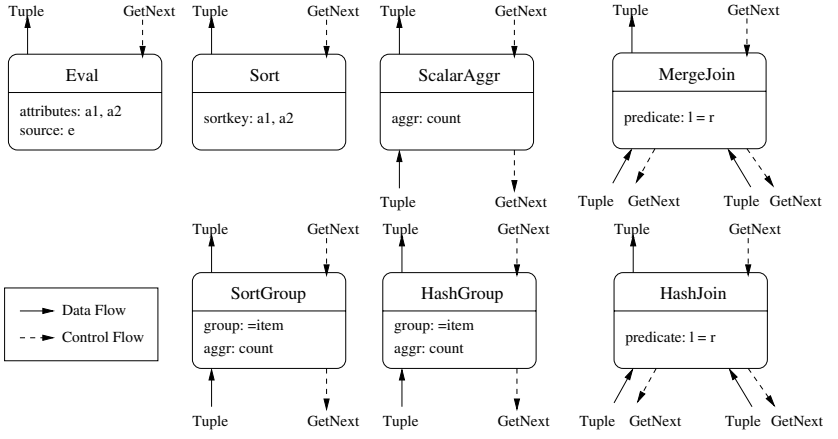


Fig. 2. Operator Notation

`Eval` This operator is not an actual operator. In this paper, it denotes XML data retrieval needed as input to the grouping operators. It evaluates a complex expression, e, in its subscript and binds the attributes mentioned. Executing expression e might include index access or navigation over an XML document. Since efficient XML data retrieval is not the focus of this paper, we use this operator as an abbreviation.

`Sort` sorts its input according to the sort key mentioned in its subscript. Our implementation uses external sorting with replacement selection.

`ScalarAggr` returns a single tuple with a set of attributes, each of which is bound to the result of an aggregate function, e.g. `fn:min`, `fn:sum`, `fn:count`, or even SQL/XML functions such as `xmlagg`.

`MergeJoin` implements a 1:N sort-merge join. Of course, its arguments must be sorted on the attributes mentioned in the join predicate. In this paper, we do not need the more general N:M sort-merge join.

`HashJoin` implements a nested-loop join where blocks of the left producer are loaded into a main-memory hash table and matched with tuples of the right producer.

`SortGroup` groups the input assuming that the sequence of tuples of the producer is sorted by the grouping attributes.

`HashGroup` employs a main-memory hash table to perform the grouping operation.

All these operator implementations are well-known from the literature [10]. Notice that the two grouping operators can also be used to remove duplicates. When we want to remove duplicates, we denote this by `DuplElim`.

In this iterator-based implementation, control flows from a consumer of tuples to its producer, and data flows into the opposite direction. As depicted in Fig. 2, we denote the former by dashed arrows and the latter by solid arrows. As our solution involves control flow beyond the direct argument relationship, we explicitly present them in our plans.

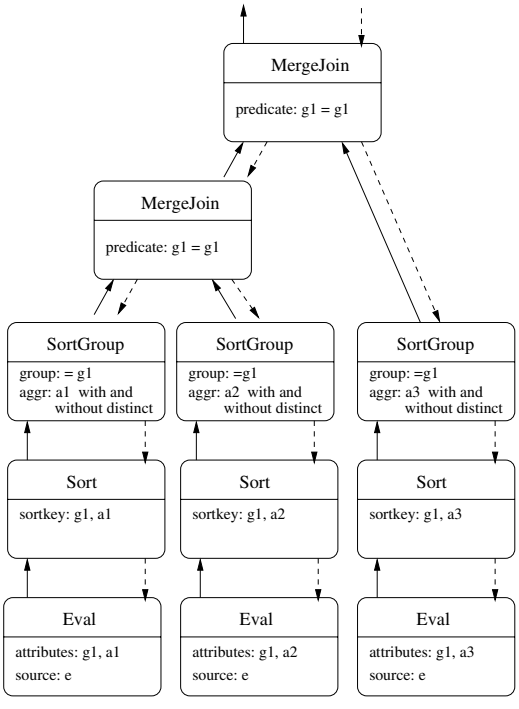


Fig. 3. Plan for sort-based strategy

3.2 Sort-Based Evaluation Strategy

The first evaluation strategy for grouping we have observed in commercial systems is shown in Fig. 3. It performs repeated scans of the input data for every attribute that contains a duplicate removal. Additional aggregate functions are piggy-backed on the branches in the plan. All grouping operators require their input to be sorted on the grouping attributes. In addition, the input must be sorted on the attributes mentioned in aggregate functions with duplicate removal. As a consequence, the sort-based grouping operator can compute aggregate functions that do not contain duplicate removals and all aggregates that contain duplicate removal on the same attribute in one scan. At the end, all partial plans are combined by a sort-merge join because this join implementation exploits the order available on the grouping attributes.

Evidently, this strategy performs the scan of the input data and, if needed, the sort operation repeatedly for every distinct attribute mentioned in an aggregate function with duplicate removal. It is our goal to share this repeated evaluation and, thereby, improve the performance of the plan.

3.3 Hash-Based Evaluation Strategy

The second strategy we have found in commercial systems employs one partial plan to compute the aggregation function for all attributes where no duplicates

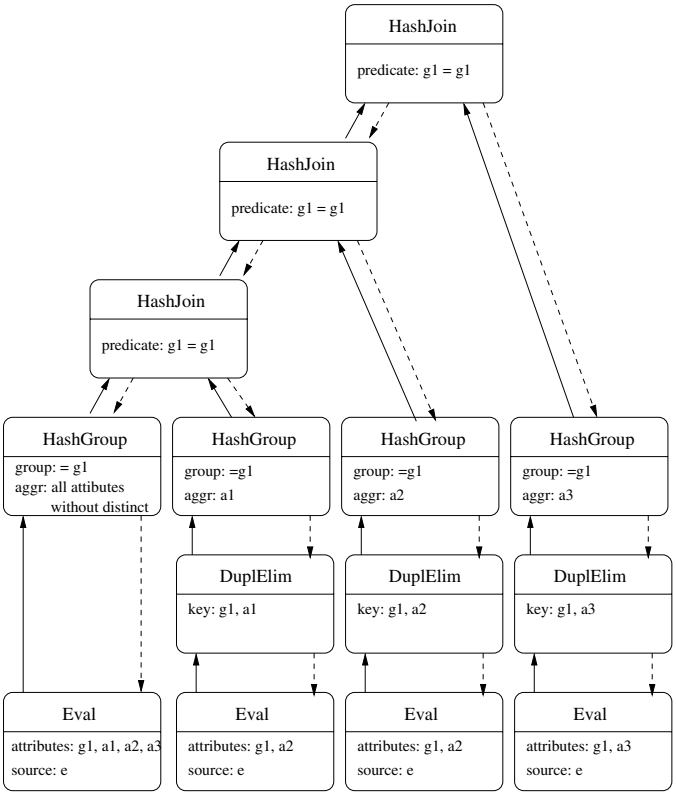


Fig. 4. Plan for hash-based strategy

are removed. For every aggregate function that contains a duplicate removal on some attribute, one partial plan is executed. All these plans are combined by joins to compute the final result. This strategy, depicted in Fig. 4, leaves the query optimizer the freedom to choose between sort-based and hash-based implementations for every partial plan. Thus, in this extreme case where we use sorting for every branch, we arrive at the sort-based strategy but with one additional partial plan.

For this reason, we investigate the case where all operations are performed by hash-based operators. The potential advantage of hash-based operators is that they avoid sorting. Notice, however, that this strategy shares the inefficiency of repeated scans (or evaluation) of the input.

3.4 Groupify and GroupApply

It is our goal to avoid the repeated evaluation of the argument expression of the **group by** operator. The key idea of our approach is to separate detecting group

boundaries from processing groups. For every group, evaluating the complex grouping conditions follows three steps. These steps are governed by two new operators, `Groupify` and `GroupApply`, and involve non-standard control and data flow between them. This is shown in Fig. 5.

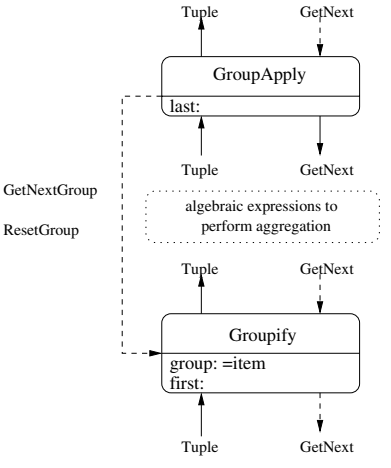


Fig. 5. Groupify and GroupApply

First, the `Groupify` operator detects group boundaries. Computing the aggregate functions for each group requires multiple passes over the tuples of this group. Therefore, the `Groupify` operator materializes the tuples of a group. Notice that a group might not completely fit into physical main memory. When we have finished processing a group, the operator discards the tuples of this group and continues with the next one. In our implementation, we expect the input to be sorted on the grouping attributes. But a hash-based implementation is also possible.

Second, aggregation and duplicate removal is done by plans that consume the tuples of a group. One advantage of our approach is that we can optimize these partial plans. For example, we can apply any of the strategies discussed in the previous two subsections. Another advantage is that processing of a group focuses on a fragment of the whole data and, thus, can benefit from the locality of data access.

Finally, we need the `GroupApply` operator to combine the results of the partial plans. After a group was detected and materialized, this operator evaluates all partial plans in a round-robin fashion. Since every partial plan is evaluated on the same sequence of tuples, the `GroupApply` operator signals the `Groupify` operator to scan the current group again from the beginning. Therefore, the `GroupApply` operator calls the function `Groupify::resetGroup`. The `GroupApply` operator combines the results of these plans into the overall result of the current group. When we have evaluated every partial plan, we can process the next group. Again, this requires communication between the `GroupApply` and `Groupify` operator. More precisely, the `GroupApply` operator uses the function `Groupify::getNextGroup` to notify that the evaluation of all argument plans is complete. Then, the `Groupify` operator discards the current group and processes the next one.

Fig. 5 shows the communication between the `Groupify` operator and the `GroupApply` operator as dashed arrows. The control flow bypasses the plans that perform the aggregation. The data flow extends the iterator model because several algebraic operators consume the tuples produced by the `Groupify` operator. Notice that we neither have to modify any iterator-based operator nor the

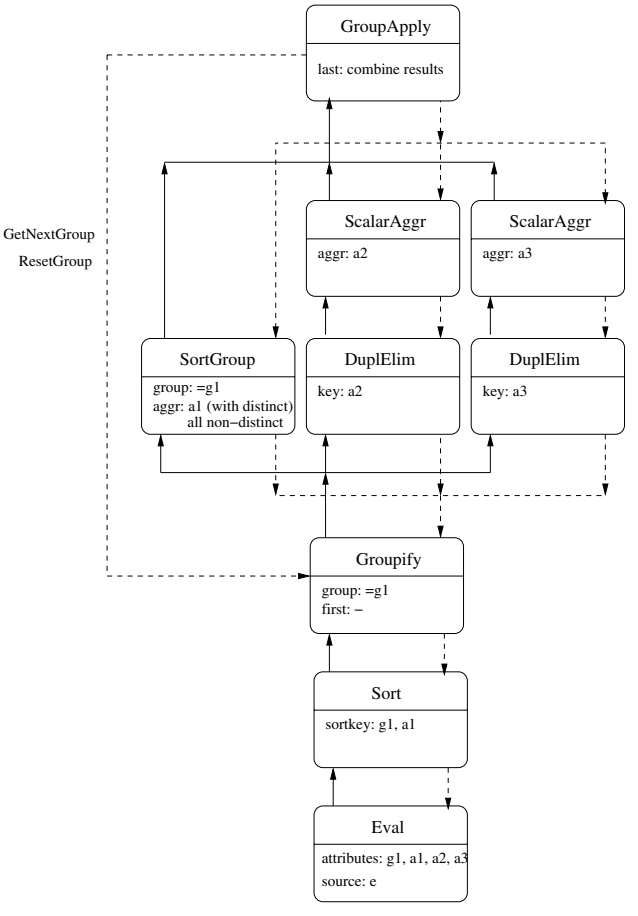


Fig. 6. Plan based on `Groupify` and `GroupApply`

general query processing architecture of our system to support this data flow. All necessary extensions are restricted to the `Groupify` and `GroupApply` operator.

3.5 Evaluation Strategy-Based on Groupify and GroupApply

Now, we discuss how we use the operators introduced in the previous subsection to improve the query execution time for queries containing grouping and duplicate removal on different attributes of tuples that are in the same partition.

From the plan structure shown in Fig. 6, it is evident that we need to evaluate the argument expression only once. The result of this expression is sorted on the grouping attributes, and the `Groupify` operator exploits the order to detect group boundaries.

The leftmost partial plan computes the aggregate functions without duplicate removal, using a sort-based grouping operator. This partial plan can benefit

from a minor sort on the attributes a_1, on which it needs to perform a duplicate removal. Notice that this plan also carries the grouping attributes needed in the final result. The remaining partial plans remove the duplicate values for a single attribute and compute the aggregate for this attribute. An advantage of this fine-grained approach is that every partial plan can be optimized by a cost-based optimizer. In particular, a sort-based or hash-based evaluation strategy can be chosen for every of these partial plans. The `GroupApply` operator combines the results of the partial plans to the final result tuple of the grouping operation.

4 Experiments

We have implemented all algorithms in Natix, our native XML database system [7]. Natix was compiled with GCC 4.1.2 and optimization level O3. All queries were executed on a Linux system with Kernel 2.6.18, an Intel Pentium 4 CPU 2.40GHz, 1 GB RAM, and IBM 18.3 GB Ultra 160 SCSI hard disk drive with 4 MB buffer. All queries were run with cold buffer cache of 8 MB size. We report the average execution time of three runs of every query.

4.1 Dataset and Queries

To investigate the performance of the three execution strategies presented in Sec. 3, we use a synthetic dataset. This setup allows us to carefully investigate the impact of different parameters both of the query and the data.

Every execution strategy retrieves the input to the **group by** clause from a materialized XML view that contains exactly the tuples needed to evaluate the query. This is reasonable because we want to isolate the effect of the different implementations for grouping. We have examined three XML views X_1, X_2, and X_3. Their cardinality and raw data size is summarized in Fig. 7.

XML view	cardinality	raw size
X_1	$2^{16} = 65$k	1.3 MB
X_2	$2^{20} = 1$M	21 MB
X_3	$2^{23} = 8$M	168 MB

Fig. 7. Dataset

In Fig. 8, we show the query pattern we used to benchmark the query performance of the different evaluation strategies. The first **for** clause retrieves the grouping attribute and the remaining **for** clauses the k attributes to aggregate. Choosing different tag names `tag_g` and `tag_i` ($i = 1 \ldots k$) in these clauses allows us to modify the number of groups or the number of distinct values for the attributes to aggregate. Thus, we can control the number of groups in the result and the cost and effect of the duplicate removal in the nesting expressions. In the **group by** clause, we have k nesting variables. Every nesting variable stores one sequence of values to aggregate in this group. For each such sequence, the **let** clause after that computes the sequence of values with duplicates removed. In the **return** clause, we apply the aggregate function `fn:sum` to the nesting sequences computed this way.

```
for $g in $doc//tag_g,
    $a1 in $g//tag_a1,
     ...
    $ak in $g//tag_ak
where P
group by $g into $gg using eq
      nest $a1 into $a1,
             ...
           $ak into $an,
      let  $a1d := distinct−values($a1),
             ...
           $and := distinct−values($ak)
return
  <result>
       { $g }
       <a1> { sum($a1) } </a1>
       <a1d> { sum($a1d) } </a1d>
         ...
       <ak> { sum($ak) } </ak>
       <akd> { sum($akd) } </akd>
  </result>
```

Fig. 8. Query Pattern

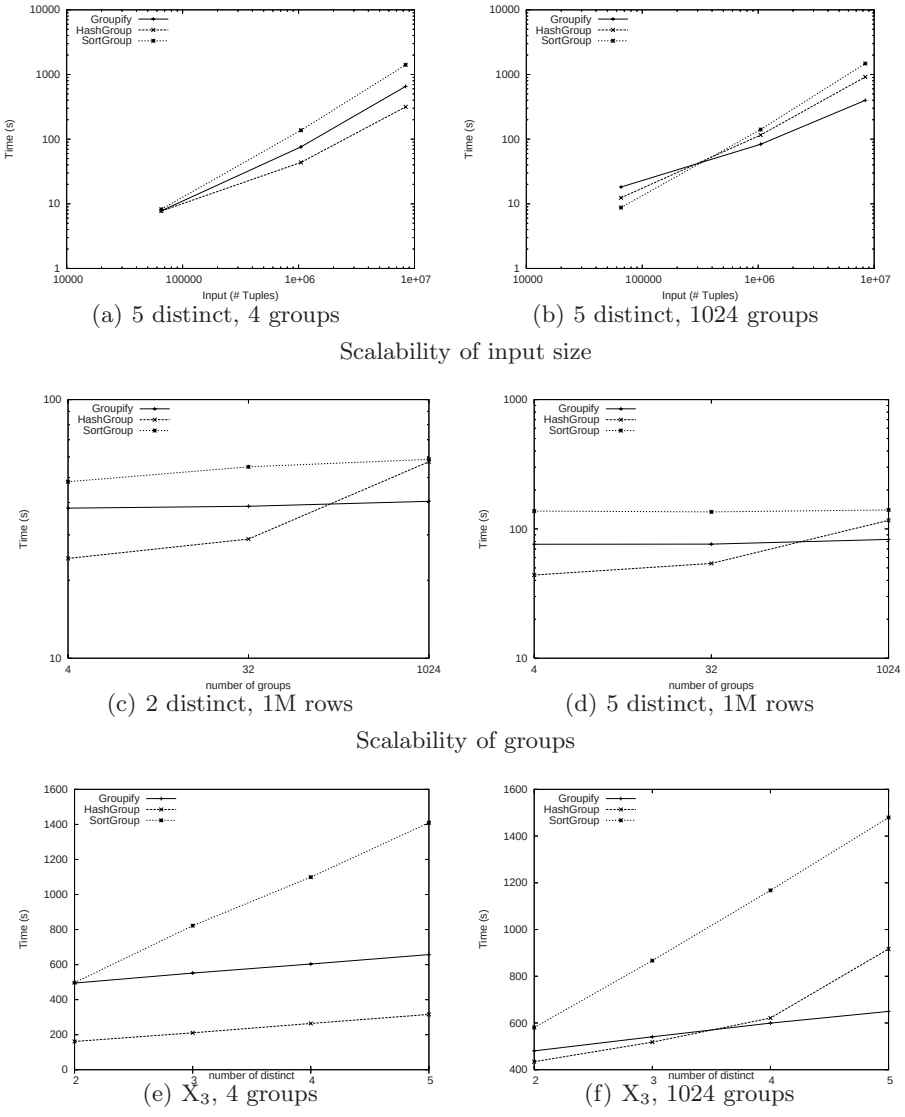
Every tuple consists of 5 integer attributes: one with s unique values, the others with 4, 32, 1024, or 65536 distinct values. Every attribute corresponds to one tag name or attribute selected from the document. For space reasons, we can only report the most interesting results.

4.2 Experimental Results

Fig. 9 summarizes the results of our experiments. We investigate how each algorithm scales with respect to the input size, the number of distinct groups, and the number of nesting expressions with duplicate removal.

Scalability of Input Size. Figs. 9(a) and 9(b) show how the three algorithms behave with increasing input size when the query contains aggregate functions on five distinct attributes with duplicate removal. We consider both only four (large) groups and 1024 (smaller) groups. Notice the logarithmic scale of both axes. Clearly, the sort-based approach scales worst among the three alternatives. From Fig. 9(a) it is evident that the hash-based plan can exploit that only four groups exist. Both grouping and the subsequent joins need to manage only few result tuples which leads to an overall performance advantage compared to the plan using `Groupify` and `GroupApply`. In Fig. 9(b), on the other hand, these two operators scale best.

Scalability of Groups. In Figs. 9(c) and 9(d), we compare the effect of the number of groups to process. The more groups we have in the input data, the smaller they become. In both plots, we report the elapsed execution times on the XML view X_2 with two or five attributes with duplicate removal. The experiments clearly show that both the sort-based strategy and `Groupify/GroupApply` are almost insensitive with respect to the number of groups. The cost of these plans is dominated by the cost for sorting, and this cost component does not change much with the size of the groups. After that, detecting group boundaries is very cheap. Independent of the number of groups, the strategy based on `GroupApply` and `GroupApply` outperforms the sort-based strategy. The performance of the hash-based plan, on the other hand, is very sensitive to the number of distinct groups. Consequently, the plan quality of this strategy strongly depends on good cardinality estimates for the number of groups. Unfortunately, estimating the number of groups is an inherently difficult task [4]. `Groupify` and `GroupApply`



(a) 5 distinct, 4 groups (b) 5 distinct, 1024 groups

Scalability of input size

(c) 2 distinct, 1M rows (d) 5 distinct, 1M rows

Scalability of groups

(e) X_3, 4 groups (f) X_3, 1024 groups

Scalability of duplicate removals

Fig. 9. Experimental Results

become faster than the hash-based method when there are more than approx. 50 groups. In further experiments, we observed similar results for larger and smaller input sizes.

Scalability of Duplicate Removals. In our final experiment, shown in Figs. 9(e) and 9(f), we scale the number of attributes on which we remove duplicates. The plots show the query execution times on XML view X_3 for four and 1024 groups.

	Sort-based	Hash-based	`Groupify/GroupApply`
Scans of input	$max(\|a_d\|, 1)$	$1 + \|a_d\|$	1
Temp+Scan	$max(\|a_d\|, 1)$	$\|a_d\|$	can be tuned
Main Memory	Sort buffers	Hash buffers	can be tuned
Combining result	Join(s)		Copy values

where a_d = attributes that occur with **distinct**

Fig. 10. Summary of the plan alternatives

Clearly, the sort-based strategy performs worst among the three alternatives. For every new attribute with duplicate removal, it has to scan and sort the whole input once more. As both operations demand I/O operations, the performance suffers.

Both the hash-based plan and `Groupify/GroupApply` scale better than the sort-based plan. Again, the hash-based algorithm is the fastest for few groups. However, for a larger number of groups, `Groupify/GroupApply` outperform the other alternatives.

Number of Duplicates to Remove. In our experiments, the query excecution times did not change significantly when we increased the number of duplicates to be removed before aggregating them. Hence, we do not present any experimental results that show the effect of duplicate elimination.

5 Conclusion

We have investigated three different strategies to evaluate grouping when duplicates are removed in several nesting expressions. Based on the algorithms underlying each strategy, we can derive the I/O operations needed for each strategy (see Fig. 10). Both the sort-based and the hash-based alternative scan the base data once for every aggregation variable which requires a duplicate elimination. The combination of `Groupify` and `GroupApply`, on the other hand, scans the base data only once. Consequently, `Groupify/GroupApply` scales better than the other two strategies with an increasing number of duplicate removals on different aggregation variables. Since this strategy keeps a single group in main memory while this group is processed by several partial plans, it avoids expensive I/O operations. If the group is too large to fit in main memory, parts of the group can be spooled to disk. Hence, I/O operations can be traded for main-memory usage. Overall, this leads to more local data access patterns.

Our novel evaluation strategy only requires two new algebraic operators in a query engine and, thus, it fits well into the standard architecture of database systems. Finally, we remark that neither alternative can be done in a fully pipelined fashion. But the group-wise processing of `Groupify/GroupApply` returns first results faster. The sort-based strategy demands several more sort operations and thus, is slower. The hash-based method must process all groups before the first result tuple is returned. Based on these observations, we plan to develop a cost model for this processing strategy.

Clearly, evaluation techniques discussed in this paper are not restricted to grouping in XQuery. It may also be useful for analytical SQL queries in a data warehouse environment. Currently, however, the SQL standard allows `DISTINCT` to be applied only to a single aggregation expression.

Acknowledgements. We would like to thank Simone Seeger for her comments on the manuscript.

