Zohra Bellahsène
Akmal B. Chaudhri
Erhard Rahm
Michael Rys
Rainer Unland (Eds.)

LNCS 2824

Database
XML Techn

First International XML Databa
Berlin, Germany, September 20
Proceedings

Lecture Notes in Computer Science 2824

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Zohra Bellahsène Akmal B. Chaudhri
Erhard Rahm Michael Rys
Rainer Unland (Eds.)

Database and XML Technologies

First International XML Database Symposium, XSym 2003
Berlin, Germany, September 8, 2003
Proceedings

Springer

Volume Editors

Zohra Bellahsène
LIRMM UMR 5506 CNRS/Université Montpellier II
161 Rue Ada, 34392 Montpellier, France
E-mail: bella@lirmm.fr

Akmal B. Chaudhri
IBM developerWorks
6 New Square, Bedfont Lakes, Feltham, Middlesex TW14 8HA, UK
E-mail: akmal.b.chaudhri@uk.ibm.com

Erhard Rahm
University of Leipzig
Augustusplatz 10-11, 04109 Leipzig, Germany
E-mail: rahm@informatik.uni-leipzig.de

Michael Rys
Microsoft Corporation
One Microsoft Way, Redmond, WA 98052, USA
E-mail: rys@acm.org, mrys@microsoft.com

Rainer Unland
University of Duisburg-Essen
Institute for Computer Science and Business Information Systems
Schützenbahn 70, 45117 Essen, Germany
E-mail: UnlandR@informatik.uni-essen.de

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

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

ISSN 0302-9743
ISBN 3-540-20055-X Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin GmbH
Printed on acid-free paper SPIN: 10953624 06/3142 5 4 3 2 1 0

PREFACE

The Extensible Markup Language (XML) is playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. The database community is interested in XML because it can be used to represent a variety of data formats originating in different kinds of data repositories while providing structure and the possibility to add type information.

The theme of this symposium is the combination of database and XML technologies. Today, we see growing interest in using these technologies together for many Web-based and database-centric applications. XML is being used to publish data from database systems on the Web by providing input to content generators for Web pages, and database systems are increasingly being used to store and query XML data, often by handling queries issued over the Internet. As database systems increasingly start talking to each other over the Web, there is a fast-growing interest in using XML as the standard exchange format for distributed query processing. As a result, many relational database systems export data as XML documents, import data from XML documents, provide query and update capabilities for XML data. In addition, so-called native XML database and integration systems are appearing on the database market, and it's claimed that they are especially tailored to store, maintain and easily access XML documents.

The first XML Database Symposium, XSym 2003, is a new forum on the combination of database and XML technologies. It is built on several previous XML, Web and database-related workshops that were held at the CAiSE 2002, EDBT 2002, NODe 2002 and VLDB 2002 conferences. The goal of this symposium is to provide a high-quality platform for the presentation and discussion of new research results and system developments. It is targeted at scientists, practitioners, vendors, and users of XML and database technologies.

The call-for-papers attracted 65 submissions from all over the world. After a careful reviewing process, the international program committee accepted 18 high-quality papers of particular relevance and quality. The selected contributions cover a wide range of exciting topics, in particular XML query processing, stream processing, XML-relational mappings, index structures, change management, and new prototypes. Another highlight of the symposium was the keynote by Mike Franklin, University of Berkeley.

As editors of this volume, we would like to thank once again all program committee members and all the external referees who gave up their valuable time to review the papers and helped in putting together an exciting program. We would also like to thank the invited speaker, authors and other individuals, without whom this

symposium would not have been possible. Moreover, our thanks go out to the local organizing committee who fulfilled with a lot of patience all our wishes. Finally, we would like to thank Alfred Hofmann from Springer-Verlag for his friendly cooperation and help in putting this volume together.

July 2003 Montpellier, Bedfont Lakes, Redmond, Leipzig, Essen,

Zohra Bellahsene (General Chair)

Akmal Chaudhri (Program Committee Co-chair)

Michael Rys (Program Committee Co-chair)

Erhard Rahm (Publicity and Communications Chair)

Rainer Unland (Publications Chair)

Program Committee

Bernd Amann, CNAM and INRIA (France)
Valeria De Antonellis, Politecnico di Milano (Italy)
Zohra Bellahsene, LIRMM (France)
Elisa Bertino, University of Milan (Italy)
Timo Böhme, University of Leipzig (Germany)
Akmal B. Chaudhri, IBM developerWorks (USA)
Sophie Cluet, INRIA (France)
Istvan Cseri, Microsoft (USA)
Gillian Dobbie, University of Auckland (New Zealand)
Mary F. Fernandez, AT&T Research (USA)
Daniela Florescu, BEA (USA)
Irini Fundulaki, Bell Labs/Lucent Technologies (USA)
Udo Kelter, University of Siegen (Germany)
Donald Kossmann, Technical University of Munich (Germany)
Mong Li Lee, National University of Singapore (Singapore)
Eng Wah Lee, Gintic (Singapore)
Stuart Madnick, MIT (USA)
Ioana Manolescu, INRIA (France)
Jim Melton, Oracle (USA)
Alberto Mendelzon, University of Toronto (Canada)
Laurent Mignet, University of Toronto (Canada)
Tova Milo, Tel Aviv University (Israel)
Guido Moerkotte, Universität Mannheim (Germany)
Allen Moulton, MIT (USA)
M. Tamer Öszu, University of Waterloo (Canada)
Shankar Pal, Microsoft (USA)
Erhard Rahm, University of Leipzig (Germany)
Michael Rys, Microsoft (USA)
Jérôme Siméon, Bell Labs (USA)
Zahir Tari, RMIT (Australia)
Frank Tompa, University of Waterloo (Canada)
Hiroshi Tsuji, Osaka Prefecture University (Japan)
Can Türker, Swiss Federal Institute of Technology, Zurich (Switzerland)
Rainer Unland, University of Essen (Germany)
Agnes Voisard, Fraunhofer ISST and Freie Universitaet Berlin (Germany)
Osamu Yoshie, Waseda University (Japan)
Jeffrey Xu Yu, Chinese University of Hong Kong (Hong Kong)

External Reviewers

Sharon Adler, IBM (USA)
Marcelo Arenas, University of Toronto (Canada)
Denilson Barbosa, University of Toronto (Canada)
Omar Benjelloun, INRIA (France)
David Bianchini, University of Brescia (Italy)
Ronald Bourret, Independent Consultant (USA)
Lei Chen, University of Waterloo (Canada)
Grégory Cobena, INRIA (France)
David DeHaan, University of Waterloo (Canada)
Hai Hong Do, University of Leipzig (Germany)
Rainer Eckstein, Humboldt-Universität zu Berlin (Germany)
Elena Ferrari, Università dell'Insubria, Como (Italy)
Leo Giakoumakis, Microsoft (USA)
Giovanna Guerrini, Università di Pisa (Italy)
Jim Kleewein, IBM (USA)
Sailesh Krishnamurthy, IBM (USA)
Allen Luniewski, IBM (USA)
Ingo Macherius, Infonyte (Germany)
Susan Malaika, IBM (USA)
Florent Masseglia, INRIA (France)
Michele Melchiori, University of Brescia (Italy)
Rosa Meo, Università di Torino (Italy)
Benjamin Nguyen, INRIA-FUTURS (France)
Yong Piao, University of Siegen (Germany)
Awais Rashid, Lancaster University (UK)
Mark Roantree, Dublin City University (Ireland)
Kenneth Salem, University of Waterloo (Canada)
Torsten Schlieder, Freie Universität Berlin (Germany)
Bob Schloss, IBM (USA)
Soumitra Sengupta, Microsoft (USA)
Xuerong Tang, University of Waterloo (Canada)
Alejandro Vaisman, University of Toronto (Canada)
Pierangelo Veltri, INRIA (France)
Brian Vickery, IBM (USA)
Sabrina De Capitani di Vimercati, Università degli Studi di Milano (Italy)
Florian Waas, Microsoft (USA)
Norbert Weissenberg, Fraunhofer ISST (Germany)
Liang-Huai Yang, National University of Singapore (Singapore)
Hui Zhang, University of Waterloo (Canada)
Ning Zhang, University of Waterloo (Canada)
Hongwei Zhu, MIT (USA)

Table of Contents

XML–Relational DBMS

XML Query Processing

Systems and Tools

XML Access Structures

Stream Processing and Updates

Design Issues

XML-to-SQL Query Translation Literature: The State of the Art and Open Problems

Rajasekar Krishnamurthy, Raghav Kaushik, and Jeffrey F. Naughton

University of Wisconsin-Madison
{sekar,raghav,naughton}@cs.wisc.edu

Abstract. Recently, the database research literature has seen an explosion of publications with the goal of using an RDBMS to store and/or query XML data. The problems addressed and solved in this area are diverse. This diversity renders it difficult to know how the various results presented fit together, and even makes it hard to know what open problems remain. As a first step to rectifying this situation, we present a classification of the problem space and discuss how almost 40 papers fit into this classification. As a result of this study, we find that some basic questions are still open. In particular, for the XML publishing of relational data and for "schema-based" shredding of XML documents into relations, there is no published algorithm for translating even simple path expression queries (with the // axis) into SQL when the XML schema is recursive.

1 Introduction

Beginning in 1999, the database research literature has seen an explosion of publications with the goal of using an RDBMS to store and/or query XML data. The problems addressed and solved in this area are diverse. Some publications deal with using an RDBMS to store XML data; others deal with exporting existing relational data in an XML view. The papers use a wide variety of XML query languages, including subsets of XQuery, XML-QL, XPath, and even "one-off" new proposals; they use a wide variety of languages or ad-hoc constructs to map between the relational and XML schema; and they differ widely in what they "push to SQL" and what they evaluate in middleware.

This diversity renders it difficult to know how the various results presented fit together, and even makes it hard to know what if any open problems remain. As a first step to rectifying this situation, we present a classification of the problem space and discuss how almost 40 papers fit into this classification. As a result of this study, we find that some basic questions are still open. In particular, for the XML publishing of relational data and for "schema-based" shredding of XML documents into relations, there is no published algorithm for translating even simple path expression queries (with the // axis) into SQL when the XML schema is recursive. It is our hope that this paper will stimulate others to refine our classification and, more importantly, to improve the state of the art and to address and solve the open problems that the classification reveals.

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 1–18, 2003.

Table 1. Summary of various published techniques

Technique	Scenario	Subproblems solved	Class of XML Schema considered	Class of XML Queries handled
XPeranto	XP/GAV	VD,QT	tree	XQuery
SilkRoute	XP/GAV	VD,QT	tree	XML-QL
Rolex	XP/GAV	QT	tree	XSLT
[17]	XP/GAV	QT	tree	XSLT
[1]	XP/GAV	VD	recursive	-
Oracle XML DB	XP/GAV, XS/SB	VD,SS,QT	recursive	SQL/XML restricted XPath[1]
SQL Server 2000 SQLXML	XP/GAV, XS/SB	VD,SS,QT	bounded depth recursive	restricted XPath[2]
DB2 XML Extender	XP/GAV, XS/SB	VD,QT	non-recursive	SQL extensions through UDFs
Agora	XP/LAV	QT	non-recursive	XQuery
MARS	XP/GAV + XP/LAV	QT	non-recursive	XQuery
STORED	XS/SO	SS,QT	all	STORED
Edge	XS/SO	SS,QT	all	path expressions
Monet	XS/SO	SS	all	-
XRel	XS/SO	SS,QT	all	path expressions
[35]	XS/SO	SS,QT	all	order-based queries
Dynamic intervals [7]	XS/SO	QT	all	XQuery
[24,32]	XS/SB	SS	recursive	-
[2,16,19,21,27]	XS/SB	SS	tree	-

XP/GAV: XML Publishing, Global-as-view XP/LAV: XML Publishing, Local-as-view, XS/SO: XML Storage, schema-oblivious XS/SB: XML Storage, schema-based
QT: Query Translation VD: View Definition SS: Storage scheme
restricted XPath[1]: child and attribute axes
restricted XPath[2]: child, attribute, self and parent axes

The interaction between XML and RDBMS can be broadly classified as shown in Figure 1. The main scenarios are:

1. XML Publishing (XP): here, the goal is to treat existing relational data sets as if they were XML. In other words, an XML view of the relational data set is defined and XML queries are posed over this view.
2. XML Storage (XS): here, by contrast, the goal is to use an RDBMS to store and query existing XML data. In this scenario, there are two main subproblems: (1) a relational schema has to be chosen for storing the XML data, and (2) XML queries have to be translated to SQL for evaluation.

In this paper, we use this classification to characterize almost 40 published solutions to the XML-to-SQL query translation problem.

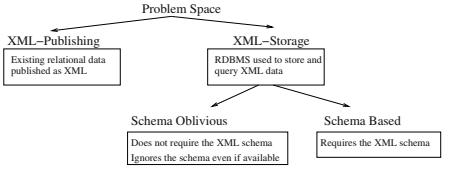


Fig. 1. High-Level Taxonomy

	Tree Schema		Recursive Schema	
Simple Queries (path expressions)	XP	lot	XP	some
	XS/SO	lot	XS/SO	lot
	XS/SB	lot	XS/SB	some
Complex Queries	XP	lot	XP	none
	XS/SO	some	XS/SO	some
	XS/SB	none	XS/SB	none

Fig. 2. Focus of published solutions

The results of our survey are summarized in Table 1, where for each technique we identify the scenario solved and the part of the problem handled within that scenario. We will look at each of these in more detail in the rest of the paper. In addition to the characteristics from our broad classification, the table also reports, for each solution, the class of schema considered, the class of XML queries handled, whether it uses the "global as view" or "local as view" approach (if the XML publishing problem is addressed), and what subproblems are solved.

The rest of the paper is organized as follows. We survey known algorithms in published literature for XML-publishing, schema-oblivious XML storage and schema-based XML storage in Sections 2, 3, and 4 respectively. For each scenario, we first survey the solutions that have been proposed in published literature, and discuss problems that remain open. When we look at XML support in commercial RDBMS as part of this survey, we will restrict our discussion to those features that are relevant to XML-to-SQL query translation. A full investigation of XML support in commercial RDBMS is beyond the scope of this survey.

2 XML Publishing

The following tasks arise in the context of allowing applications to query existing relational data as if it were XML:

- Defining an XML view of relational data.
- Materializing the XML view.
- Evaluating an XML query by composing it with the view.

In XML query languages like XPath and XQuery, part of the query evaluation may involve reconstructing the subtrees rooted at certain elements, which are identified by other parts of the query. Notice how materializing an XML view is a special case of this situation, where the entire tree (XML document) is reconstructed. In general, solutions to materialize an XML view are used as a subroutine during query evaluation.

2.1 XML View Definition

In XPeranto [29,30], SilkRoute [12,13] and Rolex [3], the view definition languages permit definition of tree XML views over the relational data. In [1], XML views corresponding to recursive XML schema (recursive XML view schema) are allowed.

In Oracle XML DB [42] and Microsoft SQL Server 2000 SQLXML [43], an annotated XSD XML schema is used to define the XML view. Recursive XML views are supported in XML DB. In SQLXML, along with non-recursive views, there is support for a limited number of depths of recursion using the max-depth annotation. In IBM DB2 XML Extender [40], a Document Access Definition (DAD) file is used to define a non-recursive XML view. IBM XML for Tables [44] provides an XML view of relational tables and is based on the Xperanto [30] project.

In the above approaches, the XML view is defined as a view over the relational schema. In a data integration context, Agora [25] uses the local-as-view approach (LAV), where the local source's schema are described as views over the global schema. Toward this purpose, they describe a generic, virtual relational schema closely modeling the generic structure of an XML document. The local relational schema is then defined as views over this generic, virtual schema. Contrast this with the other approaches where the XML view (global schema) is defined as a view over the relational schema (local schema). This is referred to as the global-as-view approach (GAV).

In Mars [9], the authors consider the scenario where both GAV-style and LAV-style views are present. The focus of [9,25] is on non-recursive XML view schema.

2.2 Materializing the XML View

In XPeranto [30], the XML view is materialized by pushing down a single "outer union" query into the relational engine, whereas in SilkRoute [12], the middleware system issues several SQL queries to materialize the view. In [1], techniques for materializing a recursive XML view schema are discussed. They argue that since SQL supports only linear recursion, the support for recursion in SQL is insufficient for this purpose. Instead, the recursive materialization is performed in middleware by repeatedly unrolling a fixed number of levels at a time. We discuss this in more detail in Section 2.4.

2.3 Evaluating XML Queries

In XPeranto [29], a general framework for processing arbitrarily complex XQuery queries over XML views is presented. They describe their XQGM query representation, an extension of a SQL internal query representation called the Query Graph Model (QGM). The XQuery query is converted to an XQGM representation and composed with the view definition. Rewrite optimizations are performed to eliminate the construction of intermediate XML fragments and to push down predicates. The modified XQGM is translated into a single SQL query to be evaluated inside the relational engine.

In SilkRoute [12], a sound and complete query composition algorithm is presented for evaluating a given XML-QL query over the XML view. An XML-QL query consists of patterns, filters and constructors. Their composition technique

evaluates the patterns on the view definition at compile-time to obtain a modified XML view, and the filters and constructors are evaluated at run-time using the modified XML view.

In [17], the authors present an algorithm for translating XSLT programs into efficient SQL queries. The main focus of the paper is on bridging the gap between XSLT's functional, recursive paradigm, and SQL's declarative paradigm. They also identify a new class of optimizations that need to be done either by the translator or by the relational engine, in order to optimize the kind of SQL queries that result from such a translation. In Rolex [22], a view composition algorithm for composing an XSLT stylesheet with an XML view definition to produce a new XML view definition is presented. They differ from [17] mainly in the following ways: (1) they produce an XML view query rather than an SQL query, (2) they address additional features of XSLT like priority and recursive templates.

As part of the Rainbow system, in [39], the authors discuss processing and optimization of XQuery queries. They describe the XML Algebra Tree (XAT) algebra for modeling XQuery expressions, propose rewriting rules to optimize XQuery queries by canceling operators and describe a cutting algorithm that removes redundant operators and relational columns from the XAT. However, the final XML to SQL query generation is not discussed.

We note here that in Rolex [3], the world view is changed so that a relational system provides a virtual DOM interface to the application. The input in this case is not a single XML query but a series of navigation operations on the DOM tree that needs to be evaluated on the underlying relational data.

The Agora [25] project uses an LAV approach and provides an algorithm for translating XQuery FLWR expressions into SQL. Their algorithm has two main steps — translating the XML query into a SQL query on the generic, virtual relational schema, and rewriting this SQL query into a SQL query over the real relational schema. In the first step, they cross the language gap from XQuery to SQL, and in the second step they use prior work on answering queries using views.

In MARS [9,10], a technique for translating XQuery queries into SQL is given, when both GAV-style and LAV-style views are present. The basic idea is to compile the queries, views and constraints from XML into the relational framework, producing relational queries and constraints. Then, a Chase and BackChase (C&B) algorithm is used to find all minimal reformulations of the relational queries under the relational integrity constraints. Using a cost-estimator, the optimal query among the minimal reformulations is obtained, which can then be executed. The MARS system also exploits integrity constraints on both the relational and XML data. The system achieves the combined effect of rewriting-with-views, composition-with-views, and query minimization under integrity constraints.

Oracle XML DB [42] provides an implementation of the majority of the operators that will be incorporated into the forthcoming SQL/XML standard [41]. SQL/XML is an extension to SQL, using functions and operators, to include pro-

cessing of XML data in relational stores. The SQL/XML operators [11] make it possible to query and access XML content as part of normal SQL operations and also provide methods for generating XML from the result of an SQL Select statement. The SQL/XML operators allow XPath expressions to be used to access a subset of the nodes in the XML view. In XML DB, the approach is to translate the XPath expression into an equivalent SQL query through a query re-write step that uses the XML view definition. In the current release (Oracle9i Release 2), simple path expressions with no wild cards or descendant axes (//) get rewritten. Predicates are supported and get rewritten into SQL predicates. The XPath axes supported are the child and attribute axis.

Microsoft SQL Server 2000 SQLXML [43] supports the evaluation of XPath queries over the annotated XML Schema. The XPath query together with the annotated schema is translated into a *FOR XML* explicit query that only returns the XML data that is required by the query. Here, *FOR XML* is a new SQL select statement extension provided by SQL Server. In the current release (SQLXML 3.0), the attribute, child, parent and self axes are supported, along with predicates and XPath variables.

In IBM DB2 XML Extender [40], powerful user-defined functions (UDFs) are provided to store and retrieve XML documents in XML columns, as well as to extract XML element or attribute values. Since it does not provide support for any XML query languages, we will not discuss XML Extender any further in this paper.

2.4 Open Problems

A number of problems remain open in this area.

1. With the exception of [1,42], the above work considers only non-recursive XML views of relational data. While Oracle XML DB [42] supports path expression queries with the child and attribute axes over recursive views, it does not support the descendant (//) axis. Translating XML queries (with the // axis) over recursive view schema remains open. In [1], the problem of materializing recursive XML view schema is considered. However, as we have mentioned, that work does not use SQL support for recursion, simulating recursion in middleware instead. The reason for this given by the authors is that the limited form of recursion supported by SQL cannot handle the forms of recursion that arise in with recursive XML schema. We return to this question at the end of this section. The following are open questions in the context of SQL support for recursion:
   - What is the class of queries/view schema for which the current support for recursion in SQL are adequate?
   - If there are cases for which SQL support for recursion is inadequate, how do we best leverage this support? (Instead of completely simulating recursion in middleware.)
2. Any query translation algorithm can be evaluated by two metrics: its functionality, in terms of the class of XML queries handled; and its performance, in terms of the efficiency of the resulting SQL query. Most of the translation

algorithms have not been evaluated thoroughly by either metric, which gives rise to a number of open research problems.

- Functionality: Among the GAV-style approaches, except XPeranto, all the above discussed work deals with languages other than XQuery. Even in the case of XPeranto, the class of XQuery handled is unclear from [29]. It would be interesting to precisely characterize the class of XQuery queries that can be translated by the methods currently in the literature.
- Performance: There has been almost no work comparing the quality of SQL queries generated by various translation algorithms. In particular, we are aware of no published performance study for the query translation problem.

3. GAV vs. LAV: While for the GAV-style approaches, XML-to-SQL query translation corresponds to view composition, for the LAV-style approaches it corresponds to answering queries with views. It is not clear for what class of XML views the equivalent query rewriting problem has published solutions. As pointed out in [25], state-of-the-art query rewriting algorithms for SQL semantics do not efficiently handle arbitrary levels of nesting, grouping etc. Similarly, [9] works under set-semantics and so cannot handle certain classes of XML view schema and aggregation in XML queries. Comparing across the three different approaches — GAV, LAV and GAV+LAV, in terms of both functionality and performance is an open issue.

Recursive XML View Schema and Linear Recursion in SQL. In this subsection we return to the problem of recursive XML view schema and whether or not they can be handled by the support for recursion currently provided by SQL.

Consider the problem of materializing a recursive XML view schema. In [1], it is mentioned that even though SQL supports linear recursion, this is not sufficient for materializing a recursive XML view. The reason for this is not elaborated in the paper. The definition of an XML view has two main components to it: the view definition language and the XML schema of the resulting view. Hence, it must be the case that either the XML schema of the view or the view definition language is more complex than what SQL linear recursion can support. Clearly, if the view definition language is complex enough (say the parent-child relationship is defined using non-linear recursion), linear recursion in SQL will not suffice. However, most view definition languages proposed define parent-child relationships through much simpler conditions (such as conjunctive queries). The question arises whether SQL linear recursion is sufficient for these view definition languages, for arbitrary XML schema.

In [6], the notion of linear and non-linear recursive DTDs is introduced. The natural question here is whether the notions of linear recursion in SQL and DTDs correspond. It turns out that the definition of non-linear recursive schema in [6] has nothing to do with the traditional Datalog notion of linear and non-linear recursion. For example, consider a classical part-subpart database. Suppose that the DTD rule for a part element is: part $\rightarrow$ pname, part*.

According to [6], this is a non-linear recursive rule as a part element can derive multiple part sub-elements. Hence, the entire DTD is non-linear recursive. Indeed, it can be shown that this DTD is not equivalent to any linear-recursive DTD. Now, suppose the underlying relational schema has two relations, Part and Subpart with the columns: (partid,pname) and (partid,subpartid) respectively. Now, the following SQL query extracts all data necessary to materialize the XML view:

```
WITH RECURSIVE AllParts(partid,pname,rtolpath) as (
           select partid,pname,''
           from Part(partid,pname)
        union all
           select P.partid,P.pname,rtolpath+A.partid
           from AllParts A, Subpart S, Part P
           where S.partid = A.partid and S.subpartid = P.partid)
select * from AllParts
```

In the above query, the root-to-leaf path is maintained for each part element through the `rtolpath` column in order to extract the tree structure. Note however that the core SQL query executes the following linear-recursive Datalog program.

AllParts(partid,pname) ← Part(partid,pname)
AllParts(subpartid,subpname) ←
AllParts(partid,pname) Subpart(partid,subpartid)
Part(subpartid,subpname)

So, we see that a non-linear recursive rule in the DTD gets translated into a linear recursive Datalog (SQL) rule. This implies that the notion of linear recursion in DTDs and SQL (Datalog) do not have a direct correspondence. Hence, the class of XML view schema/view definition languages for which SQL linear recursion is adequate to materialize the resulting XML views needs to be examined.

3 Schema-Oblivious XML Storage

Recall that in this scenario, the goal is to find a relational schema that works for storing XML documents independent of the presence or absence of a schema. The main problems addressed in this sub-space are:

1. Relational schema design: which generic relational schema for XML should be used?
2. Query translation algorithms: given a decision for the relational schema, how do we translate from XML queries to SQL queries.

3.1 Relational Schema Design

In STORED [8], given a semi-structured database instance, a STORED mapping is generated automatically using data mining techniques — STORED is a declarative query language proposed for this purpose. This mapping has two

parts: a relational schema and an overflow graph for the data not conforming to the relational schema. We classify STORED as a schema-oblivious technique since the data since data inserted in the future is not required to conform to the derived schema. Thus, if an XML document with completely different structure is added to the database, the system sticks to the existing relational schema without any modification whatsoever.

In [14], several mapping schemes are proposed. According to the Edge approach, the input XML document is viewed as a graph and each edge of the graph is represented as a tuple in a single table. In a variant known as the Attribute approach, the edge table is horizontally partitioned on the tag name yielding a separate table for each element/attribute. Two other alternatives, the Universal table approach and the Normalized Universal approach are proposed but shown to be inferior to the other two. Hence, we do not discuss these any further.

The binary association approach [28] is a path-based approach that stores all elements that correspond to a given root-to-leaf path together in a single relation. Parent-child relationships are maintained through parent and child ids.

The XRel approach [37] is another path-based approach. The main difference here is that for each element, the path id corresponding to the root-to-leaf path as well as an interval representing the region covered by the element are stored. The latter is similar to interval-based schemes for representing inverted lists proposed in [23,38].

In [35], the focus is on supporting order based queries over XML data. The schema assumed is a modified Edge relation where the path id is stored as in [37], and an extra field for order is also stored. Three schemes for supporting order are discussed.

In [7], all XML data is stored in a single table containing a tuple for each element, attribute and text node. For an element, the element name and an interval representing the region covered by the element is stored. Analogous information is stored for attributes and text nodes.

There has been extensive work on using inverted lists to evaluate path expression queries by performing containment joins [5,18,23,26,33,36,38]. In [38], the performance of containment algorithms in an RDBMS and a native XML system are compared. All other strategies are for native XML systems. In order to adapt these inside a relational engine, we would need to add new containment algorithms and novel data structures. The issue of how we extend the relational engine to identify the *use* of these strategies is open. In particular, the question of how the optimizer maps SQL operations into these strategies needs to be addressed.

In [15], a new database index structure called the XPath accelerator is proposed that supports all XPath axes. The preorder and postorder ranks of an element are used to map nodes onto a two-dimensional plane. The evaluation of the XPath axis steps then reduces to processing region queries in this pre/post plane. In [34], the focus is on exploiting additional properties of the pre/post plane to speedup XPath query evaluation and the Staircase join operator is pro-

posed for this purpose. The focus of [15,34] is on efficiently supporting the basic operations in a path expression and is complementary to the XML-to-SQL query translation issue.

In Oracle XML DB [42] and IBM DB2 XML Extender [40], a schema-oblivious way of storing XML data is provided, where the entire XML document is stored using the CLOB data type. Since evaluating XML queries in this case will be similar to XML query processing in a native XML database and will not involve XML-to-SQL query translation, we do not discuss this approach in this paper.

3.2 Query Translation

In STORED [8], an algorithm is outlined for translating an input STORED query into SQL. The algorithm uses inversion rules to create a single canonical data instance, intuitively corresponding to a schema. The structural component of the STORED query is then evaluated on this instance to obtain a set of results, for each of which a SQL query is generated incorporating the rest of the STORED query.

In [14], a brief overview of how to translate the basic operations in a path expression query to SQL is provided. The operations described are (1) returning an element with its children, (2) selections on values, (3) pattern matching, (4) optional predicates, (5) predicates on attribute names and (6) regular path queries which can be translated into recursive SQL queries.

The binary association method [28] deals with translating OQL-like queries into SQL. The class of queries they consider roughly corresponds to branching path expression queries in XQuery.

In XRel [37], a core part of XPath called XPathCore is identified and a detailed algorithm for translating such queries into SQL is provided. Since with each element, a path id corresponding to the root-to-leaf path is stored, a simple path expression query like book/section/title gets efficiently evaluated. Instead of performing a join for each step of the path expression, all elements with a matching path id are extracted. Similar optimizations are proposed for branching path expression queries exploiting both path ids and the interval encoding. We examine this in more detail in Section 3.3.

In [35], algorithms for translating order based path expression queries into SQL are provided. They provide translation procedures for each axis in XPath, as well as for positional predicates. Given a path expression, the algorithm translates one axis at a time in sequence.

The dynamic intervals approach [7] deals with a larger fragment of XQuery with arbitrarily nested FLWR expressions, element constructors and built-in functions including structural comparisons. The core idea is to begin with static intervals for each element and construct dynamic intervals for XML elements constructed in the query. Several new operators are proposed to efficiently implement the generated SQL queries inside the relational engine. These operators are highly specialized and are similar to operators present in a native XML engine.

3.3 Summary and Open Problems

The various schema-oblivious storage techniques can be broadly classified as:

1. Id-based: each element is associated with a unique id and the tree structure of the XML document is preserved by maintaining a foreign key to the parent.
2. Interval-based: each element is associated with a region representing the subtree under it.
3. Path-based: each element is associated with a path id representing the root-to-leaf path in addition to an interval-based or id-based representation.

We organize the rest of the discussion by considering different classes of queries.

Reconstructing an XML Sub-tree. This problem is largely solved. In the schema-oblivious scenario, the sub-tree corresponding to an XML element could potentially span all tables in the database. Hence, while solutions that store all the XML data in only one table need to process just that table, other solutions will need to access all tables in the database.

For id-based solutions, a recursive SQL query can be used to reconstruct a sub-tree. For interval-based solutions, a non-recursive query with interval predicates is sufficient.

Simple Path Expression Queries. We refer to the class of path expression queries without predicates as simple path expression queries. For interval-based solutions, evaluating simple path expressions entails performing a range join for each step of the path expression. For example the query book/author/name translates into a three-way join. For id-based solutions, each parent-child(/) step translates into an equijoin, whereas recursion in the path expression (through //) requires a recursive SQL query. For path-based solutions, the path id can be used to avoid performing one join per step of the path expression.

Path Expression Queries with Predicates. Predicates can be existential path expression predicates, or positional predicates. The latter is dealt with in [35,37]. We focus on the former for the rest of the section.

For id-based and interval-based solutions, a straightforward method for query translation is to perform one join per step in the path expression [8,14,38]. With path ids, however, it is conceivable that certain joins can be skipped, just as they can be skipped for some simple path expressions. A detailed algorithm for doing so is proposed in [37]. That algorithm is correct for nonrecursive data sets — it turns out that it does not give the correct result when the input XML data has an ancestor and descendant element with the same tag name. For that reason, the general problem of translation of path expressions with predicates for the path-based schema-oblivious schemes is still open.

More Complex XQuery Queries. The only published work that we are aware of that deals with more general XQuery queries is [7]. The main focus of the paper is on issues such as structural equality in FLWR where clauses, full compositionality of XML query expressions (in particular, the possibility of nesting FLWR expressions within functions), and the need for constructed XML documents representing intermediate query results. As mentioned earlier, special purpose relational operators are proposed for better performance. We note that without these operators, the performance of their translation is likely to be inferior even for simple path expressions. As an example, using their technique, the path expression /site/people is translated to an SQL query involving five temporary relations created using the *With* clause in SQL99, three of which involve correlated subqueries. To conclude, excepting [7], all prior work has been on translating path expression queries into SQL. Using the approach proposed by [7], we observe that functionality-wise, a large fragment of XQuery can be handled using dynamic intervals in a schema-oblivious fashion. However, without modifications to the relational engine, its performance may not be acceptable.

4 Schema-Based XML Storage

In this section, we discuss approaches to storing XML in relational systems that make use of a schema for the XML data in order to choose a good relational schema. The main problems to be addressed in this subspace are

1. Relational schema selection — given an XML schema (or DTD), how should we choose a good relational schema and XML-to-relational mapping.
2. Query translation — having chosen an XML-to-relational mapping, how should we translate XML queries into SQL.

4.1 Relational Schema Selection

In [32], three techniques for using a DTD to choose a relational schema are proposed — basic inlining, shared inlining, and hybrid inlining. The main idea is to inline all elements that occur at most once per parent element in the parent relation itself. This is extended to handle recursive DTDs.

In [21], a constraint preserving algorithm for transforming an XML DTD to a relational schema is presented. The authors chose the hybrid inlining algorithm from [32] and showed how semantic constraints can be generated.

In [2], the problem of choosing a good relational schema is viewed as an optimization problem: given an XML schema, an XML query workload, and statistics over the XML data choose the relational schema that maximizes query performance. They give a greedy heuristic for this purpose.

In [16,24], the theory of regular tree grammars is used to choose a relational schema for a given XML schema.

In [4], a storage mapping that takes into account the key and foreign key constraints present in an XML schema is presented.

There has been some work on using object-relational DBMS to store XML documents. In [19,27], parts of the XML document are stored using an XML

ADT. The focus of these papers is to determine *which* parts of the DTD must be mapped to relations and which parts must be mapped to the XML ADT.

In Oracle XML DB [42], an annotated XML Schema is used to define how the XML data is mapped into relations. If the XML Schema is not annotated, XML DB uses a default algorithm to decide the relational schema based on the XML Schema. This algorithm handles recursive XML schemas.

A similar approach is made in Microsoft SQL Server 2000 SQLXML [43] and IBM DB2 XML Extender [40], but they only handle non-recursive XML schemas.

4.2 Query Translation

In [32], the general approach to translating XML-QL queries into SQL is illustrated through examples without any algorithmic details.

As discussed in [31], it is possible to use techniques from the XML publishing domain in the XML storage domain. To see this, notice that once XML data is shredded into relations, we can view the resulting data as if it were pre-existing relational data. Now by defining a reconstruction view that mirrors the XML-to-relational mapping used to shred the data, the query translation algorithms in the XML publishing domain are directly applicable. Indeed, this is the approach adopted in [2]. While this approach has the advantage that solutions for XML publishing can be directly applied to the schema-based XML storage scenario, it has one important drawback. In the XML storage scenario, the data in the RDBMS originates from an XML document and there is some semantic information associated with this (like the tree structure of the data and the presence of a unique parent for each element). This semantic information can be used by the XML-to-SQL translation algorithm to generate efficient SQL queries. By using solutions from the XML publishing scenario, we are potentially making the use of this semantic information harder. We discuss this in more detail with an example in Section 4.3.

Note that even the schema-oblivious subspace can be dealt with in an analogous manner as mentioned in [31]. However, in this case, the reconstruction view is fairly complex — for example, the reconstruction view for the Edge approach is an XQuery query involving recursive functions [31]. Since handling recursive XML view schema is open (Section 2.4), this approach for the schema-oblivious scenario needs to be explored further.

In [35], as we mentioned in Section 3.2, the focus is on supporting order-based queries. The authors give an algorithm for the schema-oblivious scenario, and briefly mention how the ideas can be applied with any existing schema-based approach.

In Oracle XML DB [42], Microsoft SQL Server 2000 SQLXML [43] and IBM DB2 XML Extender [40], the XML Publishing and Schema-Based XML Storage scenarios are handled in an identical manner. So, the description of their approaches for the XML Publishing scenario presented in Section 2.3 holds for the Schema-Based XML Storage scenario. To summarize, XML DB supports branching path expression queries with the child and attribute axes, while

SQLXML supports the parent and self axes as well. XML Extender does not support any XML query language. Instead, it provides user-defined functions to manipulate XML data.

In [20], the problem of finding optimal relational decompositions for XML workloads is considered in a formal perspective. Using three XML-to-SQL query translation algorithms for path expression queries over a particular family of XML schemas, the interaction between the choice of a good relational decomposition and a good query translation algorithm is studied. The authors showed that the query translation algorithm and the cost model used play a vital role not just in the choice of a good decomposition, but also in the complexity of finding the optimal choice.

4.3 Discussion and Open Problems

There is no published query translation algorithm for the schema-based XML storage scenario. One alternative is to reduce this problem to XML publishing (using reconstruction views). Hence, from a functionality perspective, whatever is open in the XML publishing case is open here also. In particular, the entire problem is open when the input XML schema is recursive. Even for a non-recursive XML schema, a lot of interesting questions arise when the XML schema is not a tree. For example, if there is recursion in an XPath query through //, the straightforward approach of enumerating all satisfying paths using the schema and handling them one at a time is no longer an efficient approach. If we wish to reduce the problem to XML publishing, the only way to use an existing solution is to unfold the DAG schema into an equivalent tree schema.

We now examine the translation problem from a performance perspective.

Goals of XML-to-SQL Query Translation. When an XML document is shredded into relations, there is inherent semantic information associated with the relation instances given that the source is XML. For example, consider the XML schema shown in Figure 3. One candidate relational decomposition is also shown in the figure. The mapping is illustrated through annotations on the XML schema. Each node is annotated with the corresponding relation name. Leaf nodes are annotated with the corresponding relational column as well. Parent-child relationships are represented using id and parentid columns. The figure element has two potential parents in the schema. In order to distinguish between them, a parentcode field is present in the Figure relation. In this case, notice that there is inherent semantics associated with the columns parentid and parentcode given that they represent the manner in which the tree structure of the XML document is preserved.

Given this semantics, when an XML query is posed, there are several equivalent SQL queries, which are not necessarily equivalent without the extra semantics that come from knowing that the relations came from shredding XML. Consider the following query: find captions for all figures in top level sections. This can be posed as an XPath query XQ = /book/section/figure/caption. There are

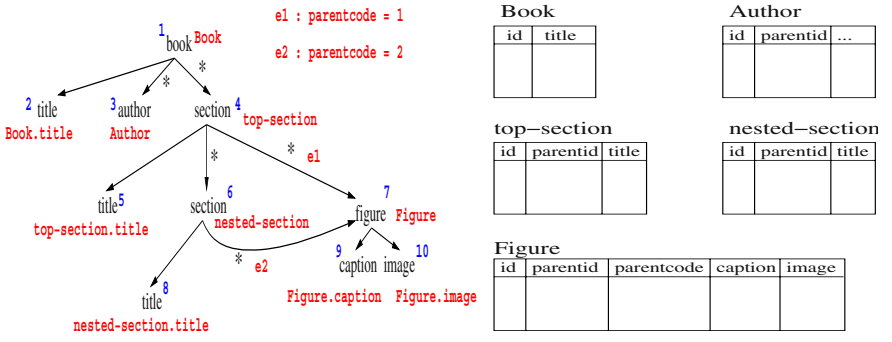


Fig. 3. Sample XML-to-Relational mapping schema

two equivalent ways in which we could translate XQ into SQL. They are shown below.

```
SQ1:
select caption
from figure
where parentcode=1
```

```
SQ2:
select caption
from   figure f, top-section ts, book b
where  f.parentcode=1 and f.parentid=ts.id
       and ts.parentid=b.id
```

While SQ1 merely performs a scan on the figure table, SQ2 roughly performs a join for each step of the path expression. SQ2 is what we would obtain by adapting techniques from XML publishing. Queries SQ1 and SQ2 are equivalent only because of the semantics associated with the parentcode and parentid columns and would not be equivalent otherwise.

Now, since the XML-to-SQL translation algorithm is aware of the semantics of the XML-relational mapping, it is better placed than the relational optimizer to find the best SQL translation. Hence, from a performance perspective, the problem of effectively exploiting the XML schema and the XML-relational mapping during query translation remains open.

Enhancing Schema-Based Solutions with Intervals/Path-ids. All the schema-based solutions proposed in published literature have been id-based. In the schema-oblivious scenario, it has been shown that using intervals and path-ids can be helpful in XML-to-SQL query translation. The problem of augmenting the schema-based solutions with some sort of intervals and/or path-ids is an interesting open problem. Note that while any id-based storage scheme can be easily augmented by adding either a path-id column or an interval for each element, developing query translation algorithms that use *both* the schema information and the interval/path information is non-trivial.

5 Summary and Conclusions

To conclude, we refer again to the summary in Table 1. From that table, we see that the community has made varying degrees of progress for different subprob-

lems in the XML to SQL query translation domain. We next summarize this progress, in terms of functionality.

- In the XML-Publishing scenario, techniques have been proposed for handling complex query languages like XQuery and XSLT over tree XML view schema. However, handling recursive XML view schema is an open problem. Even for tree XML view schema, the subset of XQuery handled by current solutions is not clear.
- In the schema-oblivious XML storage scenario, excepting [7], the focus has been on path expression queries.
- In the schema-based XML storage scenario, there is no published query translation algorithm. The only approach known to us is through a reduction to the XML publishing scenario.

Our purpose in submitting this paper to this symposium is to begin to clarify what is known and what is not known in the large, growing, and increasingly diverse literature dealing with using relational database systems for storing and/or querying XML. Other classifications of existing work are possible, and certainly our classification can be refined and improved. Furthermore, while we have tried to be complete and have surveyed close to 40 publications, there may be gaps in our coverage. It is our hope that publishing this paper in the symposium proceedings will begin a discussion that will improve the quality of this summary, and, more importantly, will spur further work to fill the gaps in the state of the art in the XML to SQL translation literature.

Acknowledgement. This work was supported in part by NSF grant ITR-0086002 and a Microsoft fellowship.

References

1. M. Benedikt, C. Y. Chan, W. Fan, R. Rastogi, S. Zheng, and A. Zhou. DTD-Directed Publishing with Attribute Translation Grammars. In *VLDB*, 2002.
2. P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML schema to relations: A cost-based approach to XML storage. In *ICDE*, 2002.
3. P. Bohannon, H. Korth, P.P.S. Narayan, S. Ganguly, and P. Shenoy. Optimizing view queries in ROLEX to support navigable tree results. In *VLDB*, 2002.
4. Y. Chen, S. B. Davidson, and Y. Zheng. Constraint preserving XML Storage in Relations. In *WebDB*, 2002.
5. S. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *VLDB*, 2002.
6. B. Choi. What Are Real DTDs Like. In *WebDB*, 2002.
7. D. DeHaan, D. Toman, M. P. Consens, and T. Ozsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *SIGMOD*, 2003.
8. A. Deutsch, M. Fernández, and D. Suciu. Storing semistructured data with STORED. In *SIGMOD*, 1999.
9. A. Deutsch and V. Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *VLDB*, 2003.

10. A. Deutsch and V. Tannen. Reformulation of XML Queries and Constraints. In *ICDT*, 2003.
11. A. Eisenberg and J. Melton. SQL/XML is Making Good Progress. *SIGMOD Record*, 31(2), 2002.
12. M. Fernandez, A. Morishima, and D. Suciu. Efficient Evaluation of XML Middleware Queries. In *SIGMOD*, 2002.
13. M. Fernández, D. Suciu, and W.C. Tan. SilkRoute: Trading Between Relations and XML. In *WWW9*, 2000.
14. D. Florescu and D. Kossman. Storing and Querying XML Data using an RDBMS. *Data Engineering Bulletin*, 22(3), 1999.
15. T. Grust. Accelerating XPath location steps. In *SIGMOD*, 2002.
16. S. Hongwei, Z. Shusheng, Z. Jingtao, and W. Jing. Constraints-Preserving Mapping Algorithm from XML-Schema to Relational Schema. In *Engineering and Deployment of Cooperative Information Systems (EDCIS)*, 2002.
17. S. Jain, R. Mahajan, and D. Suciu. Translating XSLT Programs to Efficient SQL Queries. In *WWW*, 2002.
18. H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *ICDE*, 2003.
19. M. Klettke and H. Meyer. XML and Object-Relational Database Systems - Enhancing Structural Mappings Based on Statistics. In *WebDB*, 2000.
20. R. Krishnamurthy, R. Kaushik, and J. F. Naughton. On the difficulty of finding optimal relational decompositions for xml workloads: A complexity theoretic perspective. In *ICDT*, 2003.
21. D. Lee and W.W. Chu. Constraints-preserving Transformation from XML Document Type Definition to Relational Schema. In *ER*, 2000.
22. C. Li, P. Bohannon, H. Korth, and P.P.S. Narayan. Composing XSL Transformations with XML Publishing Views. In *SIGMOD*, 2003.
23. Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of VLDB*, 2001.
24. M. Mani and D. Lee. XML to Relational Conversion Using Theory of Regular Tree Grammars. In *Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web (EEXTT)*, 2002.
25. I. Manolescu, D. Florescu, and D. Kossman. Answering XML queries over heterogeneous data sources. In *VLDB*, 2001.
26. N.Bruno, N.Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *SIGMOD*, 2002.
27. K. Runapongsa and J. M. Patel. Storing and Querying XML Data in Object-Relational DBMSs. In *EDBT Workshops*, 2002.
28. A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient Relational Storage and Retrieval of XML Documents. In *WebDB*, 2000.
29. J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML Views of Relational Data. In *VLDB*, 2001.
30. J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *VLDB*, 2000.
31. J. Shanmugasundaram, E. Shekita, J. Kiernan, R. Krishnamurthy, S. D. Viglas, J. Naughton, and I. Tatarinov. A general technique for querying xml documents using a relational database system. *SIGMOD Record*, 30(3), 2001.
32. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*, 1999.

33. D. Srivastava, S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, and Y. Wu. Structural Joins: A Primitive For Efficient XML Query Pattern Matching. In *ICDE*, 2002.
34. J. Teubner T. Grust, M. V. Keulen. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *VLDB*, 2003.
35. I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, 2002.
36. W. Wang, H. Jiang, H. Lu, and J. X. Yu. PBiTree Coding and Efficient Processing of Containment Joins. In *ICDE*, 2003.
37. M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology (TOIT)*, 1(1):110–141, 2001.
38. C. Zhang, J.F. Naughton, D.J. DeWitt, Q. Luo, and G.Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of SIGMOD*, 2001.
39. X. Zhang, B. Pielech, and E. Rundesnteiner. Honey, I Shrunk the XQuery! – An XML Algebra Optimization Approach. In *Workshop on Web Information and Data Management (WIDM)*, 2002.
40. DB2 XML Extender.
`http://www-3.ibm.com/software/data/db2/extenders/xmlext/index.html`.
41. INCITS H2.3 Task Group.
`http://www.sqlx.org`.
42. Oracle9i XML Database Developer's Guide - Oracle XML DB Release 2 (9.2).
`http://otn.oracle.com/tech/xml/xmldb/content.html`.
43. SQLXML and XML Mapping Technologies.
`http://msdn.microsoft.com/sqlxml/default.asp`.
44. XML for Tables.
`/http://www.alphaworks.ibm.com/tech/xtable`.

Searching for Efficient XML-to-Relational Mappings

Maya Ramanath[1], Juliana Freire[2], Jayant R. Haritsa[1], and Prasan Roy[3]

[1] SERC, Indian Institute of Science
{maya,haritsa}@dsl.serc.iisc.ernet.in
[2] OGI/OHSU
juliana@cse.ogi.edu
[3] Indian Institute of Technology, Bombay
prasan@it.iitb.ac.in

Abstract. We consider the problem of cost-based strategies to derive efficient relational configurations for XML applications that subscribe to an XML Schema. In particular, we propose a flexible framework for XML schema transformations and show how it can be used to design algorithms to search the space of equivalent relational configurations. We study the impact of the schema transformations and query workload on the search strategies for finding efficient XML-to-relational mappings. In addition, we propose several optimizations to speed up the search process. Our experiments indicate that a judicious choice of transformations and search strategies can lead to relational configurations of substantially higher quality than those recommended by previous approaches.

1 Introduction

XML has become an extremely popular medium for representing and exchanging information. As a result, efficient storage of XML documents is now an active area of research in the database community. In particular, the use of relational engines for this purpose has attracted considerable interest with a view to leveraging their powerful and reliable data management services.

Cost-based strategies to derive relational configurations for XML applications have been proposed recently [1,19] and shown to provide substantially better configurations than heuristic methods (*e.g.,* [15]). The general methodology used in these strategies is to define a set of *XML schema transformations* that derive different relational configurations. Given an XML query workload, the quality of the relational configuration is evaluated by a costing function on the SQL equivalents of the XML queries. Since the search space is large, greedy heuristics are used to search through the associated space of relational configurations.

In this paper, we propose **FlexMap**, a framework for generating XML-to-relational mappings which incorporates a comprehensive set of schema transformations. FlexMap is capable of supporting different mapping schemes such as ordered XML and schemaless content. Our framework represents the XML Schema [2] through type constructors and uses this representation to define several schema transformations from the existing literature. We also propose several new transformations and more powerful variations of existing ones. We utilize this framework to study, for the first time, the impact of schema

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 19–36, 2003.

transformations and the query workload on *search strategies* for finding efficient XML-to-relational mappings.

We have incorporated FlexMap in the LegoDB prototype [1]. We show that the space of possible configurations is large enough to remove the possibility of exhaustive search even for small XML schemas. We describe a series of greedy algorithms which differ in the number and type of transformations that they utilize, and show how the choice of transformations impacts the search space of configurations. Intuitively, the size of the search space examined increases as the number/types of transformations considered in the algorithms increase. Our empirical results demonstrate that, in addition to deriving better quality configurations, algorithms that search a larger space of configurations can sometimes (counter-intuitively) converge *faster*. Further, we propose optimizations that significantly speed up the search process with very little loss in the quality of the selected relational configuration.

In summary, our contributions are:

1. A framework for exploring the space of XML-to-relational mappings.
2. More powerful variants of existing transformations and their use in search algorithms.
3. A study of the impact of schema transformations and the query workload on search algorithms in terms of the quality of the final configuration as well as the time taken by the algorithm to converge.
4. Optimizations to speed up these algorithms.

Organization. Section 2 develops the framework for XML-to-relational mappings. Section 3 proposes three different search algorithms based on greedy heuristics. Section 4 evaluates the search algorithms and Section 5 discusses several optimizations to reduce the search time. Section 6 discusses related work and Section 7 summarizes our results and identifies directions for future research.

2 Framework for Schema Transformations

2.1 Schema Tree

We define a *schema tree* to represent the XML schema in terms of the following type constructors: *sequence* (","), *repetition* ("$*$"), *option* ("?"), *union* ("|"), $<$*tagname*$>$ (corresponding to a tag) and $<$*simple type*$>$ corresponding to base types (*e.g.,* integer). Figure 1 gives a grammar for the schema tree. The schema tree is an ordered tree where tags appear in the same order as the corresponding XML schema.

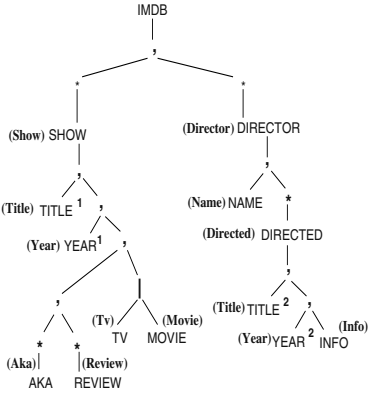
As an example, consider the (partial) XML Schema of the IMDB (Internet Movie DataBase) website [8] in Figure 2(a). Here, **Title**, **Year**, **Aka** and **Review** are simple types. The schema tree for this schema is shown in Figure 2(b) with base types ommitted and tags represented in normal font. Nodes in the tree are *annotated* with the *names* of the types present in the original schema – these annotations are shown in boldface and paranthesized next to the tags. Note that, first, there need not be any correspondence between tag names and annotations (type names). Second, the schema graph is represented as a

```
<complex type> ::=
        <simple type>
        || <complex type> , <complex type>
        || <complex type> | <complex type>
        || <complex type> *
        || <complex type> ?
        || <tagname> [<complex type>]
```

Fig. 1. Using Type Constructors to Represent XML schema Types

define element IMDB {
type **Show***, type **Director***, type **Actor*** }
define type **Show** {element SHOW {
type **Title**, type **Year**, type **Aka***,
type **Review***, (type **Tv** | type **Movie**) }}
define type **Director** {element DIRECTOR {
type **Name**, type **Directed***}}
define type **Directed** {element DIRECTED {
type **Title**, type **Year**, type **Info** }}

(a) The (partial) IMDB Schema



(b) Tree Representation

Fig. 2. The IMDB Schema

tree, where different occurrences of equivalent nodes are captured, but their content is assumed to be shared (see *e.g.,* the nodes TITLE^1 and TITLE^2 in Figure 2(b)). Finally, recursive types can be handled similarly to *shared* types, *i.e.,* the base occurrence and the recursive occurrences are differentiated, but share the same content.

Note that any subtree in the schema tree is a type regardless of whether it is annotated. We refer to annotations as the *name* of the node and use it synonymously with annotation. We also use the terms subtree, node and type interchangeably in the remainder of the paper.