References

1. Beyer, K., Chamberlin, D., Colby, L., Özcan, F., Pirahesh, H., Xu, Y.: Extending XQuery for analytics. In: SIGMOD (2005)
2. Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML Query Language. W3C (2007)
3. Borkar, V., Carey, M.: Extending XQuery for grouping, duplicate elimination, and outer joins. In: XML 2004 (2004)
4. Charikar, M., Chaudhuri, S., Motwani, R., Narasayya, V.: Towards estimation error guarantees for distinct values. In: Proc. of the ACM PODS (2000)
5. Chaudhuri, S., Shim, K.: Including group-by in query optimization. In: Proc. VLDB (1994)
6. Engovatov, D.: XML Query 1.1 Requirements. W3C Working Draft (2007)
7. Fiebig, T., Helmer, S., Kanne, C-C., Moerkotte, G., Neumann, J., Schiele, R., Westmann, T.: Anatomy of a native XML base management system. j-VLDB-J 11(4) (2002)
8. Fiebig, T., Moerkotte, G.: Algebraic XML construction and its optimization in Natix. WWW Journal 4(3) (2001)
9. Gokhale, C., Gupta, N., Kumar, P., Lakshmanan, L., Ng, R., Prakash, B.A.: Complex group-by queries for XML. In: Proc. ICDE (2007)
10. Graefe, G.: Query evaluation techniques for large databases. ACM Computing Surveys 25(2) (1993)
11. Gupta, A., Harinarayan, V., Quass, D.: Aggregate-query processing in data warehousing environments. In: Proc. VLDB (1995)
12. Kay, M.: Positional grouping in XQuery. In: <XIME-P/> (2006)
13. May, N., Helmer, S., Moerkotte, G.: Strategies for query unnesting in XML databases. ACM TODS 31(3) (2006)
14. May, N., Moerkotte, G.: Main memory implementations for binary grouping. In: XSym (2005)
15. Paparizos, S., Al-Khalifa, S., Jagadish, H.V., Lakshmanan, L., Nierman, A., Srivastava, D., Wu, Y.: Grouping in XML. In: EDBT workshops (2002)
16. Re, C., Siméon, J., Fernández, M.F.: A complete and efficient algebraic compiler for XQuery. In: ICDE (2006)
17. Wiwatwattana, N., Jagadish, H.V., Lakshmanan, L., Srivastava, D.: X^3: A cube operator for XML OLAP. In: Proc. ICDE (2007)
18. W. P. Yan, P.-Å. Larson. Performing group-by before join. In: Proc. ICDE, 1994.

On the Effectiveness of Flexible Querying Heuristics for XML Data

Zografoula Vagena, Latha Colby, Fatma Özcan,
Andrey Balmin, and Quanzhong Li

IBM Almaden Research Center
650 Harry Road, San Jose, CA

Abstract. The ability to perform effective XML data retrieval in the absence of schema knowledge has recently received considerable attention. The majority of relevant proposals employs heuristics that identify groups of meaningfully related nodes using information extracted from the input data. These heuristics are employed to effectively prune the search space of all possible node combinations and their popularity is evident by the large number of such heuristics and the systems that use them. However, a comprehensive study detailing the relative merits of these heuristics has not been performed thus far. One of the challenges in performing this study is the fact that these techniques have been proposed within different and not directly comparable contexts. In this paper, we attempt to fill this gap. In particular, we first abstract the common selection problem that is tackled by the relatedness heuristics and show how each heuristic addresses this problem. We then identify data categories where the assumptions made by each heuristic are valid and draw insights on their possible effectiveness. Our findings can help systems implementors understand the strengths and weaknesses of each heuristic and provide simple guidelines for the applicability of each one.

1 Introduction

The expressive power of XML and the many data representation alternatives that it provides can lead to the creation of datasets with very complex schemata. In certain cases, such as Web XML documents created in an ad-hoc manner, a schema might not even exist. These reasons, coupled with the fact that the main usage of XML, as a standard for data sharing, necessitates the ability to query heterogeneous data sources, have made the employment of existing structured XML query languages, such as XPath and XQuery [18], cumbersome for XML data retrieval. Without knowledge of the exact structure of the underlying data it is very difficult to come up with the right query, because XPath expressions follow the document structure. Even if the user had this knowledge, the need to query multiple heterogeneous sources may require the generation of a different query for each data source (either directly from the user or from a complex query translation module) and as a result makes the querying process both cumbersome and error prone.

D. Barbosa et al. (Eds.): XSym 2007, LNCS 4704, pp. 77–91, 2007.

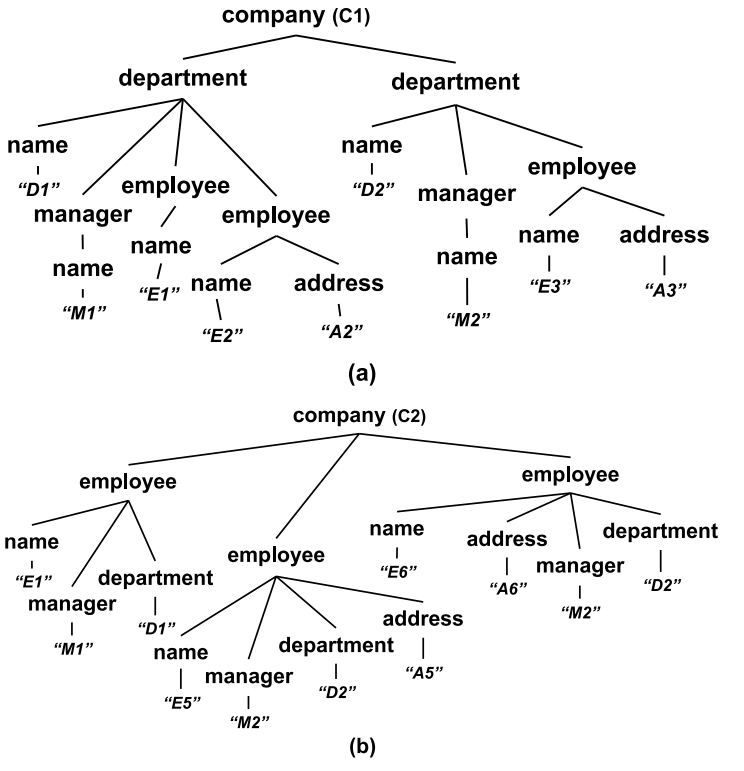


Fig. 1. Querying Heterogeneous XML Data Collections

To illustrate the problem consider the example in Figure 1, which shows two different schemata for *departments* and *employees* within two different companies, $C1$ and $C2$, which have recently merged. The data in company $C1$ (Figure 1a) are grouped by *department*, while the data in company $C2$ (Figure 1b) are grouped by *employee* that works in the company. If a user wants to retrieve information about the employees that work in department $D1$ using XPath to perform the retrieval, she has to issue two, structurally different queries over the two datasets, namely $/company/department[name = "D1"]/employee$ and $/company/employee[department = "D1"]$ respectively. With an increasing number of data sources the retrieval task will become increasingly complex.

To tackle the problem, a number of XML search engines [16,8,7,17] have been developed, which aim to leverage the keyword search paradigm to support XML data retrieval. The main advantage of keyword search is its simplicity. In particular, users do not have to know a complex query language and can query any dataset without prior knowledge of the structure of the underlying data. Nevertheless, pure keyword search is not always the appropriate querying paradigm. First, as pinpointed in [11] it is often difficult and sometimes impossible to convey semantic knowledge (e.g. that the user is looking for the *manager* of a particular *department*). Second,

the granularity of the results is fixed to entire subtrees rooted at the most promising nodes that each search engine determines.

To address the semantic and granularity problems, a number of XML retrieval schemes have appeared that utilize semantic knowledge in the query and the data in order to (a) identify sets of nodes that are relevant to the user query and have close associations and/or (b) automatically define the granularity of the output to return the more relevant nodes [5,11,13,12]. The common characteristic of the above approaches is the use of heuristics, which utilize semantic information derived from the data to decide whether a set of nodes contains meaningfully related nodes. We call those heuristics *relatedness heuristics* from now on. Having these different heuristics, which are not directly comparable, the natural question that comes to mind is how effective they are with respect to user's retrieval needs. In this paper, we address this question by studying the behavior of each heuristic for a given user query with respect to the characteristics of the underlying dataset. Our contributions can be summarized as follows:

- We abstract and define the common selection problem that is tackled by various relatedness heuristics and show how each heuristic addresses this retrieval problem.
- We identify data categories where the assumptions made by each heuristic are valid and draw some insights on their possible effectiveness on tree structured XML data.
- We show that the majority of these heuristics are not directly applicable on graph-structured XML data (where there are direct reference links, such as $IDREFs$ and $XLINKs$) and argue that different mechanisms need to be applied for such data.

The rest of the paper proceeds as follows: In section 2, we survey the related work in the field of flexible querying over XML data and define the common retrieval problem. In Section 3, we provide a detailed discussion on the effectiveness of the relatedness heuristics under consideration and we present our conclusions in Section 4.

2 Background

The ability to perform effective XML data retrieval in the absence of schema knowledge has recently received considerable attention and several different mechanisms to support flexible XML data retrieval have already appeared. These mechanisms can be classified into three main categories. In this section, we give an overview of each such category and describe the different retrieval contexts within which the heuristics that we consider in this paper have been proposed. We then abstract and define the common retrieval framework in the context of which each heuristic is evaluated.

The first category includes techniques [6,2] that start with a given XQuery or XPath query over which they perform a number of predefined relaxation primitives in order to create new queries, which produce a superset of the results

produced by the original query. These primitives provide relaxations on (a) the structure of the query [6,2], such as deletion of a node or conversion of a *child* axis to a *descendant* one etc, (b) the query node labels (based on semantic relationships derived by an ontology), and (c) the query value constraints [2]. The need to identify and return the most relevant answers among all the results that are produced by the relaxed queries has led to the development of ranking models [2], which quantify the similarity of the relaxed queries to the original query. The techniques in this category start with a set of given relationships among nodes and relax/modify them.

The second category contains a number of specialized search engines [16,10,5,8,19,7,17,9], which implement keyword search over XML data. Systems in this category identify sets of nodes that are relevant to the user query (a node is relevant if its content and/or label satisfy a query keyword) and have close associations with each other. Subsequently, they return either the identified sets [16,10,5,17,7] or appropriate subtrees that contain the nodes in these sets [8,19,9]. Some of them employ *relatedness heuristics* in order to choose combinations of relevant nodes that have close associations [8,5,19], while the majority produces all possible combinations of relevant nodes and utilizes a score function to choose the most relevant node combinations. Distance-based ranking functions [10,5,8,9,7,17] are usually used for this purpose.

Finally, the third category includes techniques [11,13,20] that enhance existing XML retrieval languages (i.e. XPath and XQuery [18]) with new operators, which enable the user to set the context of a particular query, even if she has only partial knowledge of the schema. They provide new *relatedness heuristics* (i.e. the concept of the *Meaningful Lowest Common Ancestor*, the *Amoeba operator*, and the *Closest axis*) and retrieval algorithms, which enable the identification of groups of closely associated nodes in given sets of nodes.

The techniques from the second and the third categories that we consider in this work start with an initial set of nodes whose relationships are to be established, and use relatedness heuristics to identify meaningful relationships. Moreover, some of these techniques [8,5] employ ranking functions during the retrieval process. The roles of the relatedness heuristics and the output ranking are complementary as the formers define what constitutes a meaningful answer, while the latter determines the order with which the results are presented to the user. In this work we focus on the relatedness heuristics and their effectiveness in identifying meaningful node relationships.

Retrieval Framework. At first glance, the comparison of the *relatedness heuristics* under consideration seems impossible due to the lack of a common querying framework within which each one is to be evaluated. Indeed each heuristic has so far been used for different and not directly comparable tasks. For example, the *MLCA* [11] heuristic has been used in order to set the context of a given XQuery query, while the *SLCA* [19] identifies the roots of the subtrees to be returned in response to a keyword search query. Nevertheless, they have two common operations: The first one identifies sets of nodes that satisfy some of the conditions in the user query. We will refer to these sets as *input sets*.

The second step identifies subsets of nodes from the input sets that are meaningfully related.

The input sets can be derived from appropriate retrieval conditions that exist in the user query. Each of the works under consideration specifies one of the following mechanisms to express the retrieval conditions:

- In the techniques proposed in [11], [13] and [20], the tags of the nodes that exist in each input set are explicitly specified using standard XQuery/XPath [18] node selection predicates. An input set contains those and only those nodes that satisfy the corresponding query predicates.
- In [5] the input set nodes are identified by using search terms to specify the tags and/or the content of the nodes. An input set contains those and only those nodes that satisfy the corresponding search term.
- [19] and [8] use keywords to identify input sets. An input set contains those and only those nodes whose label is equal to the corresponding keyword ([19]) or whose content contains that keyword ([8,19]).

The second step can be described abstractly as follows:

Problem Description 1. *Let the set of nodes in an XML document be N. Given m input sets (constructed in step 1) $N_i \subseteq N, i = 1, ...m$, determine the set of tuples $\{< n_1, ..., n_m > \mid < n_1, ..., n_m > \in N_1 \times N_2 \times ... \times N_m\}$ of meaningfully related nodes in the context of N.*

We note that the exact definition of *meaningfully related nodes* is different for each of the relatedness heuristics that we consider in this paper and will be described in detail in subsequent sections. We first focus our discussion on the application of the relatedness heuristics when $m = 2$. Subsequently, we extend our conclusions to cover the more general case of identifying related node sets with cardinality more than two as well.

Having described the common retrieval problem under which we are going to evaluate each of the relatedness heuristics we continue with a detailed description of each heuristic and the discussion of its relative strengths and limitations. Prior to that, we describe a simple retrieval language, which we use to express the example queries presented in the rest of the paper, as there is no uniform language used by the works in question. The language, whose syntax has been borrowed from [5], is as follows:

Definition 1 (Retrieval Query). *Let the set of nodes in an XML document be N. A query Q is a list of terms $(t_1, .., t_m)$. Each term is of the form: l::k, l::, ::k, or k, where l is a label and k a keyword. A node satisfies a term of the form:*

- *l::k if its label equals l and it contains the word k.*
- *l:: if its label equals l.*
- *::k if it contains the word k*
- *k if its label equals k or it contains the word k.*

Each term results in a separate input set containing the nodes that satisfy the term. The answer to Q is the set of tuples $\{<n_1, .., n_m>\}$*, where* $n_i \in N$ *and* n_i *satisfies* t_i *with the additional constraint that the nodes in a result tuple are meaningfully related.*

The above definition is slightly modified for the techniques that use XPath/XQuery local selection predicates for node retrieval (i.e., [11] and [13]) so as to use the XPath/XQuery [18] equality semantics when matching a keyword in a term. In this case, a node satisfies a keyword if the node's content is equal to the keyword.

3 Heuristics for Flexible Querying over XML Data

Definition 1 in Section 2 contains a requirement, namely, the *meaningful relatedness* of two nodes, whose definition depends on users' needs. In standard XML querying systems, the user describes how two nodes are related by specifying the exact structural relationships between them. In the absence of this information, retrieval systems that use the heuristics discussed in this paper assume that the user is looking for *meaningfully related* nodes that are relevant to her query. In what follows we first provide an overview of the XML node relatedness heuristics that have appeared in the literature and then discuss and compare their effectiveness under different structures and semantics of XML documents.

3.1 Relatedness Heuristics

The heuristics that we discuss in this paper include the *M*eaningful *L*owest *C*ommon *A*ncestor ($MLCA$ [11]), the *S*mallest *L*owest *C*ommon *A*ncestor ($SLCA$ [19][15][12]), the heuristic employed by the *Amoeba* operator [13], the one used in the XRank system [8], the *interconnection relationship* [4,5,3], and the *closest axis* [20]. All heuristics make the conjecture that only nodes that reside within the same document are meaningfully related while nodes that reside within different document are unrelated. Moreover, with the exception of the last heuristic, all heuristics start with the assumption that the subtree rooted by the lowest common ancestor of any two a and b nodes (LCA(a,b)) ([14,1]), defines the semantic scope within which the two nodes co-exist. This is true as long as the XML document can be modeled as a tree. We begin our discussion with the same assumption. Subsequently, we discuss the case when the data has a graph structure. The heuristics operate over two input sets A and B as follows:

- The heuristic that is utilized by the `Amoeba` operator ([13]) states that two nodes $a \in A$ and $b \in B$ are meaningfully related if $LCA(a, b) = a$ or $LCA(a, b) = b$. In other words, two nodes are meaningfully related if one node is an ancestor of the other. We will call this heuristic the *Amoeba heuristic* from now on.
- The `interconnection relationship` ([4,5,3]) views each node as an instance of the real world entity identified by its label. The heuristic states that two nodes $a \in A$ and $b \in B$ are meaningfully related unless they are

descendants of two different instances of the same real world entity. To apply this heuristic, the tree that is rooted by $LCA(a, b)$ and consists only of the paths from $LCA(a, b)$ to a (we call this path p_1) and $LCA(a, b)$ to b (we call this path p_2) is constructed. The nodes a and b are meaningfully related unless there exist nodes n_1 and n_2 such that $n_1 \neq n_2$, $n_1 \in p_1, n_1 \neq a$ and $n_2 \in p_2, n_2 \neq b$, and n_1 and n_2 have the same label. We call this heuristic the *XSearch heuristic* from now on.

- The heuristic that is used by the *closest axis* [20] defines that a node $a \in A$ is meaningfully related to the closest (in terms of number of edges) $b \in B$. We call this heuristic the *Nearest Neighbor Heuristic (NNH)* from now on.
- The XRank system [8] identifies that two nodes $a \in A$ and $b \in B$ are meaningfully related if $\nexists y \in B$ such that $LCA(a, y)$ is descendant of $LCA(a, b)$ and $\nexists x \in A$ such that $LCA(x, b)$ is descendant of $LCA(a, b)$. We call this heuristic the *XRank heuristic*.
- The remaining two heuristics (i.e. $MLCA$ and $SLCA$) state that two nodes $a \in A$ and $b \in B$ are *meaningfully related* unless there exist two nodes $x \in A$ and $y \in B$, with $a \neq x$ or $b \neq y$ such that $LCA(x, y)$ is a descendant of $LCA(a, b)$. The intuition behind those two and the *XRank* heuristics is that smaller trees contain more meaningfully related nodes. The difference between the $MLCA/SLCA$ and the *XRank* heuristics is that the first two disqualify a pair of nodes $a \in A$ and $b \in B$ if there exists any other pair (x, y) with $x \in A$ and $y \in B$ and $LCA(x, y)$ being a descendant of $LCA(a, b)$, while *XRank* has the additional constraint that $x = a$ or $y = b$. Moreover, the $MLCA$ and $SLCA$ heuristics differ on the elements that may exist within the input sets A and B. In particular, for $MLCA$ each element in A has to have the same label, and the same holds for each element in B. $SLCA$, on the other had, does not impose any restriction on the labels of the elements in A and B.

We continue with the discussion over the effectiveness of identifying meaningfully related nodes that are relevant to the user query. In our discussion we try to understand under what conditions the heuristics either fail to identify all the desired results (in which case we say that the heuristics have *false negatives*) or produce results that are not meaningfully related (in which case we say that the heuristics produce *false positives*). Taking into consideration the fact that the heuristics that we discuss are employed in an environment where the user does not know the exact structure of the underlying data, we expect at times some ambiguity about the definition of the result set. In those cases, we believe that the system should let the user choose appropriate answers.

3.2 Effectiveness of Heuristics for Tree Structured XML Data

In this section, we compare the effectiveness of each heuristic under different data characteristics. In Table 1, we summarize the results of our discussion. The first column of the table describes the characteristics of the underlying XML data and the rest shows the effect that the particular characteristic may have on

Table 1. Effectiveness of Heuristics in terms of False Positives and False Negatives

	Amoeba	**XSearch**	**XRank**	**MLCA**	**SLCA**	**NNH**
Recursive Elements	FP+FN	FP		FN	FN	FP+FN
Sibling Relationships	FN					FP+FN
Synonym Labels	FN	FP	FP	FP	FP	FP+FN
Multiple Label Contexts	FN	FN				FP+FN
Optional Nodes	FN		FP	FP	FP	FP+FN
Nested Elements	FN			FN	FN	FP+FN
Different Relation Types	FN		FN	FN	FN	FP+FN
Labels as Relations	FN	FN	FN	FN	FN	FP+FN
Graph Data	FP+FN	FP+FN	FP+FN	FP+FN	FP+FN	FP+FN

the behavior of the heuristic. In particular, we denote with FP the fact that the heuristic might produce false positives and with FN the fact that the heuristic might have false negatives. Below we describe each of the rows in the table in detail. To perform our study, we choose a pair of two nodes whose relatedness needs to be decided. For ease of presentation we separate the cases where the nodes in question are along the same root-to-leaf path and when they are on different paths.

Nodes along the same root-to-leaf path: Assume that we have two nodes $a \in A$ and $b \in B$, where A and B are input sets and a is an ancestor of b, and we want to decide whether they are meaningfully related. The *XSearch* and *Amoeba* heuristics indiscriminately decide that the two nodes are meaningfully related. As a result, they do not create false negatives. However, they might produce false positives. For example, if a user issues the query $<department::, manager::>$ over the data of Figure 3a, where the *department* is a **recursive element** both heuristics will falsely decide that the nodes *department*(1) and *manager*(2) are meaningfully related.

The *NNH* considers only the distance between nodes, without paying attention to the structure of the data. For example, the fact that b is a descendant of a, probably conveys a strong relationship between them. Nevertheless, if a sibling element $x \in A$ of a exists with a smaller distance to a than b then the heuristic will falsely decide that a and b are not meaningfully related. As a result, NNH is susceptible to both false positives and false negative, irrespective of the data characteristics.

The *MLCA* and *SLCA* heuristics check if there exists an $LCA(x, y)$ of another pair of nodes $x \in A$ and $y \in B$, such that $LCA(x, y)$ is a descendant of $LCA(a, b)$ (i.e. node a). If such a pair of nodes exists, they will reject the original nodes as meaningfully related. In Figure 2, we illustrate the case where such an $LCA(x, y)$ exists and show all the possible positions that it can have relative to a and b. As shown in the figure, the $LCA(x, y)$ can be a descendant of $LCA(a, b) = a$ and (a) neither a descendant nor an ancestor of b or, (b) an ancestor of b or, (c) a descendant of b. In the second case, the heuristics will probably make a good decision, as the node b in this case is more related to $LCA(x, y)$ than to a. In

the other two cases, however, there is no apparent reason why the relationship of x and y should suppress the relationship of a and b.

As an example, consider the dataset in Figure 3a. If the user wants to identify pairs of *departments* and their *managers* she will issue the query $<department::, manager::>$ [1]. The input sets are $\{department_1, department_2\}$ and $\{manager_1, manager_2\}$. From the document, it is very easy to conclude that the pairs $(department_1, manager_1)$ and $(department_2, manager_2)$ are valid answers, as they describe two equally strong relationships. Nevertheless, both $MLCA$ and $SLCA$ will omit the relationship of $department_1$ with $manager_1$.

The example exposes an inherent weakness of the two heuristics when **recursive elements** exist in the document. This is due to the fact that the definition of both heuristics implies that if an LCA_1 is identified, then any pair of nodes whose LCA_2 is an ancestor of LCA_1 is automatically disqualified as a meaningful relation. As a result, in cases where recursive elements exist in the data, important relationships may be missed. This problem is more profound for the $SLCA$ heuristic where the sets A and B may contain nodes with different labels. For example, if in Figure 3b a user issues the query $<department, John>$ the $SLCA$ will miss the relationship between $department_1$ and its *manager*, only because there exists an employee in another department with the same name.

The *XRank* heuristic, which also depends on the relative position of $LCAs$, will discard the relationship between a and b only in the case of Figure 2b, which as we already explained, is probably a good decision.

Nodes within different root-to-leaf paths: If the two nodes a and b do not belong to the same path then there exists a unique $LCA(a, b) \neq a, b$. The *Amoeba* heuristic will decide that the two nodes are not related, irrespective of the data characteristics. This is too restrictive, as it misses many **sibling relationships** as well as other meaningful relationships that may exist between the nodes. An example that demonstrates this problem is presented in Figure 4, where the *Amoeba* heuristic will produce no results for the query $<name::, manager::>$.

The *XSearch* heuristic looks for ancestors of a and b with the same label in order to decide on their relatedness. In the absence of such nodes, it may redundantly combine them even if they are not meaningfully related. As an example, the usage of the *XSearch* heuristic for the query $<name::, address::>$ over the data of Figure 5a, returns the pairs $\{name_1, address\}$ and $\{name_2, address\}$, although only the first pair represents meaningfully related nodes. At times, the use of an ontology can enhance the effectiveness of this heuristic, by identifying groups of **synonym labels** that may represent the same world entity. In the previous example, in the presence of an appropriate ontology the system could decide that

[1] As already mentioned, the systems that use the $MLCA$ and $SLCA$ heuristics express the node selection conditions using different means (i.e. the $MLCA$ heuristic by XQuery variables binding to the nodes $//department$ and $//manager$, and the $SLCA$ heuristic by the set of keywords $\{department, manager\}$). However, for the purposes of our discussion, where we focus on the effectiveness of the heuristics to identify related nodes, it suffices to express them with the language that we described in Section 2. As a result, we will only be using this language from now on.

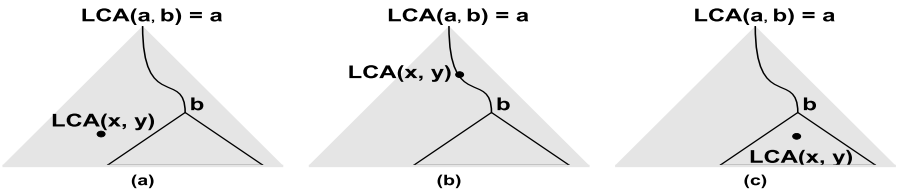


Fig. 2. Relative Position of LCA(x,y) within XML Subtree

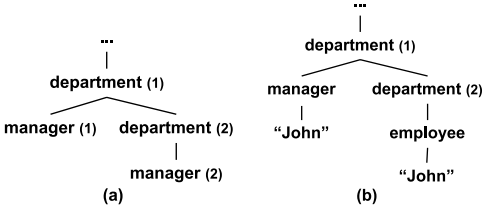


Fig. 3. False Negatives due to Recursive Elements

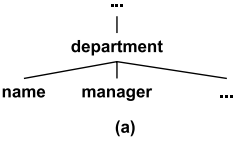


Fig. 4. FNs in Amoeba

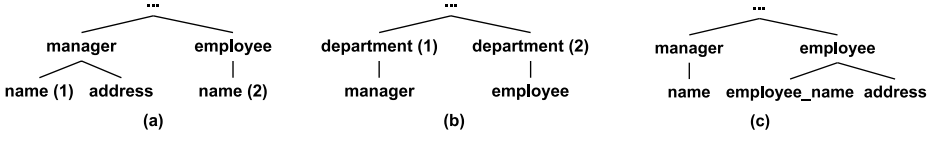


Fig. 5. (a) Synonym Labels (b) Missing Information, (c) Missing and Synonym Labels

manager ISA employee and as a result, treat the two labels as representing the same real world entity and correctly figure out that the $\{name_2, address\}$ pair does not represent a meaningful relationship. A more subtle problem with relying on the node label is that false negatives might be created due to **multiple label contexts** under which the same label might appear. In this case nodes with the same label that refer to different real world entities might cause related pairs of nodes to be falsely pruned. **Multiple label contexts** may also affect the performance of NNH as this heuristic pairs nodes based solely on the number of edges between them irrespective of the contexts within which they exist.

The $MLCA$, $SLCA$ and $XRank$ heuristics, on the other hand, do not pay attention to the label of the ancestors of a and b, and as a result would return the correct results in the previous example. Nevertheless, they may still create false positives when there is not enough information within the document for them to prune redundant relationships. This may happen, when **optional nodes** are present in the XML document. As an example consider the document fragment that is shown in Figure 5b. In this case, both *manager* and *employee* nodes are optional. If the user issues the query $< manager ::, employee ::>$ then the $SLCA$, $MLCA$ and $XRank$ heuristics will return the non meaningful

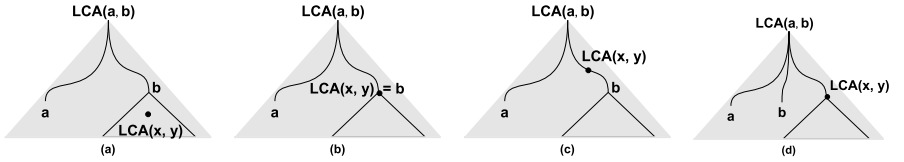


Fig. 6. Relative Position of LCA(x,y) within XML Subtree

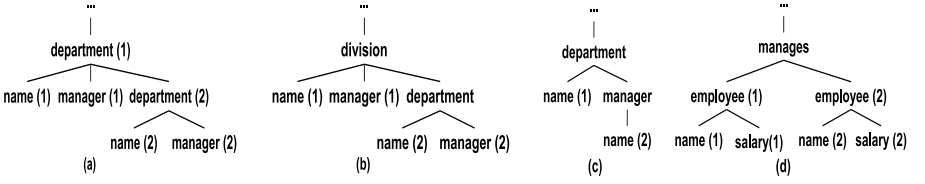


Fig. 7. (a) Recursive Elements (b) Nested Elements (c) Different Relations (d) Labels as Relations

$\{manager, employee\}$ pair. On the other hand, the *XSearch* heuristic has enough information to figure out that this is not a meaningful pair and will successfully discard it. The effect of optional nodes can be aggravated with the existence of **synonym labels**, as demonstrated in Figure 5c. If a user invokes the query $< name ::, address ::>$, all four heuristics will falsely return $\{name, address\}$, although semantically this address is related to the $employee\_name$ node.

Optional nodes can affect the behavior of NNH too. The heuristic will pair two elements $a \in A$ and $b \in B$, no matter how far away they are and how they are connected, as long as a is the closest node to b among all the other nodes in A. In the example of Figure 5b, it will also return the $\{manager, employee\}$ pair as a meaningful relationship.

The dependence of the $SLCA$ and $MLCA$ heuristics on the relative position between $LCAs$ alone can lead to problems of false negatives similar to the ones we encountered when the two nodes a and b belong to the same path. To explain this claim, we illustrate in Figure 6 the situation where an $LCA(x, y)$ exists and show all the possible positions it can have relative to the nodes a and b. As shown in the figure, the $LCA(x, y)$ can be (a) a descendant of b or, (b) equal to b or, (c) a descendant of $LCA(a, b)$ and an ancestor of b or, (d) a descendant of $LCA(a, b)$ and neither an ancestor of a nor b. In cases (a) and (d) there is again no reason why the relationship of x and y should suppress the relationship of a and b. For example, consider the datasets provided in Figure 7. In Figure 7a if a user issues the query $<name::, manager::>$ both heuristics will return only the $\{name_2, manager_2\}$ pair although $\{name_1, manager_1\}$ should also be returned, as it represents a relationship of the same type (i.e the name of a manager of a particular department). The same situation can emerge when the nodes under consideration belong to **nested elements**, as shown in Figure 7b.

Finally, the $MLCA$, $SLCA$, $XRank$ and NNH heuristics may produce false negatives when an a node in A participates in **different relationship types** with different nodes in B. Consider the example shown in Figure 7c. In this dataset, a *manager* node has two different meaningful relationships with the nodes $name_1$ and $name_2$. However, with the exception of the $XSearch$ search, all the heuristics under consideration will discard the $\{name_1, manager\}$ pair from the result of the query $<name::, manager::>$. A more subtle situation is illustrated in the dataset of Figure 7d, where the label of the node *manages* is used to define a relationship among other nodes. In this case (i.e. which presents cases when we have node **labels as relations**) if the user issues the query $<name::, salary::>$, none of the heuristics will manage to return the cartesian product of the input sets, which should be returned as all the pairs of *name* and *salary* elements have meaningful relationships.

Heuristics over Multiple Input Sets: We have discussed thus far the effectiveness of the relatedness heuristics for two input sets. In what follows, we briefly describe the behavior of the heuristics when more than two input sets exist. We omit the NNH from the discussion, as it has only been defined for the case of two input sets. Assume that we have a set of nodes $S = \{n_1, ...n_m\}, m > 2$.

The *Amoeba* heuristic states that S contains related nodes if one of them is the LCA of the others. As a result, it again suffers from the limitation of not being able to identify all possible relationships between sibling nodes.

The $XSearch$ heuristic, on the other hand, states that S contains related nodes if either (a) $\forall\{n_i, n_j\}, n_i \in S, n_j \in S$, n_i and n_j are meaningfully related according to the $XSearch$ heuristic for pairs of nodes or, (b) there exists a node n in the document so that $\forall\{n_i, n\}, n_i \in S$, n_i and n are meaningfully related according to the $XSearch$ heuristic. Consequently, for each pair of nodes that needs to be checked, the heuristic presents the same behavior that we described in the previous section.

The $MLCA$ of S ($MLCA_S$) is the LCA of $MLCA(n_i, n_j)$, $\forall n_i, n_j \in S, n_i \neq n_j$, if they all exist, and $\exists n_a, n_b \in S$ such that $MLCA_S = MLCA(n_a, n_b)$. This definition is again stated in terms of $MLCAs$ of pairs of nodes for which the relevant discussion in the previous section still holds.

Finally, the $SLCA$ and $XRank$ heuristics employ the LCA of the nodes in S (LCA_S). In this case, the nodes in S are related unless there exists a set S', such that LCA'_S is a descendant of S. The discussions in the previous section on recursive, optional and nested elements can be easily generalized, when we consider sets of more than two nodes.

3.3 Effectiveness of Heuristics for Graph Structured XML Data

So far we have focused our discussion on the tree model of XML data, which omits IDREFs, value-based relationships (such as primary-foreign key relationships), and other *direct reference links* that may exist within or among XML documents. With the exception of NNH, the heuristics that we consider in this paper omit such types of links. Nonetheless, considering the fact that each link

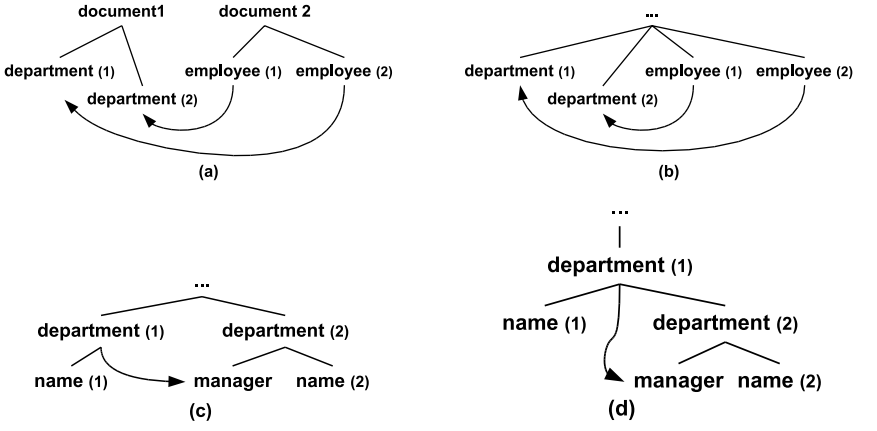


Fig. 8. Behavior of Relatedness Heuristics in the Presence of Non-Tree Edges

conveys an explicit and strong semantic relationship between two nodes, they may play an important role in identifying meaningfully related nodes in graph-structured XML data as well. As a result, it is interesting to see the behavior of the relatedness heuristics that we consider in the presence of those links. In what follows we omit NNH from the discussion as it does not differentiate between the tree and graph models. Consequently, the conclusions that we have drawn in the previous section about NNH hold for the graph model as well.

In the case of graph structured data, two nodes might be related with multiple relationships. In this case, the user might be interested in inspecting all of them, and not the one that the particular heuristic will return. Furthermore, under certain circumstances, when there are direct reference links between nodes, the LCA, as defined by the heuristics under consideration, should not be considered when checking node relatedness. That last issue is illustrated in Figures 8a and 8b. Figure 8a shows an example where the information on *departments* and *employees* of a company resides within two different XML documents. Value-based relationships, which are illustrated with arrows in the figure, create the links between each department and the employees that work in that department. If a user issues the query $<department, employee>$ none of the relatedness heuristics will return any results. However, from the figure it is apparent that the $\{department_1, employee_2\}$ and $\{department_2, employee_1\}$ pairs should be returned. The reason for the false negatives is the fact that the information resides in different documents, and, as a result, the notion of the LCA is not applicable. In Figure 8b, a similar situation is depicted where the *department* and *employee* information reside in the same document. In that case, the $XSearch$, $XRank$, $SLCA$ and $MLCA$ heuristics would return all possible combinations of *department* and *employee* elements when used to answer the previous query (the *Amoeba* heuristic still returns no results, due to the limitation of missing relationships among siblings that we described in the previous paragraphs). The reason for the false positives in this case is the fact that the LCA of nodes, although present, is only an artificial element used to create a

valid XML document and, as a result, conveys no information about the relationships amongst the underlying nodes.

The two examples that we described above summarize situations where the direct reference links are the only ones that should be checked in order to decide the relatedness of nodes. As expected, there also exist situations where the information provided by both the direct reference links and the LCA should be combined in order to determine node relatedness. Figures 8c and 8d illustrate examples where the omission of direct reference links results in false negatives. In particular, in Figure 8c when applying the $XSearch$ heuristic for the query $<manager, name>$, only the $\{manager, name_2\}$ pair would be returned although the pair $\{manager, name_1\}$ represents a meaningful relation as well.

The above discussion, although preliminary, illustrates the need to combine the information provided by $LCAs$ and direct reference links, when looking for related nodes in order to avoid missing important relationships. It is worthwhile to investigate the possibility of extending the relatedness heuristics to take into consideration such links as well, as these always express strong semantic relationships among nodes.

4 Conclusions

In this paper, we performed a qualitative evaluation of the effectiveness of the various node relatedness heuristics, namely NNH, $XSearch$, $XRank$, $SLCA$, $MLCA$ and $Amoeba$, that have been proposed in the context of flexible retrieval/querying of XML data. Their importance is attributed to the fact that they enable the creation of efficient and convenient tools for data retrieval in the absence of schema information.

We abstracted the problem of deciding node relatedness, which is addressed by the heuristics considered in this paper, in the context of schema free querying of XML data. Subsequently, starting with the tree model of XML data, we studied their behavior with respect to varying structure and semantics of the underlying data. Our discussion revealed the circumstances under which each heuristic is likely to have false positives and/or false negatives. We showed that their effectiveness depends on the application and data characteristics. We believe that our findings can help systems implementors understand the strengths and weaknesses of each heuristic and provide simple guidelines for the applicability of each one.

Finally, we demonstrated the ineffectiveness of the majority of the current heuristics on graph structured data. Taking into consideration the increased complexity that the graph model imposes on a flexible retrieval system, we believe that the investigation of additional relatedness heuristics that operate over XML graphs is an interesting path for future research.

Acknowledgments

The authors would like to thank Donald Chamberlin for his many useful suggestions as well as the anonymous reviewers for their insightful comments.

References

1. Amato, G., Debole, F., Rabiti, F., Savino, P., Zezula, P.: A Signature-Based Approach for Efficient Relationship Search on XML Data Collections. In: Proc. of XSym, Toronto, Cananda, pp. 82–96 (2004)
2. Amer-Yahia, S., Lakshmanan, L.V., Pandit, S.: FleXPath: Flexible Structure and Full-Text Querying for XML. In: Proc. of SIGMOD, Paris, France, pp. 83–94 (2004)
3. Cohen, S., Kanza, Y., Kimelfeld, B., Sagiv, Y.: Interconnection Semantics for Keyword Search in XML. In: Proc. of CIKM, Bremen, Germany (2005)
4. Cohen, S., Kanza, Y., Sagiv, Y.: Generating Relations from XML Documents. In: Proc. of ICDT, Siena, Italy (2003)
5. Cohen, S., Mamou, J., Kanza, Y., Sagiv, Y.: XSEarch: A Semantic Search Engine for XML. In: Proc. of VLDB, Berlin, Germany, pp. 45–56 (2003)
6. Delobel, C., Rousset, M.-C.: A Uniform Approach for Querying Large Tree-structured Data through a Mediated Schema. In: Foundations of Models For Information Integration Workshop (FMII) (2001)
7. Graupmann, J., Schenkel, R., Weikum, G.: The SphereSearch Engine for Unified Ranked Retrieval of Heterogeneous XML and Web Documents. In: Proc. of VLDB, Trondheim, Norway, pp. 529–540 (2005)
8. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: ranked keyword search over XML documents. In: Proc. of SIGMOD, San Diego, USA, pp. 16–27 (2003)
9. He, H., Wang, H., Yang, J., Yu, P.S.: BLINKS: Ranked Keyword Searches on Graphs. In: Proc. of SIGMOD, Beijing, China (2007)
10. Hristidis, V., Papakonstantinou, Y., Balmin, A.: Keyword Proximity Search on XML Graphs. In: Proc. of ICDE, Bangalore, India (2003)
11. Li, Y., Yu, C., Jagadish, H.V.: Schema-Free XQuery. In: Proc. of VLDB, Toronto, Canada, pp. 72–83 (2004)
12. Liu, Z., Chen, Y.: Identifying Meaningful Return Information for XML Keyword Search. In: Proc. of SIGMOD, Beijing, China (2007)
13. Saito, T., Morishita, S.: Amoeba Join: Overcoming Structural Fluctuations in XML Data. In: Proc. of WebDB, Chicago, USA, pp. 38–43 (2006)
14. Schmidt, A., Kersten, M., Windhouwer, M.: Querying XML Documents Made Easy: Nearest Concept Queries. In: Proc. of ICDE, Heidelberg, Germany, pp. 321–329 (2001)
15. Sun, C., Chan, C.-Y., Goenka, A.K.: Multiway SLCA-based Keyword Search in XML Data. In: Proc. of WWW, Singapore, Singapore (2007)
16. Theobald, A., Weikum, G.: The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. In: Proc. of EDBT, Prague, Czech Republic, pp. 477–495 (2002)
17. Theobald, M., Schenkel, R., Weikum, G.: An Efficient and Versatile Query Engine for TopX Search. In: Proc. of VLDB, Trondheim, Norway, pp. 625–636 (2005)
18. XQuery 1.0: An XML Query Language, W3C Recommendation, See (January 2007), `http://www.w3.org/TR/xquery`
19. Xu, Y., Papakonstantinou, Y.: Efficient Keyword Search for Smallest LCAs in XML Databases. In: Proc. of SIGMOD, Baltimore, USA, pp. 537–538 (2005)
20. Zhang, S., Dyreson, C.: Symmetrically Exploiting XML. In: Proc. of WWW, Edinburgh, Scotland (2006)

XML Schema Evolution: Incremental Validation and Efficient Document Adaptation

Giovanna Guerrini[1], Marco Mesiti[2], and Matteo A. Sorrenti[2]

[1] DISI - Università di Genova, Italy
guerrini@disi.unige.it
[2] DICO – Università di Milano, Italy
mesiti@dico.unimi.it

Abstract. XML Schemas describe the structure of valid documents and can be exploited for improving both the efficiency and effectiveness of queries on valid documents. XML Schemas, however, may need to be updated to adhere to new requirements and to face changes in the application domain. Starting from a set of schema modification primitives, in this paper we devise an incremental validation approach that allows to efficiently validate documents, known to be valid for the original schema, for an updated schema. Then, we enhance the approach to adapt the documents to the new schema. Experiments prove that our approach increases the performance of standard validation algorithms in this setting and that the cost of the adaptation process is limited.

1 Introduction

XML Schemas [19] describe the structure and the allowed content of XML documents. Since the contexts where XML is exploited are highly dynamic, XML Schemas frequently need to be updated to reflect changing requirements: systems need to be adapted to real-world changes, new functionalities need to be introduced, new data types need to be processed. XML data representation formats and domain-specific schemas, before being adopted as a standard, undergo several revisions resulting in many different versions and the need arises to adapt the corresponding documents.

XML Schemas can be updated in their basic components: elements declarations, simple and complex type declarations. In [11,12] a set of primitives for evolving XML Schemas has been proposed together with an analysis of the impact of such primitives on documents known to be valid for the original schema. Documents valid for the original schema, indeed, are no longer guaranteed to meet the constraints described by the evolved schema. In principle, these documents should be *revalidated* against the new schema. A naïve approach to revalidation consists in applying a standard validation algorithm (like MSXML, Xerces, and XSV) to each document and the new schema, that has been obtained by changing the original schema through an evolution primitive. This approach, however, does not take advantage of the fact that some evolution primitives are known not to impact document validity [11,12]. Moreover, also for primitives whose application can impact validity, the evolution most likely impacts a limited portion of the schema. Consequently, only restricted portions of a document need to be revalidated. The naïve approach, moreover, does not take into account that the document is

D. Barbosa et al. (Eds.): XSym 2007, LNCS 4704, pp. 92–106, 2007.

known to be valid for the original schema and that the possible effects on validity of a primitive can be foreseen. Thus, we propose in this paper an *incremental* validation approach for the validation of documents, known to be valid for an original schema sx, against an evolved schema obtained from sx through a specific evolution primitive.

If the evolution impacts validity, a related problem is how to *adapt* documents so to make them valid for the evolved schema. Adaptation by hand is error-prone and not feasible when the number of documents is high. In this paper we propose an approach in which documents are extended or pruned following a default behavior. Default adaptation is reasonable for simple evolution primitives and is very useful in some contexts, e.g., when documents are tests for statistical benchmarks.

The main contributions of this paper are an algorithm for the incremental validation of XML documents upon XML Schema evolution and an efficient algorithm for adapting the documents to the evolved schema. They have been implemented in X-Evolution [15] and experimentally evaluated. Our incremental validation algorithm outperforms the .NET validation algorithm for primitives that do not alter document validity and improves of an average 20% for other primitives. The execution time of document adaptation linearly depends on the document size.

In the remainder of the paper, Section 2 briefly surveys related work. Section 3 introduces XML Schemas and evolution primitives. Section 4 discusses validation and adaptation of complex type structures. Section 5 presents the two algorithms, that are experimentally evaluated in Section 6. Section 7 concludes the work.

2 Related Work

Schema evolution has been investigated for schemas expressed by DTDs in [14], where a set of evolution operators is proposed and discussed in detail. Problems caused by DTD evolution and the impact on existing documents are however not addressed. Moreover, since DTDs are considerably simpler than XML Schemas [5] the proposed operators do not cover all the kind of schema changes that can occur on an XML Schema. DTD evolution has also been investigated from a different perspective in [4,7]. The focus was on dynamically adapting the schema to documents. In [7] document updates invalidating some documents can lead to changes in the DTD. In [4], by contrast, modifications to the DTD are deduced by means of structure mining techniques extracting the most frequent document structures, in a context where documents are not required to exactly conform to a DTD.

In [9,18] approaches for making an XML document valid to a given DTD, by applying minimal modifications detected relying on tree edit distances, have been proposed. No knowledge of conformance of the document to a DTD is however exploited. The problem of document revalidation has been investigated in [16]. Documents to be revalidated may not be available in advance, they are known to be valid for a given schema sx_1 and must be revalidated against a different schema sx_2, but the transformations leading from sx_1 to sx_2 are not known. Incremental validation of XML documents, represented as trees, has been investigated for atomic [1,2,6] and composite [3] XML updates. Given an update operation on an XML document, the update is simulated, and only after verifying that the updated document is still valid for its schema the update is executed. An extension of the incremental validation process to document correction

is proposed in [8] where upon validation failure local corrections to the document are proposed to the user.

3 XML Schemas and Evolution Primitives

XML Schemas. We adopt the XML Schema representation of [11,12] that extends the one proposed in [16]. Let $\mathcal{EN}$ denote the set of element tags and $\mathcal{TN}$ the set of (both simple and complex) type names. Set $\mathcal{TN}$ is the union of the disjoint sets $\mathcal{TT}$ and $\mathcal{AT}$, where $\mathcal{TT}$ is the set of explicitly assigned type names and $\mathcal{AT}$ is the set of system-assigned type names (to identify anonymous types).

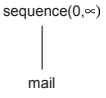
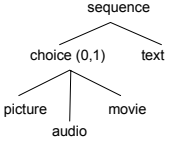
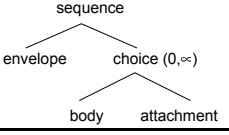
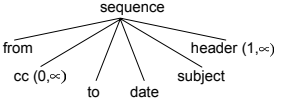
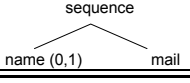
Simple types, named $\mathcal{ST}$, can be XML Schema native types in the set $\mathcal{NT}$ or can be derived through `restrict`, `list`, and `union`. Each simple type is characterized by a set of *facets* allowing to state constraints on its legal values. We assume the presence of a predicate f that represents the constraints imposed by a set of facets. The set of simple types is inductively defined as follows: native types (e.g., `decimal`, `string`, `float`, `date`) are simple types; if τ is a simple type, $\texttt{list}(\tau)$ is a simple type; if $\tau_1, \ldots, \tau_n$ are simple types, $\texttt{union}(\tau_1, \ldots, \tau_n)$ is a simple type; if τ is a simple type and f is a predicate on the facets applicable on t, $\texttt{restrict}(\tau, f)$ is a simple type. The set of legal values for type τ is denoted by $[\![\, \tau \,]\!]$. Given $\tau_1, \tau_2 \in \mathcal{ST}$, the relationship $\tau_1 \sqsubseteq \tau_2$ denotes $[\![\, \tau_1 \,]\!] \subseteq [\![\, \tau_2 \,]\!]$ and can be determined by exploiting the built-in native types hierarchy [19] and standard constraint subsumption approaches [17] when facets occur.