2.2 From Schema Trees to Relational Configurations

Given a schema tree, a relational configuration is derived as follows:

1. If N is the annotation of a node, then there is a relational table T_N corresponding to it. This table contains a *key* column and a *parent_id* column which points to the key column of the table corresponding to the *closest named ancestor* of the current node if it exists. The key column consists of ids assigned specifically for the purpose

Table **Director**	[director_key]
Table **Name**	[Name_key, NAME, parent_director_id]
Table **Directed**	[Directed_key, parent_director_id]
Table **Title**	[Title_key, TITLE, parent_directed_show_id]
Table **Year**	[Year_key, YEAR, parent_directed_show_id]
Table **Info**	[Info_key, INFO, parent_directed_id]

Fig. 3. Relational Schema for the **Director** Subtree

of identifying each tuple (and by extension, the corresponding node of the XML document).

2. If the subtree of the node annotated by N is a simple type, then T_N additionally contains a column corresponding to that type to store its values.
3. If N is the annotation of a node, and no descendant of N is annotated, then T_N contains as many additional columns as the number of descendants of N that are simple types.

Other rules which may help in deriving efficient schemas could include storing repeated types and types which are part of a union in separate tables. The relational configuration corresponding to the naming in Figure 2(b) for the **Director** subtree is shown in Figure 3. Since **Title** and **Year** are shared by **Show** and **Directed**, their parent id columns contain ids from both the **Show** and **Directed** tables. Note that the mapping can follow rules different from the ones listed above (for example, storing types which are part of unions in the same table by utilizing null values).

It is possible to support different mapping schemes as well – for example, in order to support ordered XML, one or more additional columns have to be incorporated into the relational table [16]. By augmenting the type constructors, it is also possible to support a combination of different mapping schemes. For example, by introducing an `ANYTYPE` constructor, we can define a rule mapping annotated nodes of that type to a ternary relation (edge table) [5].

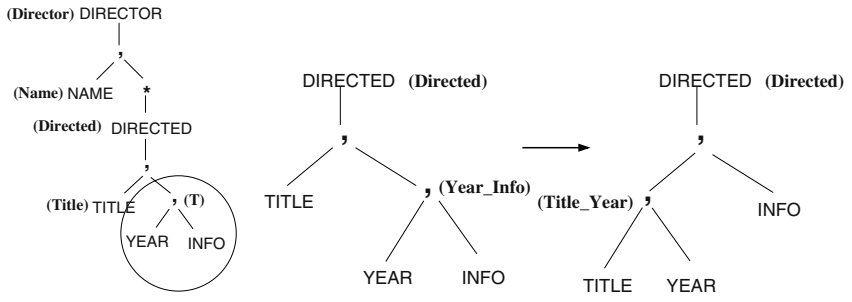
2.3 Schema Transformations

Before we describe the schema transformations we introduce a compact notation to describe the type constructors, and using this notation, we define the notion of *syntactic equality* between two subtrees. In the following, t_i and t are subtrees and a is the annotation.

Tag Constructor: $E(label, t, a)$, where $label$ is the name of the tag (such as TITLE, YEAR, etc.)

Sequence, Union, Option and Repetition Constructors: The constructors are defined as: $C(t_1, t_2, a)$, $U(t_1, t_2, a)$, $O(t, a)$, and $R(t, a)$, respectively.

Simple Type Constructor: Simple types are represented as $S(base, a)$ where $base$ is the base type (*e.g.,* integer).



(a) Introducing Annotation T

(b) Applying Associativity

Fig. 4. Grouping Elements Using Annotations

Definition 1. Syntactic Equality *Two types T_1 and T_2 are syntactically equal – denoted by $T_1 \cong T_2$ – if the following holds:*

case T_1, T_2 *of*

$| E(label, t, a), E(label', t', a') \rightarrow label = label'$ *AND* $a = a'$ *AND* $t \cong t'$

$| C(t_1, t_2, a), C(t_1', t_2', a') \rightarrow a = a'$ *AND* $t_1 \cong t_1'$ *AND* $t_2 \cong t_2'$

$| U(t_1, t_2, a), U(t_1', t_2', a') \rightarrow a = a'$ *AND* $t_1 \cong t_1'$ *AND* $t_2 \cong t_2'$

$| R(t, a), R(t', a') \rightarrow a = a'$ *AND* $t \cong t'$

$| O(t, a), O(t', a') \rightarrow a = a'$ *AND* $t \cong t'$

$| S(b, a), S(b', a') \rightarrow a = a'$ *AND* $b = b'$

Inline and Outline. An annotated node in the schema tree is *Outlined* and has a separate relational table associated with it. All nodes which do not have an annotation associated with them are *Inlined*, *i.e.,* form part of an existing table. We can outline or inline a type by respectively annotating a node or removing its annotation. Inline and outline can be used to group elements together as shown in Figure 4(a) where a new annotation T is introduced. The corresponding relational configuration will "group" **Year** and **Info** into a new table **T**.

Type Split/Merge. The inline and outline operations are analogous to removing and adding annotations to nodes. *Type Split* and *Type Merge* are based on the *renaming* of nodes. We refer to a type as *shared* when it has distinct annotated parents. In the example shown in Figure 2(b), the type **Title** is shared by the types **Show** and **Directed**.

Intuitively, the type split operation distinguishes between two occurrences of a type by renaming the occurrences. By renaming the type **Title** to **STitle** and **DTitle**, a relational configuration is derived where a separate table is created for each title. Conversely, the type merge operation adds identical annotations to types whose *corresponding subtrees* are syntactically equal.

So far, we have defined transformations which do not change the structure of the schema tree. Using only these transformations, the number of derived relational con-

figurations is already exponential in the number of nodes in the tree. That is, since any subset of nodes in the schema tree can be named (subject to the constraints of having to name certain nodes which should have separate relations of their own as outlined in Section 2.2), and the corresponding relational configuration derived, for a schema tree with N nodes, we have a maximum of 2^N possible relational configurations.

We next define several transformations which do change the structure of the schema tree which in turn leads to a further expansion in the search space. Some of these transforms were originally described in our earlier work [1]. We redefine and extend these transforms in our new framework.

Commutativity and Associativity. Two basic structure-altering operations that we consider are: *commutativity* and *associativity*. Associativity is used to *group* different types into the same relational table. For example, in the first tree of Figure 4(b), Year and Info about the type **Directed** are stored in a single table called **Year_Info**, while after applying associativity as shown in the second tree, Title and Year appear in a single table called **Title_Year**.

Commutativity by itself does not give rise to different relational mappings[1], but when combined with associativity may generate mappings different from those considered in the existing literature. For example, in Figure 4(b), by first commuting Year and Info and then applying associativity, we can get a configuration in which Title and Info are stored in the same relation.

Union Distribution/Factorization. Using the standard distribution law for distributing sequences over unions for regular expressions, we can separate out components of a union: $(a,(b|c)) \equiv (a,b)|(a,c)$. We derive useful configurations using a combination of union distribution, outline and type split as shown below:

define type **Show** { element SHOW { type **Title**, (type **Tv**|type **Movie**) }}

Distribute Union $\rightarrow$ *Outline* $\rightarrow$ *Type split* $Title \rightarrow$

define type **TVShow** { element SHOW { type **TVTitle**, type **Tv** }}
define type **MovieShow** { element SHOW { type **MovieTitle**, type **Movie** }}

The relational configuration corresponding to the above schema has separate tables for **TVShow** and **MovieShow**, as well as for **TVTitle** and **MovieTitle**. Moreover, applying this transformation enables the inlining of **TVTitle** into **TVShow** and **MovieTitle** into **MovieShow**. Thus the information about TV shows and movie shows is separated out (this is equivalent to horizontally partitioning the **Show** table). Conversely, the union factorization transform would factorize a union.

In order to determine whether there is potential for a union distribution, one of the patterns we search the schema tree is: C(X,U(Y,Z)) and transform it to U(C(X,Y),C(X,Z)). We have to determine the syntactic equality of two subtrees before declaring the pattern

[1] Commuting the children of a node no longer retains the original order of the XML schema.

to be a candidate for union factorization. Note that there are several other conditions under which union distribution and factorization can be done [11].

Other transforms, such as *splitting/merging repetitions* [1], *simplifying unions* [15] (a lossy transform which could enable the inlining of one or more components of the union), etc. can be defined similarly to the transformations described above.

We refer to Type Merge, Union Factorization and Repetition Merge as *merge transforms* and Type Split, Union Distribution and Repetition Split as *split transforms* in the remainder of this paper.

2.4 Evaluating Configurations

As transformations are applied and new configurations derived, it is important that precise cost estimates be computed for the query workload under each of the derived configurations – which, in turn, requires accurate statistics. Since it is not practical to scan the base data for each relational configuration derived, it is crucial that these statistics be accurately propagated as transformations are applied.

An important observation about the transformations defined in this section is that while merge operations preserve the accuracy of statistics, split operations do not [6]. Hence, in order to preserve the accuracy of the statistics, before the search procedure starts, *all* possible split operations are applied to the user-given XML schema. Statistics are then collected for this *fully decomposed* schema. Subsequently, during the search process, only merge operations are considered.

In our prototype, we use StatiX [6] to collect statistics and to accurately propagate them into the relational domain. The configurations derived are evaluated by a relational optimizer [12]. Our optimizer assumes a primary index on the key column of each relational table. For lack of space, we do not elaborate on these issues. More information can be found in [11].

3 Search Algorithms

In this section we describe a suite of greedy algorithms we have implemented within FlexMap. They differ in the choice of transformations that are selected and applied at each iteration of the search.

First, consider Algorithm 1 that describes a simple greedy algorithm – similar to the algorithm described in [1]. It takes as input a query workload and the initial schema (with statistics). At each iteration, the transform which results in the minimum cost relational configuration is chosen and applied to the schema (lines 5 through 19). The translation of the transformed schema to the relational configuration (line 11) follows the rules set out in Section 2.2. The algorithm terminates when no transform can be found which reduces the cost.

Though this algorithm is simple, it can be made very flexible. This flexibility is achieved by varying the strategies to select applicable transformations at each iteration (function applicableTransforms in line 8). In the experiments described in [1], only inline and outline were considered as the applicable transformations. The utility of the other transformations (*e.g.,* union distribution and repetition split) were shown qualitatively.

Algorithm 1 Greedy Algorithm

```
1: Input: queryWkld, S {Query workload and Initial Schema}
2: prevMinCost ← INF
3: rel_schema ← convertToRelConfig(S, queryWkld)
4: minCost ← COST(rel_schema)
5: while minCost < prevMinCost do
6:    S' ← S {Make a copy of the schema}
7:    prevMinCost ← minCost
8:    transforms ← applicableTransforms(S')
9:    for all T in transforms do
10:      S'' ← Apply T to S' {S' is preserved without change}
11:      rel_schema ← convertToRelConfig(S'', queryWkld)
12:      Cost ← COST(rel_schema)
13:      if Cost < minCost then
14:         minCost ← Cost
15:         minTransform ← T
16:      end if
17:   end for
18:   S ← Apply minTransform to S {The min. cost transform is applied}
19: end while
20: return convertToRelConfig(S)
```

Below, we describe variations to the basic greedy algorithm that allow for a richer set of transformations. All search algorithms described use the fully decomposed schema as the start point, and only merge operations are applied during the greedy iterations.

3.1 InlineGreedy

The first variation we consider is *InlineGreedy* (IG), which only allows the inline transform. Note that IG differs from the algorithm experimentally evaluated in [1], which we term *InlineUser* (IU), in the choice of starting schema: IG starts with the fully decomposed schema whereas InlineUser starts with the original user schema.

3.2 ShallowGreedy: Adding Transforms

The *ShallowGreedy* (SG) algorithm defines the function applicableTransforms to return *all* the applicable merge transforms. Because it follows the transformation dependencies that result from the notion of syntactic equality (see Definition 1), it only performs single-level or *shallow* merges.

The notion of syntactic equality, however, can be too restrictive and as a result, SG may miss efficient configurations. For example consider the following (partial) IMDB schema:

define type **Show** {type **Show1** | type **Show2**}
define type **Show1** {element SHOW { type **Title1**, type **Year1**, type **Tv** }}
define type **Show2** {element SHOW { type **Title2**, type **Year2**, type **Movie** }}

Unless a type merge of **Title1** and **Title2** and a type merge of **Year1** and **Year2** take place, we cannot factorize the union of **Show1** | **Show2**. However, in a run of SG, these two type merges by themselves may not reduce the cost, whereas taken in conjunction with union factorization, they may lead to a substantial cost reduction. Therefore, SG is handicapped by the fact that a union factorization will only be applied if both type merges are independently chosen by the algorithm. In order to overcome this problem, we design a new algorithm called *DeepGreedy* (DG).

3.3 DeepGreedy: *Deep* Merges

Before we proceed to describe the DG algorithm, we first introduce the notions of *Valid Transforms* and *Logical Equivalence*. A *valid transform* is an element of the set of all *applicable* transforms, S. Let V be a set of valid transforms.

Definition 2. Logical Equivalence *Two types T_1 and T_2 are* logically equivalent *under the set V of valid transforms, denoted by $T_1 \sim_V T_2$, if they can be made syntactically equal after applying a sequence of valid transforms from V.*

The following example illustrates this concept. Let $V = \{Inline\}$; $t_1 := E(\mathsf{TITLE}, S(string, -), \mathbf{Title_1})$, and $t_2 := E(\mathsf{TITLE}, S(string, -), \mathbf{Title_2})$. Note that t_1 and t_2 are not syntactically equal since their annotations do not match. However, they are *logically equivalent*: by *inlining* them (*i.e.,* removing the annotations $Title_1$ and $Title_2$), they can be made syntactically equal. Thus, we say that t_1 and t_2 are logically equivalent under the set $\{Inline\}$.

Now, consider two types T_i and T_j where $T_i := E(l, t_1, a_1)$ and $T_j := E(l, t_2, a_2)$ with t_1 and t_2 as defined above. Under syntactic equality, T_i and T_j would not be identified as candidates for type merge. However, if we relax the criteria to logical equivalence with (say) $V = \{TypeMerge\}$, then it is possible to identify the *potential* type merge of T_i and T_j. Thus, several transforms which may never be considered by SG can be identified as candidates by DG, provided the necessary operations can be fired (like the type merge of t_1 and t_2 in the above example) to *enable* the transform. Extending the above concept, we can enlarge the set of valid transforms V to contain all the merge transforms which can be fired recursively to *enable* other transforms.

Algorithm DG allows the same transforms as SG, *except* that potential transforms are identified not by direct syntactic equality, but by logical equivalence with the set of valid transforms containing all the merge operations (including inline). This allows DG to perform *deep* merges. Additional variations of the search algorithms are possible, *e.g.,* by restricting the set of valid transforms. But, they are not covered in this paper.

4 Performance Evaluation

In this section we present a performance evaluation of the three algorithms proposed in this paper: *InlineGreedy* (IG), *ShallowGreedy* (SG) and *DeepGreedy* (DG). We used a synthetically generated subset of the IMDB dataset (≈60MB) for the experiments. The user schema consisted of 22 types, with 2 unions, 3 repetitions and 2 shared types. We describe the query workloads used next. For a more detailed discussion on the schema, dataset and the query workloads, please refer to [11].

4.1 Query Workloads

A query workload consists of a set of queries with a weight (in the range of 0 to 1) assigned to each query. These weights reflect the relative importance of the queries in some way (for example, the query with the largest weight might be the most frequent). We evaluated each of the algorithms on several query workloads based on (1) the quality of the derived relational configuration in terms of the cost for executing the query workload, and (2) the efficiency of the search algorithm measured in terms of the time taken by the algorithm. These are the same metrics as those used in [1]. Note that the latter is proportional to the number of distinct configurations seen by the algorithm, and also the number of distinct optimizer invocations since each iteration involves constructing a new configuration and evaluating its cost using the optimizer.

From the discussion of the proposed algorithms in Section 3, notice that the behavior of each algorithm (which runs on the fully decomposed schema) on a given query depends upon whether the query benefits more from merge transformations or split transformations. If the query benefits more from split, then neither DG nor SG is expected to perform better than IG.

As such, we considered the following two kinds of queries: **S-Queries** which are expected to derive benefit from split transformations (Type Split, Union Distribution and Repetition Split), and **M-Queries** which are expected to derive benefit from merge operations (Type Merge, Union Factorization and Repetition Merge).

S-Queries typically involve simple lookups. For example:

```
SQ1:  for $i in /IMDB/SHOW          SQ2:   for $i in /IMDB/DIRECTOR
      where $i/TV/CHANNEL = 9              where $i/DIRECTED/YEAR = 1994
      return $i/TITLE                      return $i/NAME
```

The query SQ1 is specific about the Title that it wants. Hence it would benefit from a type split of **Title**. Moreover, it also specifies that TV Titles only are to be returned, not merely Show Titles. Hence a union distribution would be useful to isolate only TV Titles. Similarly, query SQ2 would benefit from isolating Director Names from Actor Names and Directed Year from all other Years. Such splits would help make the corresponding tables smaller and hence lookup queries such as the above faster. Note that in the example queries above, both the predicate as well as the return value benefit from splits.

M-queries typically query for subtrees in the schema which are *high up* in the schema tree. When a split operation is performed on a type in the schema, it propagates downwards towards the descendants. For example, a union distribution of **Show** would result in a type split of **Review**, which would in turn lead to the type split of **Review**'s children. Hence queries which retrieve subtrees near the top of the schema tree would benefit from merge transforms. Similarly predicates which are high up in the tree would also benefit from merges. For example:

```
MQ1: for $i in /IMDB/SHOW,              MQ2: for $i in /IMDB/ACTOR,
         $j in $i/REVIEW                         $j in /IMDB/SHOW
     return $i/TITLE, $i/YEAR, $i/AKA,      where $i/PLAYED/TITLE = $j/TITLE
            $j/GRADE, $j/SOURCE,            return $j/TITLE, $j/YEAR, $j/AKA,
            $j/COMMENTS                            $j/REVIEW/SOURCE, $j/REVIEW/GRADE,
                                                   $j/REVIEW/COMMENTS, $i/NAME
```

Query MQ1 asks for full details of a Show without distinguishing between TV Shows and Movie Shows. Since all attributes of Show which are *common* for TV as well as

Movie Shows are requested, this query is likely to benefit from reduced fragmentation, *i.e.,* from union factorization and repetition merge. For example, a union factorization would enable some types like **Title** and **Year** to be inlined into the same table (the table corresponding to Show). Similarly, query MQ2 would benefit from a union factorization of **Show** as well as a repetition merge of **Played** (this is because the query does not distinguish between the Titles of the first Played and the remaining Played). In both the above queries, return values as well as predicates benefit from merge transformations.

Based on the two classes of queries described above and some of their variations, we constructed the following six workloads. Note that each workload consists of a set of queries as well as the associated weights. Unless stated otherwise, all queries in a workload are assigned equal weights and the weights sum up to 1.

SW1: contains 5 distinct S-queries, where the return values as well as predicates benefit from split transforms.

SW2: contains 5 distinct S-queries, with multiple return values which do not benefit from split, but predicates which benefit from split.

SW3: contains 10 S-queries with queries which have: i) return values as well as predicates benefitting from split and ii) only return values benefitting from split.

MW1: contains a single query which benefits from merge transforms.

MW2: contains the same single query as in MW1, but with selective predicates.

MW3: contains 8 queries which are M-Queries as well as M-Queries with selective predicates.

The performance of the proposed algorithms on S-query workloads (SW1-3) and M-query workloads (MW1-3) is studied in Sections 4.2 and 4.3, respectively.

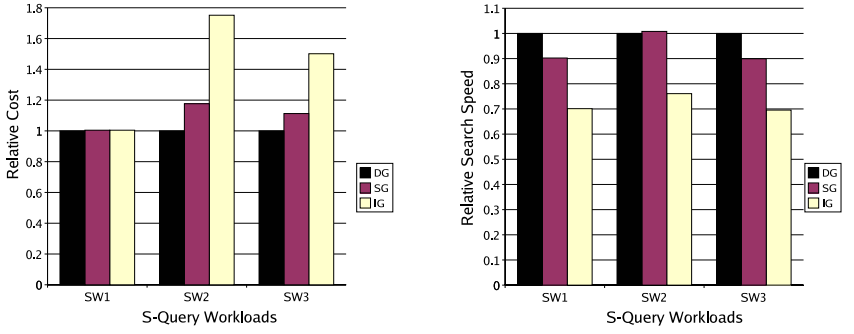
There are many queries which cannot be conclusively classified as either an S-query or an M-query. For example, an interesting variation of S-Queries is when the query contains return values which do not benefit from split, but has predicates which do (SW2). For M-Queries, adding highly selective predicates, may reduce the utility of merge transforms. For example, adding the highly selective predicate $YEAR > 1990$ (Year ranges from 1900 to 2000) to query MW1 would reduce the number of tuples.

We study workloads where the two types of transformations conflict in Section 4.4. Arbitrary queries are unlikely to give much insight because the impact of split transformations vs. merge transformations would be different for different queries. Hence, we chose to work with a *mix* of S- and M-queries where the impact of split and merge transformations is controlled using a parameter. Finally, in Section 4.5 we demonstrate the competitiveness of our algorithms against certain baselines.

4.2 Performance on S-Query Workloads

Recall that DG does "deep" merges, SG does "shallow" merges and IG allows only inlinings. S-Queries do not fully exploit the potential of DG since they do not benefit from too many merge transformations. So, DG can possibly consider transformations which are useless, making it more inefficient – *i.e.,* longer run times without any major advantages in the cost of the derived schema. We present results for the 3 workloads: SW1, SW2 and SW3.

As shown in Figure 5(a), the cost difference of the configurations derived by DG and SG is less than 1% for SW1, whereas SG and IG lead to configurations of the same



(a) Final Configuration Quality

(b) Efficiency of the Algorithms

Fig. 5. Performance of Workloads containing S-Queries

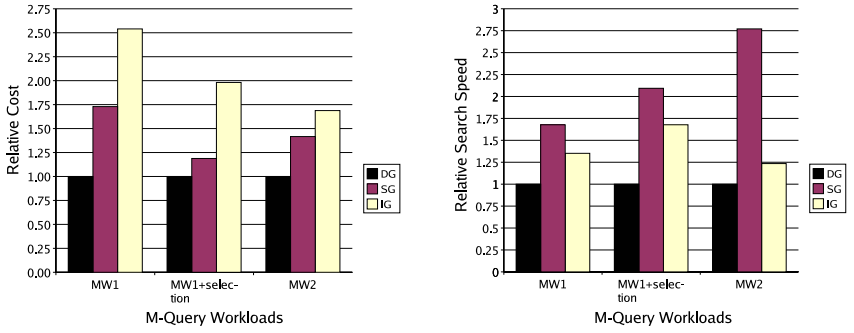
cost for SW1. This is because of the fact that all queries of SW1 benefit mainly from split transforms – in effect, DG hardly has an advantage over SG or IG. But for SW2, the cost difference between DG and SG jumped up to around 17% – this is due to the return values benefiting from merge which gives DG an advantage over SG because of the larger variety of merge transforms it can consider. The difference between DG and IG was around 48%, expectedly so, since even the merge transforms considered by SG were not considered in IG.

The relative number of configurations examined by each of DG, SG and IG are shown in Figure 5(b). In terms of the number of relational configurations examined, DG searches through a much larger set of configurations than SG, while SG examines more configurations than IG. DG is especially inefficient for SW1 where it considers about 30% more configurations than IG for less than 1% improvement in cost.

4.3 Performance on M-Query Workloads

Figure 6(a) shows the relative costs of the 3 algorithms for the 3 workloads, MW1, MW2 and MW3. As expected DG performs extremely well compared to SG and IG since DG is capable of performing deep merges which benefit MW1. Note that the effect of adding selective predicates reduces the magnitude of difference in the costs between DG, SG and IG.

In terms of the number of configurations examined also, DG performed *the best* as compared to SG and IG. This would seem counter-intuitive – we would expect that since DG is capable of examining a superset of transformations as compared to SG and IG, it would take longer to converge. However, this did not turn out to be the case since DG picked up the cost saving recursive merges (such as union factorization) fairly early on in the run of DG which reduced the number of lower level merge and inline candidates in the subsequent iterations. This enabled DG to converge faster. By the same token, we would expect SG to examine fewer configurations than IG, but that was not the case. This



(a) Final Configuration Quality

(b) Efficiency of the Algorithms

Fig. 6. Performance of Workloads containing M-Queries

is because SG was not able to perform any major cost saving merges since the "enabling" merges were never chosen individually (note the cost difference between DG and SG). Hence, the same set of merge transforms were being examined in every iteration without any benefit, while IG was not burdened with these candidate merges. But note that even though IG converges faster, it is mainly due to the lack of useful inlines as reflected by the cost difference between IG and SG.

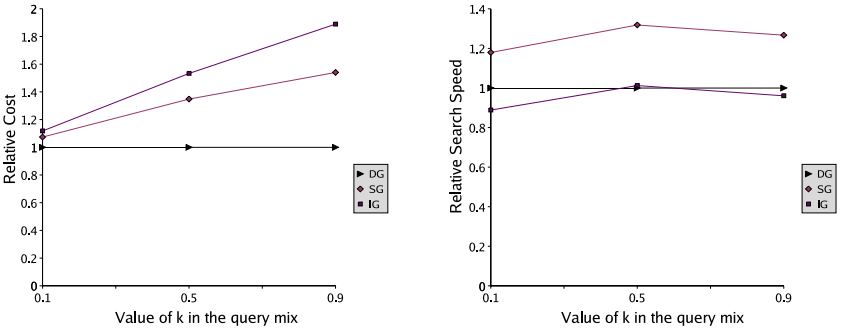
4.4 Performance on Controlled S-Query and M-Query Mixed Workloads

The previous discussion highlighted the strengths and weaknesses of each algorithm. In summary, if the query workload consists of "pure" S-Queries, then IG is the best algorithm to run since it returns a configuration with marginal difference in cost compared to DG and in less time (reflected in the results for SW1), while if the query workload consists of M-Queries, then DG is the best algorithm to run.

In order to study the behaviour of mixed workloads, we used a workload (named MSW1) containing 11 queries (4 M-Queries and 7 S-Queries).

In order to control the dominance of S-queries vs. M-queries in the workload, we use a control parameter $k \in [0, 1]$ and give weight $(1 - k)/7$ to each of the 7 S-queries and weight $k/4$ to each of the 4 M-queries. We ran workload MSW1 with 3 different values of $k = \{0.1, 0.5, 0.9\}$. The cost of the derived configurations for MSW1 are shown in Figure 7(a). Expectedly, when S-Queries dominate, IG performs quite competitively with DG (with the cost of IG being within just 15% of DG). But, as the influence of S-Queries reduce, the difference in costs increases substantially.

The number of configurations examined by all three algorithms are shown in Figure 7(b). DG examines more configurations than IG when S-Queries dominate, but the gap is almost closed for the other cases. Note that both SG and IG examine more configurations for k = 0.5 than in the other two cases. This is due to the fact that when S-Queries dominate (k = 0.1), cost-saving inlines are chosen earlier; while when M-queries dom-



(a) Final Configuration Quality (b) Efficiency of the algorithms

Fig. 7. Performance of Workloads containing both M- and S-Queries

inate (k = 0.9), both algorithms soon run out of cost-saving transformations to apply. Hence for both these cases, the algorithms converge faster.

4.5 Comparison with Baselines

From the above sections, it is clear that except when the workload is dominated by S-queries, DG should be our algorithm of choice among the algorithms proposed in this paper. In this section we compare the cost of the relational configurations derived using DG with the following baselines:

InlineUser (IU): This is the same algorithm evaluated in [1].

Optimal (OPT): A lower bound on the optimal configuration for the workload given a specific set of transformations. Since DG gives configurations of the best quality among the 3 algorithms evaluated, the algorithm to compute the lower bound consisted of transforms available to DG. We evaluated this lower bound by considering each query in the workload individually and running an *exhaustive* search algorithm on the subset of types relevant to the query. Note that an exhaustive search algorithm is possible only if the number of types involved is very small since the number of possible relational configurations increases exponentially with the number of types. The exhaustive search algorithm typically examined several orders of magnitude more configurations than DG.

We present results for two workloads, MSW1 and MSW2 (MSW1 contains 4 M- and 7 S-Queries and MSW2 contains 3 M- and 5 S-Queries). The proportion of queries in each workload was 50% each for S-Queries and M-Queries. The relative cost for each baseline is shown in Figure 8. InlineUser compares unfavorably with DG. Though InlineUser is good when there are not many shared types, it is bad if the schema has a few types which are shared or repeated since there will not be too many types left to inline. The figures for the optimal configuration show that DG is within 15% of the optimal, *i.e.,* it provides extremely high quality configurations. This also implies that the

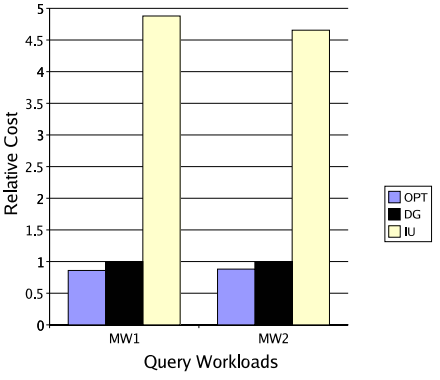


Fig. 8. Comparison of DG with the Baselines

choice of starting schema (fully decomposed) does not hamper the search algorithm in finding an efficient configuration.

5 Optimizations

Several optimizations can be applied to speed up the search process. In what follows, we illustrate two such techniques. For a more comprehensive discussion, the reader is referred to [11].

5.1 Grouping Transformations Together

Recall that in DG, in a given iteration, *all* applicable transformations are evaluated and the best transformation is chosen. In the next iteration, all the remaining applicable transformations are evaluated and the best one chosen. We found that in the runs of our algorithms, it was often the case that, in a given iteration in which n transforms were applicable, if transformations T_1 to T_n were the best n transformations in this order (that is, T_1 gave the maximum decrease in cost and T_n gave the minimum decrease), other transformations up to T_i, for some $i <= n$, were chosen in subsequent iterations. This being the case, grouping transformations T_1 to T_i together has the potential to save several iterations. Using this observation, we developed a variation of Algorithm 1, called *GroupGreedy* (GG).

We tried this optimization for DG on several workloads and the results were very encouraging. The cost of the final configuration of GG was within 1% of DG and the number of configurations examined by GG were about 30% for MW1 and about 20% for MW2 compared to DG.

5.2 Early Termination

One obvious optimization is to stop the algorithm once the decrease in the estimated cost goes below a small threshold δ. This saves several iterations which are costly to perform,

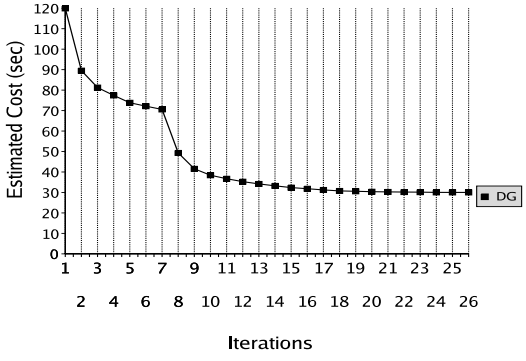


Fig. 9. Progress of DeepGreedy on Workload W

but do not give substantial decrease in cost. This optimization would be possible if the *decrease* in cost is monotonic. However, during the course of our experiments, we came across several workloads which did not exhibit this behavior. The progress of DG on such a workload, W, is shown in Figure 9. Reasons for this behaviour are analyzed in [11].

Clearly, with an unfortunate value of δ, the algorithm would terminate at iteration 7 and miss the big cost decrease at iteration 8. Thus, while this optimization would result in improved execution times, the derived schema may be suboptimal.

6 Related Work

Existing techniques for XML-to-relational storage can be broadly classified into: *generic* (*e.g.,* the edge mapping of [5]); *data-centric*, where the structure of the XML document is *mined* to guide the mapping process (*e.g.,* [4,14,18]); and *schema-centric*, which make use of schema information in the form of DTD or XML Schema in order to derive an efficient relational storage design for XML documents (see *e.g.,* [15,16,17]).

The LegoDB system [1] was the first schema-centric cost-based approach for automatically generating XML-to-relational mappings and took into account the schema, statistics and the query workload to derive a low-cost relation configuration. In this paper, we examine the search problem in detail. More recently, a cost-based approach was also described in [19] where a hill-climbing algorithm and a set of four transforms are used. Though the goals of our work and theirs is the same, we differ in the set of transformations used (they consider transforms similar to inline/outline and type split/merge). Also, we have developed a series of search strategies and proposed optimizations to prune the search space.

Formalizing the problem of finding the *optimal* XML-to-relational mapping was considered in [9]. They analyze the interaction between mapping and query translation for a subset of XML queries and XML Schemas under two simple cost metrics. In contrast, our goal in the paper is to develop *practical algorithms* for selecting *good* decompositions.

Support for XML storage is currently provided by all major commercial RDBMSs, including SQLServer [3], Oracle XML DB [10] and DB2 [7]. Although different kinds of mappings are available, these mappings either need to be defined by the user or are fixed. These systems could benefit from a cost-based approach such as the one described in this paper.

7 Conclusions and Future Work

In this paper, we described a framework for exploring the space of XML-to-relational mappings and defined several transformations which exploit the regular expressions in XML Schema (such as unions and repetitions). These transformations encompass physical database design strategies such as vertical and horizontal partitioning – through the use of inline/outline and union distribution respectively. The framework is extensible and new transformations such as some of the OO-to-relational mapping techniques [13] can be added.

We designed and implemented three greedy algorithms and studied how the quality of the final configuration is influenced by the transformations used and the query workload. We have also proposed optimizations to speed up the time taken by the search algorithm with little loss in the quality of the final relational configuration. Experimental results show that our new algorithms provide significantly improved relational schemas as compared to those derived by previous approaches in the literature.

This study can serve as a platform for further investigation into the problem of efficient storage of XML in relational backends. There are several directions for future work. For example:

1. The query workloads considered in this paper contain queries which retrieve data from the database. It would be interesting to investigate a broader range of workloads such as those which involve updates to the database. Update queries are especially significant if indexes and views are considered in the relational configuration.
2. Another scenario not so far addressed is what would happen if the application's query workload changes significantly in terms of the queries being asked. The challenge would then be to find a "minimum change" relational configuration which is efficient for the new workload as well as efficient in terms of the changes needed to the existing configuration (*i.e.,* schema evolution).

Acknowledgements. This work was supported in part by a Swarnajayanti Fellowship from the Dept. of Science & Technology, Govt. of India.

References

1. P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *Proc. of ICDE*, 2002.
2. A. Brown, M. Fuchs, J. Robie, and P. Wadler. XML Schema: Formal description, 2001. W3C working draft available at http://www.w3.org/TR/2001/WD-xmlschema-formal-20010320/.

3. A. Conrad. A survey of Microsoft SQL Server 2000 XML features. http://msdn.microsoft.com/library/en-us/dnexxml/html/xml07162001.asp?frame=true, July 2001.
4. M. Fernandez, W. C. Tan, and D. Suciu. Silkroute: trading between relations and XML. *WWW9/Computer Networks*, 33(1-6), 2000.
5. D. Florescu and D. Kossman. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3), 1999.
6. J. Freire, J. Haritsa, M. Ramanath, P. Roy, and J. Siméon. Statix: Making XML count. In *Proc. of SIGMOD*, 2002.
7. IBM DB2 XML Extender. http://www-3.ibm.com/software/data/db2/extenders/xmlext/library.html.
8. Internet movie database. http://www.imdb.com.
9. R. Krishnamurthy, V. Chakaravarthy, and J. Naughton. On the difficulty of finding optimal relational decompositions for XML workloads: a complexity theoretic perspective. In *Proc. of ICDT*, 2003.
10. Oracle XML DB: An oracle technical white paper. http://technet.oracle.com/tech/xml/content.html, 2003.
11. M. Ramanath, J. Freire, J. Haritsa, and P. Roy. Searching for efficient XML to relational mappings. Technical Report TR-2003-01, DSL/SERC, Indian Institute of Science, 2003.
12. P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proc. of SIGMOD*, 2000.
13. M. Rys. *Materialisation and Parallelism in the Mapping of an Object Model to a Relational Multi-Processor System*. PhD thesis, ETH, Zurich, 1997.
14. A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *Proc. of WebDB*, 2000.
15. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of VLDB*, 1999.
16. I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proc. of SIGMOD*, 2002.
17. Wang Xiao-ling, Luan Jin-feng, and Dong Yi-sheng. An adaptable and adjustable mapping from XML data to tables in RDB. In *First VLDB Workshop on EEXTT*, 2002.
18. M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. Xrel: A path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology*, 1(1), 2001.
19. S. Zheng, J-R. Wen, and H. Lu. Cost-driven storage schema selection for XML. In *Proc. of DASFAA*, 2003.

A Virtual XML Database Engine for Relational Databases

Chengfei Liu[1], Millist W. Vincent[1], Jixue Liu[1], and Minyi Guo[2]

[1] University of South Australia, Adelaide, SA 5095, Australia
[2] The University of Aizu, Aizu-Wakamatsu City, Fukushima, 965-8580, Japan

Abstract. While XML is emerging as the universal format for publishing and exchanging data on the Web, most business data is still stored and maintained in relational DBMSs. To enable eBusiness database applications, Web access to the legacy data managed by DBMSs needs to be provided. In this paper, we introduce a virtual XML database engine VXE-R which allows users query a relational database via XML as if they were accessing XML documents. Algorithms for schema transformation and query translation in VXE-R are presented.

1 Introduction

While XML [1,4] is emerging as the universal format for publishing and exchanging data on the Web, most business data is still stored and maintained in relational DBMSs. In fact, relational DBMSs will remain dominant in managing business data in foreseeable future because of their powerful data management services. However, relational databases are proprietary and only accessible within an enterprise. To enable eBusiness database applications, it is important for enterprises to publish their relational databases as XML documents given that XML documents are universally accessible.

A general approach to publish relational data is to create XML views of the underlying relational data. Once XML views are created over a relational database, there are two ways to use these views. A simple way is to materialize the XML views by physically creating the result XML documents specified by the views. Obviously, this may not applicable to a large view; otherwise tremendous amount of spaces may be used. Maintenance of the materialized views may also need extra computation. A better way is to support queries over XML views. SilkRoute [7] is one of the systems taking this approach. In SilkRoute, XML views of a relational database are defined using a relational to XML transformation language called RXL, and then XML-QL queries are issued against views. The queries and views are combined together by a query composer and the combined RXL queries are then translated into corresponding SQL queries. XPERANTO [5,10,11] takes a similar approach but uses XQuery [3] for user queries.

We take a different approach. Instead of defining views based on relational databases, we translate the underlying relational schema into equivalent XML

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 37–51, 2003.

schema. Then XML queries are issued directly against the XML schema. Schema mapping rules are designed to generate a normalized XML schema which bring no data redundancy from the underlying relational schema. The translated XML schema also preserves integrity constraints defined in a relational database schema. It is important for users to be aware of the constraints in the XML schema against which they are going to issue queries. In the SilkRoute and XPERANTO approaches, users cannot see the integrity constraints buried in the relational schema from the XML views defined. Another benifit of our proposed approach is that the query translation process gets simplified.

In this paper, we introduce a virtual XML database engine VXE-R which allows users query a relational database via XML as if they were accessing XML documents. VXE-R is composed of three components. A schema translator which translates the underlying relational schema into equivalent XML schema, a query translator which translates the XQuery queries against XML schema into the corresponding SQL queries against the underlying relational schema, and an XML document generator which converts SQL result tables into XML documents.

The rest of the paper is organized as follows. After the architecture of VXE-R is presented in Section 2, we discuss the translation from relational schema to XML schema in Section 3. The translation of XQuery queries to corresponding SQL queries is described in Section 4. The XML document generator is introduced in Section 5. Section 6 concludes the paper.

2 The Architecture

The architecture of the virtual XML database engine VXE-R is shown in Figure 1. There are three components:

- A schema translator
- A query translator
- An XML document generator

The schema translator is responsible to translate a relational database schema into the corresponding schema in XML Schema. We choose XML Schema [6] because Data Type Definition (DTD) has a number of limitations, e.g., it is written in a non-XML syntax; it has no support of namespaces; it only offers extremely limited data typing. XML Schema is a more comprehensive and rigorous method for defining content model of an XML document. The schema itself is an XML document, and so can be processed by the same tools that read the XML documents it describes. XML Schema supports rich built-in types and allows complex types built based on built-in types. It also supports key and unique constraints which are important to map relational databases to XML documents.

Once an XML schema is created, user queries in XQuery can be formulated against it. As the real data is stored in relational databases, it is the responsibility of the query translator to translate the XQuery queries into the corresponding SQL queries against the underlying relational schema. The translated SQL

queries are passed to a relational DBMS for execution. XQuery [3] is chosen as the XML query language since it is currently being standardized by the W3C.

After the execution of the translated SQL queries, the result relations are passed to the XML document generator which generates the result XML documents for users after possible re-structuring according to the requirements specified in the XQuery queries.

In the following sections, we describe these three components in detail.

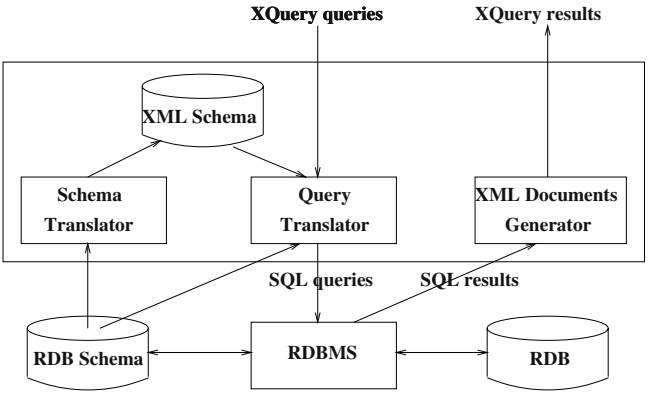


Fig. 1. Architecture of VXE-R

3 Schema Translation

In a relational database schema, different types of integrity constraints may be defined, e.g., primary keys (PKs), foreign keys (FKs), null/not-null, unique, etc. It is important to map all these constraints to the target XML schema. Also we aim to achieve high level of nesting and to avoid introducing redundancy in the target schema.

Basically, the null/not-null constraint can be easily represented by properly setting *minOccurs* of the transformed XML element for the relation attribute. The unique constraint can also be represented by the unique mechanism in XML Schema straightforwardly. In the following, we first focus on the mapping of PK/FK constraints, then we consider further on the null/not-null and unique constraints.

XML Schema supports two mechanisms to represent identity and reference: one is ID/IDREF while the other is KEY/KEYREF. There are differences in using these two mechanisms. The former supports the dereference function in path expressions in most XML query languages including XQuery, however, it only applies to a single element/attributes. The latter may apply to multiple elements/attributes but cannot support the dereference function. For schema translation, we use ID/IDREF where possible because of the dereference function sup-

port. For this purpose, we will differentiate the single attribute primary/foreign keys from multi-attribute primary/foreign keys while transforming the relational database schema to XML schema. We also classify a relation into the following four categories based on different types of primary keys:

- *regular*: the primary key of a regular relation contains no foreign keys.
- *component*: the primary key of a component relation contains one foreign key which references its parent relation. The other part of the primary key serves as a local identifier under the parent relation. A component relation is used to represent a component or a multi-valued attribute of its parent relation.
- *supplementary*: the primary key of a supplementary relation as a whole is also a foreign key which references another relation. This relation is used to supplement another relation or to represent a subclass for translating a generalization hierarchy from a conceptual schema.
- *association*: the primary key of an association relation contains more than one foreign keys, each of which references a participant relation.

Based on above discussion, we give the set of mapping rules.

3.1 Basic Mapping Rules

Given a relational database schema *Sch* with primary/foreign key definitions, we may use the following basic mapping rules to convert *Sch* into a corresponding XML schema *Sch_XML*.

Rule 1 *For a relational database schema* Sch*, a root element named* Sch_XML *is created in the corresponding XML schema as follows.*

```
<xs: element name = "Sch_XML">
  <xs: complexType>
    <xs: sequence>
      <!-- translated relation schema of Sch -->
    </xs: sequence>
  </xs: complexType>
</xs: element>
```

Rule 2 *For each regular or association relation R, the following element with the same name as the relation schema is created and then put under the root element.*

```
<xs: element name = "R" minOccurs = "0" maxOccurs = "unbounded">
  <xs: complexType>
    <xs: sequence>
      <!-- the attributes of R -->
    </xs: sequence>
  </xs: complexType>
</xs: element>
```

Rule 3 *For each component relation R_1, let its parent relation be R_2, then an element with the same name as the component relation is created and then placed as a child element of R_2. The created element has the same structure as the element created in Rule 2.*

Rule 4 *For each supplementary relation R_1, let the relation which R_1 references be R_2, then the following element with the same name as the supplementary relation schema is created and then placed as a child element of R_2. The created element has the same structure as the element created in Rule 2 except that the* maxOccurs *is 1.*

Rule 5 *For each single attribute primary key with the name PKA of regular relation R, an attribute of the element for R is created with ID data type as follows.*

```
<xs: attribute name = "PKA" type = "xs:ID"/>
```

Rule 6 *For each multiple attribute primary key PK of a regular, a component or an association relation R, suppose the key attributes are $PKA_1, \cdots, PKA_n$, an attribute of the element for R is created for each $PKA_i (1 \leq i \leq n)$ with the corresponding data type. If R is a component relation and PKA_i is a single attribute foreign key contained in the primary key, then the data type of the created attribute is* IDREF*. After that a* key *element is defined with a* selector *to select the element for R and several* fields *to identify $PKA_1, \cdots, PKA_n$. The key element can be defined inside or outside the element for R. The name of the element PK should be unique within the namespace.*

```
<xs: element name = "R" minOccurs = "0" maxOccurs = "unbounded">
  <xs: complexType>
    <xs: attribute name = "PKA1" type = "xs:PKA1_type"/>
    ... ...
    <xs: attribute name = "PKAn" type = "xs:PKAn_type"/>
  </xs: complexType>
  <xs: key name = "PK"/>
    <xs: selector xpath = "R/"/>
    <xs: field xpath = "@PKA1"/>
    ... ...
    <xs: field xpath = "@PKAn"/>
  </xs: key>
</xs: element>
```

Rule 7 *Ignore the mapping for primary key of each supplementary relation.*

Rule 8 *For each single attribute foreign key FKA of a relation R except one which is contained in the primary key of a component or supplementary relation, an attribute of the element for R is created with IDREF data type.*

```
<xs: attribute name = "FKA" type = "xs:IDREF"/>
```

Rule 9 *For each multiple attribute foreign key FK of a relation R except one which is contained in the primary key of a component or supplementary relation, suppose FK references PK of the referenced relation, and the foreign key attributes are $FKA_1, \cdots, FKA_n$, an attribute of the element for R is created for each $FKA_i (1 \leq i \leq n)$ with corresponding data type. Then a* keyref *element is defined with a* selector *to select the element for R and several* fields *to identify $FKA_1, \cdots, FKA_n$. The keyref element can be defined either inside or outside the element. The name of the element FK should be unique within the namespace and* refer *of the element is the name of the key element of the primary key which it references.*

```
<xs: element name = "R" minOccurs = "0" maxOccurs = "unbounded">
  <xs: complexType>
    <xs: attribute name = "FKA1" type = "xs:FKA1_type"/>
    ... ...
    <xs: attribute name = "FKAn" type = "xs:FKAn_type"/>
  </xs: complexType>
  <xs: keyref name = "FK" refer = "PK"/>
    <xs: selector xpath = "R/"/>
    <xs: field xpath = "@FKA1"/>
    ... ...
    <xs: field xpath = "@FKAn"/>
  </xs: keyref>
</xs: element>
```

Rule 10 *For each non-key attribute of a relation R, an element is created as a child element of R. The name of the element is the same as the attribute name.*

Rule 1 to Rule 10 are relatively straitforward for mapping a relational database schema to a corresponding XML schema. One property of these rules is redundancy free preservation, i.e., Rule 1 to Rule 10 do not introduce any data redundancy provided the relational schema is redundancy free.

Theorem 3.1. *If the relational database schema* Sch *is redundancy free, the XML schema* Sch_XML *generated by applying Rule 1 to Rule 10 is also redundancy free.*

This theorem is easy to prove. For a regular or an association relation R, an element with the same name R is created under the root element, so the relation R in *Sch* is isomorphically transformed to an element in *Sch_XML*. For a component relation R, a sub-element with the same name R is created under its parent R_p. Because of the foreign key constraint, we have the functional dependency $PK_R \rightarrow PK_{R_p}$, i.e., there is a many to one relationship from R to R_p, therefore it is impossible that a tuple of R is placed more than one time under different element of R_p. Similar to a component relation, there is no redundancy introduced for a supplementary relation.

3.2 An Example

Let us have a look of a relational database schema *Company* for a company. Primary keys are underlined while foreign keys are in *italic* font.

Employee(eno, name, city, salary, *dno*)
Dept(dno, dname, *mgrEno*)
DeptLoc(*dno*, city)
Project(pno, pname, city, *dno*)
WorksOn(*eno, pno*, hours)

Given this schema as an input, the following XML schema will be generated:

```
<xs:element name="Company_XML">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="Employee" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="salary" type="xs:int"/>
     </xs:sequence>
     <xs:attribute name="eno" type="xs:ID"/>
     <xs:attribute name="dno" type="xs:IDREF"/>
    </xs:complexType>
   </xs:element>
   <xs:element name="Dept" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="dname" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="DeptLoc" minOccurs="0" maxOccurs="unbounded">
       <xs:complexType>
        <xs:attribute name="dno" type="xs:IDREF"/>
        <xs:attribute name="city" type="xs:string"/>
       </xs:complexType>
       <xs:key name="PK_DeptLoc"/>
        <xs:selector xpath="Dept/DeptLoc/"/>
        <xs:field xpath="@dno"/>
        <xs:field xpath="@city"/>
       </xs: key>
      </xs:element>
     </xs:sequence>
     <xs:attribute name="dno" type="xs:ID"/>
     <xs:attribute name="mgrEno" type="xs:IDREF"/>
    </xs:complexType>
   </xs:element>
   <xs:element name="Project" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="pname" type="xs:string"/>
```

```
      <xs:element name="city" type="xs:string"/>
     </xs:sequence>
     <xs:attribute name="pno" type="xs:ID"/>
     <xs:attribute name="dno" type="xs:IDREF"/>
    </xs:complexType>
   </xs:element>
   <xs:element name="WorksOn" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
     <xs:element name="hours" type="xs:int"/>
     <xs:attribute name="eno" type="xs:IDREF"/>
     <xs:attribute name="pno" type="xs:IDREF"/>
     <xs:key name="PK_WorksOn"/>
      <xs:selector xpath="WorksOn/"/>
      <xs:field xpath="@eno"/>
      <xs:field xpath="@pno"/>
     </xs: key>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

The root element *Company_XML* is created for the relational database schema *Company*. Under the root element, four set elements *Employee, Dept, Project* and *WorksOn* are created for relation schema *Employee, Dept, Project* and *WorksOn*, respectively. For component relation schema *DeptLoc*, element *DeptLoc* is created under element *Dept* for its parent relation. PK/FK constraints in the relational database schema *Company* have been mapped to the XML schema *Company_XML* by using ID/IDREF and KEY/FEYREF.

3.3 Exploring Nested Structures

As we can see, the basic mapping rules fail to explore all possible nested structures. For example, the *Project* element can be moved to under the *Dept* element if every project belongs to a department. Nesting is important in XML schema because it allows navigation of path expressions to be processed efficiently. If we use IDREF instead, we may use system supported dereference function to get the referenced elements. In XML, the dereference function is expensive because ID and IDREF types are value based. If we use KEYREF, we have to put an explicit *join* condition in an XML query to get the referenced elements. Therefore, we need to explore all possible nested structure by investigating the referential integrity constraints in the relational schema. For this purpose, we introduce a reference graph as follows:

Definition 3.1. *Given a relational database schema* $Sch = \{R_1, \cdots, R_n\}$*, a* reference graph *of the schema Sch is defined as a labeled directed graph* $RG = (V, E, L)$ *where* V *is a finite set of vertices representing relation schema* R_1, $\cdots$, R_n *in Sch;* E *is a finite set of arcs, if there is a foreign key defined in* R_i

which references R_j, an arc $e =< R_i, R_j >\in E$; L is a set of labels for edges by applying a labeling function from E to the set of attribute names for foreign keys.

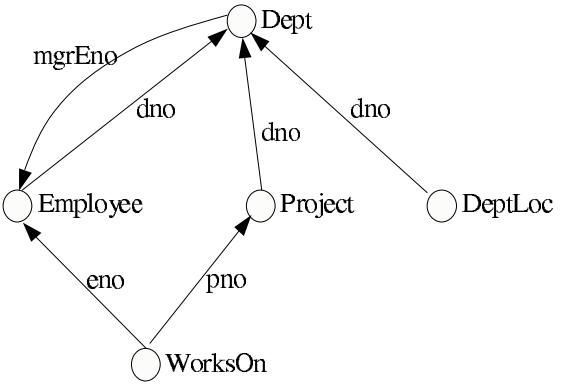


Fig. 2. A Reference Graph

The reference graph of the relational schema *Company* is shown as in Figure 2. In the graph, the element of node *DeptLoc* has been put under the element of node *Dept* by Rule 3. From the graph, we may have the following improvement if certain conditions are satisfied.
(1) The element of node *Project* could be put under the element of node *Dept* if the foreign key *dno* is defined as NOT-NULL. This is because that node *Project* only references node *Dept* and a many to one relationship from *Project* to *Dept* can be derived from the foreign key constraint. In addition, the NOT-NULL foreign key means every project has to belong one department. As a result, one project can be put under one department and cannot be put twice under different departments in the XML document.
(2) A loop exists between *Employee* and *Dept*. What we can get from this is a many to many relationship between *Employee* and *Dept*. In fact, the foreign key *mgrEno* of Dept reflects a one to one relationship from *Dept* to *Employee*. Fortunately, this semantics can be captured by checking the *unique* constraint defined for the foreign key *mgrno*. If there is such a unique constraint defined, the foreign key *mgrEno* of Dept really suggests a one to one relationship from *Dept* to *Employee*. For the purpose of nesting, we delete the arc from Dept to Employee labelled *mgrno* from the reference graph. The real relationship from *Employee* to *Dept* is many to one. As such, the element of the node *Employee* can also be put under the element of the node *Dept* if the foreign key *dno* is defined to NOT-NULL.
(3) The node *WorksOn* references two nodes *Employee* and *Project*. The element of *WorksOn* can be put under either *Employee* and *Project* if the corresponding foreign key is NOT-NULL. However, which node to choose to put under all de-

pends on which path will be used often in queries. We may leave this decision to be chosen by a designer.

Based on the above discussion, we can improve the basic mapping rules by the following theorems.

Theorem 3.2. *In a reference graph* RG*, if a node n_1 for relation R_1 has only one outcoming arc to another node n_2 for relation R_2 and foreign key denoted by the label of the arc is defined as NOT-NULL and there is no loop between n_1 and n_2, then we can move the element for R_1 to under the element for R_2 without introducing data redundancy.*

The proof of this theorem has already explained by the relationships between *Project* and *Dept*, and between *Dept* and *Employee* in Figure 2. The only arc from n_1 to n_2 and there is no loop between the two nodes represents a many to one relationship from R_1 to R_2, while the NOT-NULL foreign key gives a many to exact one relationship from R_1 to R_2. Therefore, for each instance of R_1, it is put only once under exactly one instance of R_2, no redundancy will be introduced.

Similarly, we can have the following.

Theorem 3.3. *In a reference graph* RG*, if a node n_0 for relation R_0 has outcoming arcs to other nodes $n_1, \cdots, n_k$ for relations $R_1, \cdots, R_k$, respectively, and the foreign key denoted by the label of at least one such outcoming arcs is defined as NOT-NULL and there is no loop between n_0 and any of $n_1, \cdots, n_k$, then we can move the element for R_0 to under the element for R_i ($1 \leq i \leq k$) without introducing data redundancy provided the foreign key defined on the label of the arc from n_0 to n_i is NOT-NULL.*

Rule 11 *If there is only one many to one relationship from relation R_1 to another relation R_2 and the foreign key of R_1 to R_2 is defined as NOT-NULL, then we can move the element for R_1 to under the element for R_2 as a child element.*

Rule 12 *If there are more than one many to one relationship from relation R_0 to other relations $R_1, \cdots, R_k$, then we can move the element for R_0 to under the element for R_i ($1 \leq i \leq k$) as a child element provided the foreign key of R_0 to R_k is defined as NOT-NULL.*

By many to one relationship from relation R_1 to R_2, we mean that there is one arc which cannot be deleted from node n_1 for R_1 to node n_2 for R_2, and there is no loop between n_1 and n_2 in the reference graph. If we apply Rule 11 to the transformed XML schema *Company_XML*, the elements for *Project* and *Employee* will be moved to under *Dept* as follows, the attribute *dno* with IDREF type can be removed from both *Project* and *Employee* elements.