Complex types, named $\mathcal{CT}$, are associated with a structure specifying the possible children of a given element. A type structure is represented through a labelled tree. A tree on a set of nodes N is inductively defined by stating that: (i) $v \in N$ is a tree (whose root is v); and (ii) if $T_1, \ldots, T_n$ are trees and $v \in N$ is a node not appearing in $T_1, \ldots, T_n$, $(v, [T_1, \ldots, T_n])$ is a tree (whose root is v). Given a node $v \in N$, $children(v)$ denotes the list of subtrees of v. A labelled tree is a pair (T, φ), where T is a tree and φ is a total function from the set of T nodes to a set of labels. Labels of the tree representing a type structure are pairs (l, γ), where $l \in \mathcal{EN} \cup \mathcal{OP}$ and $\gamma \in \Gamma$. The set $\mathcal{OP} = \{\texttt{SEQUENCE}, \texttt{ALL}, \texttt{CHOICE}\}$ contains the operators for building complex types. The `SEQUENCE` operator represents a sequence of elements, the `CHOICE` operator represents an alternative of elements, and the `ALL` operator represents a set of elements without order. By contrast, the set $\Gamma = \{(min, max) \mid min, max \in \mathbb{N}, min \leq max\}$ contains occurrence constraints, where min is the attribute `MinOccurs` and max is the attribute `MaxOccurs`. The default value $(1, 1)$ is not shown in our graphics. Given a tree T, let $root(T)$ denote the root of T, $l(T)$ denote the label of $root(T)$, and $l_{|_i}(T)$, $i = 1, 2$ denote the i-th component of $l(T)$.

A *type structure* is a tree T defined on the set of labels $(\mathcal{EN} \cup \mathcal{OP}) \times \Gamma$ for which:

1. $l(T) \in \mathcal{OP} \times \Gamma$;
2. for each subtree $(v, [T_1, .., T_n])$ of T, $l(v) \in \mathcal{OP} \times \Gamma$;
3. for each leaf v of T, $l(v) \in \mathcal{EN} \times \Gamma$;
4. for each subtree $(v, [T_1, \ldots, T_n])$ of T, if $l(v) = \langle \texttt{ALL}, (min, max) \rangle$ then $v = root(T)$ and $\forall i \in \{1, \ldots, n\}\ l(T_i) \in \mathcal{EN} \times \Gamma$, $l_{|_2}(T_i) = (min_i, 1)$, and $\forall j \in \{1, \ldots, n\}\ i \neq j \Rightarrow l_{|_1}(T_i) \neq l_{|_1}(T_j)$.

Table 1. `mail` schema representation

$\mathcal{EN}_G = \{mails, attachment\}, \mathcal{T} = \{mailT, envelopeT, personT\} \cup \{\tau_1, \tau_2\}$ $type_G(mails) = \tau_1, type_G(attachment) = \tau_2$		
$\rho(\tau_1)$		$mail \mapsto mailT$
$\rho(\tau_2)$		$picture \mapsto Byte$ $audio \mapsto Byte$ $movie \mapsto Byte$ $text \mapsto string$
$\rho(mailT)$		$envelope \mapsto envelopeT$ $body \mapsto string$
$\rho(envelopeT)$		$from \mapsto personT$ $cc \mapsto personT$ $to \mapsto personT$ $date \mapsto date$ $subject \mapsto string$ $header \mapsto string$
$\rho(personT)$		$name \mapsto string$ $mail \mapsto string$

The last condition imposes that `ALL` labelled nodes can only appear as children of the root element and that their children must be all distinct non repeatable elements. Let $\mathcal{T}_S$ be the set of all possible type structures.

An *XML Schema* consists of a set of global type and element definitions. As discussed above, local elements can be declared within a type definition. XML Schemas, unlike DTDs, allow an element to have different types depending on its context, but a unique type is assigned to each element of the schema depending on its context (global or local to a type τ). A *consistent XML Schema* is a 4-tuple $(\mathcal{EN}_G, \mathcal{T}, \rho, type_G)$, where:

- $\mathcal{EN}_G \subseteq \mathcal{EN}$ is the set of labels of global elements,
- $\mathcal{T} = (\mathcal{TT} \cup \mathcal{AT}) \subseteq \mathcal{TN}$ is the set of type names,
- ρ is a function associating each $\tau \in \mathcal{T}$ with its declaration, that is:
  - if $\tau \in \mathcal{ST}$, $\rho(\tau) \in \mathcal{NT} \cup \{\texttt{restrict}(\tau_1, f) \mid \tau_1 \in \mathcal{ST}\} \cup \{\texttt{list}(\tau_1) \mid \tau_1 \in \mathcal{ST}\} \cup \{\texttt{union}(\tau_1 \ldots \tau_N) \mid \tau_1, \ldots, \tau_n \in \mathcal{ST}\}$;
  - if $\tau \in \mathcal{CT}$, $\rho(\tau) = (\mathcal{EN}_\tau, t, type_\tau)$, where: $\mathcal{EN}_\tau \subseteq \mathcal{EN}$ is the set of local element names for τ; $t \in \mathcal{T}_S$ is the structure for τ; function $type_\tau : \mathcal{EN}_\tau \rightarrow \mathcal{T}$ assigns each local element of t its type;
- $type_G : \mathcal{EN}_G \rightarrow \mathcal{T}$ is a function assigning each global element its type.

Example 1. Table 1 shows the representation of our mail schema. The first row reports the sets of global element and type names, and function $type_G$ that associates each global element with the corresponding type. Then, for each complex type τ, its

Table 2. Classification of the evolution primitives

	Insertion	Modification	Deletion
Simple Type	$insert\_glob\_simple\_type^*$ $insert\_new\_member\_type^*$	$change\_restriction$ $change\_base\_type$ $rename\_type^*$ $change\_member\_type$ $global\_to\_local^*$ $local\_to\_global^*$	$remove\_type^*$ $remove\_member\_type^*$
Complex Type	$insert\_glob\_complex\_type^*$ $insert\_local\_elem^\circ$ $insert\_ref\_elem^\circ$ $insert\_operator^\circ$	$rename\_local\_elem$ $rename\_type^*$ $change\_type\_local\_elem^\circ$ $change\_cardinality^\circ$ $change\_operator^\circ$ $global\_to\_local^*$ $local\_to\_global^*$	$remove\_element^\circ$ $remove\_operator^\circ$ $remove\_substructure^\circ$ $remove\_type^*$
Element	$insert\_glob\_elem$	$rename\_glob\_elem^*$ $change\_type\_glob\_elem$ $ref\_to\_local^*$ $local\_to\_ref^*$	$remove\_glob\_elem^*$

definition $\rho(\tau)$ is provided. Specifically, the type structure t and the function $type_\tau$ that associates each local element name in $\mathcal{EN}_\tau$ with the corresponding type are given. ○

A schema is said to be *conflict-free* when in type definitions subelement names appear only once. Efficiency of incremental validation proposals discussed in Section 2 is bound to the conflict-free property of the schema. In this paper we will restrict to conflict-free schemas, both for what concerns the original schema and the evolved one. Most schemas employed on the Web do exhibit this property [10].

A unique type can always be associated with an element, given the context (global or local to a complex type). When no ambiguity arises, we will denote $type$ (thus omitting the subscript) the function that associates a (global or local) element with its type. Function $valid$ in the remainder of the paper denotes the validity of a document w.r.t. a schema or of an element w.r.t. a type, evaluated through a standard validation approach. Function $getPaths$ takes as input parameters t and e; t can be either a type, a type structure, or an element tag, whereas e is an element in the schema. It returns the XPath expressions only consisting of steps along the `child` axis from the root of the schema, passing through e, reaching an element of type, structure, or tag t. Referring to the mail schema in Table 1, $getPaths(personT, mail) = \{\ /mails/mail/envelope/from$, $/mails/mail/envelope/to, /mails/mail/envelope\ /cc\}$. Function $getPaths$ may return different sets of paths depending on the context in which it is invoked. For example, different paths are returned for element $mail$ in τ_1 and $personT$. Note that the returned set is finite also in case of recursive schemas. By contrast, function $getElems$ evaluates a set of XPath expressions (again, only consisting of / steps) on a document and returns the corresponding elements.

Evolution Primitives. In [11,12] three categories of atomic primitives have been devised: insertion, modification, and deletion of the XML Schema components (simple types, complex types, and elements). Modifications can be further classified in structural and relabelling modifications. Structural modifications allow to modify the structure of a type (subelements, operators, and cardinality constraints) while relabelling modifications allow to change the name of an element/type. Table 2 reports the evolution primitives $\mathcal{P}$ relying on the proposed classification. For simple types the operators

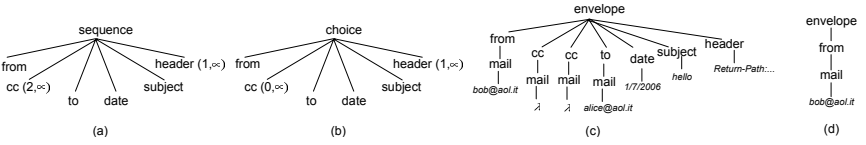


Fig. 1. Type structures with valid elements

are further specialized to handle the derived types `restrict`, `list`, and `union`. Primitives marked $*$ in Table 2 (denoted by $\mathcal{P}^*$) do not alter the validity of documents, whereas primitives marked $\circ$ in Table 2 (denoted by $\mathcal{P}^\circ$) operate on a type structure. Primitives are associated with *applicability conditions* that must hold before their application to guarantee that the updated schema is still consistent. For example, global types/elements can be removed only if elements in the schema of such a type (or that refer to such element) do not exist. Moreover, when renaming an element in a complex type τ, an element with the same tag should not occur in τ. These conditions are verified before applying the schema update.

Example 2. Let t be the structure of type `envelopeT` of schema sx in Table 1. By applying the primitive $p_1 = \texttt{change\_cardinality}((2,\infty), 3, t, sx)$, the type structure t_1 in Fig. 1(a) is obtained. By applying $p_2 = \texttt{change\_operator}(\texttt{choice}, 1, t, sx)$, the type structure t_2 in Fig. 1(b) is obtained. These primitives considers the local rank of nodes to identify the node in a type structure to be updated. ○

4 Type Structures Validity and Adaptation

The type structure t of a type $\tau \in \mathcal{CT}$ determines which subelements can occur, and in which order, in a document element declared of type τ. In this section we first consider validation and adaptation with respect to a single type structure and then introduce a relationship among type structures that will be used in the validation process.

Function $validS$. This function checks whether a sequence of document elements adheres to a type structure by taking as input: a list of sibling elements $[T_1, \ldots, T_n]$ in a document, a structure t, and a set S of expected element tags relying on t. The set of expected element tags is initially determined by an auxiliary function $init : \mathcal{T_S} \rightarrow 2^{\mathcal{EN} \cup \{\lambda\}}$. This function returns the set of tags S initially expected by $t \in \mathcal{T_S}$. S can contain the symbol λ denoting that t also admits empty content. More than one tag can occur in S because of the presence of `choice` and optional elements in t. Once the first tag of the list of sibling elements matches a tag in S, the next expected tags for t are determined by function $nxtEls : \mathcal{EN} \cup \mathcal{T_S} \rightarrow 2^{\mathcal{EN} \cup \{\lambda\}}$. This function takes as input the identified tag $l \in S$ and $t \in \mathcal{T_S}$, and returns the next set of expected tags. Consider the type structure t_2 in Table 1, $init(t_2) = \{\texttt{picture}, \texttt{audio}, \texttt{movie}, \texttt{text}\}$, $nxtEls(\texttt{text}, t_2) = \{\lambda\}$ whereas $nxtEls(\texttt{audio}, t_2) = \{\texttt{text}\}$. Function $validS$ works by using t as an automaton for accepting $[T_1, .., T_n]$. The three cases in its definition correspond, respectively, to symbol acceptance and next state transition; acceptance and transition to a final state; non acceptance. Function $validS$ is defined as follows:

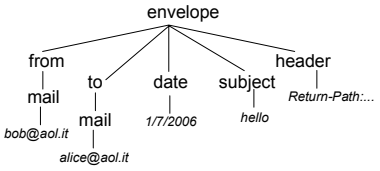


Fig. 2. The `envelope` element

$$validS([T_1,..,T_n],S,t)=\begin{cases} validS([T_2,..,T_n],nxtEls(l_1,t),t) & \text{if } l_1=l(T_1)\in S \wedge valid(T_1,type(l_1)) \\ \textbf{true} & \text{if } n=0, \lambda \in S \\ \textbf{false} & \text{otherwise} \end{cases}$$

Example 3. Consider the `envelope` element whose tree representation is shown in Fig. 2 and the structure t of type `envelopeT` of Table 1. $validS$ is initially invoked on the five subelements of `envelope` $[T_1, \ldots, T_5]$, $\{$`from`$\}$, and t. Since $l(T_1) =$ `from` and T_1 is valid for `personT`, then $validS$ is invoked on $[T_2, \ldots, T_5]$, $\{$`cc`, `to`$\}$, and t. Since $l(T_2) \in \{$`cc`, `to`$\}$ and T_2 is valid for `personT`, then $validS$ is invoked on $[T_3, \ldots, T_5]$, $\{$`date`$\}$, and t. The behavior is analogous for the remaining elements and we can conclude that the `envelope` element is valid for `envelopeT`. ○

Function $adaptS$. This function is an extension of $validS$ that alters the list of subelements $[T_1, \ldots, T_n]$ of a document element when it is not valid for a structure t. Altering $[T_1, \ldots, T_n]$ means inserting and/or deleting elements to/from the list. The function has the same input parameters as $validS$ and an additional opt parameter stating if we want to insert or delete elements. This depends on the evolution primitive and will be discussed in next section. Here we present how insertions or deletions are performed. $adaptS$ exploits the auxiliary function $genTree$ that, given an element of type τ, generates a valid instance for such a type assigning default values for data content elements and choosing the minimal structure among those that can be obtained from τ. The `envelope` element in Fig. 2 where data contents are substituted by the empty string is an example of tree generated by $genTree($`envelope`, `envelopeT`$)$. Function $adaptS$ returns a list $\mathcal{C}$ of sibling document elements valid for t obtained from $[T_1, \ldots, T_n]$ through insertions or deletions.

If $n \geq 1$ and the label l_1 of the root of T_1 belongs to S, the algorithm checks whether the content of T_1 meets the constraints imposed by the type of l_1. If it does not, the content of T_1 should be generated, otherwise it is left unchanged. In both cases, the function returns T_1 concatenated to the list of trees generated by the recursive call of $adaptS$ on the remaining tree list and the next expected elements for t.

If $n \geq 1$ and the label of T_1 does not belong to S, in case of insertion (i.e., $opt =$ `INS`) a tag s is chosen from S and an element valid for the type of s is inserted at the head of the list (according to a policy discussed below) and the label of T_1 is checked in the next expected elements. In case of deletion (i.e., $opt =$ `DEL`) T_1 is removed and the label of the next element is checked in the same set S. Whenever $n = 0$ and $\lambda \notin S$ (that is, all the sibling elements have been processed but t requires other elements), new elements are appended to the result until $\lambda \in S$.

Algorithm 1. $adaptS$

Data: $[T_1, \ldots, T_n]$: Trees, $S : 2^{\mathcal{EN}}$, $t : \mathcal{ST}$, opt : {INS,DEL}
Result: $\mathcal{C}$: Trees $valid\ for\ t$

1 Let $l_1 = l(T_1)$
2 **if** $n \geq 1 \wedge l_1 \in S$ **then**
3 **if not** $valid(T_1, type(l_1))$ **then** $T_1 = genTree(l_1, type(l_1))$
4 **return** $T_1 \cdot adaptS([T_2, \ldots, T_n], nxtEls(t, l_1), t, opt)$
5 **end**
6 **if** $n \geq 1 \wedge l_1 \notin S$ **then**
7 **if** $opt =$ INS **then**
8 $s = choose(S)$
9 **return** $genTree(s, type(s)) \cdot adaptS([T_1, \ldots, T_n], nxtEls(t, s), t, opt)$
10 **end**
11 **else return** $adaptS([T_2, \ldots, T_n], S, t, opt)$
12 **end**
13 $\mathcal{C} \leftarrow \emptyset$
14 **while** $\lambda \notin S$ **do**
15 $s = choose(S)$
16 $\mathcal{C} \leftarrow \mathcal{C} \cdot genTree(s, type(s))$
17 $S \leftarrow nxtEls(t, s)$
18 **end**
19 **return** $\mathcal{C}$

When $|S| > 1$, an appropriate heuristic, which ensures to introduce only mandatory elements with the minimal number of occurrences, is applied to choose a tag in S [13].

Example 4. Consider element `envelope` in Fig. 2, structures t_1, t_2, and primitives p_1, p_2 of Example 2. $adaptS$ is invoked with option `INS` for p_1 on $[T_1, .., T_5]$, {`from`}, t_1. Since $l(T_1) =$ `from` and T_1 is valid for `personT`, $adaptS$ is invoked on $[T_2, .., T_5]$, {`cc`}, t_1. Since $l(T_2) =$ `to` $\notin$ {`cc`}, a tree is generated for `cc` and $adaptS$ is invoked on $[T_2, .., T_5]$, {`cc`}, t_1. Again, $l(T_2) =$ `to` $\notin$ {`cc`}, thus another tree is generated for `cc` and $adaptS$ is invoked on $[T_2, .., T_5]$, {`to`}, t_1. The remaining recursive calls return $[T_2, .., T_5]$. Fig. 1(c) shows the new element `envelope`. By contrast, $adaptS$ is invoked with option `DEL` for p_2 and same parameters. Since $l(T_1) =$ `from` and T_1 is valid for `personT`, $adaptS$ is invoked on $[T_2, .., T_5]$, $\{\lambda\}$, t_2. Since $l(T_2) =$ `to` $\notin \{\lambda\}$, T_2 is removed as well as the other elements of the list by the recursive invocations. Fig. 1(d) shows the new element `envelope`. ○

$\sqsubseteq_{ST}$ **relationship**. This relationship holds between a type structure t_1 and a type structure t_2, obtained from t_1 by applying a primitive $p \in \mathcal{P}^{\circ}$, when the legal values of t_1 are known to be contained in the legal values of t_2. This relationship can be established by considering ad-hoc rules like the following ones. If p changes the cardinality of an element/operator from (min_O, max_O) to (min_N, max_N) and $min_N \leq min_O \wedge max_N \geq max_O$ (that is, the interval of allowed occurrences is extended) the elements valid for t_1 are still valid for t_2. If p changes a `sequence` operator into an `all` operator or the group bound by any operator is composed by a single element, then the elements valid for t_1 are still valid for t_2. If p introduces a new optional element/operator in the structure, then the elements valid for t_1 are still valid for t_2. If none of the elements of sx have been defined according to a complex type whose structure is t_1, then no modification to t_1 alters the validity of documents. The $\sqsubseteq_{ST}$ relationship is thus exploited in the revalidation process to avoid accessing documents when it is not strictly required.

Algorithm 2. Revalidate

```
1  input: p : P, d : DOC, sx : SX
2  output: true  <=>  d is valid for the updated schema
3  switch case p // The applicability conditions of p to sx are met
4    case (p ∈ P*) return true
5    case (p ∈ {rename_glob/local_elem})
6              let l be the element tag to remove/rename
7              return (getElems(getPaths(l, sx), d) = ∅)
8    case (p = change_type_glob/local_elem(l, τ_N, sx) ∧ τ_N ∈ CT)
9              let t_N be the structure of τ_N
10             return (∀e ∈ getElems(getPaths(l, sx), d) : validS(children(e), init(t_N), t_N))
11   case(p ∈ {change_restrict, change_base/member_type, change_type_glob/local_elem})
12             let τ_O be the old simple type and τ_N the updated one
13             if (τ_O ⊑ τ_N) then return true end-if
14             return (∀e ∈ getElems(getPaths(τ_O, sx), d) : content(e) ∈ [[ τ_N ]])
15   case(p = remove_glob_elem(l, sx))
16             return (l(root(d)) ≠ l)
17   case(p ∈ P°)
18             let t, t_N be the old and new structure
19             if (t ⊑_ST t_N) then return true end-if
20             return (∀e ∈ getElems(getPaths(t, sx), d) : validS(children(e), init(t_N), t_N))
21 end-case
```

5 Incremental Validation and Efficient Document Adaptation

Incremental Validation Algorithm. The incremental validation algorithm takes as input a schema sx, a document d valid for sx, and an evolution operation $p \in \mathcal{P}$, and outputs true if and only if d is still valid after the application of p to sx. The algorithm, relying on the applicability conditions of the evolution primitives being satisfied, tries to determine document validity from the applied evolution primitive and the schema, moving to check the document only when this is strictly needed.

If $p \in \mathcal{P}^*$, its application does not alter the validity of d. Therefore, no checks need to be performed on d. If p renames an element tagged l, the validity of d depends on the occurrence of l elements in d. Therefore, whenever an l element occurs in d, it is no longer valid. If p changes the type of an element l to the complex type τ_N, the children of elements l in d are extracted and, through function $validS$, they are checked to meet the constraints specified by the structure of τ_N. If p changes a simple type (either changing the restriction, the base/member type, or the type of a global element) the algorithm first checks whether the values of the old type τ_O are contained in the new type τ_N. If so, d is valid, otherwise, all the elements of type τ_O in d are identified and their content is checked to be a legal value of τ_N. If p removes a global element l, the root label of d is compared with l. If they are equal, d is not valid. Otherwise, it is still valid. This check is very simple since the applicability conditions of the primitive allows the removal of a global element only if no elements in the schema refers to it. If p updates the structure of a complex type through the primitives in $\mathcal{P}^\circ$, the old structure t is compared with the new structure t_N to determine whether $t \sqsubseteq_{ST} t_N$. If so, d is valid. Otherwise, the children of elements with structure t in d are extracted to check through function $validS$ whether they meet the constraints specified by t_N.

Proposition 1. *Let sx be an XML Schema and d be an XML document valid for sx. Let sx_N be an XML Schema obtained from sx by applying $p \in \mathcal{P}$. Then,*

$$valid(d, sx_N) \textbf{ iff } revalidate(p, d, sx).$$

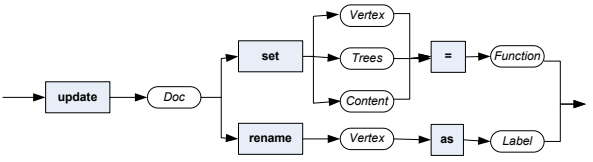


Fig. 3. An SQL-based language for the specification of document updates

Document Adaptation Algorithm. The document adaptation algorithm is an extension of the $revalidate$ algorithm (Algorithm 2) in which, when an element is not valid for the new schema, the minimal modifications are performed on d to make it valid. The modifications are minimal because they only involve the document portions affected by the primitive p and because they require to insert/eliminate the minimal number of elements to guarantee validity. Document modifications can be of different types: element renaming, removal of an element with all its content, insertion of an element. In the last case, a default value should be associated with the inserted element e (either a legal value for the (simple) type of e or a default tree generated by function $genTree$). Document modifications are specified by means of a simple SQL-based language whose syntax graph is shown in Fig. 3. Squared nodes represent keywords and oval nodes represent parameters. The new nodes and contents can be specified by means of functions $adaptS$, and $defaultVal$ that returns a default value for a simple type.

The adaptation algorithm (Algorithm 3) works on a document d valid for a schema sx on which an evolution primitive p is applied. Depending on the primitive, the algorithm determines if d is still valid for the updated schema sx_N or performs modifications to d to make it valid for sx_N. The applicability conditions of the primitive should be met, otherwise the document is not modified.

If $p \in \mathcal{P}^*$, then d is not modified at all because p does not alter validity. If p renames the l_O (either local or global) element, the occurrences of l_O in d are identified and renamed to l_N. If p removes a global element and the root of d has the same label, then document d is removed. Otherwise, the document is left unchanged. If p changes the type of an element in a complex type, all the elements of the original type are detected in the document. For element e, the children of e are checked to adhere to the new type. If not, the children of e are removed and a new content is specified for e by means of function $adaptS$ that adds subelements to e. Since an empty list of trees is passed to function $adaptS$, this function generates from scratch the content of e. This behavior is motivated by the assumption that the modification of a type is applied when the original type is deeply different from the new one. Otherwise, other primitives that locally alter the structure of the type would be employed. If p updates a simple type (including `union`, `list`, `restrict` derived types) or changes the type of a global element in a simple type, first the algorithm checks whether the values of the new simple type extends the values of the original type. If so, the document is valid as it is. Otherwise, for each element e of d of the original type its content is changed by assigning a default value of the new type.

If p updates a type structure t, the new structure t_N can require to introduce new elements or to remove existing ones, depending on the specific primitive employed and, in

Algorithm 3. Adapt

```
1  input: p : P, d : DOC, sx : SX
2  output: d' obtained from d that is valid for sx_N
3  switch case p // The applicability conditions of p to sx are met
4    case (p ∈ P*) break;
5    case (p ∈ {rename_glob/local_elem})
6              let l_O be the element tag renamed l_N
7             for e ∈ getElems(getPaths(l_O, sx), d) do UPDATE d RENAME e AS l_N
8    case (p = remove_glob_elem(l, sx))
9              if (l(root(d)) = l) then d = NULL end-if
10   case (p = change_type_glob/local_elem(l, τ_N, sx) ∧ τ_N ∈ CT)
11             let t_N be the structure of type τ_N
12             for e ∈ getElems(getPaths(l, sx), d) do
13                  if (not validS(children(e), init(t_N), t_N)) then
14                       UPDATE d SET children(e) = adaptS([], init(t_N), t_N, INS)
15                  end-if
16   case (p ∈ {change_restrict, change_base/item_type, change_type_glob/local_elem})
17             let τ_N be the new simple type updating τ_O
18             if (τ_O ⋢ τ_N) then
19                  for e ∈ getElems(getPaths(τ_O, sx), d) s.t. content(e) ∉ [[ τ_N ]] do
20                       UPDATE d SET content(e) = defaultVal(τ_N)
21             end-if
22   case (p ∈ P°)
23             let t_N be the new type structure updating t_O
24             if (t_O ⋢_ST t_N) then
25                  for e ∈ getElems(getPaths(t_O, sx), d)) do
26                       if (delElems(p)) then
27                            UPDATE d SET children(e) = adaptS(children(e), init(t_N), t_N, DEL)
28                       end-if
29                       if (addElems(p)) then
30                            UPDATE d SET children(e) = adaptS(children(e), init(t_N), t_N, INS)
31                       end-if
32             end-if
33 end-case
34 return d
```

case of insert or change of an operator, from the new operator. Table 3 reports when elements should be inserted or removed. For $change\_type\_local\ \_elem$ primitive neither insertions nor removals are required, because this primitive does not alter a type structure but the content of subelements. Primitives $insert\_operator$ and $change\_operator$ require to insert or remove elements depending on the new operator. If the new operator is `choice` it means that operator `sequence` or `all` occurred before. Therefore, from sequences of elements in the document grouped by the operator we need to choose one of them. Thus, elements need to be removed. By contrast, if the old operator was `choice`, it means that the new operator is `sequence` or `all`. Therefore, from an element in the document bound by the operator, we need to insert other elements as specified by the `sequence` or `all` operator. Thus, elements need to be inserted. For primitive $change\_cardinality$ both insertions and removals must be performed when both the minimal and maximal cardinalities are updated. This is because a single invocation of function $adaptS$ can add elements or alternatively remove elements. Thus, we first need to remove elements to adhere to the new maximal cardinality and then add elements to adhere to the new minimal cardinality.

Example 5. Starting from the type structure in Fig. 4(a) the cardinality of `b` is changed from $(0, \infty)$ to $(2, 4)$. This requires two applications of function $adaptS$. One for

(a) (b)

Fig. 4. Change of cardinality and its effects

Table 3. Output of $addElems$ and $delElems$

primitive	addElems	delElems
$insert\_local\_elem$	**true**	**false**
$insert\_ref\_elem$	**true**	**false**
$insert\_operator$ (op_N = `choice`)	**false**	**true**
$insert\_operator$ (others)	**true**	**false**
$change\_type\_local\_elem$	**false**	**false**
$change\_cardinality$	*see next table*	
$change\_operator$ (op_N = `choice`)	**false**	**true**
$change\_operator$ (others)	**true**	**false**
$remove\_operator$	**false**	**true**
$remove\_substructure$	**false**	**true**
$remove\_element$	**false**	**true**

min_N	max_N	addElems	delElems
$>$	$<$	**true**	**true**
$\leq$	$\geq$	**false**	**false**
$>$	$\geq$	**true**	**false**
$\leq$	$<$	**false**	**true**

removing elements `b` exceeding the maximal cardinality and one for adding elements `b` missing the minimal cardinality. The original and updated elements are in Fig. 4(b). ○

Once the effects of the evolution primitives have been propagated to the document making it valid for the new schema, the document itself can be returned.

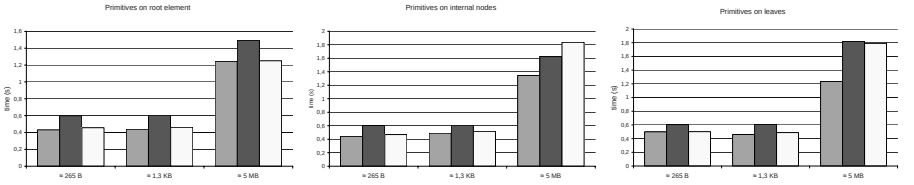
Proposition 2. *Let sx be an XML Schema and d be an XML document valid for sx. Let sx_N be an XML Schema obtained from sx by applying $p \in \mathcal{P}$. Then,*

$$valid(adapt(p, d, sx), sx_N) = \textbf{\textit{true}}\ .$$

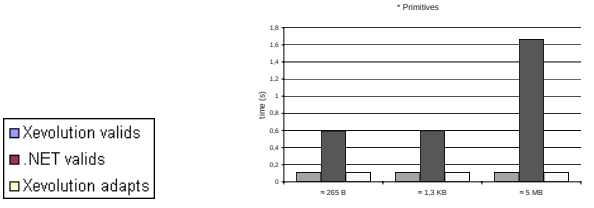
6 Experimental Evaluation

X-Evolution. X-Evolution [15] is a .NET system for handling collections of XML documents and schemas. Documents and schemas are graphically represented as trees and users can specify on the tree representation of a schema the evolution primitives according to the kind of node (element tag, simple or complex type). The $revalidate$ algorithm is applied to check whether documents valid for the original schema are still valid for the updated one. In case of invalidity, the user can then decide to adapt those documents to the new schema (using the *adapt* algorithm) or to leave them without schema. In the back end a DBMS handles documents, schemas and information of which document is valid for which schema.

Experimental Results. Different experiments have been conducted to prove the effectiveness and efficiency of our approach. We gathered from the Web different schemas and corresponding valid documents. Among them the statistics on American baseball competitions, the XML `DBLP` document (`http://dblp.uni-trier.de/xml/`), and



(a) Shallow documents with different size



(b) (c)

Fig. 5. Comparing the *revalidate*, MSXML validation and *adapt* algorithms

plays of Shakespeare collections (`http://www.ibiblio.org/xml/examples/`). The considered collections have been classified according to their size and the level of nesting. *Small* documents are those with size less than 1 KB, *average* documents are those with size between 1 KB and 1 MB, and *big* documents are those with size greater than 1 MB. *Shallow* documents are those with at most 5 levels of nesting, *average depth* documents are those with 5 levels of nesting to 10, and *deep* documents are those with more than 10 levels of nesting. The average characteristics of the documents in each class are reported in the following table.

	small	average	big
shallow	256 B	1.3 KB	5 MB
average depth	736 B	232 KB	137 MB
deep	640 B	924 KB	30 MB

On documents of each class we have applied different kinds of evolution primitives that operate on the root of the document, on internal nodes, and on leaves. We conducted many repetitions of the same evolution primitives and considered the average execution times. Moreover, we also considered the execution time for primitives in $\mathcal{P}^*$. The revalidation algorithm has been compared with MSXML validation algorithm.

Fig. 5 reports the experimental results on *Shallow* documents. The three graphics in Fig. 5(a) represent the execution time for evolution primitives applied on the root of the document, on internal nodes, and on leaves. Fig. 5(c) reports the execution times when only primitives in $\mathcal{P}^*$ are used. Each single graphic reports the execution times of *revalidate*, MSXML validation, and *adapt* algorithms applied on documents of small, average, and big dimensions (relying on the legend in Fig. 5(b)).

Fig. 5(c) points out how our revalidation algorithm outperforms the MSXML validation algorithm for primitives in $\mathcal{P}^*$. Indeed, documents are not accessed and validity is checked only through the schema (in constant time). The performance of MSXML

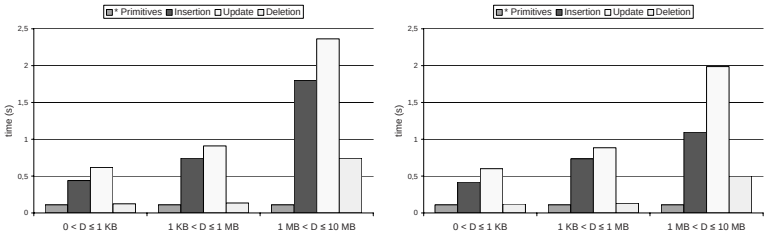


Fig. 6. Execution times of evolution primitive for revalidation and adaptation

validation algorithm is mostly the same for documents with the same size and does not depend on the level of nesting of documents. Our validation algorithm improves the performances of an average of 20% for the other primitives for documents of big size because it operates on small portions of the document. In the *revalidate* algorithm the .NET facilities for accessing documents and evaluating XPath expressions have been used. That means that exploiting indexing techniques available in the back end DBMS the performance of our algorithm would further improve.

The execution time of the *adapt* and *revalidate* algorithms have been compared. The insertion and deletion of internal nodes from the schema have a deeper impact in adapting the structure of an element through function $adaptS$. The performance of the *adapt* algorithm decreases when documents of big dimensions are handled, and specifically when document leaves need to be updated, because the entire document should be loaded in main memory and the probability of "page swapping" increases. This behavior can be however mitigated exploiting indexing techniques and standard DBMS facilities as previously described for the *revalidation* algorithm. Moreover, the graphics point out that updates of nodes deeply nested in the structure of a document require more time that those closer to the document root. To further analyze the *adapt* and *revalidation* algorithms we consider the two graphics in Fig. 6. They report the execution times in case of revalidation and adaptation for the evolution primitives that alter the validity of documents (i.e., those for inserting, deleting, and updating elements/types in the schema) and for primitives that do not alter the validity (i.e., those in $\mathcal{P}^*$). For space constraints, we only report the evaluations on documents of average nestings and primitives applied randomly in the schema. Despite the best performances for primitives in $\mathcal{P}^*$ (as expected), we can note that the execution time for revalidation and adaptation linearly increase as the size of documents increase. The update primitives are more expensive than the deletion primitives. These last ones have performances comparable to those of primitives in $\mathcal{P}^*$. A deeper analysis of primitives can be found in [13].

7 Conclusions and Future Work

In this paper we have proposed an approach for the incremental validation of XML documents upon schema evolution. The approach takes advantage of knowing the documents valid for the original schema and the applied evolution primitive to establish

what needs to be checked in the documents, if some check is needed. An efficient adaptation algorithm to make the invalidated document portions conform to the evolved schema is also proposed. Both the algorithms have been experimentally evaluated. The validation algorithm has been demonstrated to improve the performance of the naïve solution. The adaptation process execution time linearly depends on the document size.

The work presented in this paper is being extended in several directions. For what concerns the evolution primitives, primitives allowing to *move* a portion of the schema and their impact on the revalidation and adaptation processes need to be investigated. In [11] high-level primitives allowing to conveniently express common sequences of atomic primitives have been proposed. The revalidation and adaptation algorithms are currently being extended to these high-level primitives, and, more generally, to sequences of atomic primitives. Finally, the adaptation mechanism is being enhanced with the possibility of specifying through a query the new contents of the adapted documents.

References

1. Balmin, A., et al.: Incremental Validation of XML Documents. TODS 29(4), 710–751 (2004)
2. Barbosa, D., et al.: Efficient Incremental Validation of XML Documents. ICDE (2004)
3. Barbosa, D., et al.: Efficient Incremental Validation of XML Documents After Composite Updates. XSym (2006)
4. Bertino, E., et al.: Evolving a Set of DTDs according to a Dynamic Set of XML Documents. In: EDBT Workshops (2002)
5. Bex, G.J., et al.: DTDs versus XML Schema: A Practical Study. WebDB , 79–84 (2004)
6. Bouchou, B., Ferrari, M.H.: Updates and Incremental Validation of XML Documents. DBPL, 216–232 (2003)
7. Bouchou, B., et al.: Schema Evolution for XML: A Consistency-preserving Approach. MFCS, 876–888 (2004)
8. Bouchou, B., et al.: XML Document Correction: Incremental Approach Activated by Schema Validation. IDEAS, 228–238 (2006)
9. Boobna, U., de Rougemont, M.: Correctors for XML Data. XSym , 97–111 (2004)
10. Choi, B.: What are Real DTDs Like? WebDB, 43–48 (2002)
11. Guerrini, G., et al.: Impact of XML Schema Evolution on Valid Documents. WIDM (2005)
12. Guerrini, G., et al.: XML Schema Evolution, TR Uni. di Genova (2006)
13. Guerrini, G., et al.: XML Schema Evolution: Incremental Validation and Efficient Document Adaptation (extended version), TR Uni. di Genova (2007)
14. Kramer, D.K., Rundensteiner, E.A.: Xem: XML Evolution Management. RIDE-DM, 103–110 (2001)
15. Mesiti, M., et al.: X-Evolution: A System for XML Schema Evolution and Document Adaptation. EDBT, 1143–1146 (2006)
16. Raghavachari, M., Shmueli, O.: Efficient Schema-Based Revalidation of XML. EDBT, 639–657 (2004)
17. Srivastava, D.: Subsumption and Indexing in Constraint Query Languages with Linear Arithmetic Constraints. Annals of Mathematics and Artificial Intelligence 8(3-4), 315–343 (1993)
18. Staworko, S., Chomicki, J.: Validity-Sensitive Querying of XML Databases. EDBT Workshops (2006)
19. W3C. XML Schema Part 1: Structures (2004) available at: `http://www.w3.org`

Managing Branch Versioning in Versioned/Temporal XML Documents*

Luis J. Arévalo Rosado, Antonio Polo Márquez, and Jorge Martínez Gil

University of Extremadura, Department of Computer Science
Avda. de la Universidad s/n 10071 Caceres (Spain)
{ljarevalo,polo,jmargil}@unex.es

Abstract. Due to the linear nature of time, XML timestamped solutions for the management of XML versions have difficulty in supporting non-lineal versioning. Following up on our previous work, which dealt with a new technique for the management of non-lineal versions of XML graph documents, called versionstamp, we have gone a step forward by adding temporal information to each version included in the document. Not only does it allow us to query the vDocuments on a temporal and version level but also we can manage branch versioning in the temporal axis. Moreover, to check its functionality, we have compared our technique to a timestamped XML solution and a set of Web services has been developed. The easy management of multiple versioning, the large number of queries in different XML standard query languages and its implementation by using only XML technology, are some of the advantages of the proposed technique.

1 Introduction

In this collaborative society information flows through all forms of computing, however nobody looks at it in a static way because it changes throughout time and its management becomes necessary to query past information, to retrieve documents belonging to a specific version and to monitor the changes, etc. Document management has been used for years in such environments like collaborative software development, file share resources, etc and more recently, with the appearance of XML [1], it has become necessary also to manage these documents.

Versions of an XML document can be managed through traditional procedures like CVS [2] or subversion [3], the traditional adapted procedures based on XML operations change (delta XML) [4,5] or integrate the different versions into a single XML file using temporal [8,11,12,13,14] or version [9,15] technique. We consider that whatever XML versioning system should have the following main features: it should be able to, validate all XML versions of the document to its schema (the first two solutions do not take into account this fact), support branch versioning (temporal solutions do not do this) and, have the possibility to

* This work has been financed by Spanish CICYT projects "TIN2005-09098-C05-05" and "TIN2005-25882-E".

D. Barbosa et al. (Eds.): XSym 2007, LNCS 4704, pp. 107–121, 2007.

query the XML versioned documents using some XML standard query languages such as XQuery and XPath (the first solution does not do this).

To get these characteristics, we have used the technique shown in [9] that consists of marking the document with a versionstamp instead of using a timestamp. In this work we have gone even further by adding temporal information to each version allowing us to query the document either on temporal or/and version level. We have also defined the basic updated operations common to whatever XML document, describing them by means of an XML document called *XML transactional document* which allows us to manage changes for any markup language based on the XML specification. Moreover, to check its functionality, we have compared our technique to a timestamped XML solution as well as developing a set of Web Services.

The remainder of this paper is organized as follows: we begin by summarizing the current solutions for the management of XML versions. Then, we continue showing the foundations of this paper based on [9], extending it with temporal information and describing later the basic updated operations. We then follow up this by showing several queries made on a temporal and version level. After that, some implementation details and the achieved results are discussed and finally, we offer our conclusions and a look at our future work.

2 State-of-the-Art

The problem of XML document version management combines the issues of document version management [4,5,6,7] and temporal databases [22]. Document version management has been used for years mainly in collaborative environments. These traditional techniques [2,3] are based on diff lined-based algorithms to locate the differences between two versions of a text. For XML documents, where the organization in lines can be neglected, line-based approaches are inappropriate since the structure of the document is lost. The necessity to manage XML versions not only is important in XML databases but also in XML document management because nowadays more and more applications use it to store their configurations, data, etc, such as OpenOffice and Microsoft Office.

XML solutions have been centered mainly in some of the following ideas. *Delta XML management* is based on traditional change operation procedures adapted to XML [4,5]. It consists of obtaining and storing the XML differences between two versions (*delta XML*). An exhaustive study of XML diff approaches is made in [10] where the authors use an C++ implementation of [4] to manage XML OpenOffice document versions. However delta XML solutions have the same problems than traditional techniques, it means, neither XML validation nor XML query cannot be carried out in these solutions.

Multiversion XML [6,7] define an indexing technique for branched versioning which they called BT-Tree and BT-ElementList respectively, however they cannot be used in XML Standard Query languages (XQuery or XPath[2]).

Temporal XML Representation based on temporal database topics [21] representing and managing historical information in XML. In [11] a technique for

managing temporal web documents is shown using an XML/XSLT infrastructure. A data model is proposed for temporal XML documents [14] where leaf data nodes can have alternative values; however supporting different structures for non-leaf nodes is not discussed. Extensions of XPath data model are exposed in [13,12] to represent and query transactional and valid time respectively, by means of the addition of several temporal dimensions. A temporally-grouped data model is shown in [8] that gives us a way to represent the content database evolution using XML timestamps, however non-lineal versioning is not supported.

The integration of time and version concepts to manage dynamic information has been studied recently in [15,16] for XML and object-oriented databases respectively. In [15] the authors defined temporal delta (tDelta) and introduces version time in it, however query support is not discussed.

Due to the linear nature of time, XML timestamped solutions for the management of XML versions have difficulty in supporting non-lineal versioning. In collaborative scenario, due to the fact that users can update any version of the document generating a new version either from the current one or discard it and reuse an old version, branched versioning is necessary. Using our solution, called as *versionstamp* or *vstamp* [9], this feature can be modeled in a easy way.

3 XML Versioned Documents

In this section we present how to manage changes in XML document in a branch way. Firstly the foundations our work is based on [9] is shown. Then we extend it to incorporate temporal information and finally we describe a taxonomy of changes for XML documents.

3.1 Versionstamp Technique

An XML versioned graph data model, called as *V-XML* data model, was shown in [9] to represent versions in XML graph documents by means of adding versionstamp information in the graph document obtaining a new XML document which we called as *vXML Document* or *vDocument*. This is formed by two sections: The first one which stores all information about the included versions and the relationship between them and the second one being, each element in the document is transformed into a versioned element by means of defining its version validity, that is, for which version/s of the document it is valid.

In order to store the included versions, we decided to map by means of an XML document, which we called as *version_ tree,* how the versions have been made over time. Each included version is an element and represents the different snapshots of the document. If there is an parent-child relationship from V_i element to V_j element, it means that, V_j is created by updating V_i.

Once the included versions have been represented, it is necessary for each versioned element to represent its version validity. To do it, we use a *versionstamp technique*, which we called as *Version Region* [9], that is defined as a set of version identifiers from the version tree indicating for which versions of the tree it is valid

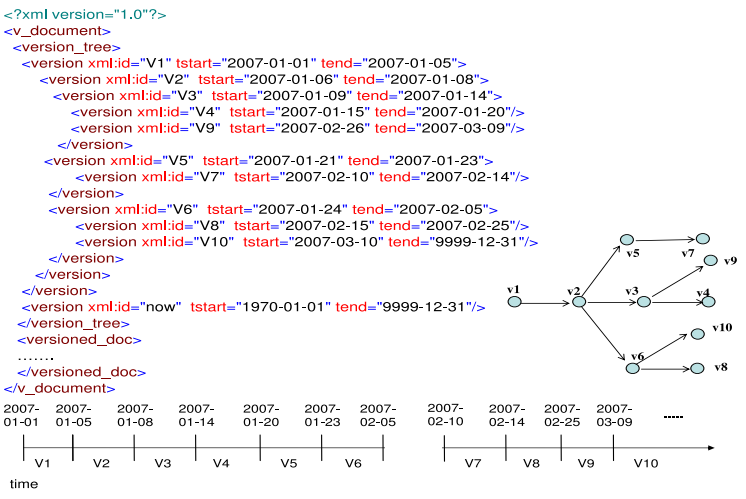


Fig. 1. XML and graphical representation of a version tree with temporal information

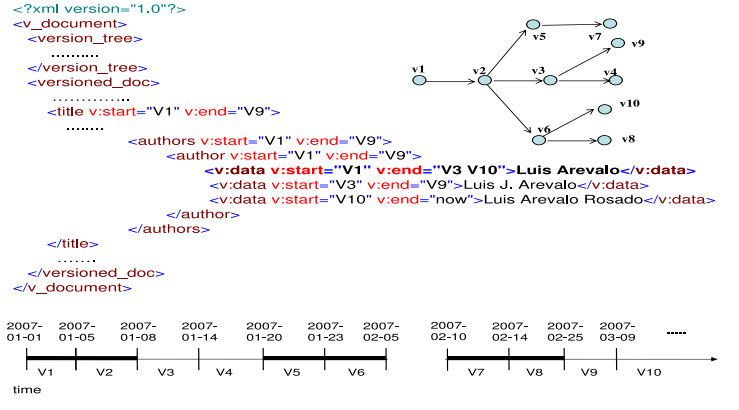


Fig. 2. Versioned elements with version region

(a sub-tree of the version tree). A version region is a [start-End] pair where the start value is a version identifier that represents the origin node of the valid area in the version tree and End is a *set* of version identifiers that indicate when each area has stopped being valid. In this way each element in the versioned document is formed by a version region that is converted into two attributes, v:start and v:end. The first one is defined as an IDREF datatype attribute which refers to a version identifier from the version tree and the second one defined as IDREFS datatype which allows us to represent a set of version identifiers from the version tree.

In figure 1 and 2 a vDocument is shown. On the one hand in figure 1 the version tree with several versions of an XML document is shown: i.e: from the version identified by V_2 several changes have been made (identified by V_3, V_5, V_6). On the other hand in figure 2 several versioned elements are shown. i.e: the first *author* element is valid from V_1 and stops being valid in V_9, this means that it is valid for all descendants-self of the V_1 version except all V_9 descendant-self versions. Another example is the first *v:data* child for *author* element which is valid from [V_1, {V_3,V_{10}}] so it is valid in the versions identified by V_1,V_2,V_5,V_7,V_6 and V_8 since all descendant-self of V_3 and V_{10} are not included meanwhile the second v:data of this *author* is only valid for all descendants versions of V_3 except descendants-self of V_9. The special value "now" in the attribute v:end indicates "no changes until now", in other words, the version region is formed by all version descendants from the v:start attribute. Obviously, we have to take into account that an element cannot exist without its ancestor elements.