```
<xs:element name="Dept" minOccurs="0" maxOccurs="unbounded">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="dname" type="xs:string"/>
```

```
   <xs:element name="city" type="xs:string"/>
   <xs:element name="DeptLoc" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
     <xs:attribute name="dno" type="xs:IDREF"/>
     <xs:attribute name="city" type="xs:string"/>
    </xs:complexType>
    <xs:key name="PK_DeptLoc"/>
     <xs:selector xpath="Dept/DeptLoc/"/>
     <xs:field xpath="@dno"/>
     <xs:field xpath="@city"/>
    </xs: key>
   </xs:element>
   <xs:element name="Project" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="pname" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
     </xs:sequence>
     <xs:attribute name="pno" type="xs:ID"/>
    </xs:complexType>
   </xs:element>
   <xs:element name="Employee" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="salary" type="xs:int"/>
     </xs:sequence>
     <xs:attribute name="eno" type="xs:ID"/>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
  <xs:attribute name="dno" type="xs:ID"/>
  <xs:attribute name="mgrEno" type="xs:IDREF"/>
 </xs:complexType>
</xs:element>
```

XML Schema offers great flexibility in modeling documents. Therefore, there exist many ways to map a relational database schema into a schema in XML Schema. For examples, XViews [2] constructs graph based on PK/FK relationship and generate candidate views by choosing node with either maximum in-degree or zero in-degree as root element. The candidate XML views generated achieve high level of nesting but suffer considerable level of data redundancy. NeT [8] derives nested structures from flat relations by repeatedly applying *nest* operator on tuples of each relation. The resulting nested structures may be useless because the derivation is not at the type level. Compared with XViews and NeT, our mapping rules can achieve high level of nesting for the translated XML schema while introducing no data redundancy provided the underlying relational schema is redundancy free.

4 Query Translation

In this section, we discuss how XQuery queries are translated to corresponding SQL queries. SQL is used to express queries on flat relations, where a join operation may be used frequently to join relations together; while XQuery is used to express queries on elements which could be highly nested by sub-elements or linked by IDREF, where navigation via path expression is the main means to link elements of a document together. As XQuery is more powerful and flexible than SQL, it is hard to translate an arbitrary XQuery query to corresponding SQL query. Fortunately, in VXE-R, the XML schema is generated from the underlying relational database schema, therefore, the structure of the mapped XML elements is *normalized*. Given the mapping rules introduced in Section 3, we know the reverse mapping which is crucial for translating queries in XQuery to the corresponding queries in SQL.

As XQuery is still in its draft version, in this paper, we only consider the translation of basic XQuery queries which do not include aggregate functions. The main structure of an XQuery query can be formulated by an FLWOR expression with the help of XPath expressions. An FLWOR expression is constructed from FOR, LET, WHERE, ORDER BY, and RETURN clauses. FOR and LET clauses serve to bind values to one or more variables using (path) expressions. The FOR clause is used for iteration, with each variable in FOR iterates over the nodes returned by its respective expression; while the optional LET clause binds a variable to an expression without iteration, resulting in a single binding for each variable. As the LET clause is usually used to process grouping and aggregate functions, the processing of the LET clause is not discussed here. The optional WHERE clause specifies one or more conditions to restrict the binding-tuples generated by FOR and LET clauses. The RETURN clause is used to specify an element structure and to construct the result elements in the specified structure. The optional ORDER BY clause determines the order of the result elements.

A basic XQuery query can be formulated with a simplified FLWOR expression:

```
FOR x1 in p1, ..., xn in pn
WHERE c
RETURN s
```

In the FOR clause, iteration variables $x_1, \cdots, x_n$ are defined over the path expressions $p_1, \cdots, p_n$. In the WHERE clause, the expression c specifies conditions for qualified binding-tuples generated by the iteration variables. Some conditions may be included in p_i to select tuples iterated by the variable x_i. In the RETURN clause, the return structure is specified by the expression s. A nested FLWOR expression can be included in s to specify a *subquery* over sub-elements.

4.1 The Algorithm

Input. A basic XQuery query Q_{xquery} against an XML schema *Sch_XML* which is generated from the underlying relational schema *Sch*.

Output. A corresponding SQL query Q_{sql} against the relational schema *Sch*.
Step 1: *make Q_{xquery} canonical* - Let p_i defined in the FOR clause be the form of $/step_{i1}/\cdots/step_{ik}$. We check whether there is a test condition, say c_{ij} in $step_{ij}$ of p_i from left to right. If there is such a step, let $step_{ij}$ be the form of $l_{ij}[c_{ij}]$, then we add an extra iteration variable y_{ij} in the FOR clause which is defined over the path expression $/l_{i1}/\cdots/l_{ij}$, and move the condition c_{ij} to the WHERE clause, each element or attribute in c_{ij} is prefixed with $\$y_{ij}/$.
Step 2: *identify all relations* - After Step 1, each p_i in the FOR clause is now in the form of $/l_{i1}/\cdots/l_{ik}$, where $l_{ij}(1 \leq j \leq k)$ is an element in *Sch_XML*. Usually p_i corresponds to a relation in *Sch* (l_{ik} matches the name of a relation schema in *Sch*). The matched relation name l_{ik} is put in the FROM clause of Q_{sql} followed by the iteration variable x_i served as a tuple variable for relation l_{ik}. If there is an iteration variable, say x_j, appears in p_i, replace the occurrence of x_j with p_j. Once both relations, say R_i and R_j, represented by p_i and p_j respectively are identified, a link from R_i to R_j is added in a temporary list *LINK*. If there are nested FLWOR expressions defined in RETURN clause, the relation identification process is applied recursively to the FOR clause of the nested FLWOR expressions.
Step 3: *identify all target attributes for each identified relation* - All target attributes of Q_{sql} appear in the RETURN clause. For each leaf element (in the form of $\$x_i/t$) or attribute (in the form of $\$x_i/@t$) defined in s of the RETURN clause, replace it with a relation attribute in the form of $x_i.t$. Each identified target attribute is put in the SELECT clause of Q_{sql}. If there are nested FLWOR expressions defined in RETURN clause, the target attribute identification process is applied recursively to the RETURN clause of the nested FLWOR expressions.
Step 4: *identify conditions* - Replace each element (in the form of $\$x_i/t$) or attribute (in the form of $\$x_i/@t$) in the WHERE clause of Q_{xquery}, then move all conditions to the WHERE clause of Q_{sql} with a relation attribute in the form of $x_i.t$. If there are nested FLWOR expressions defined in RETURN clause, the condition identification process is applied recursively to the WHERE clause of the nested FLWOR expressions.
Step 5: *set the links between iteration variables* - If there is any link put in the temporary list LINK, then for each link from R_i to R_j, create a join condition between the foreign key attributes of R_i and the corresponding primary key attributes of R_j and ANDed to the other conditions of the WHERE clause of Q_{sql}.

4.2 An Example

Suppose we want to find all departments which have office in Adelaide and we want to list the name of those departments as well as the name and salary of all employees who live in Adelaide and work in those departments. The XQuery query for this request can be formulated as follows:

```
FOR $d in /Dept, $e in $d/Employee, $l in $d/DeptLoc
WHERE $l/city = "Adelaide" and
```

```
      $e/city = "Adelaide" and
      $e/@dno = $d/@dno
RETURN
      <Dept>
        <dname> $d/dname </dname>
        <employees>
           <name> $e/name </name>
           <salary> $e/salary </salary>
        </employees>
      </Dept>
```

Given this query as an input, the following SQL query will be generated:

```
SELECT d.dname, e.name, e.salary
FROM Dept d, Employee e, DeptLoc l
WHERE l.city = "Adelaide" and
      e.city = "Adelaide" and
      e.dno = d.dno and
      l.dno = d.dno
```

5 XML Documents Generation

As seen from the query translation algorithm and example introduced in the previous section, the translated SQL query takes all leaf elements or attributes defined in an XQuery query RETURN clause and output them in a flat relation. However, users may require a nested result structure such as the RETURN structure defined in the example XQuery query. Therefore, when we generate the XML result documents from the translated SQL query result relations, we need to restructure the flat result relation by a *grouping* operator [9] or a *nest* operator for NF^2 relations, then convert it into XML documents.

Similar to SQL *GROUP BY* clause, the grouping operator divides a set or list of tuples into groups according to key attributes. For instance, suppose the translated SQL query generated from the example XQuery query returns the following result relation as shown in Table 1. After we apply grouping on the relation using dname as the key, we have the nested relation as shown in Table 2 which can be easily converted to the result XML document as specified in the example XQuery query.

Table 1. Flat Relation Example

dname	name	salary
development	Smith, John	70,000
marketing	Mason, Lisa	60,000
development	Leung, Mary	50,000
marketing	Lee, Robert	80,000
development	Chen, Helen	70,000

Table 2. Nested Relation Example

dname	name	salary
development	Smith, John	70,000
	Leung, Mary	50,000
	Chen, Helen	70,000
marketing	Mason, Lisa	60,000
	Lee, Robert	80,000

6 Conclusion and Future Work

This paper introduced the architecture and components of a virtual XML database engine VXE-R. VXE-R presents a normalized XML schema which preserves integrity constraints defined in the underlying relational database schema to users for queries. Schema mapping rules from relational to XML Schema were discussed. The Query translation algorithm for translating basic XQuery queries to corresponding SQL queries was presented. The main idea of XML document generation from the SQL query results was also discussed.

We believe that VXE-R is effective and practical for accessing relational databases via XML. In the future, we will build a prototype for VXE-R. We will also examine the mapping rules using our formal study of the mapping from relational database schema to XML schema in terms of functional dependencies and multi-valued dependencies [12,13], and investigate the query translation of complex XQuery queries and complex result XML document generation.

References

1. S. Abiteboul, P. Buneman, and D. Suciu, *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann Publishers, 2000.
2. C. Baru. XViews: Xml views of relational schemas. In *Proceedings of DEXA Workshop,* pages 700–705, 1999.
3. S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. *XQuery 1.0: An XML Query Language,* April 2002. W3C Working Draft, http://www.w3.org/TR/2002/WD-xquery-20020430/.
4. T. Bray, J. Paoli, C. Sperberg-McQueen, and E. Maler. *Extensible Markup Language (XML) 1.0 (Second Edition),* October 2000. W3C Recommendation, http://www.w3.org/TR/REC-xml.
5. M. Carey, J. Kiernan, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Middleware for publishing object-relational data as xml documents. In *Proceedings of VLDB,* pages 646–648, 2000.
6. D. Fallside. *XML Schema Part 0: Primer,* May 2001. W3C Recommendation, http://www.w3.org/TR/xmlschema-0/.
7. M. Fernandez, W. Tan, and D. Suciu. SilkRoute: Trading between relations and xml. In *Proceedings of WWW,* pages 723–725, 2000.
8. D. Lee, M. Mani, F. Chiu, and W. Chu. Nesting-based relational-to-xml schema translation. In *Proceedings of the WebDB,* pages 61–66, 2001.
9. J. Liu and C. Liu. A declarative way of extracting xml data in xsl. In *Proceedings of ADBIS,* pages 374–387, September 2002.
10. J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. Querying xml views of relational data. In *Proceedings of VLDB,* pages 261–270, 2001.
11. J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as xml documents. In *Proceedings of VLDB,* pages 65–76, 2000.
12. M. Vincent, J. Liu, and C. Liu. A redundancy free 4NF for XML. In *Proceedings of XSYM,* September 2003.
13. M. Vincent, J. Liu, and C. Liu. Redundancy free mapping from relations to xml. In *Proceedings of WAIM,* August 2003.

Cursor Management for XML Data

Ning Li, Joshua Hui, Hui-I Hsiao, and Parag Tijare

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120, USA
{ningli,jhui,hhsiao,parag}@almaden.ibm.com

Abstract. In a relational database system, cursors provide a mechanism for traversal of a set of query results. The existing cursor defined in the SQL or JDBC, which is targeted for navigating results in the relational form, is inadequate for traversing or fetching XML query results. In this paper, we propose a mechanism for efficient and flexible cursor operations on XML data sets stored in an XML repository based on a relational database system. We first define a cursor interface for traversal of XML query result, which also provides a positioned update function. To demonstrate the feasibility of the cursor interface, we then propose and examine three different implementations in a relational database system: multi-cursor, outer union, and hybrid. Our experiments using XMach [23] benchmark data sets show that the hybrid approach has the best performance among the three in a mixed workload with both sequential and structure-aware cursor traversals.

1 Introduction

XQuery [21] is rapidly emerging as the standard for querying XML data. Currently, the XQuery specification does not address the binding of the result set returned from an XML query. Such binding is crucial for most applications because XML applications need a flexible mechanism to traverse and/or fetch query results without the need for materializing a complete result set in the application space.

In a relational database system, a cursor is a mechanism that allows an application to step through a set of SQL query results. In the cursor operations defined in the SQL language or JDBC interface, the navigation is limited to forward and backward movements in a single dimension and the fetch unit is a row. Navigating through XML data or XML results, however, requires moving a cursor in multiple dimensions and the fetch unit would normally be a tree or a branch. Therefore, SQL and JDBC cursors are inadequate for navigating through or fetching result from an XML data set. The Document Object Model (DOM) [17] interface has been proposed in earlier works to navigate XML result sets. This approach works well, however, only when an entire XML result set is materialized in main-memory or the XML data is managed by a main-memory database system. Navigating materialized result sets could potentially compromise the consistency and integrity provided by the database systems. In addition, always materializing a complete result set has a very negative performance impact because, in many cases, applications only need to retrieve a sub-set or small sub-set of an XML query result. For example, a user search query returns a list of

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 52–69, 2003.

papers. The user may want to browse the table of content or abstract before retrieving the rest of the paper. Always materializing a whole paper is not needed and is a waste of resource. Therefore, a mechanism that provides materialization on-demand is highly desirable. The ROLEX system [3] provides such a solution by applying a virtual DOM interface. ROLEX works fine when applications are running in the same memory space as the ROLEX system. In a multi-tier application environment where application programs and the data set reside in different computer systems, it will not function as well. In addition, result navigation in ROLEX is specified by a declarative view query, which is not as flexible or powerful as a full-fledged cursor interface.

In this paper, we propose a mechanism for efficient and flexible cursor operations on XML data sets stored in an XML repository based on a relational database system. In particular, we make the following three contributions:

1 We propose a cursor definition for XML data or query result that includes a set of cursor operations for traversing and fetching XML data. The cursor interface allows an application to step through the XML query result in different units of granularities such as a node, a sub-tree or a whole document. Our cursor definition also supports a positioned update function. Due to space limitation, the update function will not be covered in this paper.
2 We design and study several alternatives for supporting XML cursors in an XML repository based on a relational database system. Our design exploits technologies from [10][12][13][15] whenever applicable and focuses on meeting the challenges of providing an efficient mechanism for cursor and result navigation.
3 We then provide a performance analysis of three different implementation approaches: multi-cursor, outer union, and hybrid.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 presents the details of the cursor definition. Section 4 describes the various techniques for processing XML cursor statements. It focuses on algorithms of query processing and result navigation. Performance comparisons of the various approaches are presented in Section 5. Finally, Section 6 concludes the paper and describes directions for future work.

2 Related Work

There are two aspects of our XML cursor management work, namely a flexible cursor interface definition for navigating XML data, and a robust implementation of such an interface.

In the area of cursor interface definitions, cursors are defined in SQL [4] and JDBC [6] for relational data. Such cursor interfaces are inadequate for traversing or fetching multi-dimensional XML query results. DOM [17] defines a general interface to manipulate XML documents. However, it does not address the aspect of persistent storage. In addition, our SQL/JDBC-like XML cursor interface is preferable since the relational database systems are the dominant data management systems currently and, in most cases, XML-based applications will need to interoperate with existing SQL centric applications. Oracle's JXQI [9], the Java XQuery API, also defines a SQL/JDBC-like cursor interface but with functionalities limited to one dimension cursor movement.

In order to provide cursor management for XML data on top of a relational database system, the system first requires the basic XML data management capabilities such as storage and query. Several approaches have been proposed and published [2][5][7][10][13] in this area. The work described in [11][12], on the other hand, studied how to query XML views of relational data and to publish relational data as XML documents. In addition, [16] studied storing and querying ordered XML data in a relational database system while [15] proposed a method for performing searched update of XML data. None of these works addresses XML cursor support and, to the best of our knowledge, no previous work has addressed navigation and update of persistent XML data through SQL-like cursors.

3 XML Cursor Interface

We propose a cursor interface definition that has similar syntax as the cursor definitions of embedded SQL but with much richer functionalities. Our proposal takes into consideration the multi-dimensional nature of the XML data model, which is much richer than the relational data model. In the following, we first present the data model and then give a detailed description of the proposed cursor definition.

3.1 Data Model

Our XML cursor definition is targeted for traversing the result set of an XQuery query expression, thus the data model that we adopt is similar to the XQuery data model but simpler. The XQuery and XPath Data Model [22] views XML documents as trees of nodes. An XQuery query result is an ordered sequence of zero or more nodes or atomic values. Because each node in the sequence represents the root of a tree in the node hierarchy, the result could be treated as a result tree sequence. For simplicity, we assume that the root nodes are all element nodes. We further simplify the model by attaching attributes and text to their element nodes, which means there will be no separate nodes for attributes or text. Consequently, attributes and text of the current element node can be retrieved without moving the cursor to a different node.

3.2 Cursor Definition

In the following, we first define the XML cursor interface, as an extension of XQuery, which includes cursor declaration, open and close of a cursor, cursor navigation, and result retrieval via a cursor. We then discuss the cursor position for various cursor movement operations.

Declare a Cursor

```
DECLARE CURSOR <cursor-name> [<sensitivity>] [SCROLL]
       FOR <xquery-expr> FOR <updatability> [<optimization>]
<sensitivity> : INSENSITIVE | SENSITIVE
<updatability>: READ ONLY | UPDATE
<optimization>: OPTIMIZE FOR MAX DEPTH <n>
```

The above statement defines a cursor and its properties. <xquery-expr> specifies the XQuery query expression whose result the cursor is bound to. <sensitivity> has similar semantics as that of a SQL cursor. When SCROLL is not specified, the cursor movement operation is limited to only NextNode. All supported cursor movement operations are described in later in this section. UPDATE can be specified only if each node of the XML query result has a one-to-one mapping to the backend persistent storage. <optimization> lets users provide hints of the usage pattern for better performance. A useful hint is the maximum depth of the result trees that the application will access.

Open and close the Cursor

```
OPEN CURSOR <cursor-name>
CLOSE CURSOR <cursor-name>
```

The open statement executes the XQuery query specified in the cursor's DECLARE statement. The result set will be identified, but may or may not be materialized. The close statement releases any resource associated with the cursor.

Fetch the Next Unit of Data

```
FETCH FROM CURSOR <cursor-name>
       <axis-operation> [ <fetch-unit> ][ <content-unit> ]
INTO (STRING <host-variable> | DOM <host-variable>)
<axis-operation>: NextTree | PreviousTree
       | NextNode [EXCLUDING Descendant]
       | PreviousNode [EXCLUDING Ancestor]
       | ChildNode | ParentNode | NextSibling
       | PreviousSibling
<fetch-unit>: INCLUDING SUBTREE [OF MAX DEPTH <n>]
<content-unit>: WITH (TEXT | ATTRIBUTES) ONLY
```

This statement allows a cursor to move to a desired position from its current position and to specify the content to be retrieved into a host application. <axis-operation> specifies the types of cursor movement and there are two of them. The first type is *sequential*, which follows the document order. NextNode and PreviousNode fall into this category. The other type is *structure-aware*, which includes the rest of the operations. The destination position is determined by the tree structures of the XML result. The structure-awareness of the cursor movement matches the multi-dimensional nature of XML data and is very important for many applications to step through the XQuery query result. With the supported <axis-operation>, NextTree/PreviousTree moves the cursor to the root node of the next/previous tree in the result tree sequence while NextNode/PreviousNode moves to the next/previous node in the current tree in the document order. If the cursor is not scrollable, then only NextNode is enabled and no node in the result sequence can be skipped.

<fetch-unit> specifies the unit of the data to be fetched. It could be the current node, the sub-tree which rooted at the current node, or such a sub-tree but only retrieving nodes up to depth <n>. Because attributes and text are attached to their element node in our simplified data model, <content-unit> allows user to specify whether to retrieve the text only, the attributes only, or the element node with both text and attributes. Also, the data can be fetched either as a string or in DOM representation.

Position the Cursor

```
SAVE CURRENT POSITION OF CURSOR <cursor-name> IN <host-variable>
SET CURRENT POSITION OF CURSOR <cursor-name> TO <host-variable>
```

These statements are used to bookmark (save) a given cursor position or to set the cursor to a previously saved position. This enable an application to *remember* some or all visited nodes and later jump back to any of the saved node, if needed.

3.3 Cursor Position

When a cursor is first opened, it is placed right before the root node of the first tree in the result sequence. A NextNode or a NextTree operation will bring the cursor to the root node of the first tree. After that, a cursor position is always on a node. In case of a DELETE operation, the cursor will be positioned on the prior node in the document order that is not deleted.

4 Query Processing & Navigation Techniques

In this section, we describe three approaches to support the cursor interface for XML data stored in relational databases. The major technical challenges in supporting cursor operations in XQuery are: (1) how to translate an XQuery into one or more SQL queries that will facilitate result navigation and retrieval, and (2) how to implement navigation capabilities given the SQL queries generated.

Before diving into the details of the approaches, we first describe the mapping of XML data into relational tables in our system. For simplicity, we use a direct mapping, which maps each element to a relational table with the following columns:

- one column for each attribute of the element
- a column to store the element content if content is allowed
- two columns, id and parentid, which capture the parent-child relationship of the element hierarchy; the parentid column is the foreign key of the id column of the element's parent table
- a column, docid, which stores the id of the document the element belongs to

A similar method is described in [13]. As pointed out there, a direct mapping may lead to excessive fragmentation. Various inlining and outlining techniques are described in [13] to reduce fragmentation and the number of SQL joins needed to evaluate path expressions.

The mapping scheme used will determine the top level SQL query that is generated in all of our cursor implementation approaches. Also, the number of JDBC cursor movements may be different for different mapping schemes, e.g. if the inlining technique is used, no extra SQL cursor movement is required to move to the next child element. On the other hand, if the outlining technique is used, it would require another table join to get all the parts of the same element. Since the comparison of different mapping schemes is not a goal of our work, we have chosen a single mapping scheme (i.e. the direct mapping scheme) to carry out the performance comparison of our cursor implementation approaches. The cursor implementation approaches themselves, are general and the relative performance among them will not change if other mapping schemes are used.

Several methods to support the ordered XML data model in the unordered relational model are proposed in [16]. Among the three schemes suggested, we choose the global order encoding scheme rather than the Dewey order encoding scheme mainly because the global order encoding method provides the best query performance. In addition, we accommodate its weakness for update by leaving gaps in the order numbers [8] to reduce the frequency of renumbering. When renumbering occurs, in most cases, only a small number of nodes are required to be touched. On the other hand, Dewey order encoding scheme always requires the whole subtree to be renumbered. Figure 1shows an example of such case. An extra column, "*iorder*" is used to record the global element order within each XML document.

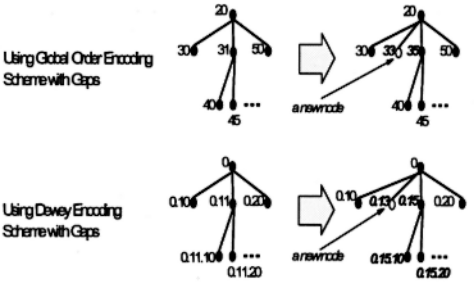


Fig. 1. A renumbering example

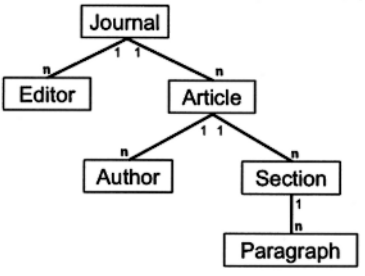


Fig. 2. Graph Representation of an XML Schema

In the following we present the query processing and the result navigation algorithms for three approaches in detail by means of concrete examples.

4.1 The Multi-cursor Approach

A straightforward approach is to open a series of database cursors on the relations that map to the result elements. We term this the *multi-cursor (MC) approach.*

4.1.1 Query Translation

This approach translates an XQuery query into a set of top-level SQL queries and constructs a list of parameterized SQL queries, one for each relation an element type in the result mapped to. An XQuery can be translated into more than one top-level SQL queries because an XQuery can construct results in different types. A query like "/Book/*" is such an example. So there will be multiple top-level queries, one for each child of the Book element. Each top-level query, when executed, produces a set of relational tuples as the root nodes of the result trees. Each of the parameterized queries, when executed with an id of a node in the result, produces the node's child nodes of one element type in the relational form. As an example, consider the graph representation of an XML schema shown in Figure 2. The XQuery query in Figure 3 will be translated into the top-level query and the list of parameterized queries shown in the same figure. In this case, all the translated queries are sorted according to the document order. Because the XML data model does not preserve order across documents, the sort is not required for the top-level queries if the query results are in the document node level.

For a given XML query, the query processor generates queries that select all the attributes of the elements as well as the id's needed for result navigation.

User Query: /Journal[./Editor/@name="James Bond"]/Article

Translated Queries:

```
1.  SELECT DISTINCT Article.docid, Article.id,
           Article.title, Article.page
    FROM Journal, Editor, Article
    WHERE (Journal.id = Editor.parentid)
    AND (Editor.name = "James Bond")
    AND (Journal.id = Article.parentid)
    ORDER BY Article.docid, Article.iorder

2.  SELECT id, name, affiliation
    FROM Author
    WHERE parentid = ? and docid = ?
    ORDER BY iorder

3.  SELECT id, number, title
    FROM Section
    WHERE parentid = ? and docid = ?
    ORDER BY iorder

4.  SELECT id, number, note
    FROM Paragraph
    WHERE parentid = ? and docid = ?
    ORDER BY iorder
```

Fig. 3. XQuery Query Example

4.1.2 Result Navigation

Given all the queries generated in the query translation phase, multiple cursors are opened by executing these queries to implement the navigation functionalities. Initially, the top-level queries are executed and their SQL result sets are pushed into a stack; and the rest of the parameterized queries are prepared. As the XML cursor moves to different nodes in the result, prepared queries corresponding to the current navigation level are executed and SQL result sets are pushed into the stack if the navigation is going down the result tree or popped out of the stack if going up the result tree. To make sure the proper order of the node is returned, firstly each query result has to be sorted according to the document order. Then among the current tuples of the current SQL result sets, we return the one with the minimum order value if the operation is a forward traversal. This also applies to the top-level queries to determine which tuple to be returned if there are multiple top-level queries and the top result nodes are not document root nodes. At any time, the tuples of the result sets on top of the stack corresponds to the current cursor level, and all its ancestors are on the stack, in sequence. The Appendix contains an example explaining the mechanism in detail.

4.2 The Outer Union Approach

In the multi-cursor (MC) approach, multiple SQL cursors are opened for an XML cursor. When an application moves an XML cursor to an element (table) of a deeper level in the result set, the corresponding set of queries is executed and the same num-

ber of SQL cursors is opened. This consumes a lot of database resources and adds extra query processing time during navigation. The *outer union (OU) approach* reduces the number of SQL cursors opened by opening a single SQL cursor for each XML cursor. It adopts a modified version of the "outer union" method, first proposed in [12], in the query processing phase to return an entire XML result set.

4.2.1 Query Translation

Among the many varieties proposed in [12], the Sorted Outer Union method is chosen because it structures relational tuples in the same order needed to appear in the XML result. Particularly, a tuple represents one XML node in the result if the Node Sorted Outer Union is used. However, the method was presented in the context of publishing relational data in XML form, which cannot be applied directly to support our cursor interface. It needs to be extended to incorporate the processing of XQuery in order to return the result of the query. Moreover, instead of sorted by the ID columns, the translated query is sorted by the document id plus two order columns. The first order column is the global order column which captures the order of the result set sequence. All the descendant elements share the same global order of their root nodes. The second one is the local order column which records the order of all the descendants of an XML node in the result sequence. An example of a translated outer union query for the same XQuery in Section 4.1.1 is given in the Appendix.

As expected, the result of this approach is the best for sequential cursor movements such as NextNode and PreviousNode. However, for operations like, NextSibling, a number of tuples need to be examined before the destination is reached. This may result in significant overhead in cursor movements. If the database system has a way, using the distance information, to jump directly to the destination node, it will avoid the need of fetching continuously until the destination node is reached. Thus a mechanism to compute the distance as part of the query translation would be very helpful. We use the DB2 OLAP partition function [14] to calculate the distance information in the query. An example using the partition function is also provided in the Appendix.

4.2.2 Result Navigation

With the XML result as relational tuples generated by executing the outer union query described above, supporting the proposed navigation functionalities is quite straightforward. A sequential movement to the next or previous XML node corresponds to one next or previous operation of the database cursor; on the other hand, multiple database cursor operations are needed for a non-sequential movement of an XML cursor unless the distance information is available.

Figure 4 shows the pseudo code for result navigation with the OU approach. Version 1 of the NextSibling does not use the distance information. The second version uses the distance information and "relativeNode()", which is a helper function that utilizes the "relative" method of SQL cursor Also, fCurrentNode is the current XML node constructed from the current tuple immediately before it is used while fSavedNode is the XML node constructed from the then current tuple at the very beginning of the axis operation Functions like "getDescendantCount()" are used to retrieve the distance information when available.

```
nextNode():
if (fResultSet.next())
   return true;

childNode():
if (nextNode())
   if (fCurrentNode.isChildOf(fSavedNode))
      return true;

Version 1. nextSibling() (without using the distance informatio):
while (nextNode()) {
   if (fCurrentNode.isSiblingOf(fSavedNode))
      return true;

   if (!fCurrentNode.isDescendantOf(
       fSaveNode.pidIndex,fSaveNode.pid))
      break;  // fail
}

Version 2. nextSibling() (using distance information):
int distance = fCurrentNode.getDescendantCount()+1;
if (relativeNode(distance))
    if (fCurrentNode.isSiblingOf(fSavedNode))
       return true;
```

Fig. 4. Pseudo Code for Result Navigation

4.3 The Hybrid Approach

While the outer union approach is well suited for sequential cursor movements, it is not suitable for the structure-aware navigation of XML cursors such as NextSibling. On the other hand, although the multi-cursor approach is usually better for structure-aware cursor movements, there are cases where multiple SQL queries have to be executed for a single navigation operation. ChildNode is one of the examples.

In order to achieve good performance for structure-aware traversal in all cases without using excess resources, we propose a hybrid (HB) approach by combining the MC and OU techniques. This approach applies the outer union technique to return all the child nodes (could be of different types) of a parent node using one SQL query, while opens much fewer SQL cursors, one for each level to the current depth, than the MC approach.

4.3.1 Query Translation

Similar to the multi-cursor approach, the hybrid approach translates an XQuery query into a top-level SQL query and a set of parameterized SQL queries. The translation to the top-level query is the same as in the multi-cursor approach, but the construction of the parameterized queries is different in the way that it utilizes the outer union technology to generate one parameterized query for each non-leaf element type which produces all the child nodes of a parent. These child nodes are produced in the document order.

The queries constructed by the PH approach are similar to the sorted outer union queries described in Section 4.2.1. The differences are: first, for a parent element type, only the ids and the attributes of its child types are included; second, a “where”

clause is added to parameterize the id of a parent node. For the same XQuery query in Section 4.1.1, the set of parameterized queries generated by this approach is shown in Figure 5.

```
For Article (two child types):
WITH
Author_t(id2, id3, a3, a4, a5, a6, iorder)
AS ( SELECT id, null, name, affiliation, null, null, iorder
     FROM Author
     WHERE Author.parentid = ? and Author.docid = ? ),
Section_t(id2, id3, a3, a4, a5, a6, iorder)
AS ( SELECT null, id, null, null, number, title, iorder
     FROM Section
     WHERE Section.parentid = ? and Section.docid = ? ),
OuterUnion_t(id2, id3, a3, a4, a5, a6, iorder)
AS ( (SELECT * FROM Author_t)
     UNION ALL
     (SELECT * FROM Section_t) )
SELECT * FROM OuterUnion_t
ORDER BY iorder

For Section (only one child type):
SELECT id, number, note, iorder
FROM Paragraph
WHERE parentid = ?
ORDER BY iorder
```

Fig. 5. Translated Queries for HB

4.3.2 Result Navigation

With this approach, multiple cursors are opened and a stack for the SQL result sets is used to support the navigation functions as in the multi-cursor approach. Axis operations such as ChildNode and NextSibling now take advantage of the fact that all the children of a parent are in one result set as shown in Figure 6.

```
childNode():
ResultSet rs = getChildResultSet();
if (rs.first()) {
   fResultSetStack.push(rs);
   return true;
}

nextSibling():
ResultSet rs = fResultSetStack.top();
if (rs.next())
   return true;
```

Fig. 6. Result Navigation for HB

A cursor movement such as NextSibling is more efficient here because it corresponds to a single next operation of the database cursor. ChildNode is also more efficient because it only requires executing a query and a database next operation, eliminating the need of executing multiple queries as in the multi-cursor approach.

5 Performance Analysis

The goal of the performance analysis is to determine which approach described in Section 4 works best for what data and what *navigation patterns*. The update per-

formance is not presented here due to the space limitation. Our implementation was written in Java, which used JDBC to communicate with a backend IBM DB2 UDB V7.2 database. The experiments were carried out on a 1.8 GHz Pentium 4 computer with 512MB main memory running Windows 2000.

5.1 Data Set and Workload

We used the Xmach-1's [23] benchmark. 10,000 XML documents were generated with a total size of 150MB, conforming to a single schema shown in Figure 7. These documents vary in size (from 2 to 100KB) as well as in structure (the depth of element hierarchy).

The schema has a recursively defined *Section* element. There are several ways to map a recursively defined element into relational tables. In our experiment, we rename and create n tables for the first n levels of recursion and one table for the rest of it. We selected and ran four XPath queries from the Xmach-1 benchmark. The four queries capture several important characteristic of an XPath query, such as forward path navigation, context position and descendant-or-self queries. Table 1 shows the four queries and the total number of result nodes for each query. We built indexes on the id and the parent id columns. In addition, we also built the index on the "cs_id" attribute of "Section".

Table 1. Queries and Numbers of Result Nodes

	Query	# of nodes
Q1	/document[./chapter/section/@cs_id = "s170"]	16,879
Q2	/document[.//section/@cs_id = "s170"]	39,103
Q3	/document[./chapter/section[1]/@cs_id = "s170"]	9,579
Q4	/document/chapter/section/section/section[./section/@cs_id = "s50"]	2,080

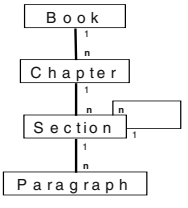


Fig. 7. Graph Representation of the XML Schema

```
ForwardStructuralAccess:
// visit the current node
if (xmlResultSet.childNode()) {
  do {
    forwardStructuralAccess(xmlResultSet);
  } while (xmlResultSet.nextSibling());
  xmlResultSet.parentNode();
}
```

Fig. 8. Pseudo Code for Forward Structural Traversal

We used two navigation patterns on the result data to evaluate the cursor movement performance of the various approaches, namely forward sequential access and forward structural (structure-aware) access. Forward sequential access calls the nextNode() operation repeatedly to fetch the nodes sequentially. It visits the nodes in the result set in the document order. Forward structural access, on the other hand, examines the result structure and uses the structure-aware operations such as childNode() and nextSibling(), to traverse the result set in a forward direction. The pseudo

code is shown in Figure 8. For all the experiments, the maximum depth of the query results is set to 5.

5.2 Performance Results

In the following figures, there are four groups of columns, one for each query in Table 1. For each group, there are four columns, illustrating the execution time of each of the four approaches described in Section 4. The total time includes the initial query execution time and the result navigation time. The query translation time is not included because it is always negligible compared to the time spent on executing the initial query. For MC and HB, the query execution time includes the time to execute the top-level query and the time to prepare the parameterized queries while the navigation time includes the executions of some parameterized queries as the cursor moves to different nodes in the result trees.

5.2.1 Forward Sequential Navigation

Figure 9 shows the performance of all the approaches for the forward sequential navigation pattern, where all the result nodes are visited. The vertical axis represents the time in seconds and the horizontal axis represents the four different approaches for each of the four queries. As shown in Figure 9, the navigation time of two outer union based approaches (OU and OU-D) is always better than that of MC and HB. This is because this pattern visits all the nodes sequentially in the document order and that is exactly how the result of the outer union query is structured. As a result, a nextNode() operation is simply translated to a single SQL Fetch Next operation. On the other hand, for the MC and HB approaches, the navigation time includes both the traversal as well as the execution of the parameterized queries as the cursor moves. Therefore, it takes much longer than that of the OU approaches.

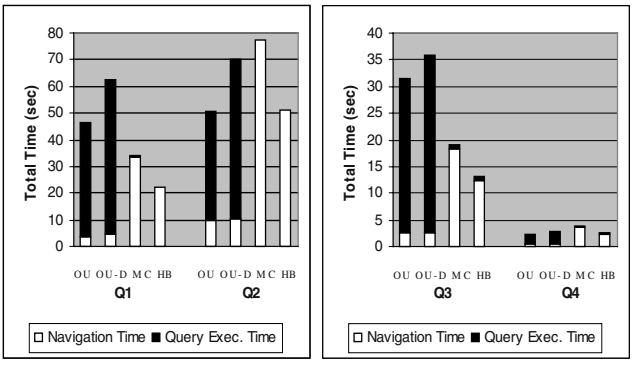


Fig. 9. Forward Sequential Navigation

It is not the case, however, when we also take into consideration of the initial query execution time. An outer union query is much more complicated than the top-level query generated in the non-OU approaches. This is because the outer union queries compute all the nodes in the result while the top-level queries only compute the root nodes. As a result, the total execution time of the two OU approaches is significantly worse than HB and MC for queries 1 and 3. For queries 2 and 4, OU and

HB perform equally well and both are about 30% better than either OU-D or MC. For OU and OU-D, the latter is always worse in this set of experiments because OU-D also calculates various distances. Between MC and HB, HB always performs better because it always executes fewer parameterized queries.

We also ran experiments where only 5% of the nodes in the query result are visited. For each of the four approaches, the query execution time remains the same while the navigation time is about 5% of the navigation time shown in Figure 9. In this case, the HB approach again provides the best performance overall and is now significantly better than OU even for queries 2 and 4.

5.2.2 Forward Structural Navigation

Figure 10 and Figure 11 show the performance results of the forward structural navigation pattern, where result nodes are accessed using structure-aware operations. In Figure 10, all the nodes in the result up to the given maximum traversal depths are accessed while only 5% of the result trees are visited in Figure 11.

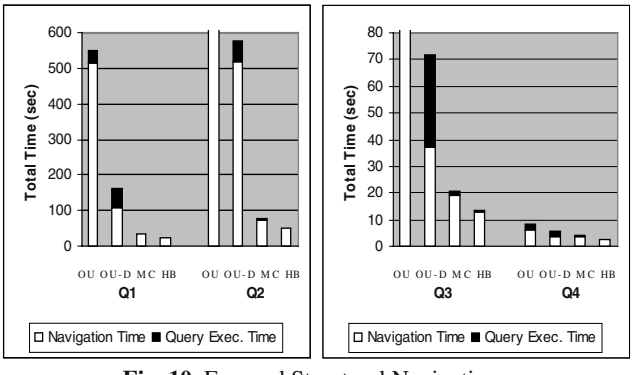


Fig. 10. Forward Structural Navigation

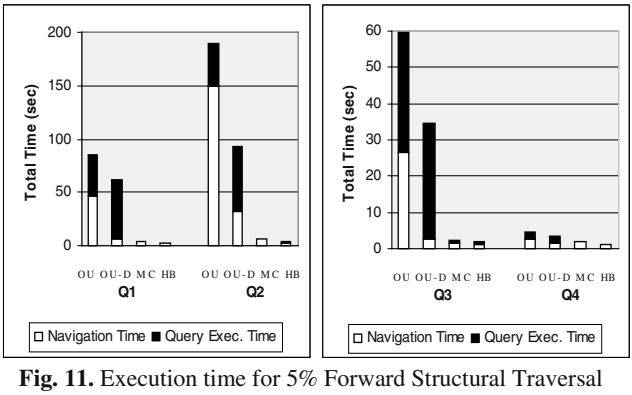


Fig. 11. Execution time for 5% Forward Structural Traversal

Figure 10 shows that the navigation time dominates the total execution time while the query execution time becomes almost negligible compared to the navigation time for the two OU approaches, which is a complete reversal of the result shown in Figure

9. This is because the OU approaches serialize the whole result tree and thus require fetching (or skipping) rows in-between in order to jump between parent and child nodes. This figure also shows that the use of distance information improves the performance by about two-third for OU-D. The reason is that, instead of fetching rows in-between, OU-D translates the traversal operation into a single SQL Fetch Relative call. However, OU-D is still worse than either MC or HB. This is because MC and HB partially preserve the tree structure of the result and thus require much less time to move the cursor between nodes. Overall, HB is again the best performer among all the approaches. Figure 11 shows the result of experiments where only 5% of the result trees were visited. As shown in the figure, the initial query execution time becomes a significant factor for the OU approaches when only 5% of the nodes are accessed. As a result, the performance of the OU approaches becomes even worst relative to the MC and HB approaches.

In both cases, the HB approach executes fewer parameterized queries than the MC approach since the sibling queries are being combined in the HB approach using the outer union technique. This results in less navigation time and makes the HB approach better than MC in total time. As shown in Figure 10 HB is about 30% better than MC. When only a small fraction of the nodes are visited, as the case in Figure 11 HB again performs better but the difference is not as significant.

5.2.3 Other Navigation Patterns

Besides the patterns discussed in the previous subsections, we also ran experiments on many other navigation patterns with mixtures of sequential and structure-aware operations. The results show that the HB approach is either among the best or is the best for all the experiments, depending on the operations used and the percentage of nodes visited. This finding is consistent with the findings for the sequential and the structure-aware access patterns discussed in the previous two subsections. Due to space limitation, we will not discuss them further here.

6 Conclusions and Future Work

XQuery is emerging as the standard for querying XML data. Navigating through XML query results without first materializing them is an important requirement for enterprise applications. To address this requirement, we proposed an XML cursor interface and described several approaches for supporting the interface in a relational database system.

To evaluate the performance of the proposed approaches, we ran a set of experiments using XMach benchmark data set. Experimental results show that the MC approach performs well in the structure-aware navigation patterns but consumes a lot of resources and performs poorly for the sequential access patterns. The OU approach, while providing good performance for sequential navigation pattern, performs very poorly for structural navigation pattern. Based on the findings of these experiments, we proposed the hybrid approach that combines the merits of both the MC and OU approaches. Our experiments using XMach benchmark data sets show that the hybrid approach has the best performance among the three, in both the sequential and the structure-aware cursor navigations as well as a mixed workload.

If application navigation patterns are predictable or known in advance, which may be possible in certain application environments, an adaptive hybrid approach could give an even better performance. Designing such an adaptive approach is a candidate for future research. Our XML cursor support mechanism assumes that XML data is stored in an XML repository based on a relational database system. Efficient query processing techniques for XML cursors for a native XML repository is another interesting research issue, which is also a topic of our future research.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. L. Wiener. The Lorel Query Language for Semistructured Data. *International Jounral on Digital Libraries 1997*.
[2] P. Bohannon, J. Freire, P. Roy, J. Simeon. From XML Schema to Relations: A Cost-based Approach to XML Storage. *ICDE 2002*.
[3] P. Bohannon, S. Ganguly, H. Korth, P. Narayan, P. Shenoy. Optimizing View Queries in ROLEX to Support Navigable result Trees. *VLDB conference 2002*.
[4] S. Cannan, G. Otten, SQL: The Standard Handbook. *McGraw-Hill*
[5] A. Deutsch, M. Fernandez, D. Suciu. Storing Semistructured Data with STORED. *SIGMOD 1999*.
[6] JDBC API 2.0 Specification, http://java.sun.com/products/jdbc/
[7] D. Florescu, D. Kossmann. Storing and Querying XML Data using an RDBMS. *IEEE Data Engineering Bulletin 2001*.
[8] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. *VLDB, 2001*.
[9] Oracle's JXQI – the Java XQuery API: http://otn.oracle.com/sample_code/tech/xml/xmldb/jxqi.html
[10] J. Shanmugasundaram et al. A General Technique for Querying XML Documents using a Relational Database System. *SIGMOD Record 2001*.
[11] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, J. Funderburk. Querying XML Views of Relational Data. *VLDB 2001*.
[12] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, B. Reinwald. Efficiently Publishing Relational Data as XML Documents. *VLDB 2000*.
[13] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. *VLDB 1999*.
[14] SQL Reference Volume 1 & 2, IBM DB2 Universal Database, Version 7.2.
[15] I. Tatarinov, Z. G. Ives, A. Y. Halevy, D. S. Weld. Updating XML. *SIGMOD 2001*.
[16] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. *SIGMOD 2002*.
[17] World Wide Web Consortium. Document Object Model (DOM) Core Specifications. W3C Recommendations.
[18] WWW Consortium. Extensible Markup Language (XML). W3C Recommendation.
[19] World Wide Web Consortium. XML Path Language (XPath). W3C Recommendation.
[20] World Wide Web Consortium. XML Schema. W3C Recommendation.
[21] World Wide Web Consortium. XQuery: An XML Query Language. W3C Working Draft.
[22] World Wide Web Consortium. XQuery and XPath Data Model. W3C Working Draft.
[23] XMach-1: A Benchmark for XML Data Management. http:// dbs.uni-leipzig.de/en/projekte/XML/XmlBenchmarking.html
[24] OMG IDL, ISO International Standard Number 14750, http://www.omg.org/cgi-bin/doc?formal/02-06-07

7 Appendix

The appendix includes three more examples of the techniques described in Section 4. The first is a result navigation example of the multi-cursor approach in Section 4.1.2. The second is a query translation example of the outer union approach in Section 4.2.1. And the third is a query example to compute distances between nodes. The technique can be applied in the outer union approach in Section 4.2.1.

7.1 A Result Navigation Example for MC

In this example, the multi-cursor approach is used to navigate the result of the XPath query at the top of Figure 3. Assume a cursor is currently positioned on the XML node corresponding to the tuple t1 in the figure below.

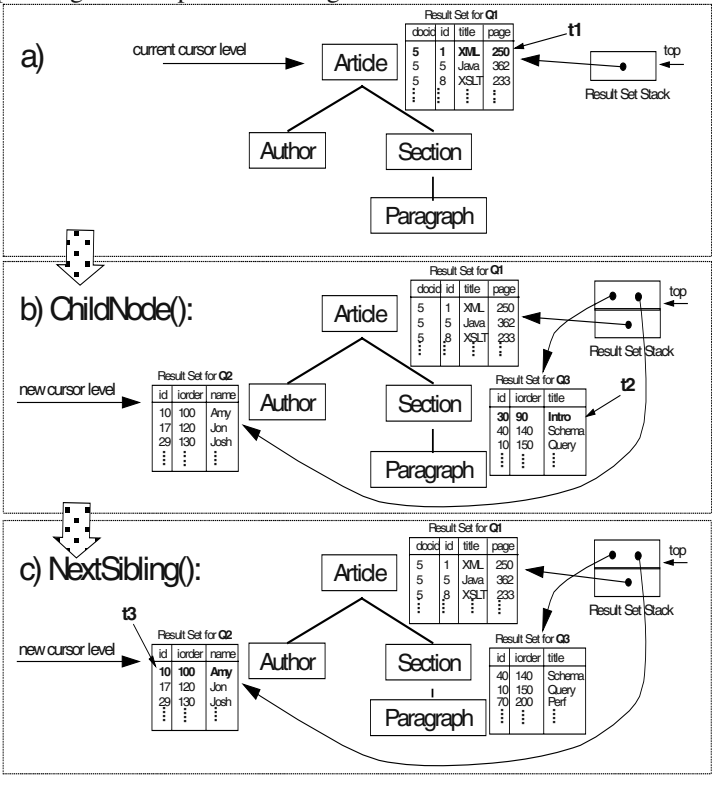


Fig. 12. A Multi-Cursor Example

Now a ChildNode operation is issued. The cursor position moves to the child level of the Article element. Since the XML schema of this document allows both Author and Section elements to be placed in any order, the next logical XML node can come from either the result set of Q2 or Q3 in Figure 3. So both queries are executed. Since each result set is already sorted by the order column, the next tuple of each result set is examined and the one with the least order value is returned. In this case, it is the

tuple t2. If a NextSibling operation is issued, since the cursor stays in the same level, tuples are continued to be fetched from the current result sets at the top of stack. In this case, the next XML node will be the tuple t3.

7.2 A Query Translation Example for OU

The following figure is the translated SQL query of the same XPath query in Figure 3 using the outer union approach.

```
WITH
Article_t(docid, id1, id2, id3, id4, g_order, l_order, a1, a2, a3, a4, a5, a6, a7, a8)
AS ( SELECT DISTINCT Article.docid, Article.id, null, null, null,
           Article.iorder as g_order, Article.iorder as l_order,
           Article.title, Article.page, null, null, null, null, null
     FROM Journal, Editor, Article
     WHERE (Journal.id = Editor.parentid)
     AND (Editor.name = "James Bond")
     AND (Journal.id = Article.parentid)),
Author_t(docid, id1, id2, id3, id4, g_order, l_order, a1, a2, a3, a4, a5, a6, a7, a8)
AS ( SELECT Article_t.docid, id1, id, null, null, g_order, Author.iorder as l_order,
           null, null, name, affiliation, null, null, null, null
     FROM Article_t, Author
     WHERE Article_t.id1 = Author.parentid ),
Section_t(docid, id1, id2, id3, id4, g_order, l_order, a1, a2, a3, a4, a5, a6, a7, a8)
AS ( SELECT Article_t.docid, id1, null, id, null, g_order, Section.iorder as l_order,
           null, null, null, null, number, title, null, null
     FROM Article_t, Section
     WHERE Article_t.id1 = Section.parentid ),
Paragraph_t(docid, id1, id2, id3, id4, g_order, l_order, a1, a2, a3, a4, a5, a6,
           a7, a8)
AS ( SELECT Section_t.docid, id1, null, id3, id, g_order, Paragraph.iorder as l_order,
            null, null, null, null, null, null, number, note
     FROM Section_t, Paragraph
     WHERE Section_t.id3 = Paragraph.parentid ),
OuterUnionTable(docid, id1, id2, id3, id4, g_order, l_order,
               a1, a2, a3, a4, a5, a6, a7, a8)
AS ( (SELECT * FROM Article_t)
     UNION ALL
     (SELECT * FROM Author_t)
     UNION ALL
     (SELECT * FROM Section_t)
     UNION ALL
     (SELECT * FROM Paragraph_t) )
SELECT * FROM OuterUnion_t
ORDER BY docid, g_order, l_order;
```

Fig. 13. A Translated Outer Union Query

Figure 13 shows the translated outer union query for the same XQuery in Section 4.1.1. The first “select” statement selects the id and the attributes of the result root nodes. The next three “select” statements select ids and attributes for Author, Section and Paragraph, respectively. For each element type, attributes of all other types are padded with NULLs so that the entire XML result can fit into one relation.

7.3 A Query Example to Compute Distances between Nodes for OU

As mentioned in Section 4.2.1, computing distances between nodes will help improve the performance of the structure-aware operations for the outer union approach. We use the DB2 OLAP partition function [14] to calculate the distances in the query. For example, the SQL statement in Figure 14 shows the additional statements being added to the query shown in Figure 13 to calculate both the distance from a node to the root of the next tree in the result sequence and the number of descendants. The number of descendants equals the distance to the next node excluding descendants. With these two pieces of information, the OU approach is now able to move the cursor directly to the next sibling or the next tree root.