3.2 Temporal Time in vDocuments

When a new version of the document is generated in a vDocument, these changes happen at some point in time. Until now, we have only represented the relationship between the versions in vDocuments without taking into account when these changes occurred, this means that, the temporal validity information associated to each version is lost. In this section we show how to integrate the valid-time axis in a vDocument calling as *VTstamp*.

Temporal database researchers have focused on three principal dimensions of time [22]: valid time, transactional time and user-defined time. In this work, we have decided to model the valid-time axis, although the other axes can be managed in the same way. The valid time of a fact is defined [22] as the time when the fact is true in the modeled reality, in our case, the valid time of a version is when the version is true. We have decided to include the valid time by means of a time interval, a pair of two time instants [t_1,t_2] that is turned into two attributes for each version defined in the document as shown in figure 1. The following restrictions must be carried out: 1) For each version defined in the version tree, the value of t_1 instant must always be less than t_2 2) Any two time intervals from the version tree cannot overlap and 3) We assume that time is bounded.

On the other hand it is also necessary to define the valid time for each tag included in the document, that is when this tag is valid. Using the version region used in our technique, we can define its temporal validity easily. Due to the fact that a specific tag is valid in a set of versions from the version tree, this means that, this tag will also be valid in each period of time for each valid version. For example in figure 2 the temporal validity of a specific v:data tag which is valid in the following version: [V_1, {V_3,V_{10}}] is shown, therefore it will be valid in the following time intervals {[01-01,01-05], [01-06,01-08], [01-21,01-23], [01-24,02-05], [02-10,02-14], [02-15,02-25]} (shown with a thick line above in the figure). Notice that some of these time intervals can be joined forming a continuous period of

time (coalesce) i.e: [02-10,02-25], however, this is not advisable since they are placed in different branches from the version tree.

3.3 Changing and Updating a VXML Document

As has been said, XML documents are not static, so it is necessary to manage inserts, deletes or updates that can modify them [20]. Beginning at the initial state of the document (version 0), new versions are then established by applying a number of changes to whatever version defined in the document. Once we know how to represent versions in XML documents, the following questions will be: what kind of change operations can generate a new version? And, how to update the XML versioned document from a change operation?.

In order to answer the first question, we have analyzed which items can be changed in an XML document and which operations can be performed on them. However, before this, it is necessary to identify thoroughly those elements which have been changed from the current version. Among the different possibilities shown in [4], we have decided to add an attribute *idf* to each element in the document in order to identify it in a vDocument, with the exception of v:data, v:attrib and v:isref because those elements are identified by its parent element. Thus, the basic structural XML operations, common in whatever document based on the XML specification, are shown in table 1.

Although *move operation* can be represented as a delete and an insert operation we have decided to include it as one of our basic operation since it is a very frequent change in XML documents. According to the consistency principle, to accept the execution of each primitive a restriction must be satisfied, that is, the document obtained must be well-formed, and each version of the document must be valid in accordance to the specifications of its XML-Schema. To guarantee this, a whole set

Table 1. XML changes primitives

Operations	Meaning
IE (idf,name,pos)	It adds a new element with the name *name* from the parent *idf* in the position *pos*.
DE(idf)	It removes the element identified by *idf*. All valid descendant elements are deleted too.
RE(idf, name)	It renames the name of the element identified by *idf*.
IA(idf, name, value)	It adds an attribute for the *idf* parent.
DA(idf, name)	It removes the attribute called *name* from the element identified by *idf*.
UA(idf, name,n_value)	It changes the value of the attribute called *name* for the parent *idf* for the *n_value*.
IC(idf,data)	It adds a PCDATA text for the *idf* parent.
DC(idf)	Removes the content for the element identified by *idf*.
UC(idf, data)	Changes the PCDATA value for *idf* to *data* value.
ME(idf,idf_from,pos)	Moves the element identified by *idf_from* and its descendants to element *idf* in the position *pos*.

of pre-conditions to be fulfilled have been defined for each single operation before producing a new version of the document. For example: 1) the "idf" parameter for all operations must exist for the version we want to update, 2) the name of the attribute in IA operation implies that another attribute for this element cannot exist from the current version (there cannot be two attributes with the same name) and 3) the DC operation cannot be carried out if there isn't any PCDATA information for the required identifier.

These basic updated operations can be obtained mainly by means of two techniques. On the one hand, obtaining the XML operational differences between two versions by means of several approaches such as [4,18,19] or on the other hand from a certain version specifying which changes we want to carry out. The technique proposed in this work is based on both solutions, needing, therefore, a mechanism to integrate them. This consists of representing each update operation exposed previously in an XML format.

In this way if an approach based on differences is chosen, then an XSLT stylesheet, which transforms this XML document with differences to our XML representation, is defined. From [10], where several XML diff approaches are analyzed, we have decided to choose JXydiff [25] which is a Java tool for detecting changes in XML documents based on Cobena's work shown in [4]. We chose this for the following reasons: 1) It has the main features to retrieve XML differences: can manage all kind of XML nodes, can detect move and update operations and is based on a tree oriented algorithm, 2) It is written in Java, so its integration in our implementation is immediate and 3) It is very easy to export its output XML differences to our XML representation by means of an XSLT stylesheet. As a future work, our idea is to use a relational-based approach [17] for detecting changes in XML documents due to scalability problem that suffers the main-memory Diff algorithms mainly in Java. On the other hand, if we decide to change the document manually, the change editor has only to generate a batch document with update operations in our XML representation.

In this way, the creation of a new version is defined by a set of the aforementioned operations represented in an XML document with changes, which we call an *XML transaction document*, as is with the concept of transaction in databases, the vDocument is updated if and only if all changes are executed. This transaction is carried out in the following three phases:

Phase 1) Retrieval of the version to modify. The document to work on will be the version of the XML document obtained from the vDocument, to which the XML change transaction will be applied.

Phase 2) Modification of the retrieved XML document.

Phase 3) Updating of the versioned document. a). Obtain the XML transaction document b). Execute each operation from this XML to the vDocument and c). The new version and its associated temporal information is added to the version tree.

In figure 3 the XML transaction schema is shown as well as a practical example. As we can see, an XML transaction document may be formed by several versions where each version may be formed either by a sole operation or by

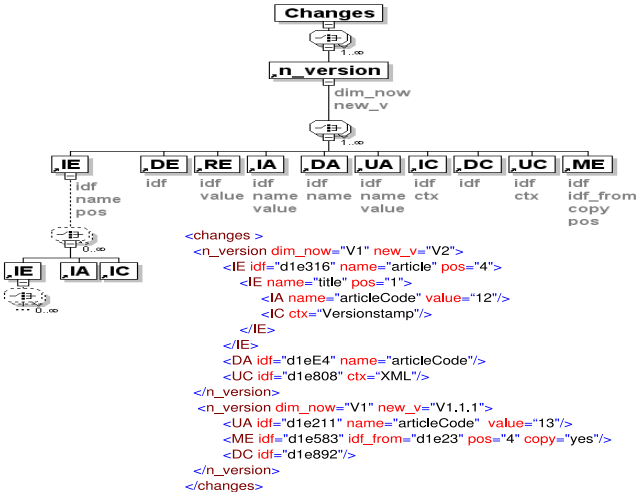


Fig. 3. Schema and an example of an XML transactional document

means of several of them (the parameters of each operation from table 1 are defined as attributes). For example, the first DA operation shown in figure 3 is formed by two attributes: *idf* that stores the parent identifier (d1eE4) and *name* (articleCode) that is the name attribute to delete. Another example is the IE operation, InsertElement, that can be formed by one or several IE/IA/IC operations as is shown in the same figure. In that case, the first IE operation inserts an element which has a child element which contains an attribute (IA) and a PCDATA content (IC).

Related to the second question about how to update a VXML document when a basic change operation is produced the following actions are carried out. When an insert operation is made, the new element/attribute/content is inserted in its position setting the v:start attribute to the new identifier version and the v:end attribute to "now" value. In the case of a delete operation, it is only necessary to change the v:end attribute of all affected items setting them to the new identifier version. For update operations the v:end attribute for the current item is set to the new identifier version and the new element/attribute/content is added and its version region attribute is set as in the insert operation. In the case of a move operation, the affected items are modified as in the update operation.

One of the most important advantages of using an XML document to define the update operations, is that it allows us to manage changes for any markup language based on the XML specification, since these update operations are common to all of them. Thus, to specify the changes of a certain XML language, it would be only necessary to define it by means of these primitives. In this way, as a future work we will use this technique to manage versions of XSLT and SVG document. Moreover, this technique can be also used to represent the version history of an XML schema document.

4 Retrieval in vDocuments

One of the main advantages of this proposal is the wide set of queries we can specify both using version and temporal axis. In this way, classical temporal XML queries can be made such as temporal projection, snapshot, etc and also version queries such as version projection, snapshot version, etc. Here, we will show some of them that are used in the following section to measure our technique.

Q1: *Version snapshot query*

In order to retrieve the valid labels for a given version it will be necessary to analyze which versions are included in a version region and check if the requested version belongs to them. This occurs only if 1) the given version is among the descendants in the "start" version identifier in the version tree or even is itself and 2) the given version is not among the descendants or is itself in all version identifiers for "end" attribute. To do this effectively, we have to obtain which versions are in a version region and check if they contain the requested version. We use the id() function provided by XPath to obtain the versions by means of dereference the version/s in the version tree which v:start and v:End refer to (they are defined as IDRef and IDRefs datatypes respectively) and thereby we can easily obtain their descendants and check the constraints said before.

We have defined a version operator called *Vmeets* as a user-defined function (line 1) that check (line 4) if the given version belongs only to the v:start attribute (line 2) and not to the v:End attribute (line 3). That query retrieves all nodes valid for V_8 version (line 6). In the same way, other version operators are able to been defined as: Vancestors, Vparent, Vcontains, etc.

```
1. declare function f:Vmeets($p,$v) as xs:boolean{
2.     let $start:=$p/id($p/@v:start)/descendant-or-self::version/@xml:id
3.     let $end:=$p/id($p/@v:end)/descendant-or-self::version/@xml:id
4.     return (($start=$v) and (not($end=$v))) };
5. <data>{
6. for $s in //versioned_ doc//*[f:Vmeets(.,'V8')]
7. return $s
8. }</data>
```

Q2: Count the number of the title element valid for version V8 using Xpath

Using the id() function, we can query the vDocument using another XML standard query language such as XPath. In the following query all title elements valid for version V_8 are counted.

```
count (//*title[not(id(./@v:end)/descendant-or-self::version/@xml:id='V8' ) and
(id(./@v:start)/descendant-or-self::version/@xml:id='V8')]
```

Q3: *Temporal snapshot query*

Since temporal information has been added to our vDocuments, we can retrieve it by means of the valid-time axis. To do this, it is necessary to find out in which version the given time belongs to. If a time instant is given, a user-defined

function called *tmeets* (line 1) retrieves which version contains this time. After that, the previous version snapshot is executed (line 5, 6). In the case of a time interval, a user-defined function called *tContain* is defined which verifies which version contains the requested time interval. Q1 query using the valid-time axis is shown below.

```
1. declare function f:tmeets($time) as xs:string{
2.    let $id:= //version[(./@tvstart<=$time) and (./@tvend>=$time)]/@xml:id
3.    return $id };
4. <data>{
5. let $version:=f:tmeets("2007-02-20")    //This instant belongs to V8
6. for $s in //versioned_ doc//*[f:Vmeets(.,$version)]
7.    return $s
8. }</data>
```

5 Experimentation and Implementation

In this section several experiments have been carried out in order to compare our technique to a timestamp XML approach and some details of its implementation are also shown.

5.1 Experimental

The testing machine is a Pentium Mobile 1,8GHz PC with Linux (Ubuntu), with 1024MB memory and a 120GB IDE hard drive. The data shown in the graphics are the performance average on 3 identical tests. We have developed a Java application to generate a large amount of version data where the operations from the table 1 are selected at random, assigning a higher probability to the insertion of elements. Once selected a primitive, the current version and the affected node are selected at random too. The tests have been carried out on cases of lineal versioning and branch versioning. In the latter case, we have selected at random the version we want to update according to the following probabilities a 20%, 50% and 80% possibility of choosing a different version from the current one.

The experiments were carried out on 5, 10 and 20 changes per version, for 100, 60 and 30 versions respectively thereby evaluating the behavior of our system in the following cases: a large number of versions with few changes (100 versions - 5 changes), a medium number of versions with some changes (60-10) and a small number of versions with many changes (30-20). In the experiment, we selected the ACM XML Sigmod Record supplied in [26] (November of 2002) where three different versions of this document were used: small, medium and large. All characteristics of these documents can be consulted in figure 4.

We have also developed a temporal timestamped XML solution (tstamp) in order to compare it with ours. In this way, we have chosen the technique shown in [8], based on adding a time interval to each label in the document, allowing the incorporation of temporal information in the XML document. All our versioned lineal XML documents have been converted to temporal ones. The resulting

		Size	Element		Attribute	Text
ACM Sigmod XML	Small	42028	687		411	417
	Medium	225487	3688		2285	2317
	Large	545368	8930		5677	5769
		Size	Element	Attribute	v:attrib	v:data
ACM Sigmod Vdocument	Small	95519	691	4545	411	417
	Medium	522093	3692	24820	2285	2317
	Large	1281000	8934	60976	5677	5769

Type. Versions/Changes		Tstamp	Lineal	20,00%	50,00%	80,00%
Vdocument	S. 100/5	191113	222245	229027	229992	236706
	S. 60/10	203406	228582	250637	240359	261261
	S. 30/20	197510	216406	219185	233103	233238
	M. 100/5	557512	640329	653778	669064	647238
	M 60/10	584815	660593	671610	658810	674334
	M. 30/20	583351	651263	659213	664908	658625
	L. 100/5	1216703	1386684	1406887	1391842	1395761
	L. 60/10	1250378	1415846	1424534	1403901	1405975
	L. 30/20	1245227	1407287	1422153	1409936	1412069

Fig. 4. a. Characteristics of the document. b. Resulting vDocument size.

version size document is shown in figure 4b where it can be seen that the size of our vDocuments are a bit higher than the timestamped solution.

The retrieval time obtained refers to the transformation time in a client application, regardless of the document loading time in memory or transmission where the retrieval time has been calculated on 3 performances. To do it, we have used the Saxon processor [27] where the following queries have been carried out:

* Q1: Version/Temporal Snapshot query using XSLT.
* Q2: Find the total number of *title* elements valid for a version in XPath.
* Q3: Retrieve those authors and their descendants valid for a version in Xquery.
* Q4: Snapshot query using an optimized XSLT.

In figure 5.a the retrieval time (measured in ms) obtained using an XSLT stylesheet is shown (query Q1). This figure shows the retrieval time using the timestamped solution (Tstamp), using the versionstamp solution (VStamp) and the versionstamp solution on a temporal level (VTstamp). As we can see, our solution here behaves less efficiently than the timestamp solution, since the time solution uses the operators $<=$ and $>=$ to verify if a time belongs to a time interval, meanwhile in our process we have to retrieve all descendant identifiers for the v:start and v:end attributes. In this way, both Vstamp and VTStamp greatly depend on the number of versions that the document has as well as the size of it. In some cases (short documents or documents with few changes) our performance is quite similar to the timestamped solution, however our solution in lineal versioning performance is poorer. This can be seen in figure 5.b and figure 6.a where the retrieval time for Q2 and Q3 query are shown.

To avoid this situation, we have developed an optimized solution that consist of storing within each version their descendants allowing us not to have to constantly recover this information in each query. Therefore, each version in the version tree will have a new attribute called *descen* that stores its descendants. In this way if we want to check if a requested version belongs to a version region, it is only necessary to verify if the *descen* attribute for the *v:start* version

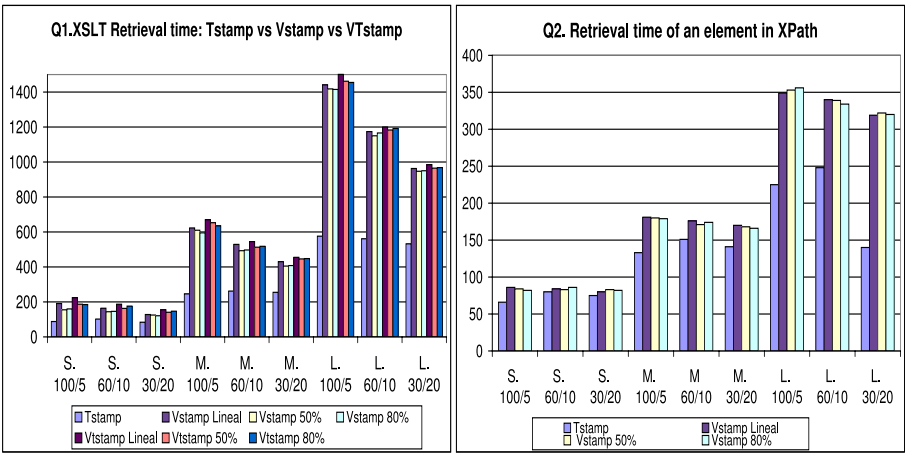


Fig. 5. Retrieval time a. Q1 query b. Q2 query

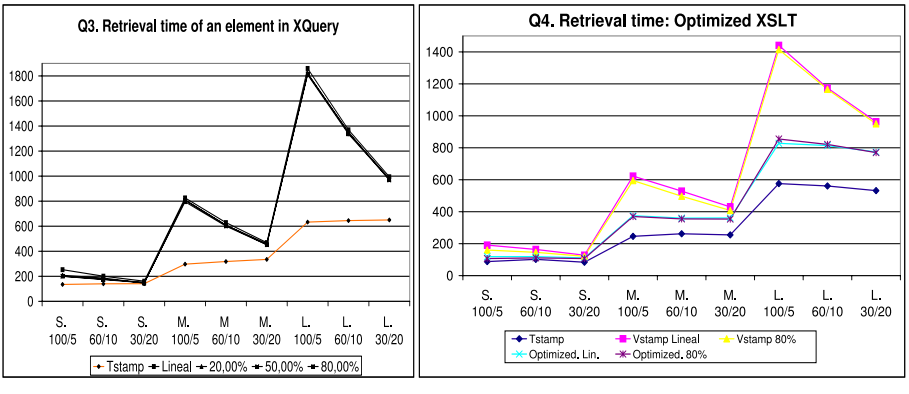


Fig. 6. Retrieval time a. Q3 query. Q4 query.

contains the given version and it is not present in the *v:end* attribute. We can see this improvement in figure 6 where we can verify that using it the retrieval time is reduced considerably and in some cases the retrieval time is quite close to the timestamped solution. In those cases that our solution had its poorest performance (large documents or several versions) this time has been reduced by up to 50%. Notice that, this solution is almost independent from the number of versions, since it is not necessary to retrieve the descendants of the v:start and v:end attribute.

Although it can be argued that our solution performs poorly in large documents, it offers many advantages that timestamped solutions cannot: we can query versioned documents both on a version and a temporal level, manage branch versioning that is not supported in timestamped solutions and extend the number of temporal/version queries that can be made.

5.2 Implementation

The system has completely developed using XML technology. To execute it we just need an XSL stylesheets, Xquery or XPath processor with support for id() XPath function (tested in Exist, Saxon and Xsltproc processor). To check its functionality, we have developed a set of Web Services to manage versions of XML documents. Our proposal is to develop a generic engine to store, manage and query the different versions from an XML document through Internet thanks to Web Services without to set any additional software. The most important advantages of this engine is the possibility to offer them to third-party clients to either version their data or to develop a more complex versioning system by means of invoking our Web Services.

Table 2. Versioning Web Services

Group	Web Service	Brief description
Conversion	doc2vdoc	Generates an XML versioned document from an XML document.
Conversion	vdoc2doc	Retrieves the reconstructed XML document from a specific version
Get	getDocs/VDocs	Retrieves a list of documents/versioned documents stored in the system for a specific user.
Get	getVersions	Retrieves the available versions for a vDocument
Get	getInfo	Gives information about each update operation (parameter, error, etc)
Query	getQuery	Executes XPath or XQuery in a vDocument.
Changes	Primitive	Used to run an update operation for a specific vDocument
Changes	execTrans	Executes an XML transaction document
Changes	exec_Randontrans	Executes an XML random transaction document
Manage	uploadXML/ VXML	Allows us to upload XML/VXML to the repository
Manage	deleteXML/ VXML	Allows us to delete a XML/versioned document from the repository
Diff.	GetDiff	Obtains the differences between two versions
Diff.	getXMLDiff	Retrieves a specific version of the document from the Vdocument.

Our Services have been developed using Java, more specifically the API called AXIS [24] from the Apache Software Foundation. AXIS has proven itself to be a reliable and stable base on which to implement Java Web services. Initially, we propose a set of 16 Web Services that can be classified in six groups as shown in table 2 (parameters of each service is omitted in this work due to lack space). In order to carry out some trials on these services we developed a client prototype too as it is shown in the following URL: http://exis.unex.es/versionado/.

6 Conclusions and Future Work

Document version management has been used for years mainly in collaborative environments by means of, on the one hand, diff lined-based approaches or delta XML, however these solutions are not recommended in XML documents since they can neither validate nor query the XML versioned document and, on the other hand, XML temporal document solutions, based on the timestamped technique, which have difficulty in supporting non-lineal versioning. To solve these problems we proposed a versionstamp technique in [9].

In this paper, we have extended it by means of adding temporal information to each version included in the vDocuments. Not only does it allow us to query the vDocuments on a temporal and version level but also we can manage branch versioning in temporal documents. Moreover we have also defined the basic updated operations common to whatever XML document, describing them by means of an XML document called *XML transactional document* which allows us to manage changes for any markup language based on the XML specification.

Finally we have compared our solution to a timestamped XML one. Although it performs poorly in some cases we have improved it by means of an optimized solution thereby offering us many advantages that timestamped solutions cannot achieve. Moreover, we have developed a set of Web services which do not have portability restrictions and allows us not only to manage the different versions of an XML document but also to validate, transform, store and query them in an easy way. Since our proposal is open, it can be used for third-party clients either to manage their documents or to extend them by incorporating new features.

As future work we propose these following steps:

* To analyze new queries in XML versioned documents as range queries, temporal/version queries, temporal overlapping queries, etc.
* Compare the results storing the documents in native XML databases and in relational databases.
* To define the version region by means of a set of sub-graph nodes allowing us to represent element temporal interval.
* To implement a versioning system based on XUpdate.
* To extend these services by incorporating some features of traditional version control systems such as security, lock files, indexing the document to run the queries faster, etc.
* To apply our versionstamp technique to other XML markup languages such as XSLT stylesheets, SVG graphics or even to XML office documents as OpenOffice or Microsoft Office.

References

1. W3C, `http://www.w3c.org`
2. CVS. Concurrent Versions System, `http://www.cvshome.org`
3. Subversion, `http://subversion.tigris.org/`

4. Cobena, G., Abiteboul, S., Marian, A.: Detecting changes in XML documents. In: Proceeding of the 18th International Conference on Data Engineering (2002)
5. Chien, S-Y., Tsotras, V.J., Zaniolo, C.: Efficient management of multiversion documents by object referencing. VLDB (2001)
6. Vagena, Z., Moro, M.M., Vassilis J.: Tsotras. Supporting Branched Versions on XML Documents. In: RIDE (2004)
7. Salzberg, B., Jiang, L., Lomet, D.B., Barrena, M., Shan, J., Kanoulas, E.: A Framework for Access Methods for Versioned Data. In: Bertino, E., Christodoulakis, S., Plexousakis, D., Christophides, V., Koubarakis, M., Böhm, K., Ferrari, E. (eds.) EDBT 2004. LNCS, vol. 2992, Springer, Heidelberg (2004)
8. Wang, F., Zaniolo, C.: XBiT: An XML-based Bitemporal Data Model. In: Atzeni, P., Chu, W., Lu, H., Zhou, S., Ling, T.-W. (eds.) ER 2004. LNCS, vol. 3288, pp. 810–824. Springer, Heidelberg (2004)
9. Rosado, L.A., Márquez, A.P., González, J.M.F.: Representing versions in XML documents using versionstamp. ECDM (2006)
10. Ronnau, S., Scheffczyk, J., Borghoff, U.M.: Towards XML Version Control of Office Document. In: Proceedings of ACM DocEng. (2005)
11. Grandi, F., Mandreoli, F.: The valid web: An XML/XSL infrastructure for temporal management of web documents. In: ADVIS (2000)
12. Dyreson, C.E.: Observing transaction-time semantics with TTXPath. In: WISE (2001)
13. Zhang, S., Dyreson, C.E.: Adding valid time to XPath. In: Bhalla, S. (ed.) DNIS 2002. LNCS, vol. 2544, pp. 29–42. Springer, Heidelberg (2002)
14. Amagasa, T., Yoshikawa, M., Uemura, S.: A data model for temporal XML documents. In: Ibrahim, M., Küng, J., Revell, N. (eds.) DEXA 2000. LNCS, vol. 1873, Springer, Heidelberg (2000)
15. Wuwongse, V., Yoshikawa, M., Amagasa, T.: Temporal Versioning of XML Documents. In: Chen, Z., Chen, H., Miao, Q., Fu, Y., Fox, E., Lim, E.-p. (eds.) ICADL 2004. LNCS, vol. 3334, Springer, Heidelberg (2004)
16. Galante, R.M., Santos, C.S., Edelweiss, N., Moreira, A.S.: Temporal and Versioning Model for Schema Evolution in Object-Oriented Databases. In: Transactions on Data and Knowledge Engineering (2005)
17. Leonardi, E., Bhowmick, S.S., Madria, S.K.: Xandy: Detecting Changes on Large Unordered XML Documents Using Relational Databases. In: Zhou, L.-z., Ooi, B.-C., Meng, X. (eds.) DASFAA 2005. LNCS, vol. 3453, Springer, Heidelberg (2005)
18. Mouat, A.: XML diff and patch utilities. Master's thesis, Heriot-Watt University, Edinburgh, Scotland (2002)
19. Wang, Y., DeWitt, D.J., Cai, J.: X-Diff: An effective change detection algorithm for XML-documents. In: Conf. on Data Engineering, IEEE CS Press, India (2003)
20. Xquery Update. `http://www.w3.org/TR/xqupdate/`
21. Snodgrass, R.T.: The TSQL2 Temporal Query Language. Kluwer Academic Publishers, Dordrecht (1995)
22. Jensen, C.S., Dyreson, C.E., et al. (eds.): The Consensus Glossary of Temporal Database Concepts (February 1998)
23. Tatarinov, I., Ives, Z.G., Halevy, A.Y., Weld, D.S.: Updating XML. In: ACM Sigmod. (2001)
24. Apache AXIS. Retrieved From: `http://ws.apache.org/axis/`
25. JXydiff. `http://potiron.loria.fr/projects/jxydiff`
26. ACM XML Sigmod Record. `http://www.sigmod.org/record/xml`
27. Saxon. `http://www.saxonica.com`

SXDGL: Snapshot Based Concurrency Control Protocol for XML Data*

Peter Pleshachkov[1] and Sergei Kuznetcov[2]

[1] Institute for System Programming RAS, Russia
peter@ispras.ru
[2] Institute for System Programming RAS, Russia
kuzloc@ispras.ru

Abstract. Nowadays, concurrency control for XML data is a big research problem. There are a number of researchers working on this problem, but most of the proposed methods are based on the two-phase locking protocol, which potentially leads to a high blocking rates in data-intensive XML-applications. In this paper we present and evaluate SXDGL, a new snapshot based concurrency control protocol for XML data. SXDGL completely eliminates data contention between read-only and update transactions. Moreover, SXDGL takes into account the hierarchical structure and semantics of XML data model determining conflicts between concurrent XML-operations of update transactions. The conducted evaluation shows significant benefits of SXDGL for processing concurrent transactions in data-intensive XML-applications.

1 Introduction

Over the last decade, native XML database management systems has received considerable attention from the research community and industrial software companies. As a result, there are a number of databases now on the market, which support XML data model. One of the key requirements for a native XML database is it's ability to provide data consistency while allowing multiple transactions to have concurrent read/write access to the XML documents. In order to meet this requirement database researchers adopted an existing multigranularity locking scheme based on 2PL protocol for XML data [10,6,17].

It is a well known fact [20] that 2PL does not efficiently support concurrent processing of complex read-only transactions (usually called *queries*) and update transactions (usually called *updaters*), causing update transactions to suffer from long delays due to data contention with read-only transactions. The problem seems to be increasingly important in the context of XML applications, because XQuery (with small extensions like XQueryP[2]) is used as a native language for development of XML-applications, and, as a result, transactions written in XQuery may be long-lived. Thus, delays for *short* updaters may be unacceptable.

This problem has been extensively studied by many researchers in the past years and various multiversion extensions to 2PL protocol (e.g. 2V2PL, MV2PL,

* This work was partially supported by the grant of RBRF N 05-07-90204.

D. Barbosa et al. (Eds.): XSym 2007, LNCS 4704, pp. 122–136, 2007.

ROMV) have been proposed in the literature [21]. Moreover, multiversion protocols have been widely accepted in the industry and implemented in many relational databases like Oracle, MS SQL Server 2005, PostgreSQL, etc. However, multiversion concurrency control for XML data has received little attention in the literature so far. The problem is challenging due to a number of reasons specific to XML data.

Firstly, it is not straightforward how to design a versioning scheme for XML database in a such way that the effectiveness of XML-document traversal operations would be preserved. It is a very important issue, because traversal operations are *intensively* performed during query/update execution, and therefore, of crucial importance to achieve high performance. Indeed, each time when we want to access a node by pointer we should perform pointer dereferencing and additionally follow the version chain in reverse chronological order to locate the appropriate version. Thus, the versioning scheme should be designed in a such way that additional overhead incurred by choosing an appropriate version would be minimized.

Secondly, to eliminate unnecessary conflicts between updaters we need to take into consideration hierarchical structure of XML data model and semantics of XQuery/XUpdate operations. Especially, we need carefully consider all different types of XML update operations and regular path expressions taking into account different conflict behavior.

Thirdly, we should reduce the locking overhead (in updaters), which can be tremendous for XML data as shown in [10].

As our main contribution we propose SXDGL, a new snapshot-based XML DataGuide locking protocol, which produces only serializable schedules and supports efficient processing of concurrent processing of transactions in XML database. Our protocol enables efficient processing by employing the following techniques. Firstly, SXDGL allows queries to execute without acquiring locks. It completely eliminates locking overhead for queries and interferences between queries and updaters. Secondly, multiversioning is implemented using adjusted memory-mapped architecture, which significantly reduces a total number of traverses of version chain to locate the appropriate version. Besides, the proposed versioning scheme restricts the maximum number of versions of data item to four. So, it significantly simplifies version management. Finally, we introduce a DataGuide-based locking scheme for isolating updaters. 11 types of new lock modes are introduced, which allow to capture the different conflict behavior of XQuery/XUpdate operations, and, as a consequence, avoid unnecessary delays. Besides, using a compact DataGuide structure for locking purposes guarantees that the locking overhead is low as opposed to approaches that set locks on the nodes of XML-document.

The SXDGL protocol have been prototyped in Sedna XML database system [13], which is successfully used in the content engineering projects, biological and Web-based applications, etc. In this paper, we present the results of some experiments which have been conducted with the aim of evaluating the performance

of SXDGL protocol. The experiments show significant benefits of SXDGL for processing concurrent transactions in XML database.

The paper is organized as follows. In Ssection 2 we introduce the XML query and update languages, we refer throughout the paper. Section 3 outlines basic principles of data organization, memory management and versioning scheme in Sedna XML database. Section 4 presents SXDGL protocol. The results of performance evaluation of our protocol are discussed in Section 5. Finally, sections 6 and 7 discuss the related work and conclude the paper.

2 XML Query and Update Languages

In Sedna we use XQuery to query XML documents. To update XML documents we use our own primitive update operations like insert, delete and rename that are familiar with W3C XUpdate [3]. It is clear, that any complex update operation may be expressed via these operations. We consider three kinds of insert operations. The operations *InsertInto(path, constr)*, *InsertAfter(path, constr)* and *InsertBefore(path, constr)* insert new node defined by *constr* as the last child, following sibling and preceding sibling respectively for each *target* node defined by *path*. Here *path* is an XPath operation (all axes are supported) used both in XQuery queries and update operations. The operation *Delete(path)* removes the *target* subtrees defined by *path*. The operation *Rename(path, QName)* assigns new name defined by *QName* to the *target* nodes defined by *path*. We denote the intermediate nodes and destination nodes of *path* as *Itm* and *Dst* respectively. Below in this paper, we use *II*, *IA*, *IB*, *RN*, *D* and *P* to denote *InsertInto*, *InsertAfter*, *InsertBefore*, *Rename*, *Delete* and *path* operations respectively.

3 Storage System

In this section we present Sedna XML storage system. Specifically, we focus on basic principles of data organization, versioning scheme and memory management.

3.1 Data Organization

The overall principles of data organization are illustrated in Fig. 1 (b). The central component is the DataGuide, which is presented as a tree of nodes. A DataGuide [8] is a compact structural summary of XML tree. A DataGuide describes every unique label path of a document exactly once, regardless of the number of times it appears in that document, and encodes no label path that does not appear in that document.

Each DataGuide node is labeled with an XML node kind name (e.g. element, attribute, text, etc.) and has a pointer to data pages where nodes corresponding to the DataGuide node are stored. Some DataGuide nodes depending on their node kinds are also labeled with names. Data pages belonging to one DataGuide node are linked via *pointers* into a bidirectional list.

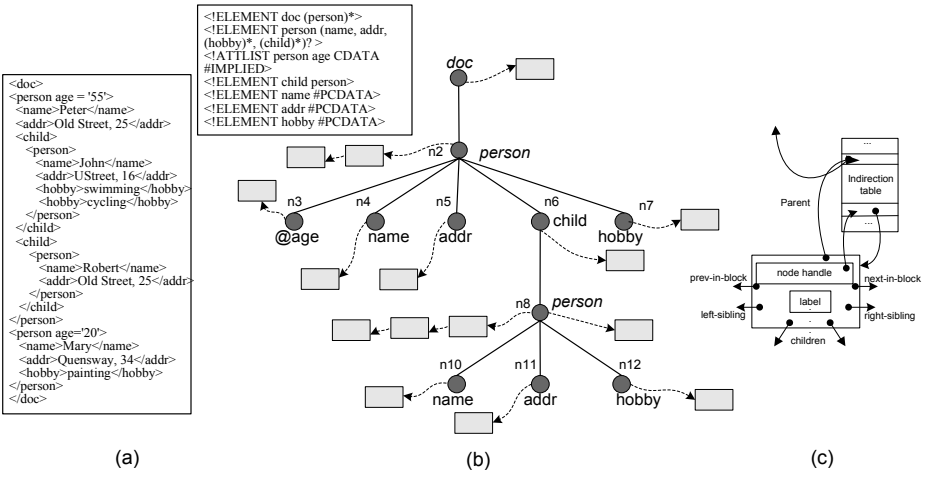


Fig. 1. An XML document Gtree and its DTD (a), Data organization scheme (b), node descriptor (c)

The structural and text parts of a node are separated. The structural part of a node reflects its relationship to other nodes (i.e. parent, children, sibling nodes) and is presented in a form of node descriptor, which structure is shown in Fig. 1 (c). The important property of node descriptors is their fixed size length in each page belonging to one DataGuide node. The text parts are stored in separated data pages and text values are a variable-length records.

All relationships between node descriptors are implemented via direct/indirect pointers. While direct pointers are preferable for querying data they are the main source of problems for efficient update execution. To support updates efficiently, we should minimize the number of modifications caused by the execution of update operation. Let us consider an operation of page splitting as a result of inserting a node into an overfilled block. If the node to be moved from one page to another page has children, they all must be modified to change their parent pointers to the node. The solution is to use indirect pointers, that is implemented via indirection table, to refer to the parent as shown in Fig. 1 (c). Note, that parent pointer is the only indirection pointer in Sedna data storage. Therefore, combining direct and indirect pointers allows to keep tradeoff between queries and updates efficiency.

The more detailed description of data organization in Sedna see in [7].

3.2 Versioning Scheme

In this subsection we describe versioning scheme implemented in Sedna. For each data item at most four versions (created by updaters) are maintained. We assume data items to be pages (further we justify our choice). Some of these versions are used as parts of logical snapshots of the database. Logical snapshot (used by queries) is a set of versions that represents transaction-consistent state

of the database at the moment of snapshot creation. In this paper we describe versioning scheme for the case of two snapshots. We will refer to them as the current snapshot and the previous snapshot.

To identify different versions we will use the following labels:

- **WV.** Working Version. This label represents version that has been created by not yet committed transaction.
- **LCV.** Last Committed Version. This is the most recent committed version.
- **CS.** Version belonging to current snapshot.
- **PS.** Version belonging to previous snapshot.

Some versions can have several labels. For example, version can be labeled as CS and PS, because it is an obsolete version that is a part of a current and previous snapshot. So, physically there may exist less than four versions of one data item. Besides, it is important to mention that labels are not physically stored in versions. Our protocol identifies version label using the auxiliry structures described below.

To dynamically identify versions we use the following structures: list containing active (i.e. not committed) transactions (we will call it $ActList$), timestamp of version (which is obtained at the moment of creation of version) and version's creator identifier (which is an identifier of corresponding updater). $ActList$ is a global list, but timestamp and identifier are version-dependent and usually are stored on the page itself. Moreover, for each snapshot we needs timestamp of its creation (T_{CS} and T_{PS} for our snapshots) and copy of $ActList$ at the moment of snapshot creation ($ActList_{CS}$ and $ActList_{PS}$).

Below we will describe algorithms how to identify LCV, WV, CS and PS versions. Consider list V, containing metadata (timestamp and creator identifier) about all versions of some data item[1]. Let us assume that this list is sorted in descending order by timestamps. We can identify versions as follows:

- **WV.** This version is always the most recent one. So if it exists, it is the first one in V. And this version exists iff the identifier of the first version in V is in the $ActList$.
- **LCV.** We have two possibilities here. If WV version exists, then LCV is the next one in V. If WV version does not exist, then LCV is the first one in V.
- **CS.** To find version belonging to current snapshot, the transaction manager (TM) first obtains sublist V' of V that contains versions with timestamps less than T_{CS}. In fact, V' contains versions that existed at the moment of snapshot creation. The problem is that this sublist can contain WV version as well, because the TM did not wait until they committed. That is where $ActList_{CS}$ comes in. Since it is a copy of $ActList$ at the moment of snapshot's creation, all that TM needs is to select LCV version from V' as described in previous item.
- **PS.** The procedure is similar to the CS version identification.

[1] In our implementation we store list V in the header of page which represents last version.

3.3 Memory Management

As pointed in the previous subsection, one of the key design choices concerning the data organization in Sedna storage system is to use direct pointers to present relationships between nodes. Therefore, traversing the nodes during query execution results in intensive pointer dereferencing. Making dereferencing as fast as possible is of crucial importance to achieve high performance. To achieve it a special memory-mapped architecture is used (see Fig. 2). The key idea of memory management in Sedna is integrating persistence with virtual memory system. To achieve this goal database address space (DAS) is divided into layers of equal size that fits virtual address space (VAS). A layer consists of pages; pages store XML data. The address of an object in DAS consists of (1) the layer number and (2) the address within the layer.

As shown in Fig. 2, an address within the layer is mapped to the address in VAS on the equality basis: the address of an object in the VAS is the address of the object within a layer. The address range of VAS is in turn mapped onto main memory by the Sedna buffer manager using memory facilities provided by the operation system. The address mapping suggested allows dereferencing the pointer very effectively, as it completely eliminates pointer swizzling overhead. An important issue here is an integration of memory-mapped architecture and versioning scheme. When a pointer dereferencing is performed and the required page is not mapped into VAS, a page-fault event is generated. As a result, the page-fault event handler is activated, which refers to the buffer manager to find an appropriate version of the required page, and then map it into VAS. It is worth mentioning that we access all versions via last version of the page, which in header contains information about all remaining versions. So, at most one additional I/O operation is performed to locate the appropriate version. The next access to the page the most likely will not generate page-fault event (because the needed version is already mapped in VAS). The mapping will be broken only if transaction requires to map a new page from other layer with the same address within the layer.

Summarizing above discussion, we conclude that an expensive procedure of locating an appropriate version will be performed relatively rarely (in most cases only when the first access to the page in transaction is occurred), and, as a result, it significantly reduces the overhead incurred by versioning scheme. Also, this is the reason why we support *page-level* granularity of versions of data item.

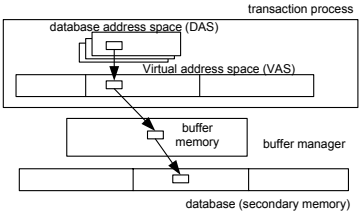


Fig. 2. Memory-mapped architecture

4 SXDGL Protocol

In this section we will describe our SXDGL protocol. The next two subsections discuss processing techniques for queries and updaters. In the last subsection we discuss some theoretical issues.

4.1 Query Processing

Queries are transactions that contain only read operations. Since such transactions are processed in a special way, the TM must know that a new transaction is a query. For efficient processing of queries SXDGL maintains two logical snapshots: the current snapshot and the previous snapshot. When a new query begins it starts reading the current snapshot, which it cannot change during execution. Moreover, it does not need to acquire any kind of locks. Since queries do not obtain any locks they never experience delays because of the data contention with updaters. Note, that queries reading the current/previous snapshot always read CS/PS versions of pages. The CS/PS version identification procedure we discussed earlier in section 3.2.

Since logical snapshots represent an obsolete state of the database, a snapshot advancement procedure should be activated periodically. To take a new snapshot the TM must have a free one, i.e. this snapshot must not be used by any transaction. New queries always start using the current snapshot (i.e. CS version). So if the current snapshot is free, the TM just takes another one by obtaining a timestamp in T_{CS} and copying the $ActList$ into $ActList_{CS}$. But if the current snapshot is used by other queries, then it must be kept intact for them during their execution. The TM uses the previous snapshot for this purpose. To backup the current snapshot it copies T_{CS} into T_{PS} and $ActList_{CS}$ into $ActList_{PS}$. However, if the previous snapshot is also in use by other queries, then the creation of a new snapshot must be delayed.

4.2 Updaters Processing

To handle concurrent updaters properly SXDGL follows the two-phase locking rule that means updater acquires an appropriate long term lock before accessing an XML-document's nodes and keeps them until the end. Besides, to ensure physical consistency of data pages we use short term latches [15] on pages, which are hold during the reading or writing the page.

SXDGL supports fine-grained lock granularity. We use DataGuide as a locking structure, i.e. we lock DataGuide nodes instead of XML-document nodes itself. Besides, SXDGL provides for annotations of locks on DataGuide nodes with simple predicates (conjunction of comparisons with constants) on XML-document nodes. SXDGL supports two kinds of long term locks: structural locks and logical locks, which we introduce in the subsections below.

Structural Locks. Below we present different types of structural locks which are used by updaters to isolate one from each other. All these locks are acquired

on the DataGuide's nodes and can be accompanied by predicates. Structural locks are divided into two classes: node locks and tree locks. The node locks impose restrictions on access to the XML-document nodes belonging to the locked DataGuide's node. The tree locks implicitly impose restrictions on access to all XML document nodes belonging to the locked DataGuide's node or DataGuide nodes located under the locked DataGuide's node.

- **P** (pass) lock. This *node lock* is used by P operation. It is set on the DataGuide's nodes, which match the *Itm* nodes of P. This lock prevents the deletion of *Itm* nodes by other updaters, but it does not prohibit any insertions of the nodes extending the sequence of *Itm* nodes (see example 1 below).
- **S** (share) lock. This *node lock* is also used by P operation. It is set on the DataGuide's nodes, which match the *Dst* nodes of P. This lock prevents *any* modifications of *Dst* nodes. For example, for query *count(/doc/person)* S lock is set on the node *n2* (see Fig. 1 (b)).
- **SI** (shared into) lock. This *node lock* is used by II operation. It is set on the DataGuide's nodes, which match the target nodes of II. This lock prevents the modification of II's target nodes and insertion of another nodes *into* the target nodes by concurrent transactions. The **SA** (shared after) and **SB** (shared before) locks are defined in a similar way.
- **XN** (exclusive new) lock. This *node lock* is used by insert operations. It is set on the DataGuide's node, which matches the newly created nodes. This lock allows passing by the new node, (i.e it compatible with P lock), but other operations (e.g. modification) on the node are prohibited.
- **X** (exclusive) lock. This *node lock* is used by operations, which modify internal nodes of the document. This lock prevents any concurrent reads and updates of the node. In our set of update operations this lock is used by RN operation since it renames the target nodes inside the document. It is set on the DataGuide's nodes which correspond to the RN's target and new nodes. Note, if we rename the leaf node then we only need to acquire X lock on the RN's target nodes and XN lock on the new nodes.
- **ST** (shared tree), **XT** (exclusive tree) locks. The ST (XT) lock is a *tree* lock. The ST (XT) lock is set on the DataGuide's node and implicitly locks all its descendants in shared (exclusive) mode. The ST (XT) lock prevents any updates (reads and updates) in the entire subtree. For example, XT lock is set on target nodes of D operation.
- **IS** (intention shared), **IX** (intention exclusive) locks. The IS (IX) lock must be obtained on each ancestor of the node, which is to be locked in one of the shared (exclusive) modes. It ensures that there are no locks on the coarser granules locking the node in conflicting mode.

To deal with value-based constraints, each structural lock has an annotated value-based predicate. The predicates locks are managed efficiently by approximating predicates using signatures [5]. The compatibility matrix for structural locks is shown in the Fig. 3. There are no strict incompatibilities in matrix.

	granted										
requested	S	P	SI	SA	SB	XN	X	ST	XT	IS	IX
S	+	+	+	+	cp	cp	+	cp	+	+	
P	+	+	+	+	+	cp	+	cp	+	+	
SI	+	+	cp	+	+	cp	cp	+	cp	+	+
SA	+	+	+	cp	+	cp	cp	+	cp	+	+
SB	+	+	+	+	cp	cp	cp	+	cp	+	+
XN	cp	+	cp	cp	cp	+	cp	cp	cp	+	+
X	cp	cp	cp	cp	cp	cp	cp	cp	cp	+	+
ST	+	+	+	+	+	cp	cp	+	cp	+	cp
XT	cp	cp	cp	cp	cp	cp	cp	cp	cp	cp	cp
IS	+	+	+	+	+	+	+	+	cp	+	+
IX	+	+	+	+	+	+	+	cp	cp	+	+

Fig. 3. Structural locks compatibility matrix

Symbol *CP* (check predicates) in matrix means that the requested lock is compatible with granted locks only if conjunction of annotated predicates is not satisfiable.

Next we study a couple of examples (using XML document presented in Fig. 1) to illustrate the locking mechanisms[2].

Example 1. Let us consider transactions `T1={II(/doc ,<person/>)}`, `T2={II(/doc,<person/>)}` and `T3={/doc/person/name}`. According to SXDGL the following sets of structural locks `{n1: (SI, IX), n2: XN}`, `{n1: (SI, IX), n2: XN}` and `{n1: P, n2: P, n4: ST}` must be obtained by transactions `T1`, `T2` and `T3` respectively. Thus, we obtain that `T1`, `T3` and `T2`, `T3` can run concurrently whereas `T1` and `T2` can not run concurrently due to the conflict of `SI` locks on node `n1` (thus we prevent document order conflict).

Example 2. For transactions `T1=D(/doc/person/@age)}` and `T2={for $v in //person return <name2>{$v/ name}</name2>}` we need to obtain the sets of structural locks `{n1: (IX, P), n2: (IX, P),n3:XT}` and `{n1: IS, n6: IS, n2: S, n8: S, n4: ST, n10: ST}` respectively. Since locks required by `T1` and `T2` are compatible, we conclude that `T1` and `T2` can be executed concurrently.

Logical Locks. Now we turn to the discussion of the logical locks, which are used to prevent phantoms. Let us show how a phantom could appear. Suppose that transaction `T1` reads all of *age* attributes in *GTree* XML-document (see Fig. 1), i.e. `T1` issued *//@age* query. In the meantime transaction `T2` inserts new *age* attribute into *person* element with *name* 'John'. The new *age* attribute is the phantom for transaction `T1`. Generally speaking, phantoms can appear when update operation extends the DataGuide[3] (adds new path to it) and this modification results in the changing of target nodes of previously executed operations.