```
WITH
Article_t(element_type, id1, id2, ...)
AS ( SELECT DISTINCT "Article", Article.id, ... ),
...
OuterUnionTable(element_type, id1, id2, ...)
SELECT id1, id2, id3, id4, g_order, l_order,
       RANK() OVER(PARTITION BY id1, g_order
                   ORDER BY l_order DESC)
       AS to_next_tree,
       CASE
        WHEN element_type = "Article"
        THEN COUNT(*) OVER(PARTITION BY id1)
        WHEN element_type = "Section"
        THEN COUNT(*) OVER(PARTITION BY id1, id3)
        ELSE 1
       END - 1 AS descendant_count
FROM OuterUnionTable
ORDER BY id1, g_order, l_order;
```

Fig. 14. Calculating Distances

Besides these two pieces of distance information, the distances to the root of the current tree and to the parent node can be computed as well. The former is similar to the next tree calculation, but with a reverse sorting order on l_order and the latter equals to its rank among all the descendants within the same parent.

Three Cases for Query Decorrelation in XQuery

Norman May, Sven Helmer, and Guido Moerkotte

Universität Mannheim
D7, 27
Mannheim
Germany
{norman|helmer|moer}@pi3.informatik.uni-mannheim.de

Abstract. We present algebraic equivalences that allow to unnest nested algebraic expressions for order-preserving algebraic operators. We illustrate how these equivalences can be applied successfully to unnest nested queries given in the XQuery language. Measurements illustrate the performance gains possible our approach.

1 Introduction

With his seminal paper Kim opened the area of unnesting nested queries in the relational context [19]. Very quickly it became clear that enormous performance gains are possible by avoiding nested-loops evaluation of nested query blocks (as proposed in [1]) by unnesting them. Almost as quickly, the subtleties of unnesting became apparent. The first bugs in the original approach were detected — among them the famous count bug [20]. Retrospectively, we can summarize the problem areas as follows:

- Special cases like empty results lead easily to bugs like the count bug [20]. They have been corrected by different approaches [7,14,18,20,23,24].
- If the nested query contains grouping, special rules are needed to pull up grouping operators [4].
- Special care has to be taken for a correct duplicate treatment [16,21,26,28].

The main reason for the problems was that SQL lacked expressiveness and unnesting took place at the query language level. The most important construct needed for correctly unnesting queries are outer joins [7,14,18,23]. After their introduction into SQL and their usage for unnesting, reordering of outer joins became an important topic [3,13,27]. A unifying framework for different unnesting strategies for SQL can be found in [24].

With the advent of object-oriented databases and their query languages, unnesting once again attracted some attention [5,6,10,29–31]. In contrast to the relational unnesting strategies, which performed unnesting at the (extended) SQL source level, most researchers from the object-oriented area preferred to describe unnesting techniques at the algebraic level. They used algebras that allow nesting. Thus, algebraic expressions can be found in subscripts of algebraic operators. For example, a predicate of a selection or join operator could

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 70–84, 2003.

again contain algebraic operators. These algebras allow a straightforward representation of nested queries, and unnesting can then take place at the algebraic level. The main advantage of this approach is that unnesting rewrites can be described by algebraic equivalences for which rigorous correctness proofs could be delivered. Further, these equivalence-based unnesting techniques remain valid independently of the query language as long as queries remain expressible in the underlying algebra. For example, they can also be applied successfully to SQL. However, the algebras used for unnesting do not maintain order. Hence, they are only applicable to queries that do not have to retain order. Fegaras and Maier describe an alternative approach in which queries are translated into a monoid comprehension calculus representation [10]. The actual unnesting of the queries is done in terms of this calculus. A disadvantage of this approach is the need for another level of representation (in addition to the algebraic representation).

XQuery[1] is a query language that allows the user to specify whether to retain the order of input documents or not. If the result's order is relevant, the unnesting techniques from the object-oriented context cannot be applied.

Consequently, the area of unnesting nested queries was reopened for XQuery by Fegaras et al. [9] and Paparizos et al. [25]. Fegaras et al. focus on unnesting queries operating on streams. It is unclear to which extent order preservation is considered (e.g. on the algebraic level hash joins are used, whose implementation usually does not preserve order). Another publication by Fegaras et al. [8] describes the translation of XML-OQL into OQL, but is not concerned with unnesting. The approach by Paparizos et al. describes the introduction of a grouping operator for a nested query. However, their verbal description of this transformation is not rigorous and indeed not complete: one important restriction that guarantees correctness is missing. We will come back to this point when discussing our counterpart of their technique. To the best of our knowledge, no other paper discusses unnesting in the ordered context.

Within this paper we introduce several different unnesting strategies and discuss their application to different query types. All these techniques are described by means of algebraic equivalences which we proved to be correct in our technical report [22]. We also provide performance figures for every query execution plan demonstrating the significant speed-up gained by unnesting.

Our Unnesting Approach consists of three steps (in this paper, we focus on the third step, details on the first two steps can be found in [22]):

1. Normalization introduces additional **let** clauses for nested queries
2. **let** clauses are translated into map operations (χ) (see Sec. 2) with nested algebraic expressions representing the nested query
3. Unnesting equivalences pull up expressions nested in a χ operator.

The remainder of the paper is organized as follows. Section 2 briefly motivates and defines our algebra. In Section 3, we introduce equivalences used for

[1] http://www.w3.org/XML/Query

unnesting queries via exemplary XQuery queries taken from the XQuery use-case document[2]. Section 4 concludes the paper.

2 Notation and Algebra

Our NAL-algebra extends the SAL-Algebra [2] developed by Beeri and Tzaban. SAL is the order-preserving counterpart of the algebra used in [5,6] extended to handle semistructured data. Other algebras have been proposed (see [22]), but we omit this discussion because this paper focuses on unnesting.

Our algebra works on sequences of sets of variable bindings, i.e. sequences of tuples where every attribute corresponds to a variable. We allow nested tuples, i.e. the value of an attribute may be a sequence of tuples. Single tuples are constructed using the standard $[\cdot]$ brackets. The concatenation of tuples and functions is denoted by $\circ$. The set of attributes defined for an expression e is defined as $\mathcal{A}(e)$. The set of free variables of an expression e is defined as $\mathcal{F}(e)$.

The projection of a tuple on a set of attributes A is denoted by $|_A$. For an expression e_1 possibly containing free variables, and a tuple e_2, we denote by $e_1(e_2)$ the result of evaluating e_1 where bindings of free variables are taken from variable bindings provided by e_2. Of course this requires $\mathcal{A}(e_2) \subseteq \mathcal{F}(e_1)$. For a set of attributes A we define the tuple constructor $\perp_A$ such that it returns a tuple with attributes in A initialized to NULL.

For sequences e we use $\alpha(e)$ to denote the first element of a sequence. We equate elements with single element sequences. The function τ retrieves the tail of a sequence and $\oplus$ concatenates two sequences. We denote the empty sequence by ϵ. As a first application, we construct from a sequence of non-tuple values e a sequence of tuples denoted by $e[a]$ in which the non-tuple values are bound to a new attribute a. It is empty if e is empty. Otherwise $e[a] = [a : \alpha(e)] \oplus \tau(e)[a]$.

By *id* we denote the identity function. In order to avoid special cases during the translation of XQuery into the algebra, we use the special algebraic operator ($\Box$) that returns a singleton sequence consisting of the empty tuple, i.e. a tuple with no attributes.

We will only define order-preserving algebraic operators. For the unordered counterparts see [6]. Typically, when translating a more complex XQuery into our algebra, a mixture of order-preserving and not order-preserving operators will occur. In order to keep the paper readable, we only employ the order-preserving operators and use the same notation for them that has been used in [5,6] and SAL [2].

Our algebra will allow nesting of algebraic expressions. For example, within a selection predicate of a select operator we allow the occurrence of further nested algebraic expressions. Hence, a join within a selection predicate containing a nested algebraic expression is possible. This simplifies the translation procedure of nested XQuery expressions into the algebra. However, nested algebraic expressions force a nested-loop evaluation strategy. Thus, the goal of the paper will

[2] http://www.w3.org/TR/xmlquery-use-cases

be to remove nested algebraic expressions. As a result, we perform unnesting of nested queries not at the source level but at the algebraic level. This approach is more versatile and less error-prone.

We define the algebraic operators recursively on their input sequences. For unary operators, if the input sequence is empty, the output sequence is also empty. For binary operators, the output sequence is empty whenever the left operand represents an empty sequence.

The order-preserving **selection** operator with predicate p is defined as

$$\sigma_p(e) := \begin{cases} \alpha(e) \oplus \sigma_p(\tau(e)) & \text{if } p(\alpha(e)) \\ \sigma_p(\tau(e)) & \text{else} \end{cases}$$

For a list of attribute names A we define the **projection** operator as

$$\Pi_A(e) := \alpha(e)|_A \oplus \Pi_A(\tau(e))$$

We also define a duplicate-eliminating projection Π^D_A. Besides the projection, it has similar semantics as the `distinct-values` function of XQuery: it does not preserve order. However, we require it to be deterministic and idempotent. Sometimes we just want to eliminate some attributes. When we want to eliminate the set of attributes A, we denote this by $\Pi_{\overline{A}}$. We use Π for renaming attributes using the notation $\Pi_{A':A}$. Here, the attributes in A are renamed to those in A'. Attributes other than those in A remain untouched.

The **map** operator is defined as follows:

$$\chi_{a:e_2}(e_1) := \alpha(e_1) \circ [a : e_2(\alpha(e_1))] \oplus \chi_{a:e_2}(\tau(e_1))$$

It extends a given input tuple $t_1 \in e_1$ by a new attribute a whose value is computed by evaluating $e_2(t_1)$. For an example see Figure 1.

R_1

A_1
1
2
3

R_2

A_2	B
1	2
1	3
2	4
2	5

$\chi_{a:\sigma_{A_1=A_2}(R_2)}(R_1) =$

A_1	a
1	$\langle [1,2],[1,3] \rangle$
2	$\langle [2,4],[2,5] \rangle$
3	$\langle\ \rangle$

Fig. 1. Example for Map Operator

We define the **cross product** of two tuple sequences as

$$e_1 \times e_2 := (\alpha(e_1) \overline{\times} e_2) \oplus (\tau(e_1) \times e_2)$$

where

$$e_1 \overline{\times} e_2 := \begin{cases} \epsilon & \text{if } e_2 = \epsilon \\ (e_1 \circ \alpha(e_2)) \oplus (e_1 \overline{\times} \tau(e_2)) & \text{else} \end{cases}$$

We are now prepared to define the **join** operation on ordered sequences:

$$e_1 \bowtie_p e_2 := \sigma_p(e_1 \times e_2)$$

The **left outer join**, which will play an essential role in unnesting, is defined as

$$e_1 \overset{g:e}{\underset{p}{⟕}} e_2 := \begin{cases} (\alpha(e_1) \bowtie_p e_2) \oplus (\tau(e_1) \overset{g:e}{\underset{p}{⟕}} e_2) & \text{if } (\alpha(e_1) \bowtie_p e_2) \neq \epsilon \\ (\alpha(e_1) \circ \perp_{\mathcal{A}(e_2)\setminus\{g\}} \circ [g : e]) \oplus (\tau(e_1) \overset{g:e}{\underset{p}{⟕}} e_2) & \text{else} \end{cases}$$

where $g \in \mathcal{A}(e_2)$. Our definition deviates slightly from the standard left outer join operator, as we want to use it in conjunction with grouping and (aggregate) functions. Consider the relations R_1 and R_2 in Figure 2. If we want to join R_1 (via left outer join) to R_2^{count} that is grouped by A_2 with counted values for B, we need to be able to handle empty groups (for $A_1 = 3$). e defines the value given to attribute g for values in e_1 that do not find a join partner in e_2 (in this case 0).

For the rest of the paper let $\theta \in \{=, \leq, \geq, <, >, \neq\}$ be a simple comparison operator. Our grouping operators produce a new sequence-valued attribute g containing "the group". We define the **unary grouping** operator in terms of the binary grouping operator.

$$\Gamma_{g;\theta A;f}(e) := \Pi_{A:A'}(\Pi^D_{A':A}(\Pi_A(e))\Gamma_{g;A'\theta A;f}e)$$

where the **binary grouping** operator (sometimes called nest-join [29]) is defined as

$$e_1\Gamma_{g;A_1\theta A_2;f}e_2 := \alpha(e_1) \circ [g : G(\alpha(e_1)] \oplus (\tau(e_1)\Gamma_{g;A_1\theta A_2;f}e_2)$$

Here, $G(x) := f(\sigma_{x|_{A_1}\theta A_2}(e_2))$ and function f assigns a meaningful value to empty groups. See also Figure 2 for an example. The unary grouping operator processes a single relation and obviously groups only on those values that are present. The binary grouping operator works on two relations and uses the left hand one to determine the groups. This will become important for the correctness of the unnesting procedure.

Given a tuple with a sequence-valued attribute, we can unnest it by using the **unnest** operator defined as

$$\mu_g(e) := (\alpha(e)|_{\overline{\{g\}}} \times \alpha(e).g) \oplus \mu_g(\tau(e))$$

where $e.g$ retrieves the sequence of tuples of attribute g. In case that g is empty, it returns the tuple $\perp_{\mathcal{A}(e.g)}$. (In our example in Figure 2, $\mu_g(R_2^g) = R_2$.)

We define the **unnest map** operator as follows:

$$\Upsilon_{a:e_2}(e_1) := \mu_g(\chi_{g:e_2[a]}(e_1))$$

This operator is mainly used for evaluating XPath expressions. Since this is a very complex issue [15,17], we do not delve into optimizing XPath evaluation

R_1

A_1
1
2
3

R_2

A_2	B
1	2
1	3
2	4
2	5

$\Gamma_{g;=A_2;count}(R_2) = R_2^{count}$

A_2	g
1	2
2	2

$\Gamma_{g;=A_2;id}(R_2) = R_2^g$

A_2	g
1	$\langle[1,2],[1,3]\rangle$
2	$\langle[2,4],[2,5]\rangle$

$R_1 \Gamma_{g;A_1=A_2;id}(R_2) = R_{1,2}^g$

A_1	g
1	$\langle[1,2],[1,3]\rangle$
2	$\langle[2,4],[2,5]\rangle$
3	$\langle\,\rangle$

Fig. 2. Examples for Unary and Binary Γ

but instead take an XPath expression occurring in a query as it is and use it in place of e_2. An optimized translation is well beyond the scope of the paper.

For **result construction**, we employ a simplified operator Ξ that combines a pair of Groupify-GroupApply operators [12]. It executes a semicolon separated list of commands and, as a side effect, constructs the query result. The Ξ operator occurs in two different forms. In its simple form, besides side-effects, Ξ is the identity function, i.e. it returns its input sequence. For simplicity, we assume that the result is constructed as a string on some output stream. Then the simplest command is a string copied to the output stream. If the command is a variable, its string value is copied to the output stream. For more complex expressions the procedure is similar. If e is an expression that evaluates to a sequence of tuples containing a string-valued attribute a that is successively bound to author names from some bibliography document, $\Xi_{"<author>";a;"</author>"}(e)$ embeds every author name into an `author` element.

In its group-detecting form, ${}^{s_1}\Xi^{s_3}_{A;s_2}$ uses a list of grouping attributes (A) and three sequences of commands. We define

$$ {}^{s_1}\Xi^{s_3}_{A;s_2}(e) := \Xi_{(s_1;\Xi_{s_2};s_3)}(\Gamma_{g;=A;\Pi_{g'}} e) $$

Like grouping in general, Ξ can be implemented very efficiently on condition that a group spans consecutive tuples in the input sequence and group boundaries are detected by a change of any of the attribute values in g. Then for every group, the first sequence of statements (s_1) is executed using the first tuple of a group, the second one (s_2) executed for every tuple within a group, and the third one (s_3) is executed using the last tuple of a group. This condition can be met by a stable sort on A. Introducing the complex Ξ saves a grouping operation that would have to construct a sequence-valued attribute.

Let us illustrate ${}^{s_1}\Xi^{s_3}_{A;s_2}(e)$ by a simple example. Assume that the expression e produces the following sequence of four tuples:

```
[a: "author1", t: "title1"]
[a: "author1", t: "title2"]
```

```
[a: "author2", t: "title1"]
[a: "author2", t: "title3"]
```

Then ${}^{s_1}\Xi^{s_3}_{a;s_2}(e)$ with

```
s1 = "<author>";"<name>";a;"</name>"
s2 = "<title>";t;"</title>"
s3 = "</author>"}
```

produces

```
<author>
  <name>author1</name>
    <title>title1</title>
    <title>title2</title>
</author>
<author>
  <name>author2</name>
    <title>title1</title>
    <title>title3</title>
</author>
```

For implementation issues concerning the operators of NAL, please consult our technical report [22].

3 Three Cases for Unnesting

In this section, we present several examples of nested queries (based on the XQuery use-case document) for which unnesting techniques result in major performance gains. For space reasons we omit the details of the normalization and translation (see [22] for details). We concentrate on the equivalences used for unnesting our example queries. The proofs of the equivalences can also be found in [22].

We verified the effectiveness of the unnesting techniques experimentally. The experiments were carried out on a simple PC with a 2.4 Ghz Pentium using the Natix query evaluation engine [11]. The database cache was configured such that it could hold the queried documents. The XML files were generated with the help of ToXgene using the DTDs from the XQuery use-case document which are shown in Fig. 3. We executed the various evaluation plans on different sizes of input documents (varying the number of elements).

3.1 Grouping

The first query is a typical example where the nested query is used to express grouping. Frequently, (after normalization) an expression bound in the **let** clause originally occurred in the **return** clause, which is an equivalent way of expressing grouping in XQuery. The normalization step takes care of the uniform treatment of these different formulations.

```
<!DOCTYPE bib [
  <!ELEMENT bib        (book*)>
  <!ELEMENT book       (title,  (author+ |
                        editor+), publisher,
                        price )>
  <!ATTLIST book       year CDATA  #REQUIRED>
  <!ELEMENT author     (last, first)>
  <!ELEMENT editor     (last, first,
                        affiliation)>
  <!ELEMENT title      (#PCDATA)>
  <!ELEMENT last       (#PCDATA)>
  <!ELEMENT first      (#PCDATA)>
  <!ELEMENT affiliation (#PCDATA)>
  <!ELEMENT publisher  (#PCDATA)>
  <!ELEMENT price      (#PCDATA)>
]>
```

```
<!DOCTYPE prices [
  <!ELEMENT prices (book*)>
  <!ELEMENT book   (title, source,
                    price)>
  <!ELEMENT title  (#PCDATA)>
  <!ELEMENT source (#PCDATA)>
  <!ELEMENT price  (#PCDATA)>
]>

<!DOCTYPE bids [
  <!ELEMENT bids      (bidtuple*)>
  <!ELEMENT bid tuple (userid, itemno,
                       bid, biddate)>
  <!ELEMENT userid    (#PCDATA)>
  <!ELEMENT itemno    (#PCDATA)>
  <!ELEMENT bid       (#PCDATA)>
  <!ELEMENT biddate   (#PCDATA)>
]>
```

Fig. 3. DTDs for the example queries

```
let $d1 := doc("bib.xml")
for $a1 in distinct-values($d1//author)
let $t1 := let $d2 := doc("bib.xml")
           for $b2 in $d2/book
           let $a2 := $b2/author,
               $t2 := $b2/title
           where $a1 = $a2
           return $t2
return
  <author>
    <name> { $a1 } </name>
    { $t1 }
  </author>
```

The translation algorithm maps the **for** clause to the Υ operator and the **let** clause to the χ operator. The subscript of these operators is defined by the binding expression of both clauses. (Frequently, these binding expressions contain XPath expressions. However, since we concentrate on unnesting techniques in XQuery in this paper, we rely on efficient translation and evaluation techniques [15,17]. These techniques can be used orthogonally to the methods presented in this paper.) The **where** clause is translated into a σ operator, and the Ξ operator constructs the query result as defined in the **return** clause. We greatly simplify the translation of the **return** clause and refer to [12] for a more advanced treatment of result construction in XQuery. The function `distinct-values` is mapped to the Π^D operator.

For the example query we get the following algebraic expression:

$$\Xi_{s1;a1;s2;t1;s3}(\chi_{t1:\Pi_{t2}(\sigma_{a1\in a2}(\hat{e}_2))}(\hat{e}_1))$$

where

$\hat{e}_1 := \Upsilon_{a1:\Pi^D(d1//author)}(\chi_{d1:doc}(\Box))$
$\hat{e}_2 := \chi_{t2:b2/title}(\chi_{a2:b2/author[a2']}(\hat{e}_3))$
$\hat{e}_3 := \Upsilon_{b2:d2/book}(\chi_{d2:doc}(\Box))$

and

```
doc = doc("bib.xml")
s1  = "<author><name>"
s2  = "</name>"
s3  = "</author>"
```

During translation, we have to ensure the existential semantics of the general comparison in XQuery. In our case, $a1 is bound to a single value and $a2 to a sequence. Consequently, we have to translate $a1 = $a2 into $a1 \in a2$. From the DTD we know that every `book` element contains only a single `title` element. Hence, we can save the introduction of an attribute $t2'$ and the invocation of a concatenation operation that is implicitly invoked in XQuery.[3] Therefore, we can apply a simple projection on $t2$ to model the `return` clause of the inner query block.

As already mentioned, the first example query performs a grouping operation, and our unnesting equivalences recognize these semantics. Potentially, this yields more efficient execution plans. We have to be careful, however, because the range of values of the grouping attributes of the inner and outer query block might differ. Therefore, we advocate the use of a left outer join. We propose the following new unnesting equivalence:

$$\chi_{g:f(\sigma_{A_1 \in a_2}(e_2))}(e_1) = \Pi_{\overline{A_2}}(e_1 \ {}^{g:f(\epsilon)}_{A_1=A_2} \ \Gamma_{g;=A_2;f}(\mu^D_{a_2}(e_2))) \tag{1}$$

which holds if

$A_i \subseteq \mathcal{A}(e_i)$, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$, $g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$, $A_1 \cap A_2 = \emptyset$,
$a_2 \in \mathcal{A}(e_2)$, $A_2 = \mathcal{A}(a_2)$, (this implies that $A_1 = \mathcal{A}(e_1)$).

Note that each book can be written by several authors. Thus, for the right hand side of the equivalence to be correct, we have to unnest these attributes before grouping them by the correlating attributes. This way, we explicitly handle the existential semantics of the general comparison. Applying this equivalence to the example query we get:

$$\Xi_{s1;a1;s2;t1;s3}(\Pi_{\overline{a2'}}(\hat{e_1} \ {}^{t1:\epsilon}_{a1=a2'} \ (\Gamma_{t1;=a2';\Pi_{t2}}(\mu^D_{a2}(\hat{e_2})))))$$

where $\hat{e_1}$, $\hat{e_2}$, s1, s2, and s3 are defined as above.

There exist alternatives. In our example, we know from the DTD that no `author` elements other than those directly beneath `book` elements can be found in the queried document. Furthermore, if we also know from the document that all authors have written at least one book, we become aware of the fact that the outer and the nested query block actually retrieve their data from the same document. In this case, we can apply the following equivalence (in fact, this condition escaped the authors of [25]):

$$\chi_{g:f(\sigma_{A_1 \in a_2}(e_2))}(e_1) = \Pi_{A_1:A_2}(\Gamma_{g;=A_2;f}(\mu^D_{a_2}(e_2))) \tag{2}$$

[3] XQuery specifies that the result sequences the `return` clause generates for every tuple binding are concatenated.

(Formally speaking, in addition to the preconditions of (1) we have to satisfy $e_1 = \Pi^D_{A_1:A_2}(\Pi_{A_2}(\mu_{a_2}(e_2)))$ to be able to apply this equivalence.)

Since for the example query this is the case when we define $e'_1 = \Pi_{a_1}(\hat{e}_1)$ and $e'_2 = \Pi_{a_2,t_2}(\hat{e}_2)$, we get:

$$\Xi_{s1;a1;s2;t1;s3}(\Pi_{a1:a2'}(\Gamma_{t1;=a2';\Pi_{t2}}(\mu^D_{a2}(e'_2))))$$

where e'_1, e'_2, s1, s2, and s3 are defined as above.

Note that although the order is destroyed on authors, both expressions produce the titles of each author in document order, as is required by the XQuery semantics for this query. While the unnesting algorithm published in [9,10] is able to unnest many more nested expressions, the resulting query does not preserve order (a hash join operator is used). In [25] unnesting is described rather informally, making it difficult to apply the technique in a general context. In our approach both the implementation of the algebraic operators and the transformations via equivalences preserve the document order (for proofs see [22]).

After renaming $a1$ to $a2'$, the expression can be enhanced further by using the group detecting Ξ operator as defined in Section 2:

$$^{s1;a2';s2}\Xi^{s3}_{a2';t2}(\mu^D_{a2}(e'_2))$$

After applying all our rewrites, we need to scan the document just once. A naive nested-loop evaluation leads to $|author| + 1$ scans of the document where $|author|$ is the number of author elements in the document. In the table below, we summarize the evaluation times for the first query. The document `bib.xml` contained either 100, 1000, or 10000 books and 10 authors per book. This demonstrates the massive performance improvements that are possible by unnesting queries.

Plan	Evaluation Time		
	100	1000	10000
nested	0.40 s	31.65 s	3195 s
outerjoin	0.09 s	0.25 s	2.45 s
grouping	0.10 s	0.27 s	2.07 s
group Ξ	0.08 s	0.17 s	1.37 s

3.2 Grouping and Aggregation

Aggregation is often used in conjunction with grouping. We illustrate further unnesting equivalences by using the second example query, which introduces an aggregation in addition to grouping.

```
let $d1 := doc("prices.xml")
for $t1 in distinct-values($d1//book/title)
let $m1 := min(let $d2 := doc("prices.xml")
               for $b2 in $d2//book
```

```
                let $t2 := $b2/title
                let $p2 := $b2/price
                let $c2 := decimal($p2)
                where $t1 = $t2
                return $c2)
return
  <minprice title="{ $t1 }">
     <price> { $m1 } </price>
  </minprice>
```

Knowing from the DTD that every `book` element has exactly one `title` child element[4], the translation yields

$$\Xi_{s1,t1,s2;m1;s3}(\chi_{m1:min(\Pi_{c2}(\sigma_{t1=t2}(\hat{e}_2)))}(\hat{e}_1))$$

where

$$\hat{e}_1 = \Upsilon_{t1:\Pi^D(d1//book/title)}(\chi_{d1:doc}(\Box))$$
$$\hat{e}_2 = \chi_{c2:decimal(p2)}(\chi_{p2:b2/price}(\hat{e}_3))$$
$$\hat{e}_3 = \chi_{t2:b2/title}(\Upsilon_{b2:d2//book}(\chi_{d2:doc}(\Box)))$$

and

```
doc = doc("prices.xml")
s1 = "<minprice title=\""
s2 = "\"><price>"
s3 = "</price></minprice>"
```

To unnest this expression, we propose the following new equivalence:

$$\chi_{g:f(\sigma_{A_1\theta A_2}(e_2))}(e_1) = \Pi_{A_1:A_2}(\Gamma_{g;\theta A_2;f}(e_2)) \tag{3}$$

which holds if

$$A_i \subseteq \mathcal{A}(e_i),\ \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset,\ g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2),\ A_1 \cap A_2 = \emptyset,$$
$$e_1 = \Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2)).$$

Unlike the equivalence for the first example query no unnesting needs to be applied before grouping because attribute A_2 is an atomic value. We have equivalences for more general cases where the last restriction is not fulfilled or where the correlation does not need be equality [22].

Let us again project unneeded attributes away and define $e'_1 := \Pi_{t1}(\hat{e}_1)$ and $e'_2 := \Pi_{t2,c2}(\hat{e}_2)$. Since only `title` elements under `book` elements are considered, the restriction $e'_1 = \Pi^D_{t1:t2}(\Pi_{t2}(e'_2))$ holds and Eqv. 3 can be applied, resulting in

$$\Xi_{s1,t1,s2;m1;s3}(\Pi_{t1:t2}(\Gamma_{m1;=t2;min\circ\Pi_{c2}}(e'_2)))$$

Below, we compare the evaluation times for the two plans with varying numbers of books.

Plan	Evaluation Time		
	100	1000	10000
nested	0.09 s	1.81 s	173.51 s
grouping	0.07 s	0.08 s	0.19 s

[4] Otherwise, translation must use '$\in$' instead of '$=$'.

Again, we observe massively improved execution times after unnesting the query because the unnested query plan needs to scan the source document only once. Assuming a naive execution strategy, the nested query plan scans the document $|title| + 1$ times. Thus, the benefit of unnesting increases with the number of title elements in the document.

3.3 Aggregation in the Where Clause

In our last example query, nesting occurs in a predicate in the **where** clause that depends on an aggregate function, in this case `count`. Our normalization heuristics moves the nested query into a **let** clause and the result of the nested query is applied in the **where** clause. Thus, the normalized query is:

```
let $d1 := doc("bids.xml")
for $i1 in distinct-values($d1//itemno)
let $c1 := count(let $d2 := doc("bids.xml")
                 for $i2 = $d2//bidtuple/itemno
                 where $i1 = $i2
                 return $i2)
where $c1 >= 3
return
  <popular_item>
     { $i1 }
  </popular_item>
```

We do not use a result construction operator on the inner query block because we do not return XML fragments but merely tuples containing a count. Hence, a projection is sufficient.

$$\Xi_{s1,i1,s2}(\sigma_{c1>=3}(\chi_{c1:count(\sigma_{i1=i2}(\hat{e}_2))}(\hat{e}_1)))$$

where

$\hat{e}_1 := \Upsilon_{i1:\Pi^D(d1//itemno)}(\chi_{d1:doc}(\Box))$ and `doc = doc("bids.xml")`
$\hat{e}_2 := \Upsilon_{i2:d2//bidtuple/itemno}(\chi_{d2:doc}(\Box))$ `s1 = "<popular_item>"`
`s2 = "</popular_item>"`

Projecting away unnecessary attributes, we define $e_1' := \Pi_{i1}(\hat{e}_1)$ and $e_2' := \Pi_{i2}(\hat{e}_2)$. Looking at the DTD of bids.xml, we see that `itemno` elements appear only directly beneath `bidtuple` elements. Thus, the condition $e_1' = \Pi^D_{i1:i2}(\Pi_{i2}(e_2'))$ holds and we can apply Eqv. 3:

$$\Xi_{s1,i1,s2}(\sigma_{c1>=3}(\Pi_{i1:i2}(\Gamma_{c1;=i2;count}(e2'))))$$

The evaluation times for each plan are given in the table below. The number of bids and items is varied between 100 and 10000. The number of items equals 0.2 times the number of bids. Again we observe that the unnested evaluation plan scales better than the nested plan.

Plan	Evaluation Time 100	1000	10000
nested	0.06 s	0.53 s	48.1 s
grouping	0.06 s	0.07 s	0.10 s

4 Conclusion

In the core of the paper we presented equivalences that allow to unnest nested algebraic expressions. Eqvs. (1) and (2) are new in both the ordered and the unordered context. An equivalent of Eqv. (2) in the ordered context appeared in [25], but without giving the important condition $e_1 = \Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$. The proofs of the equivalences are more complex in the ordered context and can be found in [22].

We demonstrated each of the equivalences by means of an example. Thereby, we showed their applicability to queries with and without aggregate functions. The experiments conducted in this paper include the first extensive performance numbers on the effectiveness of unnesting techniques for XML queries. We observed enormous performance improvements verifying the benefits of applying the rewrites. Besides our measurements, only the authors of reference [25] hint on some performance numbers for their unnesting algorithm.

The equivalences assume that the nested queries do not construct XML fragments. In many cases this restriction can be lifted by using rewrite before applying the unnesting equivalences presented here. However, including these rewrites is beyond the scope of this paper.

Acknowledgment. We thank Simone Seeger for her help in preparing the manuscript and C.-C. Kanne for his help in carrying out the experiments. We also thank the anonymous referees for their helpful comments.

References

1. M. M. Astrahan and D. D. Chamberlin. Implementation of a structured English query language. *Communications of the ACM*, 18(10):580–588, 1975.
2. C. Beeri and Y. Tzaban. SAL: An algebra for semistructured data and XML. In *ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 1999.
3. G. Bhargava, P. Goel, and B. Iyer. Hypergraph based reorderings of outer join queries with complex predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 304–315, 1995.
4. S. Chaudhuri and K. Shim. Optimizing queries with aggregate views. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 167–182, 1996.
5. S. Cluet and G. Moerkotte. Nested queries in object bases. In *Proc. Int. Workshop on Database Programming Languages*, 1993.
6. S. Cluet and G. Moerkotte. Classification and optimization of nested queries in object bases. Technical Report 95-6, RWTH Aachen, 1995.

7. U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *VLDB*, pages 197–208, 1987.
8. Leonidas Fegaras and Ramez Elmasri. Query engines for Web-accessible XML data. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *Proceedings of the Twenty-seventh International Conference on Very Large Data Bases: Roma, Italy, 11–14th September, 2001*, pages 251–260, Los Altos, CA 94022, USA, 2001. Morgan Kaufmann Publishers.
9. Leonidas Fegaras, David Levine, Sujoe Bose, and Vamsi Chaluvadi. Query processing of streamed XML data. In *Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, USA, November 4-9, 2002*, pages 126–133. ACM, 2002.
10. Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, 25(4):457–516, 2000.
11. T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a Native XML Base Management System. *VLDB Journal*, 11(4):292–314, 2002.
12. T. Fiebig and G. Moerkotte. Algebraic XML construction and its optimization in Natix. *World Wide Web Journal*, 4(3):167–187, 2002.
13. C. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Trans. on Database Systems*, 22(1):43–73, Marc 1997.
14. R. Ganski and H. Wong. Optimization of nested SQL queries revisited. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–33, 1987.
15. G. Gottlob, C. Koch, and R. Pichler. Xpath query evaluation: Improving time and space efficiency. In *Proc. IEEE Conference on Data Engineering*, page to appear, 2003.
16. W. Hasan and H. Pirahesh. Query rewrite optimization in starburst. Research Report RJ6367, IBM, 1988.
17. S. Helmer, C.-C. Kanne, and G. Moerkotte. Optimized translation of xpath expressions into algebraic expressions parameterized by programs containing navigational primitives. In *Proc. Int. Conf. on Web Information Systems Engineering (WISE)*, 2002. 215–224.
18. W. Kiessling. SQL-like and Quel-like correlation queries with aggregates revisited. ERL/UCB Memo 84/75, University of Berkeley, 1984.
19. W. Kim. On optimizing an SQL-like nested query. *ACM Trans. on Database Systems*, 7(3):443–469, Sep 82.
20. A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, 1982.
21. C. Leung, H. Pirahesh, and P. Seshadri. Query rewrite optimization rules in IBM DB2 universal database. Research Report RJ 10103 (91919), IBM Almaden Research Division, January 1998.
22. N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. Technical Report TR-03-002, Lehrstuhl für Praktische Informatik III, Universität Mannheim, 2003.
23. M. Muralikrishna. Optimization and dataflow algorithms for nested tree queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1989.
24. M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 91–102, 1992.
25. S. Paparizos, S. Al-Khalifa, H. V. Jagadish, L. V. S. Lakshmanan, A. Nierman, D. Srivastava, and Y. Wu. Grouping in XML. In *EDBT Workshops*, pages 128–147, 2002.

26. H. Pirahesh, J. Hellerstein, and W. Hasan. Extensible/rule-based query rewrite optimization in Starburst. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 39–48, 1992.
27. A. Rosenthal and C. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 291–299, 1990.
28. P. Seshadri, H. Pirahesh, and T. Leung. Complex query decorrelation. In *Proc. IEEE Conference on Data Engineering*, pages 450–458, 1996.
29. H. Steenhagen, P. Apers, and H. Blanken. Optimization of nested queries in a complex object model. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 337–350, 1994.
30. H. Steenhagen, P. Apers, H. Blanken, and R. de By. From nested-loop to join queries in oodb. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 618–629, 1994.
31. H. Steenhagen, R. de By, and H. Blanken. Translating OSQL queries into efficient set expressions. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 183–197, 1996.

A DTD Graph Based XPath Query Subsumption Test

Stefan Böttcher and Rita Steinmetz

University of Paderborn
Faculty 5 (Computer Science, Electrical Engineering & Mathematics)
Fürstenallee 11, D-33102 Paderborn, Germany
stb@uni-paderborn.de, rst@uni-paderborn.de

Abstract. XPath expressions play a central role in querying for XML fragments. We present a containment test of two XPath queries which determines whether a new XPath query XP1 can reuse a previous query result XP2. The key idea is to transform XP1 into a graph which is used to search for sequences of elements which are used in the XPath query XP2.

1 Introduction

1.1 Problem Origin and Motivation

The development of our XPath containment test was motivated by an XML database system that allows for the caching and reuse previous query results (e.g. on a mobile client) for the evaluation of a new XPath query. If we can prove that a previous query result which is already stored on a client can be reused for a new XPath query, this will have a considerable advantage over the shipping of the fragment selected by an XPath query to the mobile client again. Our goal is to prove, without any access to the data stored in the XML database, i.e. independent of the actual database state, that one XPath expression XP1 selects a subset of the data selected by another XPath expression XP2 – or as we say that *XP1 is subsumed by XP2*.

Due to the fact that we need a very fast tester, we allow our tester to be incomplete in the following sense. If our tester returns `true`, the new query XP1 definitely asks for a subset of a stored result of a previous query XP2. However, we allow our tester to return `false`, although the query is a subset of a previous query result, and in this case the data is once again requested from the database server.

In this contribution, we present only the containment tester itself, whereas the reuse of previously stored data is discussed in [2] and concurrency issues (e.g. how modifications of the original set of XML data are treated) are discussed in [3].

1.2 Relation to Other Work and Our Focus

Our contribution is related to other contributions to the area of containment tests for XPath and semi-structured data. Query containment on semi-structured data for other query languages has been examined e.g. by [5, 8]. In comparison to this, we examine the formulas of two XPath expressions XP1 and XP2 in order to decide whether XP1 selects a subset of the data selected by XP2. For this purpose, we follow [6, 9, 10, 11], which also contribute to the solving of the containment problem for two XPath ex pressions under a DTD or under other given XML schema constraints. The main is

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 85–99, 2003.

sues of the contributions [6, 9, 10, 11] are the decidability and upper and lower bounds of the complexity of the containment problem for XPath expressions. In contrast to this, we focus on an incomplete tester which can efficiently determine for a large subset of XPath queries that a previous query result can be reused.

[6] derives a set of constraints from the XPath expressions and from a DTD and it provides a set of transformation rules as to how these constraints can be normalized, so that the containment problem on a small subset of XPath and on a restricted set of DTDs can be solved in polynomial time. In comparison, our approach additionally supports the parent-axis, the ancestor-axis, and the ancestor-or-self-axis, supports the node test * at the end of XPath expressions, supports attribute variables and supports arbitrary DTDs.

The contributions [9,10,11] use tree patterns in order to normalize the XPath query expressions and compare them to the XML document tree. They consider the DTD as a set of constraints [11] or they use it as an automaton [10]. In comparison to this, we follow [1] and use the concept of a DTD graph in order to expand all paths which are selected by an XPath expression, and we right-shuffle all filter expressions within sets of selected paths. Beyond our previous work [1], this tester supports a larger subset of XPath (e.g. it includes parent-axis and ancestor-axis location steps, more node tests and nested filters), regards constraints given by the DTD and achieves a higher degree of completeness. Our transformation of an XPath query into a graph is similar to the transformation of an XPath query into an automaton, which was used in [7] in order to decide whether or not an XML document fulfills this query. However, in contrast to all other contributions, our approach combines a graph based search for paths (selected by XP1 but not by XP2) with the right-shuffling of predicate filters in such a way, that the containment test for XPath expressions which includes all axes (except preceding (-sibling) and following (-sibling)) can be reduced to a containment test of filters combined with a containment test of possible filter positions.

1.3 The Supported Subset of XPath Expressions

The XPath specification [12] defines an XPath expression as being a sequence of location steps

/<LocationStep_1> / … / <LocationStep_N>,

with <LocationStep_I> being defined as

axis-specifier_I :: node-test_I [predicate_filter_I].

As XPath is a very expressive language and our goal is to achieve a balance between a large subset of supported XPath expressions (covering most queries which are used in practice) and an efficient subsumption test algorithm on two of these XPath expressions, we restrict the axis specifiers, node tests and predicates allowed in XPath expressions as follows:

1.3.1 Axis Specifiers

We support *absolute* or *relative location paths* with *location steps* with the following *axis specifiers*: self, child, descendant, descendant-or-self, parent, ancestor, ancestor-or-self and attribute, and we forbid namespace, following (-sibling) and preceding (-sibling) axes.

1.3.2 Node Tests

We support all *node name tests* except name-spaces, but we exclude *node type tests* like text(), comment(), processing-instruction() and node(). We support wildcards (*), but only at the end of an XPath query.
For example, when A is an attribute and E is an element, then ../@A, ./ancestor::E,.//E , ./E/* and ./E//* are supported XPath expressions.

1.3.3 Predicate Filters

We restrict *predicate filters* of supported XPath expressions to be either a *simple predicate filter* [B] with a *simple filter expressions* B or to be a compound predicate filter.

1.3.3.1 Simple Filter Expressions

Let <path> be a relative XPath expression which can use parent-axis, attribute-axis and child-axis location steps, and let <value> be a constant.

- <path> is a simple filter expression. For example, when A is an attribute and E is an element, then ./@A and ../../E are simple filter expressions which check for the existence of an attribute A and of an element E which is a child of the current element's grandparent respectively.
- Each *comparison* <path> = <value> and each *comparison* <path> != <value> are simple filter expressions.[1]
- 'not <path>' is a simple predicate filter, which means that the element or attribute described by <path> does not exist.

1.3.3.2 Compound Predicate Filters

If [B1] and [B2] are supported predicate filters, then '[B1] [B2]', '[B1 and B2]', '[B1 or B2]' and '[not (B1)]' are also supported predicate filters. Note that in our subset of supported predicate filters, '[B1] [B2]' and '[B1 and B2]' are equivalent, because we excluded the sibling-axes, i.e., we do not consider the order of sibling nodes.

1.4 Used Terminology and Problem Definition

Within this section, we use the XPath expression XP=//E3//E2/E1/E4 in order to explain some terms which are used in the remainder of our presentation. The *nodes selected by XP* (in this case nodes with the element name 'E4') can be reached from the root node by a *path* which passes at least the nodes E3, E2, E1 and E4 (in this order). These paths are called *paths selected by XP* or *selected paths* for short.

Every path selected by XP must contain an element E2 which is directly followed by an element E1 which is itself directly followed by E4. We call a list of elements which are 'connected' by child-axis location steps in XP (e.g. E2/E1/E4) an *element sequence of XP*. We also use the term *element sequence* for a single element (like E3), i.e., an element where neither the location step before it nor after it are child-axis location steps. Thus, E3 and E2/E1/E4 are the element sequences of this example.

---

[1] Note that we can also allow for more comparisons, e.g. comparisons which contain the operators '<', '>', '≤' or '≥' and comparisons such as <path> <comparison operator> <path2>. In such a case the predicate filter tester which is used as part of our tester in Section 3.5 would have to check more complex formulas.

The input of our tester are the two XPath expressions (the new query XP1 on the one hand and a query XP2 representing a cached query result on the other hand) and a DTD which defines all valid XML documents. As the tester is performed on the client side, it has to decide based on this input alone, i.e. without any access to the original XML document, whether or not XP1 selects a subset of the data selected by XP2 in all the documents which are valid according to the given DTD – or as we say: the tester has to decide whether or not *XP1 is subsumed by XP2*. In other words, *XP2 subsumes XP1*, if and only if there does never exist a path selected by XP1 which is not a path selected by XP2 in any document which is valid according to the given DTD.

Section 2 describes the preparation steps for the subsumption test, i.e., how we transform the DTD into a so called DTD graph, and how we use this DTD graph in order to normalize the XPath expressions. Section 3 outlines the major parts of our subsumption test, i.e., how we compute a so called XP1 graph which describes all the paths selected by XP1, and how we try to place XP2 sequences on each path of the XP1 graph in such a way that the filter of each XP2 sequence subsumes an XP1 filter.

2 DTD Graph Construction and Normalization of XPath Expressions

2.1 Translating the DTD into a DTD Graph and a Set of Associated DTD Filters

A *DTD graph* [1] is a directed graph G=(N,V) where each node E∈N corresponds to an element of the DTD and an edge v ∈ V, v=(E1,E2) from E1 to E2 exists for each element E2 which is used to define the element E1 in the DTD. For example (Example 1), Figure 1 shows a DTD and the corresponding DTD-graph:

```
<!ELEMENT root (E1, E2)>
<!ELEMENT E1   (E2, E3, E4)>
<!ELEMENT E2   (E1)>
<!ELEMENT E3   (E2)>
<!ELEMENT E4   (#PCDATA)>
```

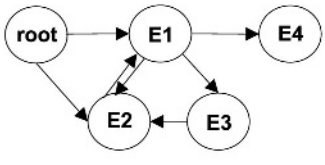


Fig. 1. DTD and corresponding DTD graph of Example 1.

We can use the DTD graph in order to check whether or not it allows for at least one path selected by XP1 (or XP2 respectively). If the DTD graph does not allow for any path selected by XP1 (or XP2 respectively), then XP1 (or XP2) selects the empty node set. We consider this to be a special case, i.e., if XP2 selects the empty node set, we try to prove that XP1 also selects the empty node set. For the purpose of the discussion which takes place in the following sections, we assume that at least one path for XP1 (and XP2 respectively) exists in the DTD graph.

Such a DTD graph does not contain all the concepts which can be expressed by a DTD, e.g., it does neither distinguish between optional and mandatory elements nor between disjunctions and conjunctions expressed in the DTD. These concepts can be added using so called *DTD filters* which are attached to the nodes of the DTD graph,

and every time, an XPath expression 'passes' this node, the DTD filter is attached to the XPath expression as a predicate filter.
If for example (Example 2) the second line of the DTD in Example 1 is changed to

```
<!ELEMENT E1 ( E2, (E3 | E4+))>  ,
```

we get the same DTD graph, but the DTD filter [./E2 and (./E3 xor ./E4) and unique(E2) and unique(E3)] is attached to the DTD graph node that is labeled E1, whereas the relative path './E2' states, that each element E1 must have a child-element E2 and 'unique(E2)' states that each element E1 has at most one child E2.

These DTD filters can also be used to discard *forbidden paths*, i.e. paths, where an inconsistency between the predicate filters and the DTD filters exists. For example, the XPath expression //E1[./E3]/E4 asks for a forbidden path to E4, because it requires an element E1 to have both children, E3 and E4 – which contradicts the DTD filter.

As we allow our tester to be incomplete, the tester can ignore some or even all of these DTD filters. Note however that the tester remains correct for the following reason. The number of valid documents described by the DTD graph may only increase and cannot decrease, when we eliminate DTD filters. Thus, a successful check that the condition 'XP1 is subsumed by XP2' holds for a larger set of valid documents, is sufficient for this condition to hold also for a subset of these documents.

2.2 Element Distance Formulas

A second initialization step involves the computation of element distance formulas for each pair of elements which occur in the DTD and to store them in the so called *DTD distance table* [1]. These element distance formulas are a key part of right-shuffling filters and of the filter subsumption test outlined in Section 3.

For example, the distance from E1 to E2 in the DTD graph of Example 1 is any number of child-axis location steps that fulfills the formula "3*x+2*y+1", i.e., the distance table contains an entry "3*x+2*y+1"$(x, y \geq 0)$" which describes the set of all possible distances from E1 to E2. The distance table entry "3*x+2*y+1"$(x, y \geq 0)$" represents two nested *loops* (one loop of the 3 elements E1, E2, and E3 and one loop of the 2 elements (E1 and E2) in the DTD graph and a direct path from E1 to E2 with a distance of 1.

If only one direct path P (i.e. a path without a circle) from E1 to E2 exists (which has the length 'k'), and exactly one circle which contains one or more nodes of P exists in the DTD graph, then the distance formula for the distance from E1 to E2 is of the form 'c*x+k (x≥0)'. Thereby, 'c' is the length of the circle, and the variable 'x' denotes the number of times the circle is passed and is called *circle variable*.

In general, a DTD graph can contain more than one direct path from E1 to E2 and a circle may be attached to any node on a path from E1 to E2. Therefore, the general form of an entry of the DTD distance table is a disjunction of linear equations

$(\sum_{1 \leq i \leq m} ci*xi + km\ (xi \geq 0)$ or ... or $\sum_{1 \leq i \leq n} ci*xi + kn\ (xi \geq 0))$,

where m (and n respectively) are the numbers of circles attached along a path of length km (and kn respectively) from E1 to E2, ci is the length of the i-th circle along such a path, and xi is the circle variable for that circle.

2.3 Transformation, Normalization, and Simplification of XPath Queries

We need an additional transformation step in order to normalize the formulas of both XPath expressions. First of all, we transform relative XPath expressions into absolute ones. Thereafter, we insert '/root' at the beginning of an XPath expression, if the XPath expression does not start with a child-axis location step, where 'root' is assumed to be the name of the root-element of the DTD.

If the XPath expression contains one or more parent-axis location steps or ancestor-axis location steps, these steps are replaced from left to right according to the following rules. Let LS1,…,LSn be location steps which neither use the parent-axis nor the ancestor-axis, and let XPtail be an arbitrary sequence of location steps. Then we replace /LS1/…/LSn/child::E[F]/../XPtail with /LS1/…/LSn[./E[F]]/XPtail .

Similarly, in order to replace the first parent-axis location step in the XPath expression /LS1/…/LSn/descendant::E[F]/../XPtail , we use the DTD graph in order to compute all parents P1,…,Pm of E which can be reached by descendent::E after LSn has been performed, and we replace the XPath expression with /LS1/…/LSn//(P1|…|Pm)[./E[F]]/XPtail .

In order to substitute an ancestor location step ancestor::E[F] in an XPath expression /LS1/…/LSn/ancestor::E[F]/XPtail, we use the DTD graph in order to compute all the possible positions between the 'root' and the element selected by LSn where E may occur. Depending on the DTD graph, there may be more than one position, i.e., we replace the given XPath expression with

(//E[F][/LS1/.../LSn] / XPtail) | (/LS1//E[F][/LS2/.../LSn]/XPtail) | ... |
(/LS1/.../LSn-1/E[F][/LSn]/XPtail) .

Similar rules can be applied in order to eliminate the ancestor-or-self-axis, the self-axis and the descendent-axis, such that we finally only have child-axis and descendant-or-self-axis-location steps (and additional filters) within our XPath expressions.

Finally, nested filter expressions are eliminated, e.g. a filter [./E1[./@a and not (@b="3")]] is replaced with a filter [./E1 and (./E1/@a and not ./E1/@b="3")] . More general: a nested filter [./E1[F1]] is replaced with a filter [./E1 and F1'] where the filter expression F1' is equal to F1 except for the modification that it adds the prefix ./E1 to each location path in F1 which is defined relative to E1. This approach to the unnesting of filter expressions can be extended to the other axes and to sequences of location steps, such that we do not have any nested filters after these unnesting steps have been carried out.

3 The Major Parts of Our Subsumption Test

Firstly, we construct a so called *XP1 graph* which contains the set of all possible paths for XP1 in any valid XML document according to the given DTD. Then, XP1 is subsumed by XP2, if the following holds for all paths for XP1 which are allowed by the DTD: the path for XP1 contains all sequences of XP2 in the correct order, and a corresponding XP1 node with a filter which is as least as restrictive as the filter attached to the XP2 element exists for each XP2 element of the sequence which has a filter. In other words, if a path selected by XP1 which does not contain all sequences of XP2 in the correct order is found, then XP1 is not subsumed by XP2.

3.1 Extending the DTD Graph to a Graph for Paths Selected by XP1

In order to represent the set of paths selected by XP1, we use a graph which we will call the *XP1 graph* for the remainder of the paper [1]. The XP1 graph can be derived from the DTD graph and the XPath expression XP1 which represents the new query by Algorithm 1 described below. Each path selected by XP1 corresponds to one path from the root node of the XP1 graph to the node(s) in the XP1 graph which represents (or represent) the selected node(s). The XP1 graph contains a superset of all paths selected by XP1, because some paths contained in the XP1 graph may be *forbidden paths*, i.e. paths that have predicate filters which are incompatible with DTD constraints and/or the selected path itself (c.f. Section 2.1). We use the XP1 graph in order to check, whether or not each path from the root node to a selected node contains all the sequences of XP2, and if so, we are then sure that all the paths selected by XP1 contain all the sequences of XP2.

Example 3: Consider the DTD graph of Example 1 and an XPath expression XP1 = /root/E1/E2//E4, which requires that all XP1 paths start with the element sequence /root/E1/E2 and end with the element E4. Figure 2 shows the XP1 graph for the XPath expression XP1, where each node label represents an element name and each edge label represents the distance formula between the two adjacent nodes.

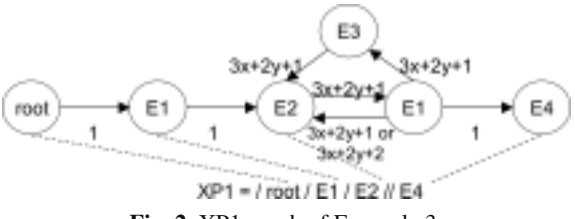


Fig. 2. XP1 graph of Example 3.

The following Algorithm 1 (taken from [1]) computes the XP1 graph from a given DTD graph and an XPath expression XP1:

```
(1)   GRAPH GETXP1GRAPH(GRAPH DTD, XPATH XP1)
(2)   { GRAPH XP1Graph = NEW GRAPH( DTD.GETROOT() );
(3)     NODE lastGoal = DTD.GETROOT();
(4)     while(not XP1.ISEMPTY()) {
(5)       NODE goalElement = XP1.REMOVEFIRSTELEMENT();
(6)       if (XP1.LOCATIONSTEPBEFORE(goalElement) == '/')
(7)         XP1Graph.APPEND( NODE(goalElement) );
(8)       else
(9)         XP1Graph.EXTEND(
(10)          DTD.COMPUTEREDUCEDDTD(lastGoal,goalElement));
(11)      lastGoal = goalElement;
(12)    }
(13)    return XP1Graph;
(14)  }
```

Algorithm 1: Computation of the XP1 graph from an XPath expression XP1 and the DTD

By starting with a node that represents the root-element (line (2)), the algorithm transforms the location steps of XP1 into a graph as follows. Whenever the actual location step is a child-axis location step (lines (6)-(7)), we add a new node to the graph and take the name of the element selected by this location step as the node label for the new node. Furthermore, we add an edge from the element of the previous location step to the element of the current location step with a distance of 1. For each descendant axis step E1//E2 Algorithm 1 attaches a subgraph of the DTD graph (the so called *reduced DTD graph*) to the end of the graph already generated. The reduced DTD graph, which is computed by the method call `ComputeReducedDTD(...,...)`, contains all paths from E1 to E2 of the DTD graph, and it obtains the distance formulas for its edges from the DTD distance table.

If XP1 ends with //*, i.e., XP1 takes the form XP1 = XP1'//*, the XP1'graph is computed for XP1'. Subsequently one reduced DTD graph which contains all the nodes which are successors of the end node of the XP1'graph is appended to the end node of the XP1 graph. All these appended nodes are then also marked as end nodes of the XP1 graph. Similarly, if XP1 ends with /*, i.e., XP1 takes the form XP1 = XP1'/*, the XP1'graph is computed for XP1'. Afterwards all the nodes of the DTD graph which can be reached within one step from the end node are appended to the end node of the XP1 graph. Furthermore, instead of the old end node now all these appended nodes are marked as end nodes of the XP1 graph.

3.2 Combining XP2 Predicate Filters within Each XP2 Sequence

Before our main subsumption test is applied, we will perform a further normalization step on the XPath expression XP2. Within each sequence of XP2, we shuffle all filters to the rightmost element itself which carries a filter expression, so that after this normalization step has been carried out all filters within this sequence are attached to one element.

The *shuffling* of a filter by one location-step to the right involves adding one parent-axis location step to the path within the filter expression and attaching it to the next location step. For example, an XPath expression XP2=//E1[./@b]/E2[./@a]/E3 is transformed into an equivalent XPath expression XP2'=//E1/E2[../@b and ./@a]/E3 .

3.3 Placing One XP2 Element Sequence with Its Filters in the XP1 Graph

Within our main subsumption test algorithm (Section 3.7), we use a Boolean procedure which we call `PlaceFirstSequence(in XP1Graph,inout XP2,inout startNode)`. It tests whether or not a given XP2 sequence can be placed successfully in the XP1 graph at a given `startNode`, such that each filter of the XP2 sequence subsumes an XP1 filter (as outlined in Section 3.4).

Because we want to place XP2 element sequences in paths selected by XP1, we define the manner in which XP2 elements correspond to XP1 graph nodes as follows. An XP1 graph node and a node name test which occurs in an XP2 location step *correspond* to each other, if and only if the node has a label which is equal to the element name of the location step or the node name test of the location step is *. We say, *a path (or a node sequence) in the XP1 graph and an element sequence of XP2 corre-*

spond to each other, if the n-th node corresponds to the n-th element for all nodes in the XP1 graph node sequence and for all elements in the XP2 element sequence.

The procedure PLACEFIRSTSEQUENCE(…,…,…) checks whether or not each path in the XP1 graph which begins at `startNode` fulfils the following two conditions: firstly that the path has a prefix which corresponds to the first sequence of XP2 (i.e. the node sequences that correspond to the first element sequence of XP2 can not be circumvented by any XP1 path), secondly, if the first sequence of XP2 has a filter, then this filter subsumes for each XP1 path at least one given filter.

In general, more than one path in the XP1 graph which starts at `startNode` and corresponds to a given XP2 sequence may exist, and therefore there may be more than one XP1 graph node which corresponds to the final node of the XP2 element sequence. The procedure PLACEFIRSTSEQUENCE(…,…,…) internally stores the final node which is the nearest to the end node of the XP1 graph (we call it the *last final node*). When we place the next XP2 sequence at or 'behind' this last final node, we are then sure, that this current XP2 sequence has been completely placed before the next XP2 sequence, whatever path XP1 will choose.

If only one path which begins at `startNode` which does not have a prefix corresponding to the first sequence of XP2 or which does not succeed in the filter implication test for all filters of this XP2 sequence (as described in Section 3.5) is found, then the procedure PLACEFIRSTSEQUENCE(…,…,…) does `not` change XP2, does `not` change `startNode` and returns `false`. If however the XP2 sequence can be placed on all paths and the filter implication test is successful for all paths, then the procedure removes the first sequence from XP2, copies the last final node to the inout parameter `startNode` and returns `true`.

3.4 A Filter Implication Test for All Filters of One XP2 Element Sequence and One Path in the XP1 Graph

For this section, we consider only one XP2 sequence E1/…/En and only one path in the XP1 graph which starts at a given node which corresponds to E1.

After the filters within one XP2 sequence have been normalized (as described in Section 3.2), each filter is attached to exactly one element which we call the *current element*. When given a `startNode` and a path of the XP1 graph, the node which corresponds to the current element is called the *current node*.

Within the first step we right-shuffle all predicate filters of the XP1 XPath expression, which are attached to nodes which are predecessors of the current node, into the current node. To *right-shuffle* a filter expression from one node into another simply means attaching $(../)^{d}$ to the beginning of the path expression inside this filter expression, whereas d is the distance from the first node to the second node. This distance can be calculated by adding up all the distances of the paths that have to be passed from the first to the second node.

By right-shuffling filters of XP1 (or XP2 respectively), we get a filter [f1]=[$(../)^{d1}$ fexp1] of XP1 (or a filter [f2]=[$(../)^{d2}$ fexp2] of XP2 respectively), where d1 and d2 are distance formulas, and fexp1 and fexp2 are filter expressions which do neither start with a parent-axis location step nor with a distance formula. Both, d1 and d2, depend on node distances which are obtained from the XP1 graph and may contain zero

or more circle variables xi. A subsumption test is performed on this right-shuffled XP1 filter [f1] and the XP2 filter [f2] which is attached to the current element. The subsumption test on filters returns that [f1] is subsumed by [f2] (i.e., [f1] is at least as restrictive as [f2]) if and only if

– every distance chosen by XP1 for d1 can also be chosen by XP2 for d2 (or as we referred to it in the next section: the *distance formula d1 is subsumed by the distance formula d2*) and
– fexp1 ⇒ fexp2 .

As both, fexp1i and fexp2i, do not contain any loops, any predicate tester which extends the Boolean logic to include features of XPath expressions (e.g. [4]), can be used in order to check whether or not fexp1i ⇒ fexp2j .

For example, a filter [f1]=[(../)^{1}@a="77"] is at least as restrictive as a filter [f2]=[../@a], because the implication [@a="77"]⇒[@a] holds, and both filters have the same constant distance d1=d2=1 (which states that the attribute a has to be defined for the parent of the current node). A predicate tester for such formulas has to consider e.g. that [not ./@a="77" and not ./@a!="77"] is equivalent to [not ./@a].

If the subsumption test for one XP2 filter returns that this filter is subsumed by the XP1 filter, this XP2 filter is discarded. This is performed repeatedly until either all XP2 filters of this sequence are discarded or until all XP1 filters which are attached to nodes which are predecessors of the current node are shuffled into the current node.

If finally not all XP2 filters of this sequence are discarded, we carry out a second step in which all these remaining filters are right-shuffled into the next node to which an XP1 filter is attached. It is again determined, whether or not one of the XP2 filters can be discarded, as this XP2 filter subsumes the XP1 filter. This is also performed until either all XP2 filters are discarded (then the filter implication test for all filters of the XP2 sequence and the XP1 path returns `true`) or until all XP1 filters have been checked and at least one XP2 filter remains that does not subsume any XP1 filter (then the filter implication test for all filters of the XP2 sequence and the XP1 path returns `false`).

3.5 A Subsumption Test for Distance Formulas

Within the right-shuffling of XP2 filters, we distinguish two cases. When an XP2 filter which is attached to an element E is right-shuffled *over* a circle '*behind*' E (i.e. the node corresponding to E is not part of the circle) in the XP1 graph (as described in the previous section), a term ci*xi+k is added to the distance formula d2 (where ci is the number of elements in the circle, k is the shortest distance over which the filter can be shuffled, and xi is the circle variable which describes how often a particular path follows the circle of the XP1 graph).

However, a special case occurs, if we right-shuffle a filter *out of a circle* (i.e. the element E to which the filter is attached belongs to the circle). For example, let the XP2 sequence consist only of one element and this element corresponds to an XP1 graph node which belongs to a circle, or let all elements of the XP2 sequence correspond to XP1 graph nodes which belong to exactly one (and the same) circle. In contrast to an XP1 filter which is right-shuffled over a circle, an XP2 filter is right-shuffled *out of a circle* by adding ci*xi′+k to the filter distance (where ci, xi, and k are defined as before and 0≤ xi′ ≤xi). While xi describes the number of times XP2

has to pass the loop in order to select the same path as XP1, the `xi'` describes the number of times the circle is passed, after XP2 has set its filter. This can be any number between and including `xi` and 0.

More general: whenever n circles on which the whole XP2 sequence can be placed exist, say with circle variables x1, …, xn, then XP2 can choose for each circle variable xi a value `xi'` ($0 \leq$ `xi'` $\leq$ xi) which describes how often the circle is passed after XP2 sets the filter, and d2 (i.e. the distance formula for the filter of the XP2 sequence) depends on `x1'`, …, `xn`' instead of on x1, …, xn.

We say, *a loop* loop1=(../)^{d1} *is subsumed by a loop*[2] loop2=(../)^{d2}, if d2 can choose every distance value which d1 can choose. In other words: no matter, what path XP1 'chooses', XP2 can choose the same path and can choose its circle variables[3] in such a way that d1=d2 holds. That is, XP2 can apply its filter to the same elements which the more restrictive filters of XP1 are applied to.

Altogether, a filter [loop1 fexp1] of XP1 is subsumed by a filter [loop2 fexp2] of XP2 which is attached to the same node, if loop1 is subsumed by loop2 and fexp1i $\Rightarrow$ fexp2j.

3.6 Including DTD Filters into the Filter Implication Test

The DTD filter associated with a node can be used to improve the tester as follows. For each node on a path selected by XP1 (and XP2 respectively) the DTD filter [FDTD] which is associated with that node must hold. For the DTD given in Example 2, we conclude in Section 2.1, that a node E1 which has both, a child node E3 and a child node E4, can not exist. Ignoring the other DTD filter constraints for E1, the DTD filter for each occurrence of E1[4] is [FDTD_E1]=[not (./E3 and ./E4)]. Furthermore, let us assume that an XP1 graph node E1 has a filter [F1]=[./E3], and the corresponding element sequence of XP2 consists of only the element E1 with a filter [F2]=[not (./E4)]. We can then conclude that

FDTD_E1 and F1 $\Rightarrow$ FDTD_E1 and F2 ,

i.e., with the help of the DTD filter, we can prove that the XP1 filter is at least as restrictive as the XP2 filter. Of course, the implication can be simplified to

FDTD_E1 and F1 $\Rightarrow$ F2 .

In more general terms: for each node E1 in the XP1 graph which is referred to by an XP2 filter, we can include the DTD filter [FDTD_E1] which is required for all elements E1, and right-shuffle it like an XP1 filter. This is how the filter implication test above and the main algorithm described in the next section can be extended to include DTD filters.

---

[2] The definition *one loop is subsumed by another* also includes paths without a loop, because distances can have a constant value.

[3] Some of the circle variables may be of the form xi while others may be of the form `xi'`.

[4] Note that the DTD filter has to be applied to each occurrence of a node E1, in comparison to an XP1 filter or an XP2 filter assigned to E1, both of which only have to be applied to a single occurrence of E1.

3.7 The Main Algorithm: Checking That XP2 Sequences Can Not Be Circumvented

The following algorithm tests for an XPath expression XP2 and an XP1 graph whether or not each path of the XP1 graph contains all element sequences of XP2, starts with nodes which correspond to the first element sequence of XP2, and ends with nodes which correspond to the last element sequence of XP2.
The main algorithm (which is outlined on the next page) searches for one XP2 sequence after the other from left to right a corresponding node sequence in the XP1 graph, starting at the root node of the XP1 graph (line(2)).
If XP2 consists of only one sequence (lines (3) and (4)), the procedure call `SequenceONALLPaths(XP1Graph,XP2,startNode)` returns whether or not each path of the XP1 graph from the current `startNode` to an end node of the XP1 graph corresponds to XP2 and the XP2 filter subsumes an XP1 filter on this path.
The case where XP2 contains more than one element sequence is treated in the middle part of the algorithm (lines (5)-(14)). The first sequence of XP2 has to placed in such a way that it starts at the root node (line (7)), i.e., if this is not possible (line (8)), the test can be aborted.
As outlined in Section 3.3, if and only if the procedure PLACEFIRSTSEQUENCE(...) returns `true`, it also removes the first sequence from XP2, and it changes `startNode` to the first possible candidate node where to place the next XP2 sequence.
The while-loop (lines 9-14) is repeated until only one element sequence remains in XP2. Firstly, the procedure SEARCH(...) searches for and returns that node which fulfills the following three conditions: it corresponds to the first element of the first element sequence of XP2, it is equal to or behind and nearest to startNode, and it is common to all paths of the XP1 graph. If such a node does not exist, the procedure SEARCH(...) returns null and our procedure XP2SUBSUMESXP1 returns false (line (11)), i.e., the test can be aborted. Otherwise (line (12)) the procedure PLACEFIRSTSEQUENCE(...) tests, whether or not the whole element sequence (together with its filters) can be successfully placed beginning at startNode. If the sequence with its filters can not be placed successfully here, (line (13)), a call of the procedure NEXTNODE(XP1Graph,startNode)computes the next candidate node, where the sequence can possibly be placed, i.e. that node behind startNode which is common to all paths of the XP1 graph and which is nearest to startNode.
When the last sequence of XP2 is finally reached, we have to distinguish the following three cases. If the last XP2 sequence represents a location step '//*' (line 15), this sequence can be successfully placed on all paths, and therefore true is returned. If the last location step of XP2 is '//*[F2]' (line (16)), we check whether or not for every path from the current startNode to an end node of the XP1 graph, the filter [F2] of XP2 subsumes an XP1 filter [F1] on this path (line (17)). Otherwise (line (18)), it has to be ensured, that each path from the current startNode to an endNode of the XP1 graph has to end with this sequence. If this final test returns true, XP1 is subsumed by XP2, otherwise the subsumption test algorithm returns false.

```
(1)   BOOLEAN XP2SUBSUMESXP1(GRAPH XP1Graph, XPATH XP2)
(2)   { startNode:= XP1Graph.getROOT() ;
(3)     if(XP2.CONTAINSONLYONESEQUENCE())
(4)       return SEQUENCEONALLPATHS(XP1Graph,XP2,startNode);
(5)     else        // XP2 contains multiple sequences
(6)     { //place the first sequence of XP2:
(7)       if(not PLACEFIRSTSEQUENCE(XP1Graph,XP2,startNode))
(8)          return false;
           // place middle sequences of XP2:
(9)       while (XP2.containsMoreThanOneSequence())
(10)      {  startNode:=SEARCH(XP1Graph,XP2,startNode);
(11)         if(startNode == null) return false;
(12)         if(not PLACEFIRSTSEQUENCE(XP1Graph,XP2,startNode))
(13)            startNode:= NEXTNODE(XP1Graph,startNode);
(14)      }
           //place last sequence of XP2:
(15)      if ( XP2 == '*' ) return true; //  XP2 is '//*'
(16)      if ( XP2 == '*[F2]' )          //     '//*[F2]'
(17)         return (for every path from startNode to an
                     end node of XP1graph, [F2] subsumes
                     an XP1 filter on this path) ;
(18)       return (all paths from startNode to an end node
                   of XP1graph contain a suffix which
                   corresponds to the XP2 sequence)
(19)    }
(20)  }
```

Main algorithm: The complete subsumption test

3.8 A Concluding Example

We complete Section 3 with an extended example which includes all the major steps of our contribution. Consider the DTD of Example 1 and the XPath expressions
XP1 = / root / E1 / E2[./@b] / E1[./@c=7] / E2 // E4[../../@a=5] and
XP2 = // E2 / E1[./../@b] // E2[./@a] // E1 / * (Example 4).

Fig. 3. XP1 graph of Example 4

Step1: Figure 3 shows the computed XP1 graph and the filters which are attached to its nodes. In order to be able to explain the algorithm, we have assigned an ID to each node of the XP1 graph in this example.

Step2: XP2 is transformed into the equivalent XPath expression XP2= / root // E2/E1 $[(../)^{1}@b]$ // E2 [./@a] // E1 / * .

Step3: Algorithm 2 is started. The corresponding node (i.e. the node with ID 1) is found for the first XP2 sequence (i.e. 'root'),. The next sequence of XP2 to be placed is E2/E1$[(../)^{1}@b]$ and one corresponding path in the XP1 graph is E2→E1, where E2 has ID 3 and E1 has ID 4. The node with ID 4 is our current node. Now the XP1 filter [./@b] of node 3 is right-shuffled into the current node and thereby transformed into $[(../)^{1}@b]$. Because this filter is subsumed by the filter of the current XP2 sequence, the filter of the current XP2 sequence is discarded. Since each filter of this XP2 sequence is discarded, the sequence is successfully placed, and the `startNode` is set as node 4.

The next sequence of XP2 to be placed is E2[./@a]. The first corresponding node is the node with ID 5, which is now the current node. The filters of node 3 and node 4 are shuffled into the current node and are transformed into one filter $[(../)^{2}@b$ and $(../)^{1}@c=7]$. However this filter is not subsumed by the filter [./@a] of the actual XP2 sequence. This is why the filter [./@a] is afterwards shuffled into the next node to which an XP1 filter is attached (i.e. into node 8). Thereby, the XP2 sequence filter [./@a] is transformed into $[(../)^{3x'+2y'+2}@a]$, ($x \geq x' \geq 0$, $y \geq y' \geq 0$) - the distance formula contains the variables x' and y', as the XP2 sequence contains only one element (i.e. E2) which corresponds to an XP1 graph node (i.e. node 5) which is part of a circle. As XP2 can assign the value 0 to x' and y' for each pair of values x,y≥0, the XP1 filter which is attached to node 8 (and which is equivalent to $[(../)^{2}@a=5]$) is subsumed by the filter of the current XP2 sequence. Altogether, the current XP2 element sequence is successfully placed, and the `startNode` is set to be node 5. As now the only remaining sequence of XP2 is E1/*, and this sequence ends with /*, it is tested whether or not each path from the `startNode` (i.e. node 5) to the predecessor of the `endNode` (i.e. node 6) has a suffix which corresponds to E1. This is true in this case.

Therefore the result of the complete test is `true`, i.e., XP1 is subsumed by XP2. XP1 therefore selects a subset of the data selected by XP2.

4 Summary and Conclusions

We have developed a tester which checks whether or not a new XPath query XP1 is subsumed by a previous query XP2. Before we apply the two main algorithms of our tester, we normalize the XPath expressions XP1 and XP2 in such a way that thereafter we only have to consider child-axis and descendent-or-self-axis location steps. Furthermore, nested filters are unnested, and thereafter within each XP2 element sequence filters are right-shuffled into the right-most location step of this element sequence which contains a filter.

In comparison to other contributions to the problem of XPath containment tests, we transform the DTD into a DTD graph and DTD filters, and we derive from this graph and XP1 the so called XP1 graph, a graph which contains all the valid paths which are selected by XP1. This allows us to split the subsumption test into two parts: first, a

placement test for XP2 element sequences in the XP1 graph, and second, an implication test on filter expressions which checks for each XP2 filter whether or not a more restrictive XP1 filter exists. The implication test on predicate filters can also be split into two independent parts: a subsumption test on distance formulas and an implication test on the remaining filter expressions which do not contain a loop any more. For the latter part, we can use any extension of a Boolean logic tester which obeys the special rules for XPath. This means that, depending on the concrete task, different testers for these filter expressions can be chosen: either a more powerful tester which can cope with a larger set of XPath filters, but may need a longer run time, or a faster tester which is incomplete or limited to a smaller subset of XPath.

To our impression, the results presented here are not just limited to DTDs, but can be extended in such a way that they also apply to XML schema.

References

[1] Stefan Böttcher, Rita Steinmetz: Testing Containment of XPath Expressions in order to Reduce the Data Transfer to Mobile Clients. ADBIS 2003.

[2] Stefan Böttcher, Adelhard Türling: XML Fragment Caching for Small Mobile Internet Devices. 2nd International Workshop on Web-Databases. Erfurt, Oktober, 2002. Springer, LNCS 2593, Heidelberg, 2003.

[3] Stefan Böttcher, Adelhard Türling: Transaction Validation for XML Documents based on XPath. In: Mobile Databases and Information Systems. Workshop der GI-Jahrestagung, Dortmund, September 2002. Springer, Heidelberg, LNI-Proceedings P-19, 2002.

[4] Stefan Böttcher, Adelhard Türling: Checking XPath Expressions for Synchronization, Access Control and Reuse of Query Results on Mobile Clients. Proc. of the Workshop: Database Mechanisms for Mobile Applications, Karlsruhe, 2003. Springer LNI, 2003.

[5] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Moshe Y. Vardi: View-Based Query Answering and Query Containment over Semistructured Data. DBPL 2001: 40-61.

[6] Alin Deutsch, Val Tannen: Containment and Integrity Constraints for XPath. KRDB 2001.

[7] Yanlei Diao, Michael J. Franklin: High-Performance XML Filtering: An Overview of YFilter, IEEE Data Engineering Bulletin, March 2003.

[8] Daniela Florescu, Alon Y. Levy, Dan Suciu: Query Containment for Conjunctive Queries with Regular Expressions. PODS 1998: 139-148.

[9] Gerome Miklau, Dan Suciu: Containment and Equivalence for an XPath Fragment. PODS 2002: 65-76.

[10] Frank Neven, Thomas Schwentick: XPath Containment in the Presence of Disjunction, DTDs, and Variables. ICDT 2003: 315-329.

[11] Peter T. Wood: Containment for XPath Fragments under DTD Constraints. ICDT 2003: 300-314.

[12] XML Path Language (XPath) Version 1.0 . W3C Recommendation November 1999. http://www.w3.org/TR/xpath.

PowerDB-XML: A Platform for Data-Centric and Document-Centric XML Processing

Torsten Grabs and Hans-Jörg Schek

Database Research Group,
Institute of Information Systems,
Swiss Federal Institute of Technology Zurich, Switzerland,
{grabs,schek}@inf.ethz.ch

Abstract. Relational database systems are well-suited as a platform for data-centric XML processing. Data-centric applications process regularly structured XML documents using precise predicates. However, these approaches come too short when XML applications also require document-centric processing, i.e., processing of less rigidly structured documents using vague predicates in the sense of information retrieval. The PowerDB-XML project at ETH Zurich aims to address this drawback and to cover both these types of XML applications on a single platform. In this paper, we investigate the requirements of document-centric XML processing and propose to refine state-of-the-art retrieval models for unstructured flat document such that they meet the flexibility of the XML format. To do so, we rely on so-called *query-specific statistics* computed dynamically at query runtime to reflect the query scope. Moreover, we show that document-centric XML processing is efficiently feasible using relational database systems for storage management and standard SQL. This allows us to combine document-centric processing with data-centric XML-to-database mappings. Our XML engine named PowerDB-XML therefore supports the full range of XML applications on the same integrated platform.

1 Introduction

The eXtended Markup Language XML [30] is highly successful as a format for data interchange and data representation. The reason for this is the high flexibility of its underlying semistructured data model [1]. The success of XML is reflected by the interest it has received by database research following its recommendation by the W3C in 1998. However, this previous stream of research has mainly focused on *data-centric XML processing*, i.e., processing of XML documents with well-defined regular structure and queries with precise predicates. This has led to important results which, for instance, allow to map data-centric XML processing onto relational database systems. XML and its flexible data model however cover a much broader range of applications, namely the full range from data-centric to document-centric processing. *Document-centric XML processing* deals with less rigidly structured documents. In contrast to data-centric

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 100–117, 2003.

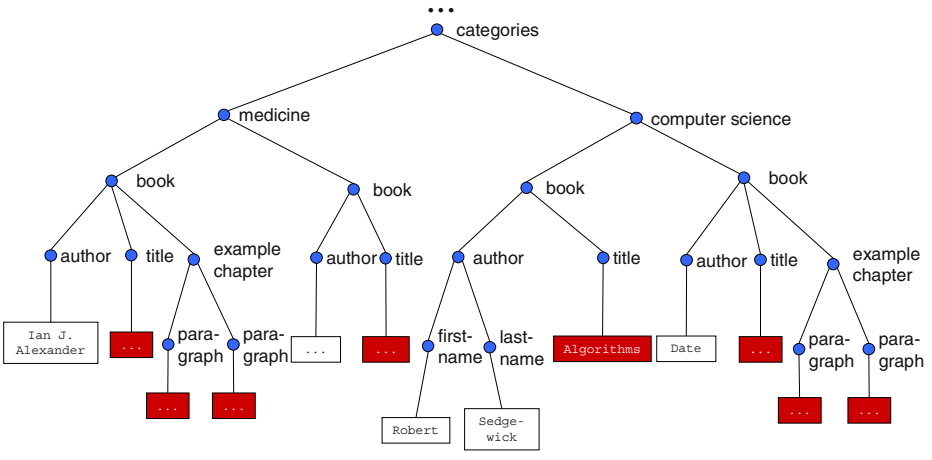


Fig. 1. Exemplary XML document with textual content represented as shaded boxes

processing, document-centric applications require processing of vague predicates, i.e., queries expressing information needs in the sense of information retrieval (IR for short).

Data-centric approaches proposed in previous work on XML processing do not cover document-centric requirements. In addition, conventional ranked retrieval on text documents is not directly applicable to XML documents because of the flexibility of the XML data model: users want to exploit this flexibility when posing their queries. Such queries usually rely on path expressions to dynamically combine one or several element types to the scope of a query. This is in contrast to conventional IR where the retrieval granularity is restricted either to complete documents or to predefined fields such as abstract or title. Dynamically defined query scopes with XML retrieval however affect retrieval, their ranking techniques, and in particular local and global IR statistics. *Local IR statistics* represent the importance of a word or term in a given text. *Term frequencies*, i.e., the number of occurences of a certain term in a given document are local statistics with the vector space retrieval model, for instance. *Global IR statistics* in turn reflect the importance of a word or a term with respect to the document collection as a whole. Taking again vector space retrieval as an example, *document frequencies*, i.e., the number of documents a given term appears in, are global statistics. State-of-the-art retrieval models such as vector space retrieval combine both local and global statistics to compute the ranked result of an IR query.

The following discussion takes vector space retrieval as a running example and illustrates shortcomings of conventional IR statistics in the context of XML retrieval.

Consider the example XML document shown in Fig. 1. The document contains information about books from the domains of medicine and computer sci-

ence. Such a document often is the only document in the collection, i.e., all information is stored in a single XML document. This represents a typical situation with many practical settings. Consequently, conventional vector space document frequencies equal either 1 or 0: 1 for all terms occurring in the document and 0 otherwise. Hence, the intention of global statistics to discriminate important and less important terms is lost when using conventional IR statistics for XML retrieval. A further observation is that conventional term frequencies do not reflect different query scopes: term frequencies are computed for the document as a whole. XML retrieval however often restricts search to certain sub-trees in the XML document, e.g., the computer science or medicine branch of the example document shown in Fig. 1. Our bottomline therefore is that conventional IR statistics need to be refined for flexible retrieval from XML documents.

The objective of our work is twofold: we want to address the aforementioned issues regarding IR statistics with so-called *query-specific IR statistics*. They are computed on-the-fly, i.e., at query processing time, and both local and global statistics reflect the scope of the query. Our second objective is to make respective document-centric XML retrieval functionality available on a platform such as a relational database system that is also well-suited for data-centric XML processing. Regarding these objectives, this current paper makes the following contributions: based on previous work [16,17], we define query-specific IR statistics for flexible XML retrieval. Moreover, we show how to realize document-centric XML processing with query-specific statistics on top of relational database systems in combination with data-centric XML processing. Our overall contribution is our XML engine called PowerDB-XML. Based on relational database systems for storage management, it realizes the envisioned platform for joint data-centric and document-centric XML processing.

The remainder of the paper discusses PowerDB-XML's approach to joint data-centric and document-centric XML processing on top of relational database systems. The following section (Sect. 2) covers related work. In Sect. 3, we discuss flexible retrieval on XML documents. The section defines query-specific statistics and explains them in more detail using vector space retrieval and *tf idf* ranking as an example. Section 4 in turn describes the system architecture of PowerDB-XML. Using again vector space retrieval, it explains how to extend relational data-centric XML mappings in order to store index data for efficient flexible retrieval from XML documents using query-specific statistics. Section 5 then discusses processing of document-centric requests over the XML collection stored. Special interest is paid to computing query-specific IR statistics dynamically, i.e., at query runtime, from the underlying IR index data using standard SQL. Section 6 concludes the paper.

2 Related Work

Data-centric XML processing has received much interest by database research after XML has been recommended by the W3C in 1998. This has led to important results such as query languages for data-centric XML processing (XPath [28]

and XQuery [29] among others). Besides query languages, several data-centric mappings for XML documents to relational databases have been proposed, e.g., EDGE and BINARY [7], STORED [5], or LegoDB [3,2]. In addition, several approaches have been devised to map XML query languages to relational storage and SQL [4,6,21,26,25]. Relational database systems are therefore well-suited as a platform for data-centric XML processing.

Recent work has extended this previous stream of research to keyword search on XML documents. Building on relational database technology, it has proposed efficient implementations of inverted list storage and query processing [8,19]. While already refining the retrieval granularity to XML elements, this previous work has still focused on simple Boolean retrieval models. Information retrieval researchers instead have investigated document-centric XML processing using state-of-the-art retrieval models such as vector space or probabilistic retrieval models. An important observation there was that conventional retrieval techniques are not directly applicable to XML retrieval [13,11]. This in particular affects IR statistics used in ranked and weighted retrieval which heavily rely on the retrieval granularities supported. To increase the flexibility of retrieval granularities for searching XML, Fuhr et al. group XML elements (at the instance level) to so-called *indexing nodes* [13]. They constitute the granularity of ranking with their approach while IR statistics such as *idf* term weights are derived for the collection as a whole. The drawback of the approach is that the assignment of XML elements to indexing nodes is static. Users cannot retrieve dynamically, i.e., at query time, from arbitrary combinations of element types. Moreover, this can lead to inconsistent rankings when users restrict the scopes of their queries to element types that do not directly correspond to indexing nodes and whose IR statistics and especially term distributions differ from the collection-wide ones. Our previous work [16] has already investigated similar issues in the context of flat document text retrieval from different domains: queries may cover one or several domains in a single query and ranking for such queries depends on the query scope. Based on this previous work, [17] proposes XML elements as the granularity of retrieval results and refines IR statistics for *tf idf* ranking [22] in this respect. Our approach derives the IR statistics appropriate to the scope of the queries in the XML documents dynamically at query runtime. This current paper extends this previous work by an efficient relational implementation which allows to combine both data-centric and document-centric XML processing on relational database systems.

3 Flexible XML Retrieval with Query-Specific Statistics

Conventional IR statistics for ranked and weighted retrieval come too short for XML retrieval with flexible retrieval granularities [13]. This section extends conventional textual information retrieval models on flat documents to flexible retrieval on semistructured XML documents. A focus of our discussion is on vector space retrieval and to refine it with query-specific statistics. Retrieval

with query-specific statistics also serves as the basic component for document-centric processing with PowerDB-XML.

3.1 Retrieval with the Conventional Vector Space Model

Conventional vector space retrieval assumes flat document texts, i.e., documents and queries are unstructured text. Like many other retrieval techniques, vector space retrieval represents text as a 'bag of words'. The words contained in the text are obtained by IR functionality such as term extraction, stopword elimination, and stemming. The intuition of vector space retrieval is to map both document and query texts to n-dimensional vectors d and q, respectively. n stands for the number of distinct terms, i.e., the size of the vocabulary of the document collection. A text is mapped to such a vector as follows: each position i $(0 < i \leq n)$ of v represents the i-th term of the vocabulary and stores the *term frequency*, i.e., the number of occurrences of t in the text. A query text is mapped analogously to q. Given some query vector q and a set of document vectors C, the document with $d \in C$ that has the smallest distance to or smallest angle with q is deemed most relevant to the query. More precisely, computation of relevance or retrieval status value (rsv) is a function of the vectors q and d in the n-dimensional space. Different functions are conceivable such as the inner product of vectors or the cosine measure. The remainder of this paper builds on the popular so-called *tf idf ranking function* [24]. *tf idf* ranking constitutes a special case of vector space retrieval. Compared to other ranking measures used with vector space retrieval, it has the advantage to approximate the importance of terms regarding a document collection. This importance is represented by the so-called *inverted document frequency* of terms, or *idf* for short. The *idf* of a term t is defined as $idf(t) = \log \frac{N}{df(t)}$, where N stands for the number of documents in the collection and $df(t)$ is the number of documents that contain the term (the so-called *document frequency* of t). Given a document vector d and a query vector q, the retrieval status value $rsv(d, q)$ is defined as follows:

$$rsv(d,q) = \sum_{t \in terms(q)} tf(t,d)\ idf(t)^2\ tf(t,q) \tag{1}$$

Going over the document collection C and computing $rsv(d, q)$ for each document-query-pair with $d \in C$ yields the ranking, i.e., the result for the query.

In contrast to Boolean retrieval, ranked retrieval models and in particular vector space retrieval assume that documents are flat, i.e., unstructured information. Therefore, a straight-forward extension to cover retrieval from semistructured data such as XML documents and to refer to the document structure is not obvious. But, ranked retrieval models are known to yield superior retrieval results [9,23]. The following paragraphs investigate this problem in more detail and present an approach that combines flexible retrieval with result ranking from vector space retrieval.

3.2 Document and Query Model for Flexible XML Retrieval

In the context of this paper, XML documents are represented as trees. We rely on the tree structures defined by the W3C XPath Recommendation [28]. This yields tree representations of XML documents such as the one shown in Fig. 1. Obviously, all textual content of a document is located in the leaf nodes of the tree (shaded boxes in the figure). For ease of presentation, we further assume that the collection comprises only a single XML document – a situation one frequently encounters also in practical settings. Note that this is not a restriction: it is always possible to add a virtual root node to compose several XML documents into a single tree representation such that the subtrees of the virtual root are the original XML documents. Moreover, we define the *collection structure* as a complete and concise summary of the structure of the XML documents in the document collection such as the DataGuide [15].

Flexible retrieval on XML now aims to identify those subtrees in the XML document that cover the user's information need. The granularity of retrieval in our model are the nodes of the tree representation, i.e., subtrees of the XML document. The result of a query is a ranked list of such subtrees. Users define their queries using so-called *structure constraints* and *content constraints*.

Structure constraints define the scope, i.e., the granularity, of the query. With our query model, the granularity of a query is defined by a label path. Taking the XML document in Fig. 1 for instance, the path /bookstore/medicine/book defines a query scope. The extension of the query scope comprises all nodes in the XML document tree that have the same path originating at the root node. The extension of /bookstore/medicine/book comprises two instances – the first and the second medicine book in the document. Users formulate their structure constraints using path expressions. With the XPath syntax [28] and the XML document in Fig. 1, the XPath expression //book for instance yields a query granularity comprising /bookstore/medicine/book and /bookstore/computer-science/book.

Content constraints in turn work on the actual XML elements in the query scope. We distinguish between so-called *vague content constraints* and *precise content constraints*. A vague content constraint defines a ranking over the XML element instances in the query scope. A precise content constraint in turn defines an additional selection predicate over the result of the ranking. In the following, we exclude precise content constraints from our discussion and focus instead on vague content constraints for ranked text search.

3.3 Result Ranking with Query-Specific Statistics

In our previous discussion, we have refined the retrieval granularity of XML retrieval to XML elements. Hence, our query model returns XML elements e in the query result, and we have to adapt the ranking function accordingly:

$$rsv(e,q) = \sum_{t \in terms(q)} tf(t,e)\ ief(t)^2\ tf(t,q) \qquad (2)$$

In contrast to Equation 1, the ranking function now computes a ranking over XML elements e under a query text q. Moreover, term frequencies *tf* and inverted element frequencies *ief* now work at the granularity of XML elements. The following paragraphs investigate the effects of these adaptations in more detail and refine global and local IR statistics to query-specific statistics.

Global IR Statistics. Different parts of a single XML document may have content from different domains. Figure 1 illustrates this with the different branches of the bookstore – one for medicine books and one for computer science books. Intuitively, the term 'computer' is more significant for books in the medicine branch than in the computer science branch. IR statistics should reflect this when users query different branches of the collection structure. The first – and simplest case – is when a query goes to a single branch of the collection structure. We denote this as *single-category retrieval*. In this case, the query-specific global statistics are simply computed from the textual content of the collection structure branch where the query goes to. The following example illustrates this retrieval type.

Example 1 (Single-Category Retrieval). Consider a user searching for relevant books in the computer science branch of the example document in Fig. 1. Obviously, he restricts his queries to books from this particular category. Thus, it is not appropriate to process this query with term weights derived from both the categories medicine and computer science in combination. This is because the document frequencies in medicine may skew the overall term weights such that a ranking with term weights for computer science in isolation increases retrieval quality.

Taking again our running example of vector space retrieval with *tf idf* ranking, global IR statistics are the (inverted) element frequencies with respect to the single branch of the collection structure covered by the query. We therefore define the element frequency $ef_{cat}(t)$ of a term t with respect to a branch *cat* of the collection structure as the number of XML element sub-trees that t occurs in. More formally:

$$ef_{cat}(t) = \sum_{e \in cat} \chi(t, e) \tag{3}$$

with $\chi(t, e)$ defined as follows:

$$\chi(t, e) = \begin{cases} 1, \text{ if } \sum_{se \in SE(e)} tf(t, se) > 0 \\ \\ 0, \text{ otherwise} \end{cases} \tag{4}$$

Thus, $\chi(t, e)$ is 1 if at least e or one of its sub-elements se contains t.

Now think of another user who wants to process a query on several categories, i.e., on several non-overlapping branches of the collection structure. We call such queries *multi-category retrieval*. In other words, a multi-category query goes over one or several single-category query scopes. The difficulty with this type of queries is again that conventional IR statistics are not meaningful in the

XML context, as already argued above. A more promising alternative in turn is to rely on query-specific statistics reflecting the scope of the query. The following example illustrates multi-category retrieval.

Example 2 (Multi-Category Retrieval). Recall the XML document from the previous example (cf. Fig. 1). The document in the figure reflects the different categories of books such as medicine or computer science with separate element types for the respective categories. Think of a user who does not care to which category a book belongs, as long as it covers the information need expressed in his query. The granularity of his query are all categories. Hence, the query is an example of *multi-category retrieval* which requires query-specific statistics. Taking again the document in Fig. 1, this means statistics must be derived from both categories medicine and computer science in combination.

With vector space retrieval and *tf idf* ranking, we define the global query-specific IR statistics as follows given a query scope *mcat*: the multi-category element frequency $ef_{mcat}(t)$ of a term t is the number of sub-trees in the XML documents t occurs in. Given this definition, the following equation holds between a multi-category query scope M_q and the single-categories it comprises.

$$ef_{mcat}(t, \mathcal{M}_q) = \sum_{cat \in \mathcal{M}_q} ef_{cat}(t) \tag{5}$$

This yields the multi-category inverted document frequency:

$$ief_{mcat}(t, \mathcal{M}_q) = \log \frac{\sum_{cat \in \mathcal{M}_q} N_{cat}}{\sum_{cat \in \mathcal{M}_q} ef_{cat}(t)} \tag{6}$$

Local IR Statistics. XML allows to hierarchically structure information within a document such that each document has a tree structure. Users want to refer to this structure when searching for relevant information. The intuition behind this is that an XML element is composed from different parts, i.e., its child elements. For instance, a chapter element may comprise a title and one or several paragraph elements. This is an issue since the children elements may contribute to the content of an XML element by different degrees. Fuhr at al. for instance reflect the importance of such composition relationships with so-called augmentation weights that downweigh statistics when propagating terms along composition relationships [13]. This also affects relevance-ranking for XML retrieval, as the following example shows.

Example 3 (Nested Retrieval). Consider again the XML document shown in Fig. 1. Think of a query searching for relevant book elements in the medicine branch. Such a query has to process content that is hierarchically structured: the title elements as well as the paragraph elements describe a particular book element. Intuitively, content that occurs in the title element is deemed more important than that in the paragraphs of the example chapter, and relevance ranking for books should reflect this.

Hierarchical document structure in combination with augmentation affects local IR statistics only. Consequently, term frequencies are augmented, i.e., downweighed by a factor $aw \in [0;1]$ when propagating them upwards from a sub-element se to an ancestor element e in the document hierarchy. This yields the following definition for term frequencies:

$$tf(t,e) = \Big(\prod_{l \in path(e,se)} aw_l \Big) tf(t,se) \tag{7}$$

After having refined the definition of global and local statistics for flexible XML retrieval, the retrieval status value of an XML element e in the query scope is given by simply instantiating the $tf\,idf$ ranking function with the appropriate global and local statistics for tf and ief. Section 5 will explain how to compute statistics for combinations of several retrieval types in a query.

4 Storage Management with PowerDB-XML

Current approaches to XML processing have focused either on the data-centric or the document-centric side. One of the promises of XML however is to reconcile these – at least in practical settings. Therefore, the objective of the PowerDB-XML project at ETH Zurich is to support both data-centric and document-centric XML processing on a single integrated platform.

A straight-forward approach is to rely on relational database systems, to deploy data-centric database mapping techniques proposed by previous work, and to extend this setting with the functionality needed for document-centric processing. Several approaches have been pursued already to combine document-centric processing with database systems: most commercially available database systems for instance feature extensions for text processing. This enables retrieval over textual content stored in database table columns. However, this does not allow for flexible weighting granularities as discussed in Sect. 3. In particular, query-specific statistics according to the scope of the query are not feasible since IR statistics are not exposed by the text extenders. Therefore, text extenders are not a viable solution.

An alternative approach is to couple a database system for data-centric processing with an information retrieval systems for document-centric processing, as pursued e.g. by [27]. This approach however suffers from the same drawback as the one previously mentioned: IR statistics are hidden by the information retrieval system and query-specific statistics are not possible.

The third approach pursued in the PowerDB-XML project is instead to rely on relational database systems and to realize document-centric functionality on top of the database system with standard SQL. This approach is based on the observation from own previous work [20] and the work by Grossman et al. [18] that information retrieval using relational database systems for storage management is efficient. The advantage of this approach is that storing IR index data in the database makes IR statistics available for document-centric XML processing with query-specific statistics. The following paragraphs discuss how

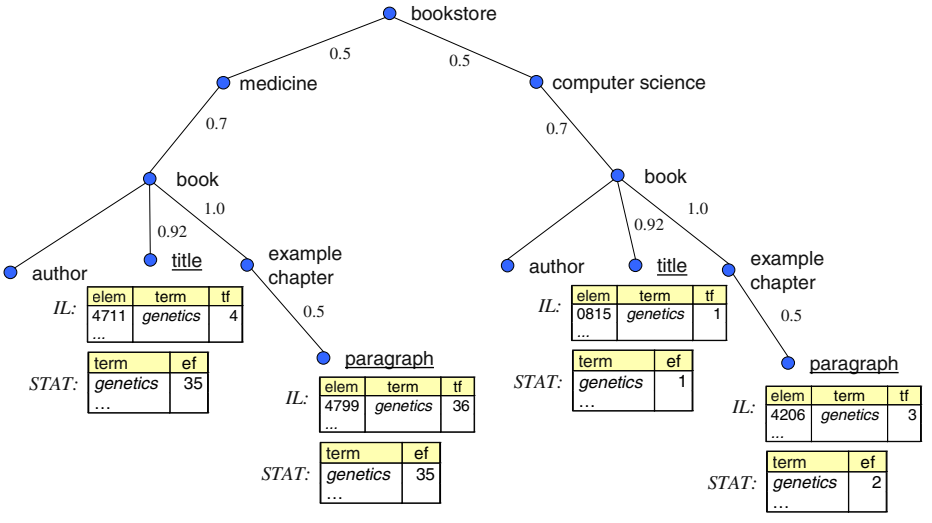


Fig. 2. Basic indexing nodes of the XML document in Fig. 1

to combine data-centric and document-centric storage management on relational database systems and outline the approach taken in PowerDB-XML.

Data-Centric Storage Management. Regarding data-centric database mappings, PowerDB-XML supports the mapping schemes proposed in previous work as discussed in Sect. 2. Our current implementation features text-based mappings, EDGE [7], and STORED [5]. An API allows users to define their own mappings and to deploy them to PowerDB-XML. An administrator then decides for a particular combination of mapping schemes that suits the XML applications running on top of PowerDB-XML.

Document-Centric Storage Management. A naive solution to support flexible retrieval with query-specific statistics would be to keep indexes and statistics for each combination of element types and element nestings that could possibly occur in a query. However, the amount of storage that this approach requires for indexes and statistics is prohibitively large and is therefore not a viable solution. Hence, we refine the notion of indexing nodes as proposed by Fuhr et al. [13] to keep indexes and statistics only for basic element types. When it comes to single-category retrieval, multi-category retrieval or nested retrieval, the approach proposed here derives the required indexes and statistics from the underlying basic ones on-the-fly, i.e., at query runtime. This has the advantage that the amount of storage needed to process IR queries on XML content is small as compared to the naive approach.

To do so, flexible retrieval on XML documents first requires to identify the basic element types of an XML collection that contain textual content. These

nodes are denoted as *basic indexing nodes*. There are several alternatives how to derive the basic indexing nodes from an XML collection:

- The decision can be taken completely automatically such that each distinct element type at the leaf level with textual content is treated as a separate indexing node.
- An alternative is that the user or an administrator decides how to assign element types to basic indexing nodes.

These approaches can further rely on an ontology that, for instance, suggests to group element types summary and abstract into the same basic indexing node. For ease of presentation, let us assume that the basic indexing nodes have already been determined, and the respective textual XML content already underwent IR pre-processing, including term extraction and stemming. PowerDB-XML then annotates the basic indexing nodes with the IR indexes and statistics derived from their textual content. Figure 2 illustrates this for the Data Guide [14,15] of the example document in Figure 1. Element types with underlined names in the figure stand for basic indexing nodes and have been annotated with inverted list tables (*IL*) and statistic tables (*STAT*) for vector space retrieval. The *IL* tables store element identifiers, term occurrences, and local IR statistics (term frequencies for vector space retrieval) in the table columns `elem`, `term`, and `tf`, respectively. The global statistics tables *STAT* in turn store term identifiers and global statistics (element frequencies for vector space retrieval) in the table columns `term` and `ef`, respectively. PowerDB-XML keeps an *IL* and a *STAT* table for each leaf node of the collection structure that has textual content (cf. Fig. 2). The annotations of the edges in the figure represent augmentation weights.

5 Operators for Flexible XML Retrieval

5.1 Operators for Combined Retrieval Types

Depending on the scope of the IR query, a combination of single-category, multi-category, and nested retrieval may be necessary to compute the ranking. Depending on the retrieval granularity and the nesting, several inverted lists and statistics tables may be relevant. The following example illustrates this.

Example 4. Consider a nested retrieval request with the query scope //book and the query text 'XML Information Retrieval' on an XML collection like the one shown in Fig. 1. The request requires the functionality of nested retrieval since the examplechapter-sub-tree and the title sub-element are searched for relevant information. Moreover, the request also requires multi-category retrieval functionality since both computer science and medicine books may qualify for the result.

As the example shows, practical settings require a combination of the different retrieval types discussed in Section 3. In order to efficiently implement

query processing for flexible IR on XML documents, this paper proposes operators called SINGLECAT, MULTICAT, NESTCAT, and AUG which encapsulate the functionality for integrating IR statistics and inverted lists. The results of a composition of these operators are integrated statistics and inverted lists for flexible retrieval from XML documents. The following paragraphs give an overview about the operators and their signatures. Subsequent paragraphs in this section then discuss their implementation in more detail.

SINGLECAT returns the IR statistics and inverted list of a given basic indexing node under a particular query. SINGLECAT takes a path expression *expr* defining a basic indexing node and a set of query terms $\{term\}$ as input parameters. The signature of SINGLECAT is:

$$SINGLECAT(expr, \{term\}) \rightarrow (IL, STAT)$$

MULTICAT in turn takes several basic indexing nodes as input and integrates their statistics to the multi-category ones using Definition 5. MULTICAT has the following signature:

$$MULTICAT(\{(IL, STAT)\}) \rightarrow (IL, STAT)$$

NESTCAT computes the integrated IR statistics for sub-trees of XML collection structures. In contrast to MULTICAT, NESTCAT relies on Definition 3 and 7 to integrate statistics:

$$NESTCAT(\{(IL, STAT)\}) \rightarrow (IL, STAT)$$

Finally, the operator AUG downweighs term weights using the augmentation weights annotated to collection structure when propagating basic indexing node data upwards. The operator takes an inverted list and IR statistics as well as an augmentation weight *aw* as input parameters:

$$AUG((IL, STAT), aw) \rightarrow (IL, STAT)$$

The following example illustrates the combination of these operators in order to integrate inverted lists and IR statistics for flexible retrieval processing.

Example 5. Consider again the query 'XML Information Retrieval' on //book elements from the previous example in combination with the document collection underlying Figure 2. The operators SINGLECAT, MULTICAT, NESTCAT, and AUG integrate IR statistics and inverted lists as stored in the underlying basic indexing nodes for the scope according to the query. This yields the operator tree shown in Figure 3.

5.2 SQL-Implementation of Flexible Retrieval Operators

The following paragraphs explain in more detail how PowerDB-XML implements the operators for processing single-category retrieval, multi-category retrieval and nested retrieval using basic indexing nodes and standard SQL.

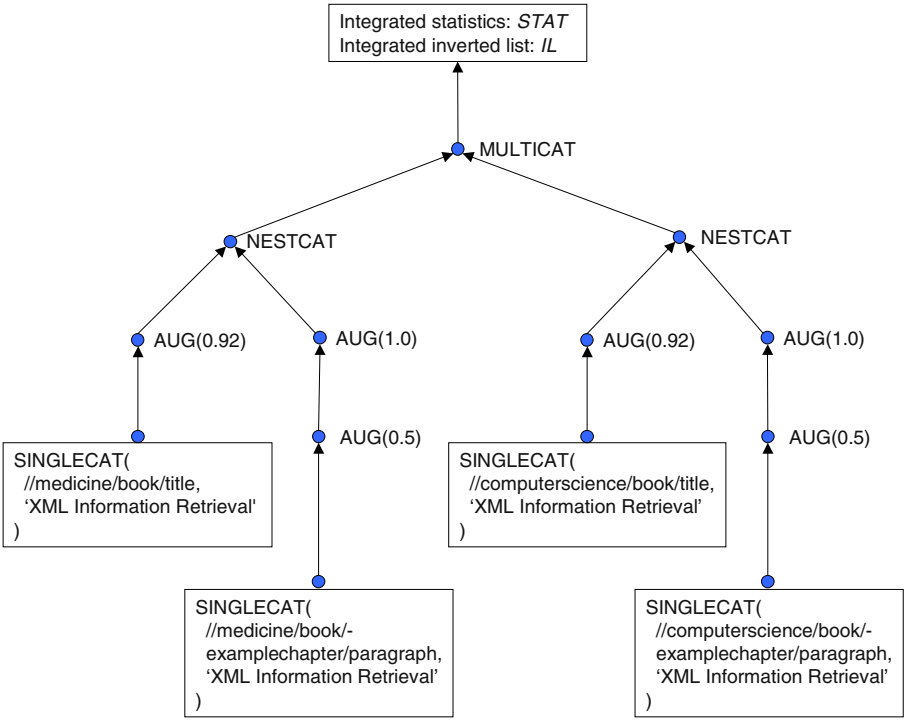


Fig. 3. Combination of retrieval types

Single-Category Retrieval Processing. Combining different retrieval types in an XML request requires to make the basic indexing node information available for further processing. The SINGLECAT operator works on the globalö and local statistics of a basic indexing node. The following SQL code shows how PowerDB-XML implements the SINGLECAT operator on input tables *IL* and *STAT*.

```
SELECT i.elem, i.term, i.tf INTO IL'
FROM IL i, query q WHERE i.term = q.term

SELECT s.term, s.ef INTO STAT'
FROM STAT s, query q WHERE s.term = q.term
```

Multi-category Retrieval Processing. Using basic indexing nodes directly for multi-category retrieval is not feasible since statistics are per basic indexing node. Hence, query processing must dynamically integrate the statistics when the query encompasses several categories.

MULTICAT relies on input provided by several – possibly parallel – invocations of the SINGLECAT operator. MULTICAT integrates their local and global

statistics. Note that a simple set union suffices to integrate the postings to the inverted lists since they only carry local statistics such as term frequencies while global IR statistics such as element frequencies require integration using Definition 5. The following SQL code shows how PowerDB-XML implements the MULTICAT operator on input tables IL_1, $STAT_1$, IL_2, and $STAT_2$.

```
SELECT i.elem, i.term, i.tf INTO IL′
FROM IL i, query q WHERE i.term = q.term
SELECT i.elem, i.term, i.tf INTO IL′
FROM IL i, query q WHERE i.term = q.term

SELECT s.term, SUM(s.ef) INTO STAT′
FROM (SELECT * FROM STAT1 UNION
      SELECT * FROM STAT2) s
GROUP BY s.term
```

Nested Retrieval Processing. The operator NESTCAT implements the functionality for integrating local and global statistics for nested retrieval. In contrast to MULTICAT, simple set union for the inverted lists does not suffice with nested retrieval. Instead, an aggregation of the term frequencies (tf) in the XML subtrees is required (cf. Def. 7). Note that the tf values are assumed to be properly augmented by previous invocations of the AUG operator. Hence, a simple SQL SUM suffices to integrate the term frequencies. The following SQL code shows how PowerDB-XML implements the NESTCAT operator on input tables IL_1, $STAT_1$, IL_2, and $STAT_2$.