Thus, we introduce two locks. The first lock is L (logical) lock, which must be set on DataGuide's node to protect the node's subtrees in the document from a phantom appearance. A logical lock specifies a set of *properties*. Essentially, a property is a logical condition on nodes. This lock prohibits the insertion of new nodes, which possess these properties. The second lock is IN (insert new node) lock, which specifies the properties of new node. The update operation,

[2] Note, that in example 2 we omit logical locks.

[3] In our set of update operations *II*, *IA*, *IB* and *RN* operations can extend the DataGuide.

which extends the DataGuide, should obtain the IN lock on each ancestor of the new node.

Here we list all possible combinations of properties for L lock: (1) node-name='name1' (e.g. *//person*), (2) node-name='name1', node-value *relop* 'val1' (e.g. *//name[.≠'John']*), (3) node-name='name1', child-name='name2', child-value *relop* 'val1' (e.g. *//person[name ≠ 'John']*). Here *relop* is a comparison operation.

To check that the new node's properties do not interfere with the L lock properties, the IN lock should specify three properties of a new node: new-node-parent-name, new-node-name, new-node-value.

Thus, L and IN locks are incompatible if one of the following conditions holds:

- If IN's new-node-name equals to a node-name of L lock and L does not contain any other properties (case (1) from the above)[4].
- If IN's new-node-name and new-node-value both match appropriate values of L lock consisting of two properties. That is, node-name=new-node-name and new-node-value relop 'val1' ≠ #f (case (2) from the above).
- If all IN's properties match three properties of L lock. That is node-name= new-node-parent-name, child-name=new-node-name and new-node-value relop 'val1' ≠ #f (case (3)).

If node's name is a wildcard '*' then it equals to any node-name.

DTD-Based Concurrency Enhances. The main benefit of using DataGuide as a locking structure is that the locking overhead is reduced significantly. However, there are some drawbacks. The problem with DataGuide is that it lacks the notion of document order. Therefore, the evaluation of ordered-based axes (e.g. preceding-sibling, following-sibling) on DataGuide may result in the lock of unnecessary DataGuide nodes. As a consequence, we can get unnecessary conflicts among transactions.

To reduce the number of such conflicts we use DTD, which specifies the document order. For instance, on the basis of the *GTree*'s DTD information `<!ELEMENT person (name, addr, (hobby)*, (child)*)?!>` we can resolve the conflict between transactions `T1={/doc/person/ addr/preceding-sibling::*}` and `T2={D(/doc/person/hobby)}`. Indeed, without DTD information `T1` must acquire `ST` lock on all DataGuide's nodes of level 2 (`n3`, `n4`, `n5`, `n6`, `n7`), whereas `T2` must obtain `XT` lock on node `n7` of level 2. Thus, there is a conflict between `T1` and `T2` on node `n7`. But using DTD we know that preceding sibling of *addr* element can only be *name* element, and `T1` must acquire `ST` lock only on node `n4` thereby allowing `T2` to run concurrently.

Transaction Dependency Graph. In this subsection we discuss some complications, which arise due to the fact that a granularity of locks is less than a granularity of versions. Specifically, there may be a situation when some pages

[4] In case of RN operation we compare only new-node-name property of IN lock with node-name property of L lock. Another properties of IN lock are not considered.

would always keep updates produced by a non-committed updater. As a result, last updates performed by already committed updaters would not be visible for queries even if the snapshot advancement procedure is taken regularly. Please note that queries must always see a transaction-consistent snapshot.

To cope with this problem we introduce a transaction dependency graph (we call it *TDG*). *TDG* is an undirected graph whose nodes are the updaters. The new edge between updaters U_i and U_j is added in *TDG* if the following rules hold: (1) $U_i, U_j \in ActList$, (2) U_i tries to update the page that already has been updated by noncommitted U_j.

To provide an efficient way to check the (1) and (2) rules we use a lock manager. Recall, that before updating a page updater must acquire an X latch on this page. When updater releases the X latch from the page the TM converts it to C lock. The C locks are compatible with every other types of locks/latches, even with the exclusive ones. So C locks are not locks in the common sense of this word, but they can be easily implemented within a lock manager. The C locks are released by updaters only at commit time. So, a new edge between U_i and U_j is added in *TDG* if (1) U_i tries to update a page, and (2) there is a C lock on that page hold by U_j.

Besides, before updater U commits, it must perform the following actions:

1. If $U \in TDG$ then mark the node in *TDG* corresponding to U with P ("prepared").
2. If $U \notin TDG$ then remove U from *ActList*.

So, after U commits, it is removed from *ActList* and would be visible for new queries after next snapshot advance only if it is not a member of *TDG*. But updates produced by updaters that are the members of *TDG* would not be visible for queries. However, when the maximum subgraph *SG* of *DTG* (i.e. there is no node $N \in TDG \wedge N \notin SG$ such that the edge $(N, K) \in TDG \wedge K \in SG$) consists of nodes all of which are marked with P, then the *SG* subgraph can be removed from *TDG*, and the corresponding U's can be removed from *ActList*. If the number of *TDG*'s nodes exceeds a given threshold the TM turn on a propagation mode. In propagation mode the TM changes the compatibility rules for C lock. Particularly, it requires the incompatibility of the X and C latches. It guarantees that *TDG* will not grow infinitely and after certain period of time it will be empty. Then the TM again turn on a normal mode, i.e. C locks become compatible with any other kind of latches/locks.

Also, it is significant that updater creates a new version of page only in case if the last version is committed. Otherwise, it performs update in last version of the page.

4.3 Correctness

Theorem 1. *SXDGL guarantees serializability of all transactions.*

Due to space restrictions we skip the formal proof of this theorem. A formal proof of this theorem can be found in extended version of this paper [16]. Note,

however, that serializability is the key feature for any concurrency control protocol. Many applications rely on serializability and cannot simply sacrifice it for performance gains.

5 Experimental Evaluation

In this section we present an experimental evaluation of SXDGL. We compare it with the conventional two-phase multigranularity locking protocol (we call it DGL), which employs DataGuide as a locking structure. To conduct such evaluation we have prototyped both protocols in Sedna XML database. Sedna server was installed on Intel Pentium 4 2.8GHz, 512Mbs RAM machine, run under Windows XP. We use a standard XMark tool [1] to generate XML-documents. The 100Mb XML-documents were loaded into Sedna database and then we run experiments using this database. Since XMark benchmark does not cover updates, we generated our own update operations, e.g. converting open auction to a closed auction, adding new bidders, changing shipment types, etc.

In our experiments workload contains 25% of queries and 75% of updaters running concurrently. Selectivity factor for queries is varied from 5% to 55% of the whole XML-document, but within the same experimental run it stays fixed. Updaters on the other hand are relatively short, containing approximately 20 update operations.

Fig. 4 (a) shows updaters throughput. As we can see, for SXDGL selectivity is not an issue. Queries just read snapshots, while updaters write new versions. Updaters do not experience long delays, because they do not intersect with queries. Any delays would be because of data contention between updaters themselves, but the number of such delays is reduced significantly by means of smart detection of conflicts. Data contention becomes a problem for DGL, though. In this case queries and updaters read the same nodes. In fact, the queries are executed as updaters here, obtaining an appropriate lock before accessing nodes in XML-document. It does not create any problems when query selectivity is about 5%, because in this case queries are relatively small and data contention is not high enough to cause problems. However, increasing query selectivity also increases execution time of the queries and data contention. At 55% selectivity almost all new updaters conflict with queries and, as a result, spend most of the time waiting for them. The data contention problem for updaters run under DGL is increased because of the great number of unnecessary conflicts with queries. The most artificial conflicts arise because DGL always uses tree locks, but in most cases updaters do not read/update the whole subtrees.

Fig. 4 (b) shows query throughput. Here we see that DGL and SXDGL demonstrates approximately the same query throughput. In fact, this experiment compares the overhead of query processing for DGL and SXDGL. For DGL the overhead consists of locking on DataGuide's nodes and delays caused by concurrent updaters. For SXDGL the overhead includes one possible additional read operation of a page on each basic read. In this case the query first reads an LCV version, which contains information about all remaining versions, determines

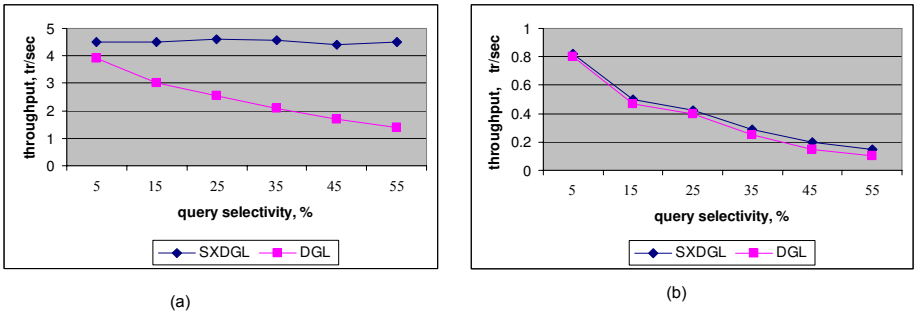


Fig. 4. Updater throughput (a) and query throughput (b)

the version for reading and reads it then. If an LCV version is a suitable version itself, no additional read operation occurs. As we can see in Fig. 4 (b), the DGL overhead slightly exceeds multiversioning overhead of SXDGL. So, query throughtput is slightly lower under DGL.

Also, we would like to mention that we also evaluated the locking overhead in DGL and SXDGL. The experiments confirmed that the locking overhead is much smaller than the overhead incurred by delays caused by concurrent updaters (under DGL) and additional I/O operations (under SXDGL), respectively.

Summarizing our experiments show that SXDGL has benefits in both query and updater processing. More importantly, it offers significantly higher updater throughput.

6 Related Work

Various concurrency control protocols have been developed to deal with XML data effectively. Most of them are based on the multigranularity locking techniques, which are extensively used in relational databases.

In [10,12] authors presented locking protocols, which introduce new lock modes tailored especially for efficient concurrent processing of DOM-like operations. Moreover, in [11] Haustein et al. proposed a new DeweyID labeling scheme to acquire intention locks efficiently. These protocols seem to work perfectly for DOM-like operations, but there is no research done whether they could suite well for concurrent processing of XQuery/XUpdate operations.

The protocols developed in [6,4,14] rely on the assumption that XML is accessed by means of XPath query language. They propose to use "path locks" to achieve higher concurrency. However, these methods have a number of shortcomings. In [6], the authors deal with too restrictive subset of XPath. The other works deal with more complex XPath queries, but the conflict determination for them becomes too expensive. Besides, these methods cause excessive locking overhead if we have to deal with huge XML-documents.

Another approach to deal with concurrent processing of XML-transactions in RDBMSs was presented in [9]. The authors proposed DGLOCK protocol, which

employs DataGuide structure for locking purposes. Unfortunately, DGLOCK protocol is suitable only for limited subset of XPath excluding such important axes like descendant, following-sibling, preceding-sibling, etc. Moreover, DGLOCK uses conventional multigranularity locking scheme, which leads to a great number of pseudoconflicts. Finally, DGLOCK protocol does not ensure serializability of produced schedules. In [18] the authors extended Grabs et al.'s approach and eliminated the most their shortcomings. However, the applicability of these methods is restricted to RDBMSs with XML support while in this paper we focus on concurrency control for native XML databases.

In our earlier work [17] we presented XDGL, an XPath-based locking protocol, which also uses DataGuide as a locking structure. In that work a full set of XPath axes was considered. Actually, the SXDGL protocol presented in this paper is an extension of XDGL in a number of different directions. Firstly, in SXDGL we consider XQuery as a query language. Secondly, we extended SDXGL with support of multiversioning that enables to execute queries without unnecessary delays. Finally, SXDGL introduces new lock modes, which allow to capture the semantics of XQuery/XUpdate operations determining conflicts between updaters. So, a number of pseudoconflicts is significantly reduced.

To the best of our knowledge, there is only one short paper [19], which suggests multiversioning approach for concurrency control in XML. The authors proposed SnaX protocol. Unfortunately, due to space limitations the authors did not present many important details about smart conflict detection and implementation. The overhead of their protocol is not evaluated. Moreover, SnaX protocol does not guarantee serializability of produced schedules.

7 Conclusion

In this paper we presented a new approach, called snapshot based XML DataGuide locking protocol (SXDGL), which supports efficient processing of concurrent transactions in native XML database. SXDGL supports two classes of transactions: queries and updaters. SXDGL guarantees non-interference of queries and updaters, but queries still may access a slightly obsolete data. A special memory-mapped architecture is introduced to reduce the multiversioning overhead.

Our protocol introduces smart detection of conflicts between updaters exploiting semantics of XQuery/XUpdate operations. Moreover, we use DataGuide as a locking structure, which usually relatively small, and, as a result, the locking overhead is significantly reduced as opposed to approaches that use XML-document nodes for locking purpose. However, in case of complex predicates used in XQuery/XUpdate operations the granularity of locking may be slightly more coarse-grained than a individual node. Hence SXDGL allows to keep trade-off between granularity of locks and locking overhead.

Finally, we prototyped SXDGL in Sedna and conducted an experimental evaluation, which confirms that SXDGL provides an efficient solution to the problem of processing of concurrent transactions in native XML database.

References

1. XMark - An XML Benchmark Project. http://monetdb.cwi.nl/xml/
2. Chamberlin, D., Carey, M., Florescu, D., Kossmann, D., Robie, J.: XQueryP: Programming with XQuery. In: XIME-P Workshop (2006)
3. Chamberlin, D., Florescu, D., Robie, J.: XQuery Update Facility. W3C Consortium (2006), http://modis.ispras.ru/sedna/
4. Choi, H.: XPath-based Concurrency Control for XML Data. In: DEWS (2003)
5. Dadam, P., Pistor, P., Schek, H.-J.: A predicate oriented locking approach for integrated information systems. In: IFIP Congress, pp. 763–768 (1983)
6. Dekeyser, S., Hidders, J.: Conflict scheduling of transactions on XML documents. In: ADC, pp. 93–101 (2004)
7. Fomichev, A., Grinev, M., Kuznetsov, S.: Sedna: a Native XML DBMS. In: SOFSEM, pp. 272–281 (2006)
8. Goldman, R., Widom, J.: Dataguides: Enabling Query Formulation and Optimization in Semistructured Databases. In: VLDB, pp. 436–445 (1997)
9. Grabs, T., Böhm, K., Schek, H.-J.: XMLTM: Efficient Transaction Management for XML Documents. In: CIKM, pp. 142–152 (2002)
10. Haustein, M.P., Härder, T.: Adjustable transaction isolation in XML database management systems. In: Bellahsène, Z., Milo, T., Rys, M., Suciu, D., Unland, R. (eds.) XSym 2004. LNCS, vol. 3186, pp. 173–188. Springer, Heidelberg (2004)
11. Haustein, M.P., Härder, T., Mathis, C., 0002, M.W.: Deweyids - the key to fine-grained management of XML documents. SBBD, 85–99 (2005)
12. Helmer, S., Kanne, C.-C., Moerkotte, G.: Lock-based protocols for cooperation on XML documents. In: DEXA Workshops, pp. 230–234 (2003)
13. ISP RAS. Sedna, a Native XML Database. http://modis.ispras.ru/sedna/
14. Jea, K.-F.J., Chen, S.-Y., Wang, S.-H.: Concurrency control in XML document databases: XPath locking protocol. In: ICPADS, pp. 551–556 (2002)
15. Mohan, C., Haderle, D.J., Lindsay, B.G., Pirahesh, H., Schwarz, P.M.: ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. Database Syst. 17(1), 94–162 (1992)
16. Pleshachkov, P., Kuznetsov, S.: SXDGL: Snapshot based Concurrency Control Protocol for XML Data (extended version). Technical Report, ISP RAS (2006), http://modis.ispras.ru/downloads/publications/sxdgl-extended.pdf
17. Pleshachkov, P., Chardin, P., Kuznetsov, S.: A DataGuide-based concurrency control protocol for cooperation on XML data. In: Eder, J., Haav, H.-M., Kalja, A., Penjam, J. (eds.) ADBIS 2005. LNCS, vol. 3631, pp. 268–282. Springer, Heidelberg (2005)
18. Pleshachkov, P., Kuznetsov, S.: Transaction management in RDBMSs with XML support. Programming and Computing Software. 32(5), 3–20 (2006)
19. Sardar, Z., Kemme, B.: Don't be a pessimist: Use snapshot based concurrency control for XML. In: ICDE (poster paper), p. 130 (2006)
20. Thomasian, A.: Performance limits of two-phase locking. In: ICDE, pp. 426–435 (1991)
21. Weikum, G., Vossen, G.: Transactional information systems. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)

The Generation Y of XML Schema Matching
Panel Description

Avigdor Gal

Technion – Israel Institute of Technology
avigal@ie.technion.ac.il

Schema matching is the task of matching between concepts describing the meaning of data in various heterogeneous, distributed data sources (*e.g.* XML DTDs and XML Schemata). Schema matching is recognized to be one of the basic operations required by the process of data integration [3], and thus has a great impact on its outcome. Schema mappings (the outcome of the matching process) can serve in tasks of generating global schemata, query rewriting over heterogeneous sources, duplicate data elimination, and automatic streamlining of workflow activities that involve heterogeneous data sources. As such, schema matching has impact on numerous applications. It impacts business, where company data sources continuously realign due to changing markets. It also impacts life sciences, where scientific workflows cross system boundaries more often than not.

The BABY-BOOM GENERATION of schema matching involves two decades of research, summarized in surveys (*e.g.*, [19,6,21]) and various online lists (*e.g.*, OntologyMatching[1], Ziegler[2], DigiCULT[3], SWgr[4]). A significant body of work was devoted to the identification of *schema matchers*, heuristics for schema matching. Examples include COMA [7], Cupid [13], OntoBuilder [11], Autoplex [2], Similarity Flooding [15], Clio [16], Glue [8], to name just a few. The main objective of schema matchers is to provide schema mappings that will be effective from the user point of view, yet computationally efficient (or at least not disastrously expensive). Such research has evolved in different research communities, including databases, information retrieval, information sciences, data semantics, and others. Research papers in different communities yielded overlapping, similar, and sometimes identical results.

Some works in XML schema matching (*e.g.*, Cupid) use the rich(er) structure of XML in generating heuristics for better matching. Others (*e.g.*, [5,22]) aim at modeling and using the semantics that can be derived from XML schemata to map attributes. Efficiency was also tackled in [20,22]. Systems like XLearner [17] and SMART [18] provide more matching techniques for purposes such as query rewriting.

The rise of GENERATION X of schema matching in recent years stems from the understanding that the approaches taken by baby boomers did not deliver

[1] http://www.ontologymatching.org/
[2] http://www.ifi.unizh.ch/~pziegler/IntegrationProjects.html
[3] http://www.digicult.info/pages/resources.php?t=10
[4] http://www.semanticweb.gr/modules.php?name=News&file=categories&op=newindex&catid=17

D. Barbosa et al. (Eds.): XSym 2007, LNCS 4704, pp. 137–139, 2007.

satisfactory results as of yet [4,9]. For example, on a recent OAEI benchmark of a real-life matching task[5], participating matchers provide 30-40% precision and recall of 13-45%. The reason may be that the right "silver bullet" is yet to be found among the existing approaches or that we have been searching in the wrong place all along. More rigorous attempts at formalizing the problem of schema matching were performed [3,14,12,10,1]. Aspects of uncertainty and inferencing were given more attention, striving to better understand the difficulties this area is facing.

While Generation X is still here, and not yet fulfilled its potential, this panel will investigate the next generation of schema matching, Generation Y. The panelists will provide their vision on the following three questions:

- **The true goal of schema matching:** Are we aiming at a fully automatic or a semi automatic schema matching? are we aiming at 100% precision? 100% recall? a little bit of both?
- **The non-linear climb from 30% precision and recall to 80-90%**: What does it take to reach 40% precision and recall? 50%?...
- **Promising directions:** will Generation Y be defined by the success or failure of Generation X? will it take a totally new approach?

References

1. Benerecetti, M., Bouquet, P., Zanobini, S.: Soundness of schema matching methods. In: Gómez-Pérez, A., Euzenat, J. (eds.) ESWC 2005. LNCS, vol. 3532, pp. 211–225. Springer, Heidelberg (2005)
2. Berlin, J., Motro, A.: Autoplex: Automated discovery of content for virtual databases. In: Batini, C., Giunchiglia, F., Giorgini, P., Mecella, M. (eds.) CoopIS 2001. LNCS, vol. 2172, pp. 108–122. Springer, Heidelberg (2001)
3. Bernstein, P.A., Melnik, S.: Meta data management. In: Proceedings of the IEEE CS International Conference on Data Engineering, IEEE Computer Society Press, Los Alamitos (2004)
4. Bernstein, P.A., Melnik, S., Petropoulos, M., Quix, C.: Industrial-strength schema matching. SIGMOD Record 33(4), 38–43 (2004)
5. Bossung, S., Stoeckle, H., Grundy, J.C., Amor, R., Hosking, J.G.: Automated data mapping specification via schema heuristics and user interaction. In: 19th IEEE International Conference on Automated Software Engineering (ASE 2004), Linz, Austria, 20-25 September 2004, pp. 208–217. IEEE Computer Society Press, Los Alamitos (2004)
6. Do, H., Melnik, S., Rahm, E.: Comparison of schema matching evaluations. In: Proceedings of the 2nd Int. Workshop on Web Databases (German Informatics Society), 2002 (2002)
7. Do, H.H., Rahm, E.: COMA - a system for flexible combination of schema matching approaches. In: Proceedings of the International conference on Very Large Data Bases (VLDB), pp. 610–621 (2002)
8. Doan, A., Madhavan, J., Domingos, P., Halevy, A.: Learning to map between ontologies on the semantic web. In: Proceedings of the eleventh international conference on World Wide Web, pp. 662–673. ACM Press, New York (2002)

[5] http://oaei.ontologymatching.org/2006/results/directory/

9. Gal, A.: Why is schema matching tough and what can we do about it? SIGMOD Record 35(4), 2–5 (2007)
10. Gal, A., Anaby-Tavor, A., Trombetta, A., Montesi, D.: A framework for modeling and evaluating automatic semantic reconciliation. VLDB Journal 14(1), 50–67 (2005)
11. Gal, A., Modica, G., Jamil, H.M., Eyal, A.: Automatic ontology matching using application semantics. AI Magazine 26(1), 21–32 (2005)
12. Madhavan, J., Bernstein, P.A., Domingos, P., Halevy, A.Y.: Representing and reasoning about mappings between domain models. In: Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI), pp. 80–86 (2002)
13. Madhavan, J., Bernstein, P.A., Rahm, E.: Generic schema matching with Cupid. In: Proceedings of the International conference on Very Large Data Bases (VLDB), Rome, Italy, pp. 49–58 (September 2001)
14. Melnik, S.: Generic Model Management: Concepts and Algorithms. Springer, Heidelberg (2004)
15. Melnik, S., Rahm, E., Bernstein, P.A.: Rondo: A programming platform for generic model management. In: Proceedings of the ACM-SIGMOD conference on Management of Data (SIGMOD), San Diego, California, pp. 193–204. ACM Press, New York (2003)
16. Miller, R.J., Hernàndez, M.A., Haas, L.M., Yan, L.-L., Ho, C.T.H., Fagin, R., Popa, L.: The Clio project: Managing heterogeneity. SIGMOD Record 30(1), 78–83 (2001)
17. Morishima, A., Kitagawa, H., Matsumoto, A.: A machine learning approach to rapid development of xml mapping queries. In: Proceedings of the IEEE CS International Conference on Data Engineering, pp. 276–287. IEEE Computer Society Press, Los Alamitos (2004)
18. Morishima, A., Okawara, T., Tanaka, J., Ishikawa, K.: Smart: a tool for semantic-driven creation of complex xml mappings. In: Proceedings of the ACM-SIGMOD conference on Management of Data (SIGMOD), pp. 909–911. ACM Press, New York (2005)
19. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. VLDB Journal 10(4), 334–350 (2001)
20. Rahm, E., Do, H.H., Massmann, S.: Matching large xml schemas. SIGMOD Record 33(4), 26–31 (2004)
21. Shvaiko, P., Euzenat, J.: A survey of schema-based matching approaches. Journal of Data Semantics 4, 146–171 (2005)
22. Smiljanic, M., van Keulen, M., Jonker, W.: Formalizing the xml schema matching problem as a constraint optimization problem. In: Andersen, K.V., Debenham, J., Wagner, R. (eds.) DEXA 2005. LNCS, vol. 3588, pp. 333–342. Springer, Heidelberg (2005)

Author Index