```
SELECT e.elem, i.term, SUM(i.tf) INTO IL′
FROM elements e, IL1 i
WHERE DescendantOrSelf(e.elem, i1.elem)
SELECT e.elem, i.term, SUM(i.tf) INTO IL′
FROM elements e, IL2 i
WHERE DescendantOrSelf(e.elem, i2.elem)

SELECT s.term, COUNT(DISTINCT i.elem) INTO STAT′
FROM IL′
GROUP BY i.term
```

Note that repeated application of the binary versions of the operators implements the n-ary ones.

Processing of Augmentation. The operator AUG implements augmentation weighting for flexible retrieval processing. As the following SQL code shows, it simply returns a weighted projection of the local statistics of the input table IL which correspond to term frequencies with vector space retrieval. Global statistics are not affected by augmentation. Hence, AUG simply propagates them without any changes to subsequent operator instances.

```
SELECT elem, term, aw * tf INTO IL′
FROM IL
```

Computing Retrieval Status Values with Query Specific Statistics. Previous work has proposed implementations using standard SQL for data access with Boolean, vector space and probabilistic retrieval models [10,18]. Based on this work, the following SQL code for flexible XML retrieval using *tf idf* ranking takes integrated query-specific statistics *STAT* and *IL* from a composition of the operators discussed previously as an input.

```
SELECT elem, SUM(i.tf * ief(s.ef) * ief(s.ef) * q.tf) rsv
FROM IL i, STAT s, query q
WHERE i.term = s. term AND s.term = q.term
GROUP BY elem
```

The SQL statement yields the ranking, i.e., the XML element identifiers from the query scope and their retrieval status values.

Preliminary Experimental Results with INEX 2002. INEX – short for the INitiative for the Evaluation of XML retrieval – is an ongoing effort to benchmark the retrieval quality of XML retrieval systems [12]. INEX comes with a document collection of roughly 500 MB of XML documents representing about 12,000 IEEE Computer Society publications. Marked up in XML, the document collection comprises about 18.5 million XML elements. 60 different topics have been developed by the initiative, including relevance assessments. INEX differentiates between so-called *content-only (CO) topics* and *content-and-structure (CAS) topics*. CO topics specify a query text or a set of keywords for relevance-oriented search. Hence, each of the 18.5 million XML elements in the collection is a potential results to a CO topic. CAS topics in addition pose structural constraints such as path expressions on the result elements.

Using PowerDB-XML as retrieval engine, we have run the INEX 2002 benchmark on a single PC node with one 1.8 GHz Pentium processor, 512 MB RAM, and a 40 GB IDE disk drive. We deploy Microsoft Windows 2000 Server as operating system and Microsoft SQL Server 2000 for storage management. After having loaded PowerDB-XML with the complete document collection, we have run all topics and measured their response times. A positive finding from this series of experiments is that CAS topic processing is interactive, i.e., response times are in the order of seconds. However, some CO topics yield response times in the order of minutes (but less than 10 minutes). The reason for this is that CO topics require to compute a ranking for potentially all 18.5 million XML elements since constraints on the document structure to cut down the result space are not available with this topic type. The bottleneck of the topics with high response times is the inverted list lookup of terms in combination with processing element containment relationships. Note that the overhead of computing and integrating global statistics such as *ief* values is not significant with both topic types. We therefore plan to investigate combining query-specific statistics with more efficient representations of element containment relationships in relational database systems as discussed, e.g., in [19].

6 Conclusions

So far, database research has focused on data-centric processing of XML documents. This has left aside a large number of XML applications which require document-centric XML processing. An explanation for this might be that document-centric processing and in particular ranked and weighted retrieval from XML content is still an open research issue in the IR community. Our work on PowerDB-XML took over the vector-space model and *tf idf* ranking from flat document retrieval and refined it to flexible retrieval on XML documents using query-specific statistics. We see this as an important foundation for document-centric XML processing and we are currently evaluating its retrieval quality in the INEX initiative [11].

As our discussion has shown, flexible XML retrieval with query-specific statistics nicely maps to relational storage managers and an efficient implementation with standard SQL. This allows to combine state-of-the-art data-centric database mapping schemes with our relational implementation of document-centric XML processing. We pursue this approach in the PowerDB-XML project at ETH Zurich. As a result, our XML engine covers the full range from data-centric to document-centric XML applications on a single integrated platform using XML for data representation.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web – From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, 2000.
2. P. Bohannon, J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. LegoDB: Customizing Relational Storage for XML Documents. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB2002), August 20–23, 2002, Hongkong, China*, pages 1091–1094. Morgan Kaufmann, 2002.
3. P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML Schema to Relations: A Cost-based Approach to XML Storage. In *Proceedings of the 18th International Conference on Data Engineering (ICDE2002), February 26 - March 1, 2002, San Jose, CA, USA*. Morgan Kaufmann, 2002.
4. M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *Proceedings of 26th International Conference on Very Large Data Bases (VLDB2000), September 10–14, 2000, Cairo, Egypt*, pages 646–648. Morgan Kaufmann, 2000.
5. A. Deutsch, M. F. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1–3, 1999, Philadelphia, Pennsylvania, USA*, pages 431–442. ACM Press, 1999.
6. M. F. Fernandez, W. C. Tan, and D. Suciu. SilkRoute: Trading between Relations and XML. *WWW9 / Computer Networks*, 33(1-6):723–745, 2000.
7. D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

8. D. Florescu, D. Kossmann, and I. Manolescu. Integrating Keyword Search into XML Query Processing. In *Proceedings of the International WWW Conference, Amsterdam, May 2000*. Elsevier, 2000.
9. E. Fox and M. Koll. Practical Enhanced Boolean Retrieval: Experiments with the SMART and SIRE Systems. *Information Processing and Management*, 24(3):257–267, 1988.
10. O. Frieder, A. Chowdhury, D. Grossman, and M. McCabe. On the Integration of Structured Data and Text: A Review of the SIRE Architecture. In *Proceedings of the First DELOS Network of Excellence Workshop on Information Seeking, Searching and Querying in Digital Libraries, Zurich, Switzerland, 2000*, pages 53–58. ERCIM, 2000.
11. N. Fuhr, N. Gövert, G. Kazai, and M. Lalmas. INEX: Initiative for the Evaluation of XML Retrieval. In R. Baeza-Yates, N. Fuhr, and Y. S. Maarek, editors, *Proceedings of the ACM SIGIR Workshop on XML and Information Retrieval, Tampere, Finland*, pages 62–70. ACM Press, 2002.
12. N. Fuhr, N. Gövert, G. Kazai, and M. Lalmas, editors. *Proceedings of the First Workshop of the Initiative for the Evaluation of XML Retrieval (INEX), 9–11 December 2002, Schloss Dagstuhl, Germany*. ERCIM DELOS, 2003. ERCIM-03-W03.
13. N. Fuhr and K. Großjohann. XIRQL: A Query Language for Information Retrieval in XML Documents. In W. B. Croft, D. J. Harper, D. H. Kraft, and J. Zobel, editors, *Proceedings of the 24th Annual ACM SIGIR Conference on Research and Development in Information Retrieval, New Orleans, USA*, pages 172–180. ACM Press, 2001.
14. R. Goldman, J. McHugh, and J. Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *ACM SIGMOD Workshop on The Web and Databases (WebDB'99), June 3–4, 1999, Philadelphia, Pennsylvania, USA*, pages 25–30. INRIA, Informal Proceedings, 1999.
15. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of 23rd International Conference on Very Large Data Bases, 1997, Athens, Greece*, pages 436–445. Morgan Kaufmann, 1997.
16. T. Grabs, K. Böhm, and H.-J. Schek. PowerDB-IR – Information Retrieval on Top of a Database Cluster. In *Proceedings of the 10th International Conference on Information and Knowledge Management (CIKM2001), November 5–10, 2001 Atlanta, GA, USA*, pages 411–418. ACM Press, 2001.
17. T. Grabs and H.-J. Schek. Generating Vector Spaces On-the-fly for Flexible XML Retrieval. In R. Baeza-Yates, N. Fuhr, and Y. S. Maarek, editors, *Proceedings of the ACM SIGIR Workshop on XML and Information Retrieval, Tampere, Finland*, pages 4–13. ACM Press, 2002.
18. D. A. Grossman, O. Frieder, D. O. Holmes, and D. C. Roberts. Integrating Structured Data and Text: A Relational Approach. *Journal of the American Society for Information Science (JASIS)*, 48(2):122–132, Feb. 1997.
19. L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In A. Y. Halevy, Z. G. Ives, and A. Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9–12, 2003*, pages 16–27. ACM, 2003.
20. H. Kaufmann and H. J. Schek. Text Search Using Database Systems Revisited - Some Experiments -. In *Proceedings of the 13th British National Conference on Databases*, pages 18–20, 1995.

21. M. Rys. Bringing the Internet to Your Database: Using SQLServer 2000 and XML to Build Loosely-Coupled Systems. In *Proceedings of the 17th International Conference on Data Engineering, 2001, Heidelberg, Germany*, pages 465–472. IEEE Computer Society, 2001.
22. G. Salton. *The SMART Retrieval System : Experiments in Automatic Document Processing*. Prentice-Hall, 1971.
23. G. Salton, E. A. Fox, and H. Wu. Extended Boolean Information Retrieval. *Commun. ACM*, 26(12):1022–1036, 1983.
24. G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.
25. J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML Views of Relational Data. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *Proceedings of 27th International Conference on Very Large Data Bases, September, 2001, Roma, Italy*, pages 261–270. Morgan Kaufmann, 2001.
26. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *Proceedings of 25th International Conference on Very Large Data Bases (VLDB'99), September 7-10, 1999, Edinburgh, Scotland, UK*, pages 302–314. Morgan Kaufmann, 1999.
27. M. Volz, K. Aberer, and K. Böhm. Applying a Flexible OODBMS-IRS-Coupling for Structured Document Handling. In S. Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, New Orleans, Louisiana, USA*, pages 10–19. IEEE Computer Society, 1996.
28. W3C – World Wide Web Consortium (J. Clark, S. DeRose, editors). XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath`, Nov. 1999.
29. W3C – World Wide Web Consortium (S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Siméon, editors). XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery`, Nov. 2002.
30. W3C – World Wide Web Consortium (T. Bray, J. Paoli, C. M. Sperberg-McQueen, editors). Extensible Markup Language (XML) 1.0. `http://www.w3.org/TR/1998/REC-xml-19980210`, Feb. 1998.

An XML Repository Manager for Software Maintenance and Adaptation

Elaine Isnard[1], Radu Bercaru[2], Alexandra Galatescu[2], Vladimir Florian[2], Laura Costea[2], and Dan Conescu[2]

[1] Prologue Software/MEMSOFT Multilog Edition, Parc de Haute Technologie Bat 8, Avenue du Docteur Donat 06250 Mougins, Sophia-Antipolis, Nice, France
ielaine@proloque-software.fr
[2] National Institute for R&D in Informatics,
8-10 Averescu Avenue, 71316 Bucharest 1, ROMANIA
{radu,agal,vladimir,laura,dconescu}@.ici.ro

Abstract. In the framework of the IST project called MECASP (Maintenance and improvement of component-based applications diffused in ASP mode), an XML repository manager (RM) has been conceived, aiming at the maintenance and adaptation of heterogeneous software. It raises many problems and implies additional work for the connection of the involved open source software, for adding the missing functionality (usually with a high degree of complexity) and for wrapping it with MECASP-specific functionality. The repository contains versioned models in XML of the existing applications, subject to maintenance and adaptation (i.e. creation and management of their versions). The involved open source products are Castor, Xindice, XML:DB API, Slide. The paper briefly presents the place of RM in the overall architecture of MECASP, the open source-based architecture of the RM and the main problems which had to be solved for its implementation (correlated with MECASP specific features introduced at the beginning of the paper). The paper also presents the basic features of the version merger, a key tool in MECASP.

1 Introduction

The XML repository manager (RM) represents the core of the set of tools for software maintenance and adaptation, which will result from the ongoing European IST project, called MECASP (Maintenance and improvement of component-based applications diffused in ASP mode). The repository contains versioned models (descriptions in XML) of existing applications and resources, subject to maintenance and adaptation (i.e creation and management of their versions).

The implementation of the MECASP repository manager raises many problems and requires new solutions, with a high degree of complexity and difficulty, as this paper

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 118–132, 2003.

will try to reveal. The paper will focus on the issues related to the XML model-based solution for software maintenance and adaptation.

MECASP is mainly devoted to the maintenance of application software, but it will be useful for general purpose software as well. The main features of MECASP, that differentiate it from the existing version management products, are:

- *maintenance and adaptation of heterogeneous software* (built with heterogeneous development tools). Because of the XML-based approach of the software architecture, MECASP can be used to maintain/ adapt Java projects, objects in relational databases, forms, reports, graphical objects, documents etc;
- *versioning the XML descriptions of the software* (models in MECASP), tightly related to the changes in the corresponding pieces of software;
- *automatic merge of the versions* for heterogeneous software, relying on rule-based automatic decisions to solve inconsistencies;
- *synchronous and asynchronous multi-user work* on the same version of a MECASP-compliant software;
- *installation of new versions of a running application*, by the installation of changes relative to it;
- *uniform interface for editing heterogeneous types of resources*, using a MECASP-specific browser.

Similar issues for software maintenance or software deployment using XML-based software architecture descriptions are treated in other projects as well. The representation languages for software architectures (e.g. [12], [13], [14]) are now adapted to XML (e.g. [4], [5], [6]). Versioning and detecting changes in XML documents are still subject to debates (e.g. [7], [8], [9]). Transaction management and locking on XML databases (e.g. [26]) are still in an incipient research stage and do not have practical results.

Theoretical results have also been obtained in version management and merge algorithms (e.g. [15, [16], [17]), unfortunately without practical results and tools.

The software description using meta-models and models has also been experimented. The starting point and the inspiration source for MECASP was Oxygene ++ [18], devoted to application development relying on external descriptions (models) of the graphical objects and interfaces, of the menus, reports and databases..

PIROL (Project Integrating Reference Object Library) [19] relies on the Portable Common Tool Environment (PCTE), the reference model of ECMA [20]. Concerto [21] is a project for XML representation of parallel adaptive components. The components (described in XML) collect from the environment information on the hardware or software resources and dynamically create XML definitions for these resources. GraphTalk [22] is a graphical editor generator adapted to semi-formal and informal models. It builds meta-models relying on design technologies (e.g. UML).

Also, research results have been obtained for the automatic generation of XML documents from MOF meta-models [23], for the conversion between XML and relational schemas [24], for the semantic enhancement of the XML schemas [25], etc .

Most of today's research results treat separate aspects needed in MECASP: software architecture representation, software meta-modeling, generation of XML meta-

models from existing applications, version management and merge algorithms, management of the changes in XML documents, transaction management etc. MECASP is supposed to unify research results in all these fields.

Section 2 briefly describes the general architecture of MECASP. Section 3 gives a brief description of the XML model-based representation of the applications in MECASP. Section 4 presents the architecture and enumerates the most important problems that had to be solved for the implemetation of the open source-based XML Repository Manager in MECASP. Section 5 gives the basic features of the merger tool in MECASP.

2 General Architecture of MECASP

MECASP platform is divided into three layers (Figure 1):

- *browsing layer* with an advanced graphical interface.
- *tool layer*, composed of Configuration manager, a set of Editors, Security manager, Merger, Installer, Meta-model generator, Import tool, manager of the configuration files, wizards.
- *repository layer*, composed of the repository and its manager (basically composed of the meta-model manager and model manager). The repository is composed of application meta-models and models, briefly described in Section 3. The issues for the implementation of the repository manager will be described in Section 4.

The repository layer represents the core of MECASP. All tools communicate with the browser (in both directions) by means of RM. RM handles the storage of the XML models and meta-models and it also co-ordinates the actions of the other pieces of software in MECASP. It allows the extraction or building of an XML model or a part of a model. It gives information on the model, manages versions, workspaces and transactions. It sends 'Refresh' messages to the repository clients.

The *browser* instantiates four graphical clients (model explorer, object editor, property editor, action menu) which are graphical views on the information inside the XML repository. These clients allow the navigation through the model, object update in the model (e.g. move an object, update a property, etc) and launching actions or specific editors. The browser sends messages to RM specifying the types of changes the user wants to perform. RM processes the updates and replies with a 'refresh' message to all clients handling the updated nodes.

The *security manager* is accessed by RM in order to establish a user session or to check the access rights on the software objects handled by the user. When the user is versioning his work, a delta is calculated. The *delta* contains the user's changes on the current version of the XML model.

MECASP components are implemented in Java. *Scripts* are also written in Java, used to specialize MECASP generic behaviour. They execute code when an action is launched or when a trigger is fired.

An important role have the *converters*, with two functionalities (Figure 2):

- import of the user's files into an MECASP-compliant XML model. The files can contain descriptions of 4GLs, java code, forms, reports, database structures etc. The import has two steps: (1) conversion of the user's file into a standard XML file and (2) transformation of the XML file into an XML model in MECASP.
- generation of a file from a part of an MECASP-compliant XML model. When a user modifies an object (e.g. java source code, menu, screen etc), he needs to compile, execute or view the corresponding physical object. The generation process will convert the nodes in the model into the corresponding files.

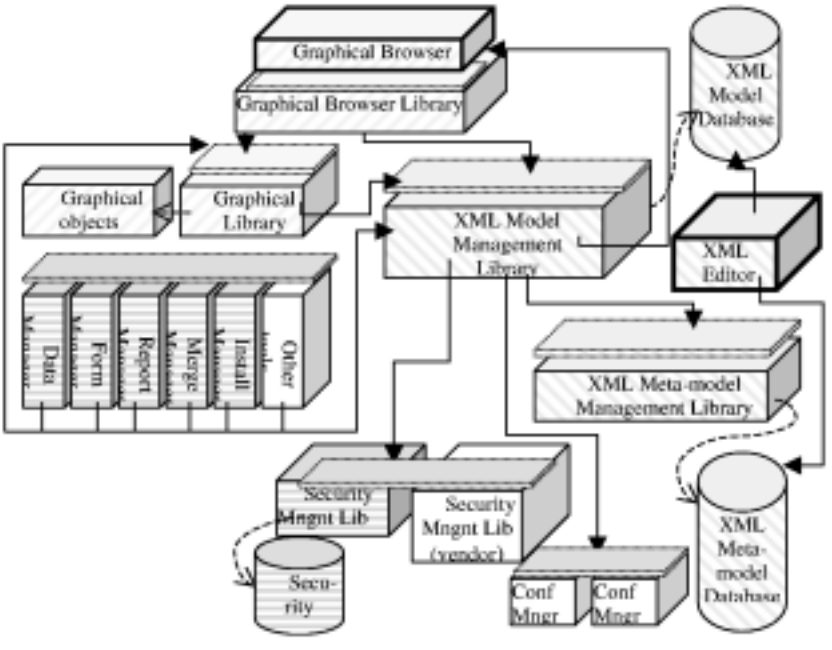


Fig. 1. General Architecture of MECASP

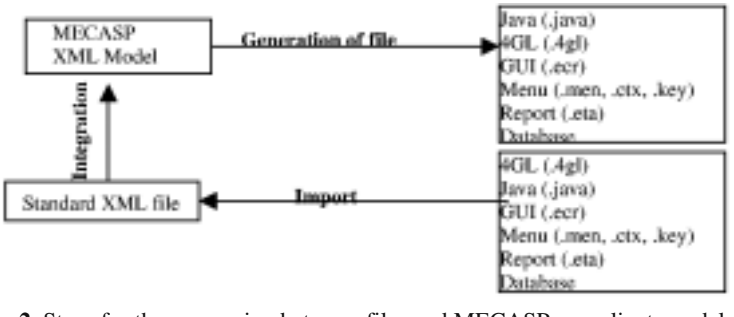


Fig. 2. Steps for the conversion between files and MECASP-compliant models

3 Versioned XML Models for Software Representation

The general architecture of MECASP in Figure 1 reveals the central place of the XML Model and Meta-model Managers in MECASP. They compose the XML Repository Manager (RM).

For each software (application), the XML repository contains the application model and the version graph for the application maintenance and adaptation. Each application is seen as a MECASP project.

The *representation of the application architecture* in MECASP has three levels:

- *XML schema,* that describes the general architecture for all types of software evolving under MECASP,
- *XML meta-models* (predefined or imported from existing development tools), complying with the general architecture in XML schema. The meta-models represent templates in XML for the types of projects allowed in MECASP: Oxygene++ projects, Java projects, database applications, graphical applications, Web applications etc or any combination of them.
- *XML models*, that describe general software or domain-specific applications maintained with MECASP. Each model represents a MECASP project and results from the customization of the definitions in a certain meta-model provided in MECASP.

The XML schema and XML meta-models are predefined in MECASP and the XML models are user-defined and application-specific.

The XML meta-models (and implicitly, the models) integrate data, control and presentation levels in the application architecture.

From the *data* point of view, the XML meta-models are *object-oriented*, hierarchically organized in XML. Their common schema was conceived as general as possible, in order to comply with any type of application. It contains the following types of elements: Project, Object, Property, Attribute.

The project is composed of objects, the objects are qualified by properties and can embed other objects (atomic or container-like objects) and the object properties are described and managed by attributes.

The relationships between objects are 'composition' and 'property inheritance'.

Figure 3 exemplifies several types of objects allowed in a MECASP meta-model for describing Oxygene++ projects.

From the *control* point of view, the XML meta-models are based on *actions*. MECASP manages predefined change actions upon application objects. These actions can be: (1) standard actions like "create", "delete", "update", "move" objects or properties or (2) non-standard actions like "compile" or "search and replace". In meta-models, the actions are represented as particular types of properties of the objects upon which they act. These actions can be executed by *scripts* or *external tools*, represented in the meta-models as particular types of objects.

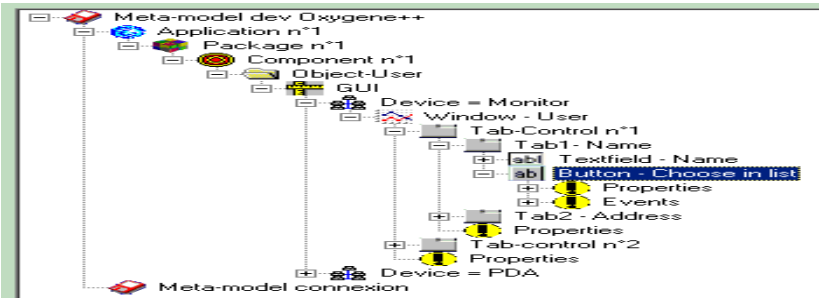


Fig. 3. Types of objects in Oxygene++ meta-model

From the *presentation* point of view, the objects are described in the meta-models by default values of their properties, that can be further changed by the users.

Each XML meta-model provided in MECASP can be further customized and transformed into models for domain specific applications. For one meta-model, many models can be built to create many different applications.

When an application is changed, the application model is changed as well and a new version is created and managed in the XML repository. On the developer's request, a new version can also be created by merging two existing versions of the application. Each new version of the XML model has attached a delta structure composed of all standard and non-standard change actions applied to the original version of the model.

4 An Open Source-Based Repository Manager

This section enumerates the open source products used for the implementation of RM and the most important problems that had to be solved during MECASP project, most of them from scratch. The architecture of RM is represented in Figure 4.

4.1 Open Source-Based Architecture of the Repository Manager

An implementation objective in MECASP is to rely on existing open source and, as much as possible, standard and portable software. For the implementation of the repository manager, four open source software products have been chosen:

- **Castor**, an open source data binding framework for Java [1]. It is used in MECASP for the mapping between (1) XML schema and Java classes and between (2) XML documents and Java objects;
- **Xindice**, a native XML (semi-structured) database server [2]. It is used in MECASP to store the XML meta-models and models, version graphs, etc.

- **XML:DB** API for Java development [10]. It is used as API for Xindice but, at the same time, it brings portability to the XML database in MECASP, because it is vendor neutral and may be used to access all the XML:DB compliant databases
- **Slide**, a Jakarta project [3] as framework for managing content.(of the software, in the case of MECASP). It can also provide a consistent interface to a variety of repositories or data stores: JDBC databases, source code or document repositories, file systems etc. Its methods for content and structure management and, also, for security and locking are wrapped with MECASP RM functionality.

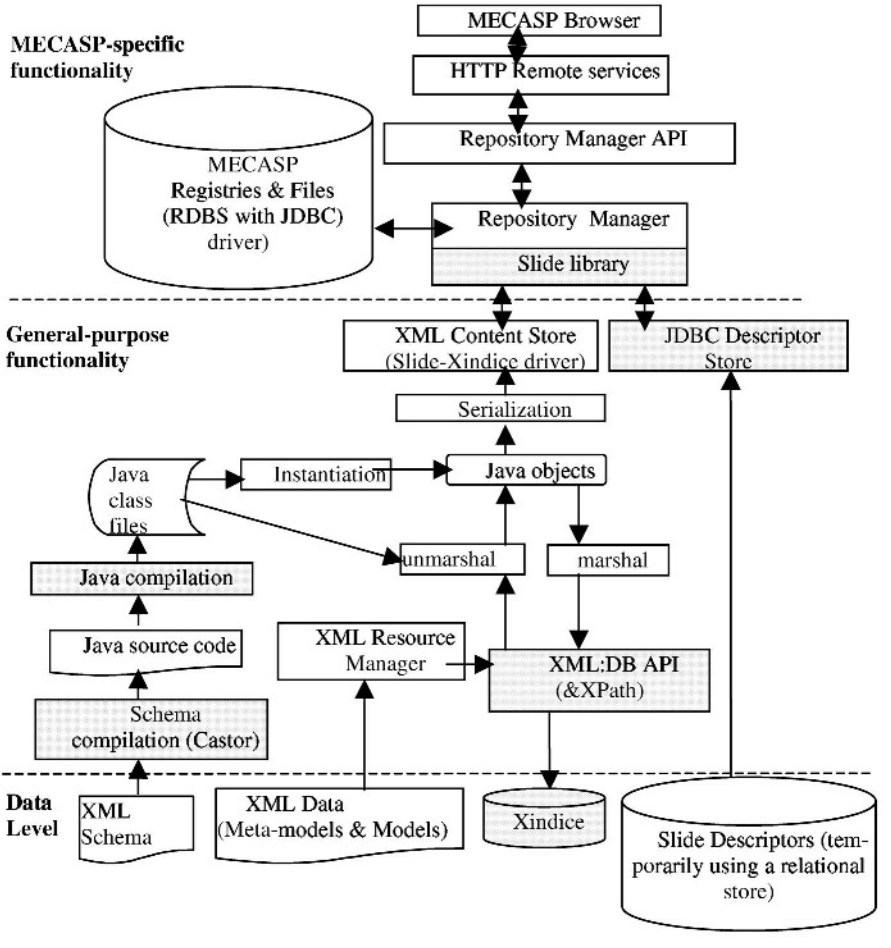


Fig. 4. Architecture of the Repository Manager in MECASP

4.2 Issues for the Implementation of the XML Repository Manager

The main problems for the implementation of the open source-based architecture of the repository manager, in correlation with the MECASP intended features (see Section 1) are briefly enumerated below.

Populating the XML Database with Meta-Models. Most complex meta-models in MECASP are obtained by the conversion from the definitions/ schemas of the existing applications/ resource types (e.g. a generic database schema, a generic Java project, graphical objects, etc). This conversion is accomplished in two phases: (1) conversion of the application schema into an XML document; (2) conversion of the XML document into a MECASP-specific XML meta-model.

Versioning Heterogeneous Resources. MECASP cannot benefit from existing version management tools like CVS or Microsoft VSS (VisualSource Safe), because (1) they deal with the versioning of text files only and (2) they have a primitive mechanism for tracking and merging changes.

For instance, in the CVS delta-like files (files that contain the differences between two versions of the same application), any change is tracked by a combination of the 'delete' and/ or 'append' operations. In the case of a database, these two operations are not appropriate to switch two columns, for example, in an already populated database, because the existing data will be lost during 'delete'. So, a 'move' operation was necessary, along with an algorithm for the semantic interpretation of the change (standard and non-standard) actions. Also, a MECASP specific delta representation and processing have been implemented in order to maintain non-text resources.

Delta Management. Slide helps manage versions of XML models, but does not help manage deltas (changes from the initial version to the new one).

In MECASP, there are two ways to define deltas: (1) *intensionally* (in each object description in the meta-model, by a property that defines a standard or non-standard type of change action), (2) *extensionally* (by aattaching all saved change actions on all objects in a project to the project model). A model in MECASP is stored along with the list of change actions, also represented in XML .

MECASP repository manager provides its own mechanism for delta management. The deltas are bi-directional and this mechanism allows the merge and version restoration in both directions (forward and backward), in comparison with the existing tools for version management, that allow only backward restoration of the versions.

Also, MECASP repository manager has its own mechanism for delta interpretation and merge. For example, suppose a field is deleted in a table. This action fires a trigger and launches a script that deletes the reference to the field in all windows in the application. In the delta, the delete action is stored only once. During the application

version restoration or during the merge process, the trigger is fired to adapt the change to the current context, depending on the object relationships.

Locking and Transaction Management on Hierarchical Objects. Because the multi-user work will be the basic work method with MECASP, it must implement powerful locking and transaction mechanisms.

The hierarchical representation of the application XML models leads to the need for specific mechanisms for locking and transaction management on XML hierarchical objects. These mechanisms are not implemented yet in the existing open source XML database servers (including Xindice) [26].

Consequently, for the open source-based MECASP repository manager, these mechanisms are now implemented at MECASP level, relying on Slide's functionality for locking, with a high degree of generality (in order to cope with a potential further substitution of Xindice by another XML database).

Synchronous and Asynchronous Multi-user Work on MECASP Projects. In MECASP, the implementation of the multi-user work is directed to:

- asynchronous sharing of the same project/ object version, by independent users. In this case, the save operations are independent and result into different versions of the project.
- synchronous sharing of the same project version, by the users of the same team. A Publish/Refresh mechanism is implemented to synchronize the work of all users. This is not appropriate while the users work on text documents (e.g. source code), when the first solution is is suitable. The source code should not be synchronized in real time (in standard cases) but, a web server or a database definition could be.

Because several tools can access the same resource, the locking must be implemented even in the single user mode in order to prevent data corruption.

The current implementation of the multi-user work on an XML database in MECASP relies on: (1) a MECASP locking mechanism, at node and document level, relying on Slide's functionality for locking, along with a two-phase locking (2PL) algorithm (not an XBMS included locking mechanism); (2) the implementation of a multi-user synchronous refresh mechanism; (3) the implementation of a mechanism for the multi-user work recovery from the server and repository manager crashes.

Installation of a New Version of a Running Application. Besides the initial installation of the repository and RM, using an existing installer, MECASP provides for the installation of a new version of a running application, by the installation of the changes relative to the schema of the original version. It uses the results of the merge operation, i.e the change files depending on the application type. For instance, for installing a new version of a database application, without interrupting the work with it, the following operations will be performed: (1) change the schema of the running application, by the execution of the SQL scripts resulting from the merge of the two versions of the application model; (2) import the data from the running version into

the new one; (3) discard the old version and start the new one. The schema transformation and the data import are supposed to run without schema inconsistencies (which have been solved during the previous merge operation).

Recovery from Crashes. Repository and RM crashes are prevented by a specific mechanism for: (1) the management of temporary files for the currently used versions and the current changes, not saved yet in the XML database; (2) the restoration of the user/ team working space.

5 Merge of Versions in MECASP

This section will briefly describe the basic functionality of a key tool in MECASP: the version merger.

Delta Representation. MECASP has its own representation and management strategy for the deltas (lists of change actions from an XML model version to another). Their representation is enhanced with the semantics of the change operations, as one may notice in the following examples of the change actions for: creation of a new object, modification of an object attribute or property attribute and move of an object. Also, in Figure 5, one may notice that the actions are bidirectionally described (by the new and old values), in order to allow the forward and backward merge.

Features of the Merger in MECASP. When the user changes the definition of a physical object (e.g. the schema of a table in a database), automatically the changes are reflected into the XML model of the respective object. The merge operation applies to the versions of the XML models, not to the versions of the physical objects and applications. This strategy allows the merge of versions for any type of objects, not only for text files. Other distinctive features of the MECASP-specific merger are:

- it semantically interprets and processes the change actions stored in deltas (e.g. move an object, delete an object, delete or change a property, etc);
- it implements an automatic rule-based decision mechanism for conflicts resolution. According to these rules, the list of change actions is simplified and the change operations of several users are chronologically interleaved. Special types of change operations (e.g. compile, search and replace etc), also tracked in deltas, are treated by specific merge rules.
- it creates change files, further used for the installation of a new version of a running application. These files depend on the type of application. For example, for a database application, they are SQL scripts and for a Java project, they might represent the new executable file.

```
<Obj id="12" refObjMM="107" parentId="7">
   <Attr name="type" value="delta" />
      <Attr name="Visible" value="false" />
      <Attr name="actionType"
            value="createObject" />
      <Attr name="propagate" value="" />
      <Attr name="behaviour" value="" />
      <Attr name="parentNodeId" value="725" />
      <Attr name="objectId" value="726" />
      <Attr name="refNodeMM" value="16" />
      <Attr name="objectType" value="general"/>
</Obj>

<Obj id="15" refObjMM="115" parentId="7">
   <Attr name="type" value="delta" />
      <Attr name="Visible" value="false" />
      <Attr name="actionType"
            value="setPropertyAttribute" />
      <Attr name="propagate" value="" />
      <Attr name="behaviour" value="" />
      <Attr name="objectId" value="726" />
      <Attr name="propertyName"
          value="g_width" />
     <Attr name="attributeName"value="value"/>
      <Attr name="oldValue" value="0" />
     <Attr  name="newValue"  value="100"  />
</Obj>
```

```
<Obj id="14" refObjMM="114" parentId="7">
   <Attr name="type" value="delta" />
      <Attr name="Visible" value="false" />
      <Attr name="actionType"
            value="setObjectAttribute" />
      <Attr name="propagate" value="" />
      <Attr name="behaviour" value="" />
      <Attr name="objectId" value="726" />
      <Attr name="propertyName" value="" />
      <Attr name="attributeName"
            value="content" />
      <Attr name="oldValue"
            value="PositionnerDebut" />
      <Attr name="newValue"
        value="BoiteMessage &quot;Operation
        impossible en mode visualisation&quot;"/>
</Obj>

<Obj id="16" refObjMM="109" parentId="7">
   <Attr name="type" value="delta" />
      <Attr name="Visible" value="false" />
      <Attr name="actionType"
          value="moveObject" />
      <Attr name="propagate" value="" />
      <Attr name="behaviour" value="" />
      <Attr name="objectId" value="726" />
      <Attr name="oldParentId" value="725" />
      <Attr name="newParentId" value="690" />
</Obj>
```

Fig. 5. Examples of definitions for the change actions in MECASP

Types of Merge. MECASP Merger implements two kinds of merge (Figure 6):

- *merge by integration*, when the merge of B with C is an adaptation of B.
- *complete merge*, when the new version is generated starting from A, by applying the changes in 'delta1' and 'delta2'. The new version D becomes the child of A (the parent of the merged versions).

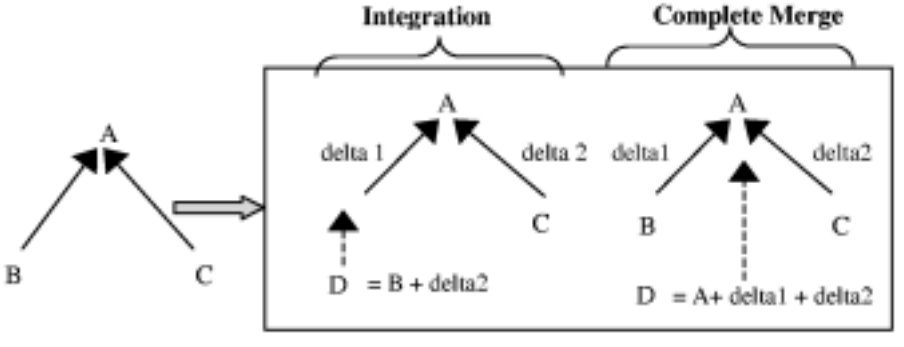


Fig. 6. Two kinds of merge: by integration and complete merge

Merge Process. The merge process is represented in Figure 7. Its main steps are:

- ***Choice of versions*:** The user chooses (1) the two versions of the model he wants to merge: the Donator and the Receptor [17], and (2) the *type of merge*: by integration or complete merge.
- ***Initialization*:** the Model Manager calculates the *nearest common parent version* and the *tree of change actions* from this common parent to the current state of the Donator and Receptor.

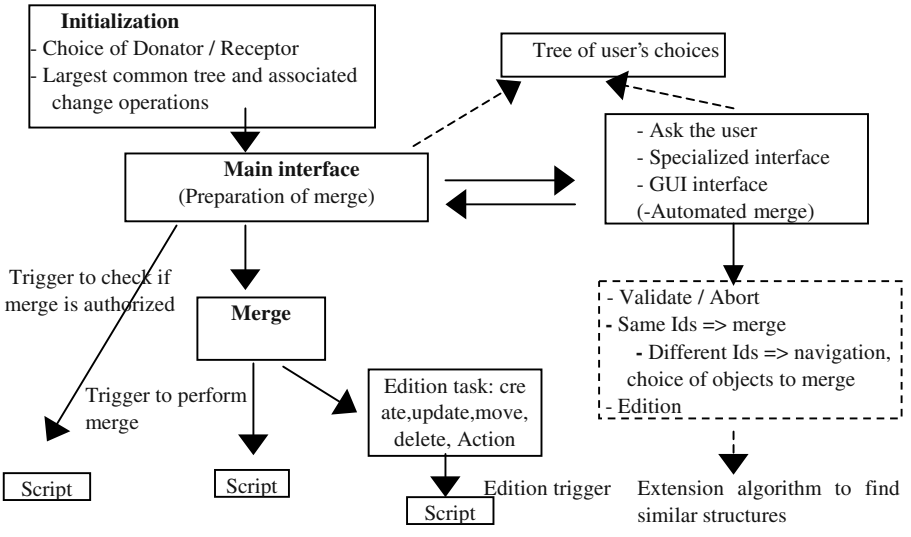


Fig. 7. The general merge process in MECASP

- ***Preparation and execution of the merge process* :**
  - o A graphical interface allows the user to plan the merge steps, to make choices and to launch the merge. The goal is to allow the user to plan most of the merge steps in advance but, sometimes, he must make choices interactively.
  - o The merge process in MECASP implies the execution of several ***steps*** :
    - *generation of the sequence of change actions*, by the generation of two lists of ordered actions, relying on the two trees of change actions attached to the Donator and Receptor;
    - *simplification of the action sequence in each list* by (1) marking the useless actions (e.g. repeated 'Compile' action on the same objects), (2) simplifying the sequence of change actions that occurred between two non-standard actions. The sequence of change actions on the same element is simplified according to predefined rules. For example, the sequence 'move'/'update' + 'delete' becomes 'delete', the sequence 'create + 'move'/'update' + 'delete' becomes an ineffective operation, etc.

- *synchronization of the change actions in the two lists*, by links between the equivalent standard or non-standard change actions on the same nodes.
- *conflict resolution*, depending on the type of objects intended for merge (see examples below);
- *calculation of priorities of the nodes/ change actions*. The priorities defined on Donator and Receptor's nodes are represented as properties in the model. They might alter the default behaviour of the merge process.
- *interactions with the user* in order to choose objects to merge, navigate into models, plan the merge process, specialize the merge process.
- *management of the tree of user's choices*.

The merger in MECASP has a default behaviour, expressed by three *default states* of the merge process: automatic merge, ask the user before merging, merge refused. A state depends on the type of change action that is going to be merged (create, delete, modify, etc) and on the type of objects it acts upon. Different *rules* are applied to calculate the default state of the merge process. Here are a few examples:

- an object is created in the Donator and it never existed in the Receptor. Default behaviour: *automatic merge* (the new object is automatically added into the Receptor).
- an object is deleted in the Donator. Default behaviour: *ask the user* working on the Receptor if he wants to delete this object in his version.
- a property of an object in the Donator is updated. There are two cases:
  - o if the property has never been modified in the Receptor then the default behaviour is *automatic merge* (it is automatically updated).
  - o if the property has been modified in the Receptor then the default behaviour is *ask the user* working on the Receptor if he wants to update this property.

The automatic decision during the conflict resolution and on the merge result relies on predefined rules that depend on the type of change actions, on the type of objects they act upon and on the role of the version in the merge operation: donator or receptor. These premises impact on the type of merge and on the *merge result*: fully automatic merge, merge after user decision, refused merge, recommended merge.

Examples of Conflict Resolution in MECASP. They are briefly enumerated below:

- ***Different types for a column in a table:*** When the user updates the table definition, if the same column in the two versions of the definition has different types, the merge process will choose (with the user's help) between the two types and will convert the data according to the chosen type.
- ***Different positions of a column in a table:*** The user can switch two columns, so that each column will have a different position in the two versions of the table definition. In order to avoid losing data (as with the classical 'delete' and 'append' actions of the other version management tools), the merge process

automatically saves the data set and rebuilds it correctly after the change of the table definition.

- ***Different lengths for a text field:*** If a text field has different lengths in the two versions of the model, the merger will automatically choose the maximum length.
- ***Same location in a window for two controls:*** When merging two windows, some controls can have the same location and, partly, they are not visible. The merge process detects such situations and, depending on the user's choice, automatically changes the position of the controls or asks the user to do it.***Search and replace text:*** This example shows that the order of the change actions is important. Suppose that the developer chooses 'Search and Replace' action to change all occurences of the word 'Window' into the word 'TextField', and then to change only one word 'Textfield' into 'Window'. So, there is only one word 'Window' left. But, if the developer changes one word 'Textfield' into 'Window' first and then he changes all words 'Window' into Textfield', there is no word 'Window' left. So, the information has been lost. For this reason, the merger in MECASP memorizes the order of these actions.

Graphical User Interfaces for Merge. These interfaces are :

- a *browser tree*, representing a model, with a red mark for the nodes in conflict. When clicking on this mark, a specialized panel is opened explaining the nature of the conflict and proposing the possible choices. This panel can call, via an interface, a specialized window to refine the choice. When a choice is made, the browser tree is updated and so are the edition panels.
- three grouped *edition panels*. Two panels represent the current state of the original versions (non editable) and the third one represents the merged version (editable). This third panel can be used to edit the new merged version during the merge process. When clicking on the red mark in the browser tree, the panels are also auto-positioned on the selected object.

Each time a choice is made, the tables of change actions in memory are recalculated, and the GUIs are refreshed automatically.

6 Conclusions

The paper first gives a brief presentation of the basic features and general architecture of MECASP. It then presents the application architecture in MECASP repository, represented by XML meta-models and models. The most important benefit drawn from the external description of an application is the possibility to maintain heterogeneous types of applications (in comparison with the existing tools for version management that maintain only text files).

The paper reveals the most important problems faced during the development of the open source-based XML repository manager. The solutions for most problems have been conceived and implemented from scratch in MECASP, as stated in Section 4.

The concept and basic features of the version merger are outlined, as it is a key powerful tool in the MECASP architecture, relying on the functionality provided by the XML repository manager

References

1. Exolab,"Castor project", http://castor.exolab.org/index.html
2. Apache, "Xindice Users Guide", http://xml.apache.org/xindice/
3. Jakarta, Slide project, http://jakarta.apache.org/
4. E.M. Dashofy, A.Hoek, R. N. Taylor, "A Highly-Extensible, XML-based Architecture Description Language", *Proc. of Working IEE/ IFIP Conference on Software Architecture*, 2001
5. E.M. Dashofy, "Issues in generating Data Bindings for an XML Schema-based Language", *Proc. of XML Technology and Software Engineering*, 2001
6. R.S.Hall, D. Heimbigner, A.L. Wolf, "Specifying the Deployable Software Description Format in XML", *CU-SERL-207-99*, University of Colorado
7. S-Y. Chien, V.J. Tsotras, C. Zaniolo, "Version Management of XML Documents", *Proc. of 3rd International Workshop on the Web and Databases (WebDB'2000)*, Texas, 2000. In conjunction with ACM SIGMOD'2000
8. A. Marian, S. Abiteboul, G. Cobena, L. Mignet, "Change-Centric Management of Versions in an XML Warehouse", *Proc. of 27th International Conference on Very Large DataBases (VLDB' 2001)*, Italy, 2001
9. Y. Wang, D. J. DeWitt, J.Cai, "X-Diff: An Effective Change Detection Algorithm for XML Documents", *Proc. of 19th International Conference on Data Engineering ICDE 2003*, March 5–8, 2003, Bangalore, India
10. XML:DB, "XML:DB Initiative", http://www.xmldb.org/
11. Cederqvist P. et al., "Version Management with CVS", http://www.cvshome.org/
12. Open Group, "Architecture Description Markup Language (ADML)", 2002, http://www.opengroup.org/
13. Kompanek A. "Modeling a System with Acme", 1998, http://www-2.cs.cmu./~acme/acme-home.htm
14. Garlan D., Monroe R., Wile D. (2000). "Acme: Architectural Description of Component-Based Systems". In *Foundations of Component-based Systems*, Cambridge University Press, 2000
15. Conradi R., Westfechtel B. "Version Models for Software Configuration Management". In *ACM Computing Surveys, Vol. 30, No. 2, June* 1998, http://isi.unil.ch/radixa/
16. Christensen H. B. "The Ragnarok Architectural Software Configuration Management Model". In *Proc. of the 32nd Hawaii International Conference on System Sciences,* 1999. http://www.computer.org/proceedings/
17. Christensen H. B. 99). Ragnarok: An Architecture Based Software Development Environment. In *PhD Thesis,* 1999, Centre for Experimental System Development Department of Computer Science University of Aarhus DK-8000 Århus C, Denmark. http://www.daimi.aau.dk/
18. Prologue-Software. "Documentation of Oxygene++". Technical documentation at Prologue Software/MEMSOFT Multilog Edition.

19. Groth B., Hermann S., Jahnichen S., Koch W. " PIROL: An object-oriented Multiple-View SEE". *Proc. of Software Engineering Environments Conference (SEE'95)*, Netherlands, 1995
20. ECMA (European Computer Manufacturers Association). "Reference Model for Frameworks of Software Engineering Environments". Technical Report, ECMA, 1993
21. Courtrai L., Guidec F., Maheo Y. " Gestion de ressources pour composants paralleles adaptables". *Journees "Composants adaptables"*, Oct. 2002, Grenoble
22. Parallax (Software Technologies). "GraphTalk Meta-modelisation Manuel de Reference, 1993
23. Blanc X., Rano A., LeDelliou. "Generation automatique de structures de documents XML a partir de meta-models MOF", Notere, 2000
24. Lee D., Mani M.,Chu W. W. "Efective Schema Conversions between XML and Relational Models", *Proc. European Conf. on Artificial Intelligence (ECAI), Knowledge Transformation Workshop*, Lyon, France, July, 2002.
25. Mani M., Lee D., Muntz R. R.. "Semantic Data Modeling using XML Schemas", *Proc. 20th Int'l Conf. on Conceptual Modeling (ER)*, Yokohama, Japan, November, 2001.
26. Helmer S., Kanne C., Moerkotte. "Isolation in XML Bases". Technical Report of The University of Mannheim, 2001.

XViz: A Tool for Visualizing XPath Expressions

Ben Handy and Dan Suciu

University of Washington Department of Computer Science
handyman@u.washington.edu
suciu@cs.washington.edu

Abstract. We describe a visualization tool for XPath expressions called XViz. Starting from a workload of XQueries, the tool extracts the set of all XPath expressions, and displays them together with some relationships. XViz is intended to be used by an XML database administrator in order to assist her in performing routine tasks such as database tuning, performance debugging, comparison between versions, etc. Two kinds of semantic relationships are computed and displayed by XViz, ancestor/descendant and containment. We describe an efficient, optimized algorithm to compute them.

1 Introduction

This paper describes a visualization tool for XPath expressions, called XViz. The tool starts from a workload of XQuery expressions, extracts all XPath expressions in the queries, then represents them graphically, indicating certain structural relationships between them. The goal is to show a global picture of all XPath expressions in the workload, indicating which queries use them, in what context, and how they relate to each other. The tool has been designed to scale to relatively large XQuery workloads, allowing a user to examine global structural properties, such as interesting clusters of related XPath expressions, outliers, or subtle differences between XPath expressions in two different workloads. XViz is not a graphical editor, i.e. it is not intended to create and modify queries.

The intended user of XViz is an XML database administrator, who could use it in order to perform various tasks needed to support applications with large XQuery workloads. We mention here some possible usages of XViz, without being exhaustive. One is to identify frequent common subexpressions used in the workload; such XPath expressions will be easily visible in the diagram produced by XViz because they have a long list of associated XQuery identifiers. Knowledge of the set of most frequent XPath expressions can be further used to manually select indexes, or to select efficient mappings to a relational schema. Clusters of almost identical XPath expressions can also be visually identified, giving the administrator more guidance in selecting indexes. A similar application consists of designing an efficient relational schema to store the XML data: a visual inspection of the graph produced by XViz can be used for that. Another application is to find performance bugs in the workload, e.g. redundant //'s or $*$'s. For example if both `/a/b/c` and `/a/b//c` occur in the workload then XViz will draw

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 134–148, 2003.

a line between them, showing the structural connection, and the administrator can then examine whether the // in the second expression is indeed necessary or is a typo. Finally, XViz can be used to study the relationship between two versions of the same workload, for example resulting from two different versions of an application. The administrator can either compare the graphs produced by XViz for the two different applications, or create a single graph with XPath expressions from both workloads, and see how the XPath expressions from the two versions relate.

In some cases there exist techniques that automatize some of these tasks: for example an approach for index selection is discussed in [1], and efficient mappings to relational storage are described in [3]. However, these techniques are quite complex, and by no means universally available. A lightweight tool like XViz, in the hands of a savvy administrator, can be quite effective. More importantly, like any visualization tool, XViz allows an administrator to see interesting facts even without requiring her to describe what she is looking for.

XViz starts from a text file containing a collection of XQueries, and extracts all XPath expressions occurring in the workload. This set may contain XPath expressions that are only implicitly, not explicitly used in the workload. For example, given the query:

```
for $x in /a/b[@c=3],
    $y in $x/d
. . .
```

XViz will display two XPath expressions: both `/a/b[@c=3]` (for `$x`) and `a/b[@c=3]/d` (for `$y`).

Next, XViz establishes two kinds of interesting relationships between the XPath expressions. The first is the *ancestor relationship*, checking whether the nodes returned by the first expression are ancestors of nodes returned by the second expression. The second is the *containment relationship*: this checks whether the answer set of one expression contains that for the other expression. Both relationships are defined semantically, not syntactically; for example XViz will determine that `/a//b[c//@d=3][@e=5]` is an ancestor of[1] `a/b[@e=5][@f=7][c/@d=3]/g/h`, even though they are syntactically rather different.

Finally, the graph thus computed is output in a dot file then passed to GraphViz, the graph visualization tool[2]. When the resulting graphs are large, they can easily clutter the screen. To avoid this, XViz provides a number of options for the user to specify what amount of detail to include.

The core of XViz is the module computing the relationships between XPath expressions, and we have put a lot of effort into making it as complete and efficient as possible. Recent theoretical work has established that checking containment of XPath expression is computationally hard, even for relatively simple

[1] We will define the ancestor/descendant relationship formally in Sec. 4.

[2] GraphViz is a free tool from AT&T labs, available at `http://www.research.att.com/sw/tools/graphviz/`.

fragments. As we show here, these hardness results also extend to the ancestor/descendant relationship. For example, checking for containment is co-NP complete, when the expressions are using `//`, `*`, and `[ ]` (predicates) [10]. When disjunctions or DTDs are added then the complexity becomes PSPACE complete, as shown in [11]; and it is even higher when joins are considered too [9]. For XViz we settled on an algorithm for checking the ancestor/descendant and the containment relationships that is quite efficient (it runs in time $O(mn)$ where m and n are the sizes of the two XPath expressions) yet as complete as possible, given the theoretical limitations; the algorithm is adapted from the homomorphism test described in [10].

Related work. Hy+ is a query and data visualization tool [8,7], used for object-oriented data. It is also a graphical editor. For XML languages, several graphical editors have been described. The earliest is for a graphical query language, XML-GL [5]. More recently, a few graphical editors for XQuery have been described. QURSED is a system that includes a graphical editor for XQuery [13], and XQBE is graphical editor designed specifically for XQuery [2].

What sets XViz aside from previous query visualization tools and query editors is its emphasis on finding and illustrating the semantics relationships between XPath expressions. For XML, this has been made possible only recently, through theoretical work that studied the containment problem for XPath expressions, in [9,10,11].

2 A Simple Example

To illustrate our tool and motivate the work, we show it here in action on a simple example. Consider a file `f.xquery` with the following workload:

```
Q1:

FOR $x in /a/b
WHERE sum($x/c) > 5
RETURN <result> $x/d </result>

Q2:

FOR $u in /a/b[c=6],
    $v in /a/b
WHERE $u/d > $v/c
RETURN $v/d
```

The tool is invoked like this:

```
xviz -i f.xquery -o f.eps -p -q
```

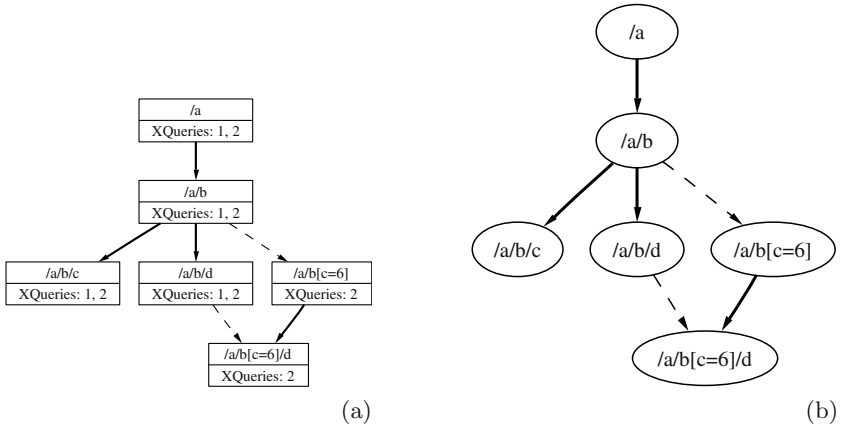


Fig. 1. Example: with XQuery IDs (a) and without (b).

This generates the output file `f.eps`, which is shown in in Fig. 1 (a). Notice that there is one node for each XPath expression in each query, and for each prefix of such an expression, with two kinds of edges: solid edges denote ancestor/descendant relationships and dashed edges denote containment relationships.

There are several flags for the command line that control what pieces of information is displayed. The flags are shown in Figure 2. For example, the first one, `-p`, determines only the XPath expression to be displayed, i.e. drops the XQuery identifiers. When used this way on our example, XViz produces the graph in Fig. 1 (b).

3 Architecture

The overall system architecture is shown in Fig. 3. The input consists of a text file containing a workload of XQuery expressions. The file does not need to contain pure XQuery code, but may contain free text or code in a different programming language, interleaved with XQuery expressions: this is useful for example in cases where the workload is extracted from a document or from an application. The XQuery workload is input to an XPath extractor that identifies and extracts all XPath expressions in the workload. The extractor uses a set of heuristics to identify the XQuery expressions, and, inside them the XPath expressions. Next, the set of XPath expressions are fed into the graph constructor. This makes several calls to the XPath containment algorithm (described in Sec. 4) in order to construct the graph to be displayed. Finally, the graph is displayed using GraphViz. A variety of output formats can be generated by GraphViz: postscript, gif, pdf, etc.

Flag	Meaning	Sample output
`-p`	displays the XPath expression	`/a/b[c=6]/d`
`-q`	displays the query where the expressions occurs	`XQuery: 2, 5, 9`
`-f`	displays for each query the FLWR statement where it occurs	`XQuery: 2(F), 5(W), 9(W)`
`-v`	displays the variable name that is bound to it	`XQuery: 2($x), 5(-), 9($y,-)`
`-b`	brief: do not include prefixes of the XPath expressions	
`-l`	display left-to-right (rather than top-to-bottom)	

Fig. 2. Flags used in conjunction with the `xviz` command.

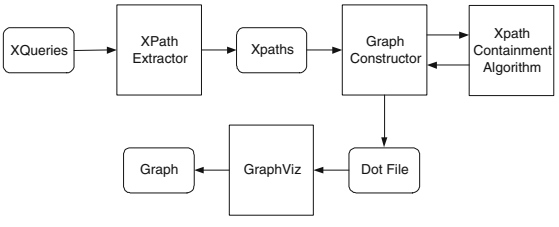


Fig. 3. The System's Architecture

4 Relationships between XPath Expressions

XViz computes the following two relationships between XPath expressions: ancestor/descendant, and containment. We define them formally below. Notice that both definitions are *semantic*, i.e. independent of the particular syntactic representation of the XPath expression. This is important for XViz applications, since it is precisely these hard to see semantic relationships that are important to show to the user.

We denote with p some XPath expression, and with t some XML tree. Then $p(t)$ denotes the set of nodes obtained by evaluating p on the XML tree t. We shall always assume that the evaluation starts at the root of the XML tree, i.e. all our XPath expressions start with /.

We denote nodes in t with symbols $x, y, \ldots$ If x is a proper ancestor of y then we write $x \ll y$: that is x can be y's parent, or its parent's parent, etc.

Ancestor/Descendant. We say that p' and p are in the ancestor/descendant relationship, denoted $p' \ll p$, if for every tree t and for any node $y \in p(t)$

there exists some node $x \in p'(t)$ such that $x \ll y$. Notice that the definition is semantic, i.e. in order to apply it directly one would need to check all possible XML trees t. We will give below a practical algorithm for checking $\ll$.

Example 1. The following are examples and counterexamples of ancestor/descendant relationships:

$$\begin{aligned} \texttt{/a/b[c=6]} &\ll \texttt{/a/b[c=6]/d} \\ \texttt{/a/b[c=6]} &\ll \texttt{/a/b[c=6]/d[e=9]/f} \\ \texttt{/a/b} &\not\ll \texttt{/a/b} \\ \texttt{/a//b} &\ll \texttt{/a/b[c=6]/d} \\ \texttt{/a/b[c=6]} &\not\ll \texttt{/a/b/d} \\ \texttt{/a//b[c//@d=3][@e=5]} &\ll \texttt{a/b[@e=5][@f=7][c/@d=3]/g/h} \end{aligned} \tag{1}$$

The first two examples should be clear. The third illustrates that the ancestor/descendant relationship is strict (i.e. anti-reflexive: $p \not\ll p$). The next example, (1), shows a particular choice we made in the definition of $\ll$. A node y returned by `/a/b[c=6]/d` always has an ancestor x (namely the `b` node) that is also returned by `/a//b`; but `/a//b` can return nodes that are not ancestors of any node satisfying `/a/b[c=6]/d`. The next two examples further illustrates this point. There are some theoretical arguments in favor of our choice of the definition (the elegant interaction between $\ll$ and $\supseteq$ defined below), but other choices are also possible.

Containment. We say that p' contains p, in notation $p' \supseteq p$, if for every XML tree t, the following inclusion holds: $p'(t) \supseteq p(t)$. That is, the set of nodes returned by p' includes all nodes returned by p. Notice that we place the larger expression on the left, writing $p' \supseteq p$ rather than $p \subseteq p'$ as done in previous work on query containment [9,10,11], because we want in the graph an arrow to go from the larger to the smaller expression.

Example 2. The following illustrate some examples of containment and non-containment:

$$\begin{aligned} \texttt{/a/b} &\supseteq \texttt{/a/b[c=6]} \\ \texttt{/a//e} &\supseteq \texttt{/a/b[c=6][d=9]/e} \\ \texttt{/a//*/e} &\supseteq \texttt{/a/*//e} \\ \texttt{/a/b[c=6]} &\supseteq \texttt{/a/b[c=6][d=9]} \\ \texttt{/a/b} &\not\supseteq \texttt{/a/b/c} \end{aligned}$$

Here too, the definition is semantic: we will show below how to check this efficiently. Notice that it is easy to check equivalence between XPath expressions by using containment: $p \equiv p'$ iff $p \supseteq p'$ and $p' \supseteq p$.

4.1 Reducing Ancestor/Descendant to Containment

The two relationships can be reduced to each other as follows:

$$[t]p' \ll p \Longleftrightarrow p'//* \supseteq p$$
$$p' \supseteq p \Longleftrightarrow p'/a \ll p/a/*$$

Here a is any tag name that does not occur in p'. We use the first reduction in order to compute $\ll$ using an algorithm for $\supseteq$. We use second reduction only for theoretical purposes, to argue that all hardness results for $\supseteq$ also apply to $\ll$. For example, for the fragment of XPath described in [10], checking the relationship $\ll$ is co-NP complete.

4.2 Computing the Graph

XViz uses the relationships $\ll$ and $\supseteq$ to compute and display the graph. A relationship $p' \ll p$ will be displayed with a solid edge, while $p' \supseteq p$ is displayed with a dashed edge.

Two steps are needed in order to compute the graph. First, identify equivalent expressions and collapse them into a single graph node. Two XPath expressions are equivalent, $p \equiv p'$ if both $p \supseteq p'$ and $p' \supseteq p$ hold. Once equivalent expressions are identified and removed, only $\supset$ relationships remain between XPath expressions.

Second, decide which edges to represent. In order to reduce clutter, redundant edges need not be represented. An edge is *redundant* if it can be inferred from other edges using one of the four implications below:

$$p_1 \supset p_2 \wedge p_2 \supset p_3 \Longrightarrow p_1 \supset p_3$$
$$p_1 \ll p_2 \wedge p_2 \ll p_3 \Longrightarrow p_1 \ll p_3$$
$$p_1 \ll p_2 \wedge p_2 \supset p_3 \Longrightarrow p_1 \ll p_3$$
$$p_1 \supset p_2 \wedge p_2 \ll p_3 \Longrightarrow p_1 \ll p_3$$

The first two implications state that both $\ll$ and $\supset$ are transitive. The last two capture the interactions between them.

Redundant edges can be naively identified with three nested loops, iterating over all triples (p_1, p_2, p_3) and marking the edge on the right hand side as redundant whenever the conditions on the left is satisfied. This method takes $O(n^3)$ steps, where n is the number of XPath expressions. We will discuss a more efficient way in Sec. 6.

5 An Application

We have experimented with XViz applied to three different workloads: the XMark benchmark [12], the XQuery Use Cases [6], and the XMach benchmark [4]. We describe here XMark only, which is shown in Fig. 4. The other

two are similar: we show a fragment of the XQuery Use cases in Fig. 5, but omit XMach for lack of space.

The result of applying XViz to the entire XMark benchmark[3] is shown in Fig. 4. It is too big to be readable in the printed version of this paper, but can be magnified when read online.

Most of the relationships are ancestor/descendant relationships. The root node `/` has one child, `/site`, which in turn has the following five children:

```
/site/people
/site//item
/site/regions
/site/open_auctions
/site/closed_auctions
```

Four of them correspond to the four children of `site` in the XML schema, but `/site//item` does not have a correspondence in the schema. We emphasize that, while the graph is somewhat related to the XML schema, it is different from the schema, and precisely these differences are interesting to see and analyze.

For example, consider the following chain in the graph:

```
/site ≪ /site//item
      ⊃ /site/regions//item
      ⊃ /site/regions/europe/item
      ≪ /site/regions/europe/item/name
```

Or consider the following two chains at the top of the figure, that start and end at the same node (showing that the graph is a DAG, not a tree):

```
/site/people/person ⊃ /site/people/person[@id='person0']
                    ≪ /site/people/person[@id='person0']/name
/site/people/person ≪ /site/people/person/name
                    ⊃ /site/people/person[@id='person0']/name
```

They both indicate relationships between XPath expressions that can be of great interest to an administrator, depending on her particular needs.

For a more concrete application, consider the expressions:

```
/site/people/person/name
/site/people/person[@id='person0']/name
```

The first occurs in XQueries 8, 9, 10, 11, 12, 17 is connected by a dotted edge (i.e. ⊃) to the second one, which also occurs in XQuery 1. Since they occur in relatively many queries, are good candidates for building an index. Another such candidate consists of `p` = `/site/closed_auctions/closed_auction`, which occurs in queries 5, 8, 9, 15, 16, together with several descendants, like `p/seller`, `p/price`, `p/buyer`, `p/itemref`, `p/annotation`.

[3] We omitted query 7 since it clutters the picture too much.

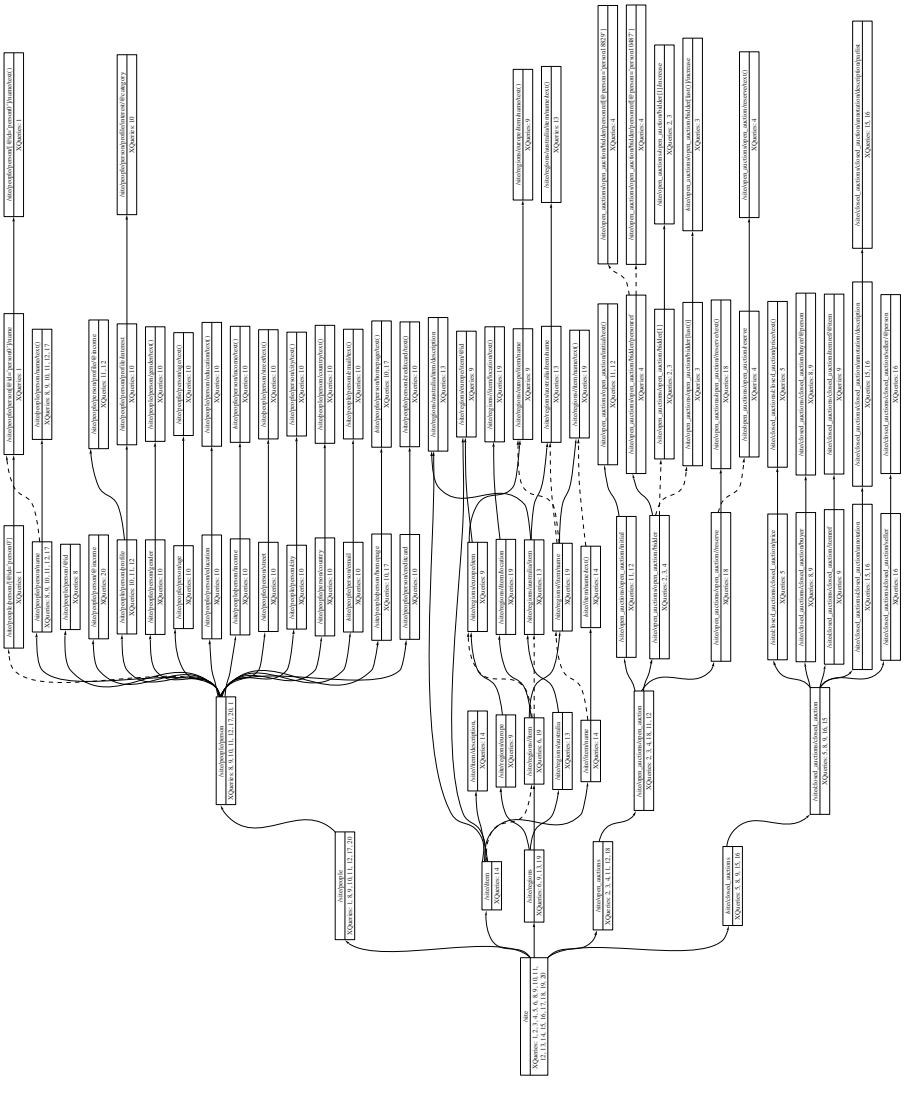


Fig. 4. XViz showing the entire XMark Benchmark workload.

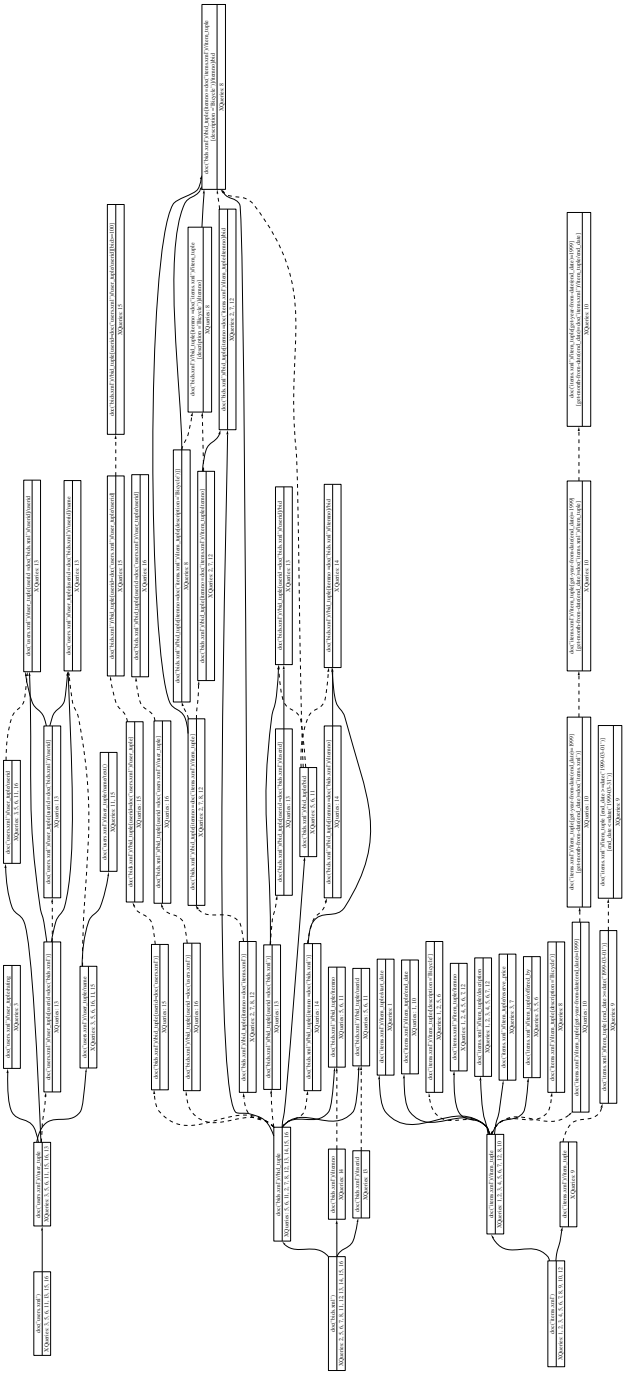


Fig. 5. XPath Expressions from the "R" Section of the XQuery Use Cases.

```
FlwrExpr ::= (ForClause | letClause)+ whereClause? returnClause
ForClause ::= 'FOR' Variable 'IN' Expr (',' Variable IN Expr)*
LetClause ::= 'LET' Variable ':=' Expr (',' Variable := Expr)*
WhereClause ::= 'WHERE' XPathText
ReturnClause ::= 'RETURN' XPathText
Expr ::= XPathExpr | FlwrExpr
```

Fig. 6. Simplified XQuery Grammar

6 Implementation

We describe here the implementation of XViz, referring to the Architecture in Fig. 3.

6.1 The XPath Extractor

The XPath extractor identifies XQuery expressions in a text and extracts as many XPath expressions from these queries as possible. It starts by searching for the keywords `FOR` or `LET`. The following text is then examined to see if a valid XQuery expression follows. We currently parse only a fragment of XQuery, without nested queries or functions. The grammar that we support is described in Fig. 6.

In this grammar, each Variable is assumed to start with a $ symbol and each XPathExpr is assumed to be a valid XPath expression. XPathText is a body of text, usually a combination of XML and expressions using XPaths, that we can extract any number of XPath expressions from. After an entire XQuery has been parsed, each XPath Expression is expanded by replacing all variables with their declared expressions. Once all XPath expressions have been extracted from a query, the Extractor continues to step through the text stream in search of XQuery expressions.

6.2 The XPath Containment Algorithm

The core of XViz is the XPath containment algorithm, checking whether $p' \supseteq p$ (recall that this is also used to check $p' \ll p$, see Sec. 4.1). If the XQuery workload has n XPath expressions, then the containment algorithm may be called up to $O(n^2)$ times (some optimizations may reduce this number however, see below), hence we put a lot of effort in optimizing the containment test. Namely, we checked containment using homomorphisms, by adapting the techniques in [10]. For presentation purposes we will restrict our discussion to the the XPath fragment consisting of tags, wildcards $*$, /, //, and predicates [ ], and mention below how we extended the basic techniques to other constructs.

Each XPath expression p is represented as a tree. A node, x, carries a label $\text{LABEL}(x)$, which can be either a tag or $*$; $\text{NODES}(p)$ denotes the set of nodes.

Edges are of two kinds, corresponding to / and to // respectively, and we denote $\text{EDGES} = \text{EDGES}_{/} \cup \text{EDGES}_{//}$.

A *homomorphism* from p' to p is a function from $\text{NODES}(p')$ to $\text{NODES}(p)$ that maps each node in p' to a matching node in p (i.e. it either has the same label, or the node in p' is $*$), maps an /-edge to an /-edge, and maps a //-edge to a path, and maps the return node in p' to the return node in p. Fig. 7 illustrates a homomorphism from $p' = /a/a[.//b]/*[c]//a/b$ to $p = /a/a/[.//c]/d[c]//a[a]/b$. Notice that the edge $a//b$ is mapped to the path $a/d//a/b$.

If there exists a homomorphism from p' to p then $p' \supseteq p$. This allows us to check containment by checking whether there exists homomorphism. This is done bottom-up, using dynamic programming. Construct a boolean table $\mathcal{C}$ where each entry $\mathcal{C}(x, y)$ for $x \in \text{NODES}(p), y \in \text{NODES}(p')$ contains 'true' iff there exists a homomorphism mapping y to x. The table $\mathcal{C}$ can be computed bottom up since $\mathcal{C}(x, y)$ depends only on the entries $\mathcal{C}(x', y')$ for y' a child of y and x' a child or a descendant of x. More precisely, $\mathcal{C}(x, y)$ is true iff $\text{LABEL}(y) = *$ or $\text{LABEL}(y) = \text{LABEL}(x)$ and, for every child y' of y the following conditions holds. If $(y, y') \in \text{EDGES}_{/}(p')$ then $\mathcal{C}(x', y')$ is true for some /-child of x:

$$\bigvee_{(x,x') \in \text{EDGES}_{/}(p)} \mathcal{C}(x', y')$$

If $(y, y') \in \text{EDGES}_{/}(p')$ then $\mathcal{C}(x', y')$ is true for some descendant x' of x:

$$\bigvee_{(x,x') \in \text{EDGES}^{+}(p)} \mathcal{C}(x', y') \tag{2}$$

Here $\text{EDGES}^{+}(p)$ denotes the transitive closure of $\text{EDGES}(p)$. This can be directly translated into an algorithm of running time $O(|p|^2|p'|)$.

Optimizations. We considered the following two optimizations.

The first addresses the fact that there are some simple cases of containment that have no homomorphism. For example there is no homomorphism from $/a//*/b$ to $/a/*//b$ (see Figure 8 (a)) although the two expressions are equivalent. For that we remove in p' any sequence of $*$ nodes connected by / or // edges and replace them with a single edge, carrying an additional integer label that represents the number of $*$ nodes removed. This is shown in Figure 8 (b). The label thus associated to an edge (y, y') is denoted $k(y, y')$. For example $k(y, y') = 1$ in Fig. 8 (b).

The second optimization reduces the running time to $O(|p||p'|)$. For that, we compute a second table, $\mathcal{D}(x, y')$, which records whenever there exists a descendant x' of x s.t. $\mathcal{C}(x', y')$ is true. Moreover, $\mathcal{D}(x, y')$ contains the actual distance from x to x'. Then, we can avoid a search for all descendants x' and replace Eq.(2) with the test $'D(x, y') \geq 1 + k(y, y')$. Both $\mathcal{C}(x, y)$ and $\mathcal{D}(x, y)$ can now be computed bottom up, in time $O(|p||p'|)$, as shown in Algorithm 1.

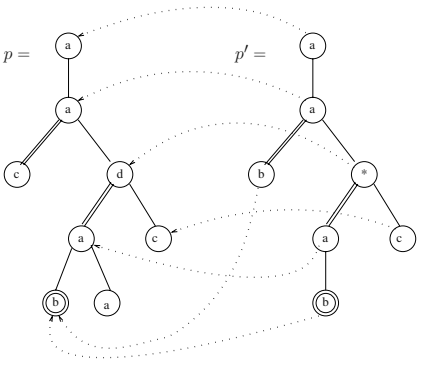


Fig. 7. Two tree patterns p, p' and a homomorphism from p' to p, proving $p' \supseteq p$.

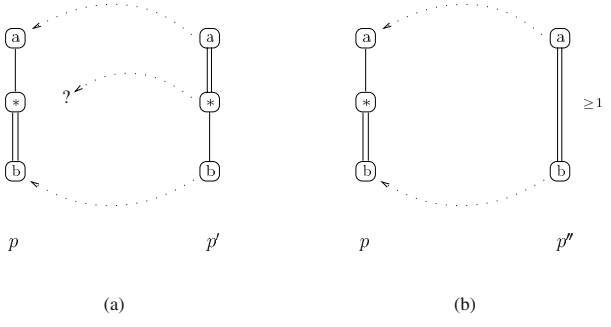


Fig. 8. (a) Two equivalent queries p, p' with no homomorphism from p' to p; (b) same queries represented differently, and a homomorphism between them.

Other XPath Constructs. Other constructs, like predicates on atomic values, `first()`, `last()` etc, are handled by XViz by extending the notion of homomorphism in a straightforward way. For example a node labeled `last()` has to be mapped into a node that is also labeled `last()`. Additional axes can be handled similarly. The existence of a homomorphism continues to be a sufficient, but not necessary condition for containment.

6.3 The Graph Constructor

The Graph Constructor takes a set of n XPath expressions, $p_1, \ldots, p_n$, computes all relationships $\ll$ and $\supseteq$, eliminates equivalent expressions, then computes a minimal set of solid edges (corresponding to $\ll$) and dashed edges (corresponding to $\supseteq$) needed to represent all $\ll$ and $\supseteq$ relationships, by using the four implications in Sec. 4.2.

Algorithm 1 Find homomorphism $p' \to p$

for x in NODES(p) **do** {The iteration proceeds bottom up on nodes of p}
  for y in NODES(p') **do** {The iteration proceeds bottom up on nodes of p'}
    compute $\mathcal{C}(x,y) = (\text{LABEL}(y) = \text{“}*\text{”} \vee \text{LABEL}(x) = \text{LABEL}(y)) \wedge$
      $\bigwedge_{(y,y') \in \text{EDGES}_{/}(p')} (\bigvee_{(x,x') \in \text{EDGES}_{/}(p)} \mathcal{C}(x',y')) \wedge$
      $\bigwedge_{(y,y') \in \text{EDGES}_{//}(p')} (\mathcal{D}(x,y') \geq 1 + k(y,y'))$
    if $\mathcal{C}(x,y)$ **then**
      $d = 0$;
    else
      $d = -\infty$
    compute $\mathcal{D}(x,y) = \max(d, 1 + \max_{(x,x') \in \text{EDGES}_{/}(p)} \mathcal{D}(x',y),$
      $1 + \max_{(x,x') \in \text{EDGES}_{//}(p)} (k(x,x') + \mathcal{D}(x',y)))$
return $\mathcal{C}(\text{ROOT}(p), \text{ROOT}(p'))$

A naive approach would be to call the containment test $O(n^2)$ times, in order to compute all relationships[4] $p_i \ll p_j$ and $p_i \supseteq p_j$, then to perform three nested loops to remove redundant relationships (as explained in Sec. 4.2), for an extra $O(n^3)$ running time.

To optimize this, we compute the graph G incrementally, by inserting the XPath expressions $p_1, \ldots, p_n$, one at a time. At each step the graph G is a DAG, whose edges are either of the form $p_i \ll p_j$ or $p_i \supset p_j$. Suppose that we have computed the graph G for $p_1, \ldots, p_{k-1}$, and now we want to add p_k. We search for the right place to insert p_k in G, starting at G's roots. Let G_0 be the roots of G, i.e. the XPath expressions that have no incoming edges. First determine if p_k is equivalent to any of these roots: if so, then merge p_k with that root, and stop. Otherwise determine whether there exists any edge(s) from p_k to some XPath expression(s) in G_0. If so, add all these edges to G and stop: p_k will be a new root in G. Otherwise, remove the root nodes G_0 from G, and proceed recursively, i.e. compare p_k with the new of roots in $G - G_0$, etc. When we stop, by finding edges from p_k to some p_i, then we also need to look one step “backwards” and look for edges from any parent of p_i to p_k. While the worst case running time remains $O(n^3)$, with $O(n^2)$ calls to the containment test, in practice this performs much better.

7 Conclusions

We have described a tool, XViz, to visualize sets of XPath expressions, together with their relationships. The intended use for XViz is by an XML database administrator, in order to assist her in performing various tasks, such as index selection, debugging, version management, etc. We put a lot of effort in making the tool scalable (process large numbers of XPath expressions) and usable (accept flexible input).

[4] Recall that $p_i \ll p_j$ is tested by checking the containment $p_i//* \supseteq p_j$.

We believe that a powerful visualization tool has great potential for the management of large query workloads. Our initial experience with standard workloads, like the XMark Benchmark, gave us a lot of insight about the structure of the queries. This kind of insight will be even more valuable when applied to workloads that are less well designed than the publicly available benchmarks.

References

1. S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 496–505. Morgan Kaufmann, 2000.
2. E. Augurusa, D. Braga, A. Campi, and S. Ceri. Design of a graphical interface to XQuery. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 226–231, 2003.
3. P. Bohannon, J. Freire, P. Roy, and J. Simeon. From xml schema to relations: A cost-based approach to xml storage. In *ICDE*, 2002.
4. T. Böhme and E. Rahm. Multi-user evaluation of XML data management systems with XMach-1. In *Proceedings of the Workshop on Efficiency and Effectiveness of XML Tools and Techniques (EEXTT)*, pages 148–158. Springer Verlag, 2002.
5. S. Ceri, S. Comai, E. Damiani, P. Fraternali, and S. Paraboschi. XML-gl: a graphical language for querying and restructuring XML documents. In *Proceedings of WWW8*, Toronto, Canada, May 1999.
6. D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: an XML query language, 2001. available from the W3C, `http://www.w3.org/TR/query`.
7. M. Consens, F. Eigler, M. Hasan, A. Mendelzon, E. Noik, A. Ryman, and D. Vista. Architecture and applications of the hy+ visualization system. *IBM Systems Journal*, 33:3:458–476, 1994.
8. M. P. Consens and A. O. Mendelzon. Hy: A hygraph-based query and visualization system. In *Proceedings of 1993 ACM SIGMOD International Conference on Management of Data*, pages 511–516, Washington, D. C., May 1993.
9. A. Deutsch and V. Tannen. Optimization properties for classes of conjunctive regular path queries. In *Proceedings of the International Workshop on Database Programming Lanugages*, Italy, Septmeber 2001.
10. G. Miklau and D. Suciu. Containment and equivalence of an xpath fragment. In *Proceedings of the ACM SIGMOD/SIGART Symposium on Principles of Database Systems*, pages 65–76, June 2002.
11. F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *International Conference on Database Theory*, 2003.
12. A. Schmidt, F. Waas, M. Kersten, D. Florescu, M. Carey, I. Manolescu, and R. Busse. Why and how to benchmark XML databases. *Sigmod Record*, 30(5), 2001.
13. V. V. Yannis Papakonstantinou, Michalis Petropoulos. QURSED: querying and reporting semistructured data. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 192–203. ACM Press, 2002.

Tree Signatures for XML Querying and Navigation

Pavel Zezula[1], Giuseppe Amato[2], Franca Debole[2], and Fausto Rabitti[2]

[1] Masaryk University, Brno, Czech Republic,
zezula@fi.muni.cz
http://www.fi.muni.cz
[2] ISTI-CNR, Pisa, Italy,
{Giuseppe.Amato,Franca.Debole,Fausto.Rabitti}@isti.cnr.it
http://www.isti.cnr.it

Abstract. In order to accelerate execution of various matching and navigation operations on collections of XML documents, new indexing structure, based on tree signatures, is proposed. We show that XML tree structures can be efficiently represented as ordered sequences of preorder and postorder ranks, on which extended string matching techniques can easily solve the tree matching problem. We also show how to apply tree signatures in query processing and demonstrate that a speedup of up to one order of magnitude can be achieved over the containment join strategy. Other alternatives of using the tree signatures in intelligent XML searching are outlined in the conclusions.

1 Introduction

With the rapidly increasing popularity of XML, there is a lot of interest in query processing over data that conforms to a labelled-tree data model. A variety of languages have been proposed for this purpose, most of them offering various features of a pattern language and construction expressions. Since the data objects are typically trees, the tree pattern matching and navigation are the central issues of the query execution.

The idea behind evaluating tree pattern queries, sometimes called the *twig queries*, is to find all the ways of embedding a pattern in the data. Because this lies at the core of most languages for processing XML data, efficient evaluation techniques for these languages require relevant indexing structures. More precisely, given a query twig pattern Q and an XML database D, a match of Q in D is identified by a mapping from nodes in Q to nodes in D, such that: (i) query node predicates are true, and (ii) the structural (ancestor-descendant and preceding-following) relationships between query nodes are satisfied by the corresponding database nodes. Though the predicate evaluation and the structural control are closely related, in this article, we mainly consider the process of evaluating the structural relationships, because indexing techniques to support efficient evaluation of predicates already exist.

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 149–163, 2003.

Available approaches to the construction of structural indexes for XML query processing are either based on mapping pathnames to their occurrences or on mapping element names to their occurrences. In the first case, entire pathnames occurring in XML documents are associated with sets of element instances that can be reached through these paths. However, query specifications can be more complex than simple path expressions. In fact, general queries are represented as pattern trees, rather than paths. Besides, individual path specifications are typically *vague* (containing for example *wildcards*), which complicates the matching. In the second case, element names are associated with *structured references* to the occurrences of names in XML documents. In this way, the indexed information is *scattered*, giving more freedom to ignore unimportant relationships. However, a document structure reconstruction requires expensive merging of lengthy reference lists through *containment joins*.

Contrary to the approaches that accelerate retrieval through the application of joins [10,1,2], we apply the *signature file* approach. In general, signatures are compact (small) representations of important features extracted from actual documents, created with the objective to execute queries on the signatures instead of the documents. In the past, see e.g. [9] for a survey, such principle has been suggested as an alternative to the *inverted file* indexes. Recently, it has been successfully applied to indexing of multi-dimensional vectors for similarity-based searching, image retrieval, and data mining.

We define the *tree signature* as a sequence of tree-node entries, containing node names and their structural relationships. In this way, incomplete tree inclusions can be quickly evaluated through extended string matching algorithms. We also show how the signature can efficiently support navigation operations on trees. Finally, we apply the tree signature approach to a complex query processing and experimentally compare such evaluation process with the structural join.

The rest of the paper is organized as follows. In Section 2, the necessary background is surveyed. The tree signatures are specified in Section 3. In Section 4, we show the advantages of tree signatures for XPath navigation, and in Section 5 we elaborate on the XML query processing application. Performance evaluation is described and discussed in Section 6. Conclusions and a discussion on alternative search strategies are available in Section 7.

2 Preliminaries

Tree signatures are based on a sequential representation of tree structures. In the following, we briefly survey the necessary background information.

2.1 Labelled Ordered Trees

Let Σ be an alphabet of size $|\Sigma|$. An ordered tree T is a rooted tree in which the children of each node are ordered. If a node $i \in T$ has k children then the children are uniquely identified, left to right, as $i_1, i_2, \ldots, i_k$. A labelled tree T

associates a label $t[i] \in \Sigma$ with each node $i \in T$. If the path from the root to i has length n, we say that $level(i) = n$. Finally, $size(i)$ denotes the number of descendants of node i – the size of any leaf node is zero. In the following, we consider ordered labelled trees.

2.2 Preorder and Postorder Sequences and Their Properties

Though there are several ways of transforming ordered trees into sequences, we apply the *preorder* and the *postorder* ranks, as recently suggested in [5]. The *preorder* and *postorder* sequences are ordered lists of all nodes of a given tree T. In a preorder sequence, a tree node v is traversed and assigned its (increasing) preorder rank, $pre(v)$, before its children are recursively traversed from left to right. In the postorder sequence, a tree node v is traversed and assigned its (increasing) postorder rank, $post(v)$, after its children are recursively traversed from left to right. For illustration, see the sequences of our sample tree in Fig. 1 – the node's position in the sequence is its preorder/postorder rank.

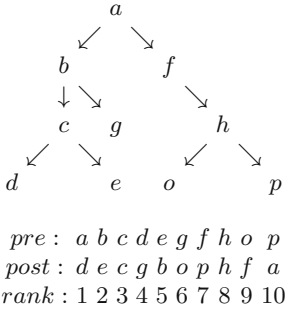


Fig. 1. Preorder and postorder sequences of a tree

Given a node $v \in T$ with $pre(v)$ and $post(v)$ ranks, the following properties are of importance to our objectives:

- all nodes x with $pre(x) < pre(v)$ are either the *ancestors* of v or nodes *preceding* v in T;
- all nodes x with $pre(x) > pre(v)$ are either the *descendants* of v or nodes *following* v in T;
- all nodes x with $post(x) < post(v)$ are either the *descendants* of v or nodes *preceding* v in T;
- all nodes x with $post(x) > post(v)$ are either the *ancestors* of v or nodes *following* v in T;
- for any $v \in T$, we have $pre(v) - post(v) + size(v) = level(v)$.

As proposed in [5], such properties can be summarized in a two dimensional diagram, as illustrated in Fig. 2, where the *ancestor* (A), *descendant* (D), *preceding* (P), and *following* (F) nodes of v are separated in their proper regions.

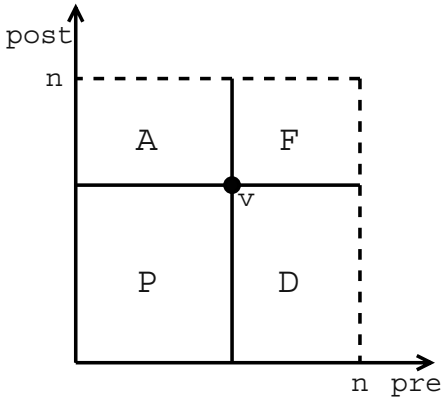


Fig. 2. Properties of the preorder and postorder ranks.

2.3 Longest Common Subsequence

The *edit distance* between two strings $x = x_1, \ldots, x_n$ and $y = y_1, \ldots, y_m$ is the minimum number of the *insert*, *delete*, and *modify* operations on characters needed to transform x into y. A dynamic programming solution of the edit distance is defined by an $(n+1) \times (m+1)$ matrix $M[\cdot, \cdot]$ that is filled so that for every $0 < i \leq n$ and $0 < j \leq m$, $M[i, j]$ is the minimum number of operations to transform $x_1, \ldots, x_i$ into $y_1, \ldots, y_j$.

A specialized task of the edit distance is the *longest common subsequence* (l.c.s.). In general, a *subsequence* of a string is obtained by taking a string and possibly deleting elements. If $x_1, \ldots, x_n$ is a string and $1 \leq i_1 < i_2 < \ldots < i_k \leq n$ is a strictly increasing sequence of indices, then $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ is a subsequence of x. For example, **art** is a subsequence of **a**lgo**r**i**t**hm. In the l.c.s. problem, given strings x and y we want to find the longest string that is a subsequence of both. For example, **art** is the longest common subsequence of **a**lgo**r**i**t**hm and p**ar**achu**t**e.

By analogy to edit distance, the computation uses an $(n + 1) \times (m + 1)$ matrix $M[\cdot, \cdot]$ such that for every $0 < i \leq n$ and $0 < j \leq m$, $M[i, j]$ contains the length of the l.c.s. between $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$. The matrix has the following definition:

- $M[i, 0] = M[0, j] = 0$, otherwise
- $M[i, j] = max\{M[i-1, j]; M[i, j-1];\ M[i-1, j-1] + eq(x_i, y_j)\}$, where $eq(x_i, y_j) = 1$ if $x_i = y_j$, $eq(x_i, y_j) = 0$ otherwise.

Obviously, the matrix can be filled in $O(n \cdot m)$ time. But algorithms such as [7] can find l.c.s. much faster.

The Sequence Inclusion. A string is *sequence-included* in another string, if their longest common subsequence is equal to the shorter of the strings. Assume

strings $x = x_1, \ldots, x_n$ and $y = y_1, \ldots, y_m$ with $n \leq m$. The string x is sequence-included in the string y if the l.c.s. of x and y is x. Note that sequence-inclusion and string-inclusion are different concepts. String x is included in y if characters of x occur contiguously in y, whereas characters of x might be interspersed in y with characters not in x for the sequence-inclusion. If string x is string-included in y, it is also sequence-included in y, but not the other way around.

For example, the matrix for searching the l.c.s. of "art" and "parachute" is:

	λ	p	a	r	a	c	h	u	t	e
λ	0	0	0	0	0	0	0	0	0	0
a	0	0	1	1	1	1	1	1	1	1
r	0	0	1	2	2	2	2	2	2	2
t	0	0	1	2	2	2	2	2	3	3

Using the l.c.s. approach, one string is sequence-included in the other if $M[n, m]$-$= min\{m, n\}$. Because we do not have to compute all elements of the matrix, the complexity is $O(p) \mid p = max\{m, n\}$.

3 Tree Signatures

The idea of the tree signature is to maintain a small but sufficient representation of the tree structures, able to decide the tree inclusion problem as needed for XML query processing. We use the preorder and postorder ranks to linearize the tree structures, which allows to apply the sequence inclusion algorithms for strings.

3.1 The Signature

The tree signature is an ordered list (sequence) of pairs. Each pair contains a tree node name along with the corresponding postorder rank. The list is ordered according to the preorder rank of nodes.

Definition 1. *Let T be an ordered labelled tree. The signature of T is a sequence, $sig(T) = \langle t_1, post(t_1); t_2, post(t_2); \ldots; t_m, post(t_m)\rangle$, of $m = |T|$ entries, where t_i is a name of the node with $pre(t_i) = i$. The $post(t_i)$ is the postorder value of the node named t_i and the preorder value i.*

Observe that the index in the signature sequence is the node's preorder, so the value serves actually two purposes. In the following, we use the term preorder if we mean the rank of the node, when we consider the position of the node's entry in the signature sequence, we use the term index. For example, $\langle a, 10; b, 5; c, 3; d, 1; e, 2; g, 4; f, 9; h, 8; o, 6; p, 7\rangle$ is the signature of the tree from Fig. 1. By analogy, tree signatures can also be constructed for query trees, so $\langle h, 3; o, 1; p, 2; \rangle$ is the signature of the query tree from Fig. 3.

A sub-signature $sub\_sig_S(T)$ is a specialized (restricted) view of T through signatures, which retains the original hierarchical relationships of nodes in T.

Considering $sig(T)$ as a sequence of individual entries representing nodes of T, $sub\_sig_S(T) = \langle t_{s_1}, post(t_{s_1}); t_{s_2}, post(t_{s_2}); \ldots; t_{s_k}, post(t_{s_k})\rangle$ is a sub-sequence of $sig(T)$, defined by the ordered set $S = \{s_1, s_2, \ldots, s_k\}$ of indexes (preorder values) in $sig(T)$, such that $1 \leq s_1 < s_2 < \ldots < s_k \leq m$. For example, the set $S = \{2, 3, 4, 5, 6\}$ defines a sub-signature representing the subtree rooted at the node b of our sample tree.

Tree Inclusion Evaluation. Suppose the data tree T specified by signature

$$sig(T) = \langle t_1, post(t_1); t_2, post(t_2); \ldots; t_m, post(t_m)\rangle,$$

and the query tree Q defined by its signature

$$sig(Q) = \langle q_1, post(q_1); q_2, post(q_2); \ldots; q_n, post(q_n)\rangle.$$

Let $sub\_sig_S(T)$ be the sub-signature of $sig(T)$ induced by a sequence-inclusion of $sig(Q)$ in $sig(T)$, just considering the equality of node names. The following lemma specifies the tree inclusion problem precisely.

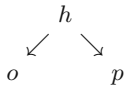
Lemma 1. *The query tree Q is included in the data tree T if the following two conditions are satisfied: (1) on the level of node names, $sig(Q)$ is sequence-included in $sig(T)$ determining $sub\_sig_S(T)$ through the ordered set of indexes $S = \{s_1, s_2, \ldots, s_n\} | q_1 = t_{s_1}, q_2 = t_{s_2}, \ldots, q_n = t_{s_n}$, (2) for all pairs of entries i and $i + j$ in $sig(Q)$ and $sub\_sig_S(T)$ $(i, j = 1, 2, \ldots |Q| - 1$ and $i + j \leq |Q|)$, $post(q_{i+j}) > post(q_i)$ implies $post(t_{s_{i+j}}) > post(t_{s_i})$ and $post(q_{i+j}) < post(q_i)$ implies $post(t_{s_{i+j}}) < post(t_{s_i})$.*

Proof. Because the index i increases in (sub-)signatures according to the preorder rank, node $i + j$ must be either the descendent or the following node of i. If $post(q_{i+j}) < post(q_i)$, the node $i+j$ in the query is a descendent of the node i, thus also $post(t_{s_{i+j}}) < post(t_{s_i})$ is required. By analogy, if $post(q_{i+j}) > post(q_i)$, the node $i+j$ in the query is a following node of i, thus also $post(t_{s_{i+j}}) > post(t_{s_i})$ must hold.

A specific query signature can determine zero or more data sub-signatures. Regarding the node names, any $sub\_sig_S(T) \equiv siq(Q)$, because $q_i = t_{s_i}$ for all i, see point (1) in Lemma 1. But the corresponding entries can have different postorder values, and not all such sub-signatures necessarily represent qualifying patterns, see point (2) in Lemma 1.

The complexity of tree inclusion algorithm according to Lemma 1 is $\sum_{i=1}^{n-1} i$ comparisons. Though the number of the query tree nodes is usually not high, such approach is computationally feasible. Observe that Lemma 1 defines the *weak inclusion* of the query tree in the data tree, in the sense that the parent-child relationships of the query are implicitly reflected in the data tree as only the ancestor-descendant. However, due to the properties of preorder and postorder ranks, such constraints can easily be strengthened, if required.

For example, consider the data tree T in Fig. 1 and the query tree Q in Fig. 3. Such query qualifies in T, i.e. $sig(Q) = \langle h, 3; o, 1; p, 2\rangle$ determines a



$sig(Q) = \langle h, 3; o, 1; p, 2\rangle$

Fig. 3. Sample query tree Q

compatible $sub\_sig_S(T) = \langle h, 8; o, 6; p, 7\rangle$ through the ordered set $S = \{8, 9, 10\}$, because (1) $q_1 = t_8$, $q_2 = t_9$, and $q_3 = t_{10}$, (2) the postorder of node h is higher than the postorder of nodes o and p, and the postorder of node o is smaller than the postorder of node p (both in $sig(Q)$ and $sub\_sig_S(T)$). If we change in our query tree Q the lable h for f, we get $sig(Q) = \langle f, 3; o, 1; p, 2\rangle$. Such a modified query tree is also included in T, because Lemma 1 does not insist on the strict parent-child relationships, and implicitly consider all such relationships as ancestor-descendant. However, the query tree with the root g, resulting in $sig(Q) = \langle g, 3; o, 1; p, 2\rangle$, does not qualify, even though the query signature is also sequence-included (on the level of names) determining the sub-signature $sub\_sig_S(T) = \langle g, 4; o, 6; p, 7\rangle | S = \{6, 9, 10\}$. The reason for the false qualification is that the query requires the postorder to go down from node g to o (from 3 to 1) , while in the sub-signature it actually goes up (from 4 to 6). That means that o is not a descendant node of g, as required by the query, which can be verified in Fig. 1.

Extended Signatures. In order to further increase the efficiency of various matching and navigation operations, we also propose the *extended signatures*. For motivation, see the sketch of a signature in Fig. 4, where A, P, D, F represent

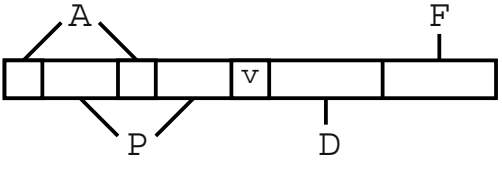


Fig. 4. Signature structure

areas of ancestor, preceding, descendant, and following nodes with respect to the generic node v. Observe that all descendants are on the right of v before the following nodes of v. At the same time, all ancestors are on the left of v, acting as separators of subsets of preceding nodes. This suggests to extend entries of tree signatures by two preorder numbers representing pointers to the *first following*, ff, and the *first ancestor*, fa, nodes. The general structure of the extended signature of tree T is

$$sig(T) = \langle t_1, post(t_1), ff_1, fa_1; t_2, post(t_2), ff_2, fa_2; \ldots; t_m, post(t_m), ff_m, fa_m \rangle,$$

where ff_i (fa_i) is the preorder value of the first following (ancestor) node of that with the preorder rank i. If no terminal node exists, the value of the first ancestor is zero and the value of the first following node is $m+1$. For illustration, the extended signature of the tree from Fig. 1 is

$$sig(T) = \langle a, 10, 11, 0; b, 5, 7, 1; c, 3, 6, 2; d, 1, 5, 3; e, 2, 6, 3;$$

$$g, 4, 7, 2; f, 9, 11, 1; h, 8, 11, 7; o, 6, 10, 8; p, 7, 11, 8 \rangle$$

Given a node with index i, the cardinality of the descendant node set is $size(i) = ff_i - i - 1$, and the level of the node with index i is $level(i) = i - post(i) + ff_i - i - 1 = ff_i - post(i) - 1$. Further more, the tree inclusion problem can be solved in linear time, as the following lemma obviates.

Lemma 2. *Using the extended signatures, the query tree Q is included in the data tree T if the following two conditions are satisfied: (1) on the level of node names, $sig(Q)$ is sequence-included in $sig(T)$ determining $sub\_sig_S(T)$ through the ordered set of indexes $S = \{s_1, s_2, \ldots s_n\} | q_1 = t_{s_1}, q_2 = t_{s_2}, \ldots, q_n = t_{s_n}$, (2) for $i = 1, 2, \ldots |Q| - 1$, if $post(q_i) < post(q_{i+1})$ (leaf node, no descendants, the next is the following) then $ff(t_{s_i}) \leq s_{i+1}$, otherwise (there are descendants) $ff(t_{s_i}) > s_{ff(q_i)-1}$.*

4 Evaluation of XPath Expressions

XPath [3] is a language for specifying navigation within an XML document. The result of evaluating an XPath expression on a given XML document is a set of nodes stored according to document order, so we can say that the result nodes are selected by an XPath expression.

Within an XPath `Step`, an `Axis` specifies the *direction* in which the document should be explored. Given a context node v, XPath supports 12 axes for navigation. Assuming the context node is at position i in the signature, we describe how the most significant axes can be evaluated through the extended signatures, using the tree from Fig. 1 as reference:

Child. The first child is the first descendant, that is a node with index $i + 1$ such that $post(i) > post(i + 1)$. The second child is indicated by pointer ff_{i+1}, provided the value is smaller than ff_i, otherwise the child node does not exist. All the other children nodes are determined recursively until the bound ff_i is reached. For example, consider the node b with index $i = 2$. Since $ff_2 = 7$, there are 4 descending nodes, so the node with index $i+1 = 3$ (i.e. node c) must be the first child. The first following pointer of c, $ff_{i+1} = 6$, determines the second child of b (i.e. node g), because $6 < 7$. Due to the fact that $ff_6 = ff_i = 7$, there are no other child nodes.

Descendant. The descendant nodes (if any) start at position $i+1$, and the last descendant object is at position $ff_i - 1$. If we consider node b (with $i = 2$), we immediately decide that the descendants are at positions starting from $i + 1 = 3$ to $ff_2 - 1 = 6$, i.e. nodes c, d, e, and g.

Parent. The parent node is directly given by the pointer fa. The **Ancestor** axis is just a recursive closure of **Parent**.

Following. The following nodes of the reference at position i (if they exist) start at position ff_i and include all nodes up to the end of the signature sequence. All nodes following c (with $i = 3$) are in the suffix of the signature starting at position $ff_3 = 6$.

Preceding. All preceding nodes are on the left of the reference node as a set of intervals separated by the ancestors. Given a node with index i, fa_i points to the first ancestor (i.e. the parent) of i, and the nodes (if they exist) between i and fa_i precede i in the tree. If we recursively continue from fa_i, we find all the preceding nodes of i. For example, consider the node g with $i = 6$: following the ancestor pointer, we get $fa_6 = 2, fa_2 = 1, fa_1 = 0$, so the ancestors nodes are b and a, because $fa_1 = 0$ indicates the root. The preceding nodes of g are only in the interval from $i - 1 = 5$ to $fa_6 + 1 = 3$, i.e. nodes c, d, and e.

Following-sibling. In order to get the following siblings, we just follow the ff pointers while the following objects exist and the fa pointers are the same as fa_i. For example, given the node c with $i = 3$ and $fa_3 = 2$, the ff_3 pointer moves us to the node with index 6, that is the node g. The node g is the sibling following c, because $fa_6 = fa_3 = 2$. But this is also the last following sibling, because $ff_6 = 7$ and $fa_7 \neq fa_3$.

Preceding-sibling. All preceding siblings must be between the context node with index i and its parent with index $fa_i < i$. The first node after the i-th parent, which has the index $fa_i + 1$, is the first sibling. Then use the **Following-sibling** strategy up to the sibling with index i. Consider the node f ($i = 7$) as the context node. The first sibling of the i-th parent is b, determined by pointer $fa_7 + 1 = 2$. Then the pointer $ff_2 = 7$ leads us back to the context node f, so b is the only preceding sibling node of f.

Observe that the postorder values, $post(t_i)$, are not used for navigation, so the size of a signature for this kind of operations can even be reduced.

5 Query Processing

A query processor can also exploit tree signatures to evaluate *set-oriented* primitives similar to the XPath axes. Given a set of elements R, the evaluation of $Parent(R, \texttt{article})$ gives back the set of elements named `article`, which are parents of elements contained in R. By analogy, we define the $Child(R, \texttt{article})$ set-oriented primitive, returning the set of elements named `article`, which are children of elements contained in R. We suppose that elements are identified by their preorder values, so sets of elements are in fact sets of element identifiers.

Verifying structural relationships can easily be integrated with evaluating content predicates. If indexes are available, a preferable strategy is to first use these indexes to obtain elements satisfying the predicates, and then verify the structural relationships using signatures. Consider the following XQuery [4] query:

```
for $a in //people
where
      $a/name/first="John" and
      $a/name/last="Smith"
return $a/address
```

Suppose that content indexes are available on the `first` and `last` elements. A possible efficient execution plan for this query is:

1. let $R_1 = ContentIndexSearch($`last = "Smith"`$);
2. let $R_2 = ContentIndexSearch($`first = "John"`$);
3. let $R_3 = Parent(R_1$,`name`$)$;
4. let $R_4 = Parent(R_2$,`name`$)$;
5. let $R_5 = Intersect(R_3,R_4)$;
6. let $R_6 = Parent(R_5$,`people`$)$;
7. let $R_7 = Child(R_6$,`address`$)$;

First, the content indexes are used to obtain R_1 and R_2, i.e. the sets of elements that satisfy the content predicates. Then, tree signatures are used to navigate through the structure and verify structural relationships.

Now suppose that a content index is only available on the `last` element, the predicate on the `first` element has to be processed by accessing the content of XML documents. Though the specific technique for efficiently accessing the content depends on the storage format of the XML documents (plain text files, relational transformation, etc.), a viable query execution plan is the following:

1. let $R_1 = ContentIndexSearch($`last = "Smith"`$);
2. let $R_2 = Parent(R_1$,`name`$)$;
3. let $R_3 = Child(R_2$,`first`$)$;
4. let $R_4 = FilterContent(R_3$,`John`$)$;
5. let $R_5 = Parent(R_4$,`name`$)$;
6. let $R_6 = Parent(R_5$,`people`$)$;
7. let $R_7 = Child(R_6$,`address`$)$.

Here, the content index is first used to find R_1, i.e. the set of elements containing `Smith`. The tree signature is used to produce R_3, that is the set of the corresponding `first` elements. Then, these elements are accessed to verify that their content is `John`. Finally, tree signatures are used again to verify the remaining structural relationships.

Obviously, the outlined execution plans are not necessarily optimal. For example, they do not take into consideration the selectivity of predicates. But the query optimization with tree signatures is beyond the scope of this paper.

6 Experimental Evaluation

The length of a signature $sig(T)$ is proportional to the number of the tree nodes $|T|$, and the actual length depends on the size of individual signature entries. The postorder (preorder) values in each signature entry are numbers, and in many cases even two bytes suffice to store such values. In general, the tag names are of variable size, which can cause some problems when implementing the tree inclusion algorithms. But also the domain of tag names is usually a closed domain of known or upper-bounded cardinality. In such case, we can use a dictionary of the tag names and transform each of the names to its numeric representation of fixed length. For example, if the number of tag names and the number of tree nodes are never greater than $65,536$, both entities of a signature entry can be represented by 2 bytes, so the length of the signature $sig(T)$ is $4 \cdot |T|$ for the short version, and $8 \cdot |T|$ for the extended version. With a stack of maximum size equal to the tree hight, signatures can be generated in linear time.

In our implementation, the signature of an XML file was maintained in a corresponding signature file consisting of a list of records. Each record contained two (for the short signature) or four (for the extended signature) integers, each represented by four bytes. Accessing signature records was implemented by a seek in the signature file and by reading in a buffer the corresponding two or four integers (i.e. 8 or 16 bytes) with a single read. No explicit buffering or paging techniques were implemented to optimize access to the signature file. Everything was implemented in Java, JDK 1.4.0 and run on a PC with a 1800 GHz Intel pentium 4, 512 Mb main memory, EIDE disk, running Windows 2000 Professional edition with NT file system (NTFS).

We compared the extended signatures with the Multi Predicate MerGe JoiN (MPMGJN) proposed in [10] – we expect to obtain similar results comparing with other join techniques as for instance [1]. As suggested in [10], the *Element Index* was used to associate each element of XML documents with its start and end positions, where the start and end positions are, respectively, the positions of the start and the end tags of elements in XML documents. This information is maintained in an inverted index, where each element name is mapped to the list of its occurrences in each XML file. The inverted index was implemented by using the BerkeleyDB as a B^+-tree. Retrieval of the inverted list associated with a key (the element name) was implemented with the bulk retrieval functionality, provided by the BerkeleyDB.

In our experiments, we have used queries of the following template:

```
for $a in //<e_name>
where <pred($a)>
return
      <result> $a/<e_1> ...$a/<e_n> </result>
```

Table 1. Selectivity of element names

element name	# elements
phdthesis	71
book	827
inproceedings	198960
author	679696
year	313531
title	313559
pages	304044

In this way, we are able to generate queries that have different *element name selectivity* (i.e. the number of elements having a given element name), *element content selectivity* (i.e. the number of elements having a given content), and the number of navigation steps to follow in the pattern tree (twig). Specifically, by varying the element name <e_name> we can control the element name selectivity, by varying the predicate <pred($a)> we can control the content selectivity, and by varying the number of expressions n in the return clause, we can control the number of navigation steps.

We run our experiments by using the XML DBLP data set containing 3,181,-399 elements and occupying 120 Mb of memory. We chose three degrees of the element name selectivity by setting <e_name> to phdthesis for high selectivity, to book for medium selectivity, and to inproceedings for low selectivity. The degree of content selectivity was controlled by setting the predicate <pred($a)> to $a/author="Michael J. Franklin" for high selectivity, $a/year="1980" for medium selectivity, and $a/year="1997" for low selectivity. In the return clause, we have used title as <e_1> and pages as <e_2>. Table 1 shows the number of occurrences of the element names that we used in our experiments, while Table 2 shows the number of elements satisfying the predicates used.

Each query generated from the previously described query template is coded as "QNCn", where N and C indicate, respectively, the element name and the content selectivity, and can be H(igh), M(edium), or L(ow). The parameter n can be 1 or 2 to indicate the number of steps in the return clause.

The following execution plan was used to process our queries with the signatures:

1. let $R_1 = ContentIndexSearch($<pred>$)$;
2. let $R_2 = Parent(R_1,$<e_name>$)$;
3. let $R_3 = Child(R_2,$<e_1>$)$;
4. let $R_4 = Child(R_2,$<e_2>$)$.

The content predicate is evaluated by using a content index. The remaining steps are executed by navigating in the extended signatures.

The query execution plan to process the queries through the containment join is the following:

Table 2. Selectivity of predicates

predicate	# elements
`$a/author="Michael J. Franklin"`	73
`$a/year="1980"`	2595
`$a/year="1997"`	21492

1. let $R_1 = ContentIndexSearch($`<pred>`$)$;
2. let $R_2 = ElementIndexSearch($`<e_name>`$)$;
3. let $R_3 = ContainingParent(R_2, R_1)$;
4. let $R_4 = ElementIndexSearch($`<e_1>`$)$;
5. let $R_5 = ContainedChild(R_4, R_3)$;
6. let $R_6 = ElementIndexSearch($`<e_2>`$)$;
7. let $R_7 = ContainedChild(R_6, R_3)$.

By analogy, we first process the content predicate by using a content index. Containment joins are used to check containment relationships: first the list of occurrences of necessary elements is retrieved by using an element index (*ElementIndexSearch*); then, structural relationships are verified by using the containment join (*ContainingParent* and *ContainedChild*).

For queries with $n = 1$, step 4, for the signature based query plan, and steps 6 and 7, for the containment join based query plan, do not apply.

Analysis. Results of performance comparison are summarized in Table 3, where the processing time in milliseconds and the number of elements retrieved by each query are reported. As intuition suggests, performance of extended tree signatures is better when the selectivity is high. In such case, improvements of one order of magnitude are obtained.

The containment join strategy seems to be affected by the selectivity of the element name more than the tree signature approach. In fact, using high content selective predicates, performance of signature files is always high, independently of the element name selectivity. This can be explained by the fact that, using the signature technique, only these signature records corresponding to elements that have parent relationships with the few elements satisfying the predicate are accessed. On the other hand, the containment join strategy has to process a large list of elements associated with the low selective element names.

In case of low selectivity of the content predicate, we have a better response than containment join with the exception of the case where low selectivity of both content and names of elements are tested. In this case, structural relationships are verified for a large number of elements satisfying the low selective predicate. Since such queries retrieve large portions of the database, they are not supposed to be frequent in practice.

The difference in performance of the signature and the containment join approaches is even more evident for queries with two steps. While the signature

Table 3. Performance comparison between extended signatures and containment join. Processing time is expressed in milliseconds.

Query	Ext. sign	Cont. join	#Retr. el
QHH1	80	466	1
QHM1	320	738	1
QHL1	538	742	1
QMH1	88	724	1
QMM1	334	832	9
QML1	550	882	60
QLH1	95	740	38
QLM1	410	1421	1065
QLL1	1389	1282	13805
QHH2	90	763	1
QHM2	352	942	1
QHL2	582	966	1
QMH2	130	822	1
QMM2	376	1327	9
QML2	602	1220	60
QLH2	142	1159	38
QLM2	450	1664	1065
QLL2	2041	1589	13805

strategy has to follow only one additional step for each qualifying element, that is to access one more record in the signature, containment joins have to merge potentially large reference lists.

7 Concluding Remarks

Inspired by the success of signature files in several application areas, we propose tree signatures as an auxiliary data structure for XML databases. The proposed signatures are based on the preorder and postorder ranks and support tree inclusion evaluation. Extended signatures are not only faster than the short signatures, but can also compute node levels and sizes of subtrees from only the partial information pertinent to specific nodes. Navigation operations, such as those required by the XPath axes, are computed very efficiently. We demonstrate that query processing can also benefit from the application of the tree signature indexes. For highly selective queries, i.e. typical user queries, query processing with the tree signature is about 10 times more efficient, compared to the strategy with containment joins.

In this paper, we have discussed the tree signatures from the traditional XML query processing perspective, that is for navigating within the tree structured documents and retrieving document trees containing user defined query twigs. However the tree signatures can also be used for solving queries such as:

> Given a set (or bag) of tree node names, what is the most frequent structural arrangement of these nodes.

Or, alternatively:

What set of nodes is most frequently arranged in a given hierarchical structure.

Another alternative is to search through tree signatures by using a query sample tree as a paradigm with the objective to rank the data signatures with respect to the query according to a convenient proximity (similarity or distance) measure. Such an approach results in the implementation of the *similarity range* queries, the *nearest neighbor* queries, or the *similarity joins*.

In general, ranking of search results [8] is a big challenge for XML searching. Due to the extensive literature on string processing, see e.g. [6], the string form of tree signatures offers a lot of flexibility in obtaining different and more sophisticated forms of comparing and searching. We are planning to investigate these alternatives in the near future.

References

1. Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, pp. 310–321, Madison Wisconsin, USA, June 2002*. ACM, 2002.
2. S. Chien, Z. Vagena, D.Zhang, V.J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proceedings of the 28rd VLDB Conference, Honk Kong, China*, pages 263–274, 2002.
3. World Wide Web Consortium. XML path language (XPath), version 1.0, W3C. Recommendation, November 1999.
4. World Wide Web Consortium. XQuery 1.0: An XML query language. W3C Working Draft, November 2002. http://www.w3.org/TR/xquery.
5. Torsten Grust. Accelerating XPath location steps. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data, 2002, Madison, Wisconsin*, pages 109–120. ACM Press, New York, NY USA, 2002.
6. D. Gusfield. *Algorithms on Strings, trees, and Sequences*. Cambridge University Press, 1997.
7. J.W. Hunt and T.G. Szymanski. A fast algorithm for computing longest common subsequences. *Comm. ACM*, 20(5):350, 353 1977.
8. Anja Theobald and Gerhard Weikum. The index-based XXL search engine for querying XML data with relevance ranking. In Christian S. Jensen, Keith G. Jeffery, Jaroslav Pokorný, Simonas Saltenis, Elisa Bertino, Klemens Böhm, and Matthias Jarke, editors, *Advances in Database Technology - EDBT 2002, 8th International Conference on Extending Database Technology, Prague, Czech Republic, March 25–27, Proceedings*, volume 2287 of *Lecture Notes in Computer Science*, pages 477–495. Springer, 2002.
9. Paolo Tiberio and Pavel Zezula. Storage and retrieval: Signature file access. In A. Kent and J.G. Williams, editors, *Encyclopedia of Microcomputers*, volume 16, pages 377–403. Marcel Dekker Inc., New York, 1995.
10. Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In Walid G. Aref, editor, *ACM SIGMOD Conference 2001: Santa Barbara, CA, USA, Proceedings*. ACM, 2001.

The *Collection Index* to Support Complex Approximate Queries

P. Ciaccia and W. Penzo

DEIS – IEIIT-BO/CNR
University of Bologna, Italy
{pciaccia,wpenzo}@deis.unibo.it

Abstract. The presence of structure in XML documents poses new challenges for the retrieval of data. Answering complex structured queries with predicates on *context* where data is to be retrieved, implies to find results that match *semantic* as well as *structural* query conditions. Then, the structural heterogeneity and irregularity of documents in large digital libraries make necessary to support *approximate queries*, i.e. queries where matching conditions are *relaxed* so as to retrieve results that possibly partially satisfy user's query conditions.
Exhaustive approaches based on sequential processing of documents are not adequate as to response time. In this paper we present an indexing method to execute efficiently approximate complex queries on XML documents. Approximations are both on content and document's structure. The proposed index provides a great deal of flexibility, supporting different query processing strategies, depending on the constraints the user might want to set to possible approximations on query results.

1 Introduction and Related Work

XML is announced to be the standard for future representation of data, thanks to the capability it offers to compose semi-structured documents that can be checked by automatic tools, as well as the great flexibility it provides for data modelling. The presence of nested tags inside XML documents leads to the necessity of managing *structured* information. In this scenario, traditional IR techniques need to be adapted, and possibly redesigned, to deal with the structural information coded in the tags. When querying XML data, the user's is allowed to express structural conditions, i.e. predicates that specify the *context* where data is to be retrieved. For instance, the user might want to retrieve: "Papers having title dealing with *XML*" (Query1). Of course, the user is not interested in retrieving whatsoever is containing the keyword "XML". This implies to find both a *structural match* for the context (title of papers) and a (traditional IR) semantic match for the content (the "XML" issue) *locally* to the matched context. Then, the structural heterogeneity and irregularity of documents in large digital libraries, as well as user's ignorance of documents structure, make necessary to support *approximate queries*, i.e. queries where matching conditions are *relaxed* so as to retrieve results that possibly *partially* satisfy user's query

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 164–179, 2003.

Fig. 1. Sample XML documents

conditions. Powerful query tools should be able to efficiently answer structural queries, providing results that are ranked according to possible relaxations on the matching conditions. As an example, consider document `Doc1` in Fig. 1 and the query: "Retrieve CDs with songs having the word *love* in the title" (Query2). *Doc1* contains a relevant answer that should be returned, although some dissimilarities are present with respect to the query structural condition: 1) The `track` element indeed can be considered a synonym for `song` in this context (*semantic relaxation*), and 2) an additional `tracklist` element is present between `cd` and `track` (*structural relaxation*). An exact match of the query on `Doc1` would have returned no result. The above query is more properly a *path query*, i.e. a query where structural condition is expressed by a single sequence of nested tags. Things are more complicated when answering complex queries, i.e. queries with two or more *branching* conditions. Finding structural approximate answers to this kind of query is known to be a hard task, namely, finding a solution to the *unordered tree embedding problem*[1] [5,6,8,16,17], which is proved to be NP-complete [12]. In [17] some critical assumptions of the problem are simplified to reduce the overall complexity, as to support approximate structural queries.

In this paper we present an indexing structure, the *Collection Index*, that effectively and efficiently supports *complex approximate queries* on XML data. The Index aims to reduce the complexity of finding approximate query patterns, avoiding the sequential examination of all documents in the collection. This issue has been recently covered by several approaches [10,11,13,15]. However, all these works do not deal with semantic nor structural approximations, except for the explicit use of wildcards in query paths. Actually, wildcards do not contribute to result ranking: Data is asked to satisfy a relaxed pattern, no matter the "grade"

[1] According to the XML Information Set W3C Recommendation [4], XML documents can be represented as trees. Complex queries can be also represented as *pattern trees* to be searched in the documents.

of relaxation. Thus, for instance, *cohesion* of results is not taken into account for ranking. Approximation is investigated in [19], although the proposed method focuses only on *semantic* similarity. Structural relaxations are supported by the index structures proposed in [14], though these are limited to deal with path expressions. The indexing method we propose is based on an *intensional view* of the data, similarly in spirit with traditional database modeling where data is separated from the schema. The Collection Index is a concise representation of the structure of all the documents in the collection, in that each document can be mapped exactly in one part of the index. The structure we propose resembles a DataGuide [11], but overcomes its limitations.[2] The Collection Index efficiently supports queries on XML document collections producing relevant *ranked results* that even partially satisfy query requirements, also in absence of wildcards in the query. Results are ranked according to their approximation to both structural and semantic query conditions.

The paper is organized as follows: In Section 2 we compare our approach with ApproXQL [17], a similar system that supports complex approximate query processing on XML data. Section 3 presents the Collection Index, as an *extended template* for the structure of a set of XML documents. In Section 4 we show how the Collection Index *efficiently* and *flexibly* supports approximate query processing: Different query processing strategies can be applied on the Index, depending on the constraints the user might want to set to possible approximations on query results. Complexity of query processing is also discussed, showing the feasibility of our approach. In Section 5 we sketch some experiments on a large heterogeneous collection of XML documents [18], and finally in Section 6 we draw our conclusion and discuss future developments.

2 Comparison with ApproXQL

Supporting approximate complex queries are work in progress (or future work) of most of the mentioned proposals that deal with efficient techniques to query XML data [14]. To authors' knowledge, ApproXQL [17] is currently the more complete system that enables the user to retrieve all approximate results to complex structured queries. ApproXQL applies different approximations to the query tree, allowing for renaming, insertion, and deletion of query nodes. We will refer to this system to compare the efficiency and the effectiveness of our method. The focus of the problem is finding an efficient algorithm to solve the tree embedding problem [12], properly reformulated to escape NP-completeness.

Reformulating the Tree Embedding Problem. Both queries and documents are assumed to be tree-structured. We recall that, given two trees t_1 and t_2, an injective function f from nodes of t_1 to nodes of t_2 is an *embedding* of t_1 into t_2, if it preserves labels, and ancestorship in both directions [12].

In order to make the embedding function efficiently computable, ApproXQL, as well as our approach, guarantees ancestorship only in one sense, from query

[2] Basically, a DataGuide would have not been helpful in answering Query2, since it does not allow for approximate navigation.

tree to document tree. This means that siblingness is not guaranteed by the mapping, in that sibling nodes in the query may be mapped to ancestor-descendant nodes in the data. For convenience, we will use the term embedding to denote such reformulated tree embedding problem, according to the one-way ancestorship preservation. For the sake of simplicity, we limit to *insertion* of nodes the approximations on results. This is compliant with the definition of tree embedding (as to ancestorship). We will complete discussion on renaming and deletion of query nodes in Section 4.3.

According to this new formulation of tree embedding problem, the result of a complex query for the ApproXQL method is a set $\mathcal{R}$ of nodes of the document tree, such that each $r \in \mathcal{R}$ is the root of a document (sub)tree that matches the query root and contains at least one embedding of the query tree. Using the Collection Index, we return the set $\mathcal{S}$ of minimal[3] document (sub)trees that contain at least one embedding of the query tree. It is easy to show that the relationship between ApproXQL and Collection Index results is: $\mathcal{R} = \bigcup_i \text{root}(t_i)$ $t_i \in \mathcal{S}$. We can derive that both ApproXQL and the Collection Index retrieve (or provide a reference to) the same embeddings.

A View to Complexity. Processing a complex structured query[4] is proved to require time $O(n_q{*}l{*}s)$, with n_q number of query nodes, l maximal number of repetition of a given label along a document path, and s maximal number of occurrences of a label in the document tree [17]. The goal of the Collection Index is to reduce this complexity. We will show that our method obtains the same results as ApproXQL in time $O(n_q{*}l{*}s_c^I{*}p^I)$, with $s_c^I{*}p^I \leq s$, where s_c^I is the maximal number of *contextual occurrences* of a label l_i in all index subtrees rooted at a given label l_j (each subtree is called a *context* for label l_j), and p^I is the maximal number of occurrences of a given index path in the document tree. It is worth to note that in most cases $s_c^I{*}p^I << s$, depending on the heterogeneity of the document collection. More details are given in Section 4.2.

3 The Collection Index

The extensional representation of a large XML document collection presents a great deal of redundancy because of the repeated tags to specify data semantics and structural organization for each document. Consider Fig. 1, showing two sample XML excerpts from XML Sigmod [1] (`Doc2`) and dblp [2] (`Doc3`) collections.[5] Whilst redundancy is evident for homogeneous collections (`Doc2`), the presence of repeated structural information is very common also in heterogeneous collections (e.g. *article* and *phdthesis* elements in `Doc3`). In general, the collected documents can be partitioned into homogeneous groups, according to the same basic structure they agree upon. Basically, the Collection Index is an

[3] Among all document (sub)trees that contain n embeddings, the *minimal* subtree has no proper subtrees that contain the same n embeddings.

[4] Recall that here only insertions of data nodes are allowed.

[5] We consider a single large XML document gathering several records as a collection of basic documents, one for each XML record.

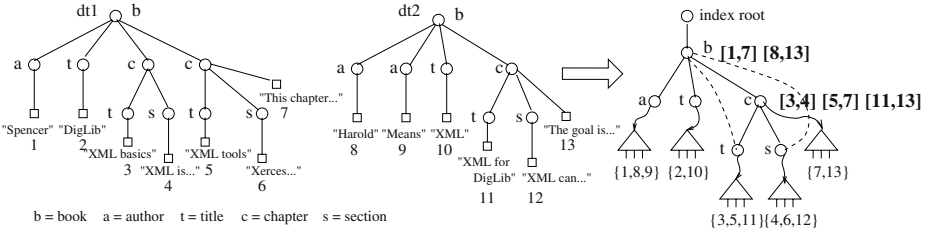


Fig. 2. Insertion of two documents in the index

extended template for the structure of a set of XML documents. The Index synthesizes the documents in the collection, and provides a skeleton in accordance with the structural relationships occurring in the documents.

3.1 Index Overview

The index is tree-structured: Nodes represent elements and attributes, and arcs define hierarchical relationships according to the nesting of elements/attributes in the documents. This structure forms the basic template of the document collection. We start simple, giving an intuition of index construction, and deferring a formal description of its components to Section 3.2.

The index is created incrementally, one document at a time. Before entering the documents in the index, a tree representation is provided for each document. Basically, each sequence[6] of tags $(t_1, \ldots, t_k)$, with t_1 document root, referencing a CDATA/PCDATA content defines a *path instance* in the document tree, i.e. a sequence of nodes $(n_1, \ldots, n_k)$ where n_1 is the tree root, such that $\forall i$, n_i is labelled with t_i, and n_k is a *tree leaf* that refers the data content. Given a path instance $p = (n_1, \ldots, n_k)$, the sequence $p^s = (label(n_1), \ldots, label(n_k))$ defines the *path schema* of p . Each referred CDATA/PCDATA section is stored in what we call a *data leaf*. All data leaves are *globally* numbered with increasing integer values from left to right, according to a depth-first visit of each document's tree. This numbering scheme allows for uniquely identifying each data leaf to be referenced by the index. Two sample document trees *dt1* and *dt2* are shown on the left side of Fig. 2: Squares denote data leaves, whereas tree nodes are marked by circles. The insertion of a document tree t_d in the index is equivalent to inserting (without repetitions) all the path instances that compose t_d. The path instances of t_d are assigned to *equivalence classes* according to the sequence of node labels (i.e. document tags) they are made of.

Definition 1 (Path Equivalence Class (PEC)). *A Path Equivalence Class Ξ is a set of path instances with the same path schema. Formally: Given two path instances $p_1 = (x_1, .., x_k)$ and $p_2 = (y_1, .., y_h)$, $\{p_1, p_2\} \subseteq \Xi \iff k = h$ and $\forall\ i \in [1..k]$ label(x_i)=label(y_i). For each path instance p_i $\exists$ a PEC Ξ_j such that $p_i \in \Xi_j$.*

Any (tree-structured) document collection $\mathcal{C}$ can be partitioned into a set of PECs, according to the path instances that compose documents in $\mathcal{C}$.

[6] Elements and attributes are considered equivalent. This implies that sequences might end with attributes.

Equivalence classes are created and grow as documents are processed. Equivalent path instances *share* the same path in the index, i.e. only one path per class is present in the index: This *template path* has the same schema of the path instances of the class, i.e. the *schema* of the class.

Definition 2 (PEC Schema). *Given any path instance $p \in \Xi$, the PEC schema Ξ^s of Ξ is p^s, the path schema of p.*

This leads to the following definition, which is the milestone of the structure of the Collection Index.

Definition 3 (Collection Index Path). *Given a document collection $\mathcal{C}$, a path p_I with schema p_I^s is a Collection Index Path for $\mathcal{C} \iff \exists$ a PEC instance Ξ of $\mathcal{C}$, with schema Ξ^s, and $\Xi^s = p_I^s$. For each pair of PEC instances Ξ_i, Ξ_j, let p_i and p_j be their corresponding Collection Index Paths, and let p_M^s be the maximal prefix[7] of PEC schemas Ξ_i^s and Ξ_j^s. Then, p_M^s is also the maximal prefix of p_i^s and p_j^s and p_i and p_j share the (sub)path with schema p_M^s in the Collection Index.*

For instance, in Fig. 2 the two paths in *doc1* referring to data leaves numbered 3, and 5, and the path in *doc2* referring to data leaf numbered 11, respectively, belong to the same PEC Ξ_i having the same schema $\Xi_i^s = (b, c, t)$. On the other hand, the paths in *doc1* leading to data leaves numbered 4, and 6, and the path in *doc2* referring to data leaf numbered 12, respectively, belong to the same PEC Ξ_j, having the same schema $\Xi_j^s = (b, c, s)$. Ξ_i^s and Ξ_j^s share their maximal prefix (b, c) in the index.[8]

Each index (sub)path[9] references the set of data leaves referred by the document paths it has been derived from. In our example, the index path $/b/c/t$ references data leaves numbered 3, 5, and 11, and the index (sub)path $/b/c$ references data leaves numbered 7, and 13. The index provides an intensional view of the documents structure, where each occurrence of a path in a document is present only once in the index. The extensional representation of the document collection is given by the set of data leaves referenced by the index, since the documents structure is implicitly specified by the index itself. Each index (sub)path define a *semantic context* where the underneath data leaves are referenced according to IR access structures local to the (sub)path, e.g. B+-tree on leaf numbers, and inverted lists on terms. These access structures are depicted by triangles in Fig. 2. We call each structure a *value-index*.

[7] Given two sequences $s_1 = (x_1, .., x_k)$, and $s_2 = (y_1, .., y_h)$, the *maximal prefix* of s_1 and s_2 is the longest sequence s such that $prefix(s, s_1) \wedge prefix(s, s_2)$, where $prefix$ is defined as follows:

$$prefix(x, y) = \begin{cases} true & \text{if } x = (x_1, \ldots, x_k) \\ & \quad \wedge\ y = (y_1, \ldots, y_k, y_{k+1}, \ldots, y_h) \\ & \quad \wedge\ x_i = y_i\ \forall i \in [1..k] \\ false\ otherwise \end{cases}$$

[8] The dashed lines and the numbers between square brackets keep additional information that will be explained in Section 3.2.

[9] We always refer to (sub)paths that start from root.

3.2 The Index Structure

In order to support data retrieval, the index skeleton sketched above needs additional information regarding the document instances indexed. After a document is added to the index, information on the new referenced data leaves is propagated to the inner nodes. Each inner node n keeps a pair of integer values $[l_{min}, l_{max}]$, for each occurrence of the (sub)path from index root to n occurring in the data tree. Each range denotes the set of leaves underneath the instance data node, such that l_{min} is the lowest leaf number, and l_{max} is the highest one, respectively. As an example, in Fig. 2 ranges are assigned to index nodes labelled "b" and "c", denoting books and chapters, respectively. The ranges assigned to index node "b" say that two occurrences of book elements are present in the collection, and that the data leaves they refer to are 1 to 7, and 8 to 13, respectively. Similarly, ranges assigned to index node "c" say that three occurrences of chapter elements are present in the collection, each one referring the sets of data leaves 3 to 4, 5 to 7, and 11 to 13, respectively.[10] Note that, each sequence of ranges is *locally ordered* by increasing values of l_{min}, and that ranges are *disjoint*. This is due to the numbering scheme of the data leaves when processing documents to generate the Collection Index paths.

Insertion of documents is an incremental process that does not require the index to be restructured. The index begins with a dummy root node. All the data paths are inserted starting from the root, which collects all the path entries. This allows for indexing also heterogeneous collections of documents. There is no need for a priori knowledge of the schema of the data, since the paths we encode are extracted from the data itself. Actually, the complete index is an *extended tree* in that it is more properly a single-rooted DAG. Each inner node, index root included, references its descendants in the tree. We take advantage of this information to efficiently perform *approximate* retrieval on structure.[11] In Fig. 2, arcs depicted by dashed lines denote ancestorship between nodes: For clarity, parentship is expressed by solid arcs, and references to descendants of the index root are omitted. Formally: Let $\mathcal{N}$ be a set of nodes, *NID* set of index node ids, *DID* set of (sub)document ids,[12] Λ set of node labels, Ψ set of document paths, and *VID* set of value-index ids. The Collection Index is defined as follows.

Definition 4 (Collection Index). A Collection Index is a tree $\Im = (\mathcal{N}, r)$, with $\mathcal{N} = \mathcal{N}_I \cup \mathcal{N}_L$, with $\mathcal{N}_I$ set of inner nodes, $\mathcal{N}_L$ set of leaves, and r $\in \mathcal{N}_I$ root of the tree, s.t.:

$$\mathcal{N}_I = \{n = (nid, l, \delta, \rho, vid) \,|\, n \in NID \times \Lambda \times 2^{NID \times \Lambda} \times 2^{\mathbb{N} \times \mathbb{N} \times DID} \times VID\}$$
$$\mathcal{N}_L = \{l = (lid, l, p, vid) \,|\, lid \in NID \times \Lambda \times \Psi \times VID\}$$

where $\forall\, n \in \mathcal{N}_I$, δ is a set of labelled id's denoting descendants of n in $\Im$, and ρ is a set of tuples $t = (min, max, did)$ each denoting a range $[min, max]$ originated

[10] Ranges can be omitted for index leaves since their values can be obtained from the value-indices.

[11] Of course, this choice is not costless. The overhead of this additional information is discussed in Section 12.

[12] Each node of a document tree is supposed to be uniquely identified.

from (sub)document *did* in the document collection; $\forall\, l \in \mathcal{N}_L$, $p \in paths(\Im)$ and $p = (r, \ldots, l)$.

Index Size. As introduced in Section 2, we compare the Index Collection with ApproXQL [17]. In order to perform structural approximations, the Collection Index takes advantage of the lists of labelled descendant nodes δ assigned to each index node $n \in \mathcal{N}_I$. The list immediately points out nodes having a given label, located in specific parts of the index, namely in the subtree of the node n itself (denoting the scope of the search). The size of collecting these lists can be proved to be at most $O(n^I{*}h^I)$, with n^I number of nodes in the Collection Index, and h^I height of the Index. Depending on the grade of heterogeneity of the document collection, the value of n^I can be very small, in many cases comparable to the number of labels of a DTD for the collection. On the other hand, the value of h^I is generally small: it represents the maximal nesting level of tags in the documents. This means that the skeleton of the Index in most cases has a minimal impact as to its size. However, additional space is required for keeping information on ranges of data leaves assigned to each occurrence of document (sub)trees. The size of the ρ lists can be estimated as $O(n_d)$, with n_d number of nodes in the data tree. This evaluation is comparable with the size of the global inverted lists used in the ApproXQL approach, where each distinct label in the data is assigned the list of its occurrences in the document tree.[13] In comparison with the size of the ρ lists, in the general case, the quantity $O(n^I{*}h^I)$, stating the size of the Index skeleton, can be considered negligible.

4 Approximate Query Processing

We assume a query t_q to be tree-structured, expressing both semantic and complex structural conditions. Rather than attempting a tree embedding of t_q for each document, which is a time-consuming task, we rely on the Collection Index. Basically, the Index is navigated top-down following the arcs in accordance with the query structural conditions.

Various navigation strategies can be applied on the Collection Index, depending on the constraints the user might want to set to possible approximations on query results. We start presenting the navigation method to obtain approximate embeddings that relax the query conditions to possible insertion of nodes in the data. Other kinds of approximations, namely, renaming and deletion of query nodes are discussed in Section 4.3, together with the strategies to obtain the corresponding sets of approximate embeddings of the query tree.

4.1 Navigating the Collection Index

Approximate embeddings of the query tree in the Collection Index are retrieved in two steps, according to Algorithm `Solve` in Fig. 3 : 1) Top-down navigation for selecting relevant contexts, and 2) Bottom-up construction of final results.

[13] This computation includes also the inverted lists for the content-based retrieval of data leaves. As discussed in Section 3.1 we use inverted lists which are *local* to each structural context. However, their size is comparable to the size of the global lists used in [17] for content retrieval.

```
Solve(IndexTree I, QueryTree q)
begin
1.  q' ← SetMatches(I,q)
2.  FindEmbeddings(I,q')
end
```

Fig. 3. Algorithm `Solve`

```
SetMatches(IndexTree I, QueryTree q)
begin
1.  for each t_i in δ of I s.t. label(t_i)=label(q)
2.    append (&I,&t_i) to list of context. matches of q
3.    for each q_j ∈ children(q)
4.      q_j ← SetMatches(t_i,q_j)
5.  return q
end
```

Fig. 4. Algorithm `SetMatches`

Exploration. The navigation step extends the query tree: Each query node is assigned a *list of matches* with nodes of the index tree. More precisely, each match is a *contextual match*, in that it is a pair $m = (\&context, \&node)$ such that $\&node$ is a pointer to the matching node in the index, and $\&context$ is a pointer to the index node that specifies the context where $\&node$ has been retrieved. Each context node is a match for the parent of the current query node. The context node of the query root is the Collection Index root.

The goal is to find all the *structural* matches of the query tree in the Collection Index. Semantic match of query leaves containing CDATA/PCDATA content is not considered at this step: This will be the starting point of Algorithm `FindEmbeddings` for the construction of results. Exploration of the Collection Index is guided by a query tree q as follows (algorithm `SetMatches`):

1. Given a context I in the Collection Index, descendant index nodes in I having the same label as the query root's label are set as matching nodes of q (lines 1 – 2). This means that only *relevant contexts* of the index are explored. Each match found is the root of a potential approximate embedding schema.[14]
2. The search proceeds recursively for each query node (lines 3 – 4), according to a pre-order visit of the query tree. Each matched index node found at the previous step is taken as the *current context* in the Collection Index.

Figure 5 presents the result of the `SetMatches` Algorithm to answer a structured query. For clarity, only node matchings are shown, depicted by dashed bold lines.

Construction of Results. Recall from Section 2 that the evaluation of a query q on a document d with the Collection Index returns a set of subtrees of d, each one denoting an approximate result for query q. Each returned subtree collects the set of approximate embeddings of q rooted at the same (sub)document *did* in d. These id's indicate the roots of the instances of the query tree in document d. Note that each (sub)document *did* that is the root of a returned subtree is in the list ρ of an index node that matched the query root. In order to determine which are the *did*'s in the ρ lists that actually represent the root of relevant results, semantic conditions of q are to be satisfied in the data leaves of the

[14] Recall that the Collection Index provides an intensional representation of the structural organization of the data, basically it collects all structural schemas that may occur in the data.

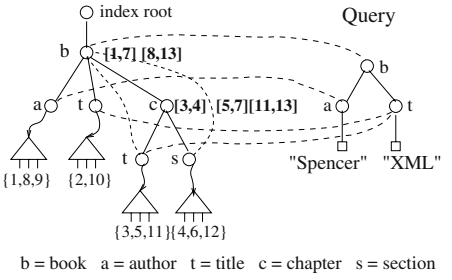


Fig. 5. Query embedding in CI

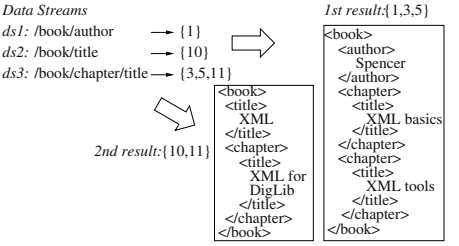


Fig. 6. Sample query results

(sub)document instance. At this purpose, only *relevant* data leaves are considered, in that we take advantage of the value-indices of index nodes that matched the leaves of the query tree[15] to retrieve the data that satisfies the corresponding query semantic conditions, i.e. data that contains the keyword(s) specified in the query. We call each list of relevant data leaves returned by a value-index a *data stream*. For instance, in Fig. 6 three data streams are shown: One for each match of the query (tree) leaves in the index. As an example, the data stream $ds2$ is a list made of a single element, the data leaf numbered 10: In fact, as shown in Fig. 2, only leaf 10 is relevant to the query path $/b/t/"XML"$. This retrieval is performed by the value-index assigned to the index path $/b/t$. Each stream denotes a different data source according to different structural approximations. For instance, in Fig. 5, the data streams assigned to index paths $/b/t$ and $/b/c/t$ represent data sources assigned to exact match and approximate match of query path $/b/t/"XML"$, respectively.

However, the assignment of data streams does not suffice. Although each data stream returns relevant data leaves, we have to verify that they belong to the same (sub)document instance. As an example, according to the document collection shown in Fig. 2, only document $dt1$ is relevant to the query shown in Fig. 6, since "Spencer" appears as author only in $dt1$. This implies that only data leaves numbered 1, 3, and 5 compose a relevant result. On the other hand, data leaves numbered 10 and 11 belong to an incomplete result that does not satisfy condition on author.[16] In order to assert which data of n data streams belongs to the same relevant result, we take advantage of the list ρ of ranges assigned to index inner nodes.

Construction of results proceeds according to a post-order visit of the extended query tree obtained at step 1), exploiting additional information which is recursively attached to each query node. In fact, at the end of the construction process, each node of the query tree is assigned a list of *contextual ranges*, i.e. a list of pairs `(context,Ranges)`, such that `Ranges` is a list of ranges of relevant data leaves, and `context` is the context in the index where these have been retrieved. For instance, in Fig. 5 query node b is assigned the pair

[15] Recall that we refer to *data leaves* as CDATA/PCDATA content, and to *(tree) leaves* as tree nodes that refer to data leaves.

[16] The last incomplete result will be taken into account in Section 4.3 when dealing with approximate embeddings allowing renaming and deletion of query nodes.

```
FindEmbeddings(IndexTree I, QueryTree q)
begin
1.  if leaf(q)
2.    for each m=(context,match) ∈ list of contextual matches of q
3.      Ranges ← getDataStream(I,match) s.t. semantic condition of q is satisfied
4.      append r=(context,Ranges) to list of contextual ranges of q
5.  else
6.    for each q_j ∈ children(q)
7.      FindEmbeddings(I,q_j)
8.    ChildrenLists ← get lists of contextual ranges from children of q
9.    ChildrenList ← merge ChildrenLists per context
10.   IndexList ← ∅
11.   for each m=(context,match) ∈ list of contextual matches of q
12.     append (m,getRangeList(I,match)) to IndexList
13.   RangeList ← InclusionCheck(ChildrenList,IndexList)
14.   assign RangeList as list of contextual ranges of q
end
```

Fig. 7. Algorithm `FindEmbeddings`

(`index_root`,⟨`[1,7]`⟩), since relevant data leaves 1, 3, and 5 that compose the first result shown in Fig. 6 belong to range [1,7]. This implies that, the *did* assigned to range [1,7] in the ρ list of index node b is the id of the document that satisfies the query. For a query (tree) leaf l, the list `Ranges` correspond to a list of data streams, one for each match of l in the index. The list of contextual ranges for a query node n, each denoting a (sub)query embedding is constructed recursively, according to Algorithm `FindEmbeddings`:

1. For each contextual match of a query leaf l, the data stream of the match is assigned to l as list of ranges of relevant data leaves in the context of the match. This is the basic case of single-element ranges, one per relevant data leaf (lines 2 – 4).
2. As to each inner query node n, the algorithm proceeds recursively to generate a list of contextual ranges for each child node of n (lines 6 – 7). Then, all lists of contextual ranges are collected from children nodes of n (line 8). Contextual ranges that share the same context c are merged in order to collect all relevant data leaves in a single occurrence of context c (line 9). Thus, `ChildrenList` contains the list of contextual ranges referring relevant data leaves that satisfy conditions specified by children nodes of n. Recall that these ranges have to be aggregated according to their membership to each (sub)document instance. This is accomplished by taking advantage of the ρ lists of the contextual matches of n, which are collected in the list `IndexList` (lines 10 – 12). Recall that each range in `IndexList` references a (sub)document instance, and that ranges of relevant data leaves in `ChildrenList` belong to the same (sub)document instance d_i if they are included in the range r in `IndexList` that refers d_i. This implies to check inclusion of ranges of `ChildrenList` into ranges of `IndexList` (line 13). Ranges in `IndexList` that satisfy inclusion are assigned as contextual ranges of node n. For each of these ranges, each context is given by the context assigned to the corresponding match of n in the index.

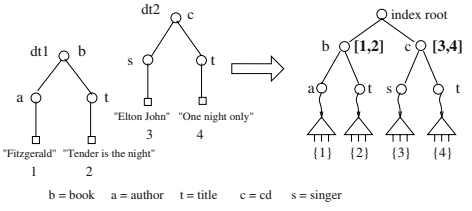


Fig. 8. Heterogeneous documents in CI

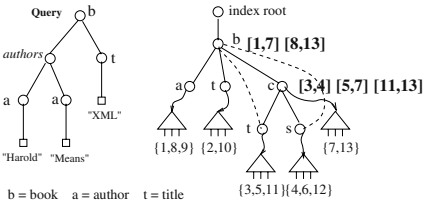


Fig. 9. Sample query on the CI

In order to guarantee that all query conditions are satisfied, the check inclusion step requires that at least one data leaf per query path belongs to the same contextual range (*completeness check*).[17] At the end of the construction process, the contextual ranges assigned to the root of the query denote the (sub)document instances that satisfy the query conditions. In our example of Fig. 6, only the first result is returned. Approximation is due to the presence of the `chapter` element between the `book` and `title` in the query path */book/title/"XML"*. This result complies with the definition of tree embedding. Element `book` is the root of two query embeddings: The former matching book's author and the title of first chapter; The latter matching book's author and the title of second chapter. This implies that the algorithm `FindEmbeddings` finds out several occurrences of the same pattern in a document.

4.2 Complexity of Query Processing

The basic feature of the index is the list of labelled descendants (δ) for each inner node $n \in \mathcal{N}_I$. Efficiency and effectiveness of of top-down navigation of the Collection Index are improved: Query execution is lightened, since intermediate nodes leading to each descendant node are not visited; On the other hand, the navigation strategy leads to exploring only *relevant contexts*, a priori excluding parts of the index tree denoting different scopes. As an example, consider Fig. 8 and query */book/title/"night"*, looking for books having the keyword "night" in the title. The navigation step looking for element `book` reduces the scope of the search to the left part of the Collection Index, whereas approaches "à la ApproXQL" check for relevant data leaves also for titles of CD's.

Recalling Section 2, the complexity of performing approximate complex queries with the Collection Index is $O(n_q{*}l{*}s_c^I{*}p^I)$. More precisely, this is given by the sum of two quantities: $O(n_q{*}l{*}s_c^I) + O(n_q{*}l{*}s_c^I{*}p^I)$. Quantity $O(n_q{*}l{*}s_c^I)$ is due to Index navigation to set matches for query nodes (Algorithm `SetMatches`): In fact, for each query node at most s_c^I matches exist in all the *contexts* of the parent node, and all of these can be found at most as descendants of l contexts.
Quantity $O(n_q{*}l{*}s_c^I{*}p^I)$ is due to construction of results (Algorithm `FindEmbeddings`): For each query leaf, the number of contextual ranges is at most $l{*}s_c^I$, one for each contextual match. For each inner query node, the merge step (line 9)

[17] We will relax this step in Section 4.3 to capture also partial embeddings.

requires $O(l^* s_c^I)$ operations, since lists are ordered per context,[18] and $l^* s_c^I$ is the maximal length of a list of contextual ranges. For each pair of contextual ranges merged at this step, the corresponding lists of ranges are to be collected as a single list per context. This additional merging step requires $O(p^I)$ operations since lists of ranges are ordered[19], and each list contains at most p^I elements. Thus, operation at line 9 costs $O(l^* s_c^I * p^I)$. After this merging step, `ChildrenList` contains at most s_c^I contextual ranges, each one having a range list of length at most $fo_{max} * p^I$, where fo_{max} is the maximal fan-out of nodes in the Collection Index. `IndexList` contains at most $l^* s_c^I$ ordered elements, each referring $O(p^I)$ ordered ranges (lines 10 – 12). Thus, the inclusion check step for ranges (line 13) costs $O(l^* s_c^I * p^I)$, since lists are ordered. In conclusion, Algorithm `FindEmbeddings`, which dominates the overall complexity, can be executed in time $O(n_q * l^* s_c^I * p^I)$.

4.3 Retrieving All Approximate Embeddings

For the sake of simplicity, in Section 4 we dealt with approximate embeddings that relax query conditions only to insertion of nodes in the data. Now we consider two additional kinds of approximations: *Renaming* and *deletion* of query nodes. Consider the query shown in Fig. 5, where label `writer` is set in place of `author`. Results would be empty since no index node exists having label `writer`. Renaming of query nodes can be easily supported by the Collection Index: Traversal of arcs can be relaxed to descendants having *similar* labels. This allows for *semantic approximation*. Similarity can be measured as a semantic relationship between node labels.[20] Consider again query in Fig. 5, with name of the author changed to "Brown". No document in our sample data collection satisfies all query conditions. Following a more flexible approach, relaxation on completeness of results would produce both (approximate) results shown in Fig. 6. However, this is not the only relaxation that can be made as to query partial matching, since the removal of the completeness check step corresponds to deletion of query data leaves. Consider Fig. 9: results are empty because label `authors` does not appear in the Index. To overcome this drawback, the navigation strategy (Algorithm `SetMatches`) can be modified: a query node that does not find any match in the index should be marked as *unsatisfied*, and navigation should proceed with next query node.[21]

Note that the last navigation strategy leads to *controlled* structural approximations. In fact, a query node is allowed to be deleted only if it does not find a match in the index. This implies that only the *structurally best-matching* results are returned. As an example, consider the query */book/chapter/title/"DigLib"*.

[18] Recall that the δ list of each index node contains pointers to descendants obtained by a pre-order visit of the Index. This implies that the list of contextual matches (obtained from the δ list, in Algorithm `SetMatches`) is ordered according to this visit. As a consequence, elements in each list of contextual ranges (obtained at line 2 of Algorithm `FindEmbeddings`) respect the same order.

[19] As discussed in Section 3.2.

[20] To this end, in our implementation we relied on the WordNet semantic network [3].

[21] ApproXQL only allows for deletion of nodes having at most one child.

In Fig. 2 the query pattern occurs in document *dt*2, which is relevant to the query. This is the only result captured by the navigation strategy proposed, that does not allow query node `chapter` to be deleted, since at least one structural occurrence */book/chapter* is present in the collection. On the other hand, ApproXQL retrieves a larger set of approximate embeddings, namely those obtainable by the deletion of any query node, except the root. According to this approach, also document *dt*1 is relevant to the above query, since it is a book having in the title the searched keyword. Of course, also document *dt*1 might be of interest for the user. This approximation can be obtained by means of an additional relaxation of the navigation strategy: At each navigation step all arcs leading to descendants having a label among the labels of the query pattern are traversed. Thus, depending on the query constraints the user may want to set to possible approximations on query results, a different navigation strategy can be applied on the Collection Index. It is worth to note that, the retrieval of all approximate embeddings according to the deletion of any query node, on one hand guarantees to obtain all relevant data, on the other hand, it may generate a huge amount of results that satisfy the query pattern only minimally, thus lowering the precision of the result set.

Ranking Results. The relevance of results takes into account several aspects. According to the *SATES* proposal [7,9], we compute the score of results as a combination of semantic and structural similarities, providing a ranking according to correctness, completeness, and cohesion of the data retrieved.

5 Experiments

We used the XMach collection [18] for testing our index. The collection presents some interesting characteristics that emphasize the importance of providing approximate query capabilities: The documents have *irregular* structure and a high nesting level (up to 13), with label repetition along a path which is at most 11. In such a structural scenario, the recall of an exact match approach would not be satisfactory. XML files in the collection respect a DTD[22] that models documents having an author, a title and 1 or more chapters, each one with usual information, like author, sections, etc. We experienced the Collection Index on XMach subcollections ranging from 1,000 to 150,000 documents, with total size ranging from 16Mb to 2.3Gb, respectively. The index scales linearly in space, and for large collections exceeds the original size of the dataset by 1,5-2%. As to time scalability, the Collection Index proved to perform linearly with the size of the dataset.

6 Conclusion and Future Work

We have presented the Collection Index, an indexing method that efficiently supports the retrieval of approximate results for complex structured queries on large collections of XML documents. To authors' knowledge it is the first index that

[22] We remind that the Collection Index does not require the presence of a DTD.

supports complex branching queries with both content and structural approximations, also capturing the presence of *multiple occurrences* of query conditions. A similar approach having the same goals has been presented in [17]. The ApproXQL system copes with the intrinsic complexity of finding unordered tree embeddings [12], and proposes a method that solves a reformulation of the classical problem in polinomial time. We proved the Collection Index to support a more efficient query processing, through the selection of *relevant data contexts*. Results are *ranked* according to an effective measure that takes into account semantic and structural correctness, completeness, and cohesion of results [6,8].

The Collection Index seems to be an effective and flexible indexing structure, where different strategies can be applied to personalize the user needs. Studying which strategies may prune less relevant results, with the aim of scaling down further the complexity of finding approximate embeddings for a query tree is future work. Then, we plan to exploit the numbering scheme applied to the data leaves, to support queries with ordering requirements. Query processing optimization is a further direction we intend to follow, by studying the evaluation ordering of structural predicates to improve index performance.

References

1. ACM Sigmod Online. `http://www.acm.org/sigmod/record/xml/`.
2. DBLP XML records. `http://www.informatik.uni-trier.de/ ley/db/index.html`.
3. WordNet Home Page. `http://www.cogsci.princeton.edu/ wn/`.
4. XML Information Set. `http://www.w3.org/TR/xml-infoset`.
5. S. Amer-Yahia, S. Cho, and D. Srivastava. Tree Pattern Relaxation. In *Proc. of the 8th Int. Conf. on Extending Database Technology (EDBT 2002)*, March 2002.
6. P. Ciaccia and W. Penzo. Adding Flexibility to Structure Similarity Queries on XML Data. In *Proc. of 5th Int. FQAS Conf. (2002) LNAI 2522*, pages 124–139.
7. P. Ciaccia and W. Penzo. Adding Flexibility to Structure Similarity Queries on XML Data. In *Proc. of the 5th Int. Conf. on Flexible Querying Answering Systems (FQAS 2002) LNAI 2522*, pages 124–139, Copenhagen, Denmark, October 2002.
8. P. Ciaccia and W. Penzo. Relevance Ranking Tuning for Similarity Queries on XML Data. In *Proc. of the 1st VLDB Workshop EEXTT 2002*, China, 2002.
9. P. Ciaccia and W. Penzo. Relevance Ranking Tuning for Similarity Queries on XML Data. In *Proc. of the 1st VLDB Workshop on Efficiency and Effectiveness of XML Tools, and Techniques (EEXTT 2002)*, Hong Kong, China, August 2002.
10. B. F. Cooper, N. Sample, M. J. Franklin, and M. Shadmon G. R. Hjaltason. A Fast Index for Semistructured Data. In *Proc. of the 27th VLDB Conf.*, 2001.
11. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. of 23rd Int. VLDB Conf.*, 1997.
12. P. Kilpeläinen. *Tree Matching Problems with Application to Structured Text Databases*. PhD thesis, Dept. of Computer Science, Univ. of Helsinki, SF, 1992.
13. Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. of the 27th VLDB Conf.*, pages 361–370, Rome, Italy, 2001.
14. T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proc. of the 8th Int. Conf. on Database Theory (ICDT'99)*, pages 277–295, Jerusalem, Israel, 1999.

15. F. Rizzolo and A. Mendelzon. Indexing XML Data with ToXin. In *Proc. of 4th Int. Work. on the Web and Databases (WebDB01)*, 2001.
16. T. Schlieder. Similarity Search in XML Data Using Cost-Based Query Transformations. In *Proc. of 4th Int. Work. on the Web and Databases (WebDB01)*, 2001.
17. T. Schlieder. Schema-Driven Evaluation of Approximate Tree-Pattern Queries. In *Proc. of the 8th Int. EDBT Conf.*, 2002.
18. E. Rahm T. Böeme. XMach-1: A Benchmark for XML Data Management. In *Proc. of Conf. on Database Systems for Business, Technology ad Web (BTW 2001)*.
19. A. Theobald and G. Weikum. The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. In *Proc. of the 8th Int. EDBT Conf.*, 2002.

Finding ID Attributes in XML Documents

Denilson Barbosa and Alberto Mendelzon

University of Toronto
Toronto, Ontario, Canada
{dmb,mendel}@cs.toronto.edu

Abstract. We consider the problem of discovering candidate ID and IDREF attributes in a schemaless XML document. We characterize the complexity of the problem, propose a heuristic algorithm for it, and discuss experimental results.

1 Introduction

XML documents may be accompanied by schemas that specify their structure, and impose restrictions on the *values* of some elements or attributes. The WWW Consortium has defined two schema formalisms for XML: Document Type Definitions (DTDs) [13] and XML Schema [14]. Although these languages differ in several aspects, both allow the specification of *ID*, *IDREF* and *IDREFS* attributes. ID attributes are unique identifiers for the elements that bear them; IDREF attributes are logical pointers to ID attributes; IDREFS attributes are pointers to *sets* of ID attributes. IDREF(S) attributes establish references among elements in the document, turning the XML trees into graphs. XML query languages have ways of "navigating" these graphs seamlessly via tree edges and reference edges (see [1]). In a sense, ID attributes serve as keys for the elements, while IDREF(S) attributes serve as foreign keys. However, as we shall see, there are some significant differences between ID/IDREF(S) attributes and keys.

Schemas are important tools that enable data exchange, efficient storage, and query formulation. Because many XML databases are likely to be missing their schemas [8], there has been interest in the literature in the problem of *inferring* a schema for a document. Current approaches, however, focus on extracting the portions of the schema that define the tree structure of the document. In this paper, we consider the problem of discovering the portions of the schema that constrain the non-tree edges in the graph, that is, discovering candidate ID and IDREF attributes in a schemaless document.

1.1 Related Work

To the best of our knowledge, there is no previous work on the problem of discovering ID and IDREF attributes for XML documents.

Grahne and Zhu [6] consider the problem of finding approximate keys in XML documents. That work differs from ours in the following ways. First, a

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 180–194, 2003.

key is only required to be unique over the set of elements of the same type, while ID attributes are required by the XML specification to be unique over the entire document. Second, the keys considered in [6] are not necessarily unary, as are ID attributes. Finally, that work does not consider finding foreign keys, the equivalent to IDREF attributes.

Garofalakis *et al.* [5] consider the problem of extracting DTDs from XML documents. However, the authors consider only structural constraints, and do not deal with attribute values.

Buneman *et al.* [3] define keys for XML. That work lays the foundations and also the basic notation used here and in [6]. Arenas *et al.* [2] study the interaction of DTDs with keys and foreign-keys for XML.

A similar problem in the relational setting is finding keys and foreign keys from a set of relation instances (see, e.g., Mannila and Räihä [7]). That problem differs from ours in two ways: foreign keys are typed, while IDREF(S) attributes are not; and the scope of a key is a relation, while, as we mentioned above, the scope of ID attributes is the entire document.

Organization of the Paper. Section 2 introduces the machinery we will use throughout the paper. The semantics of ID and IDREF attributes in XML, and how they are represented in our notation are covered in Section 3. The complexity of finding a good set of ID attributes is given in Section 4; a heuristic algorithm for this problem is given in Section 5. We consider the easier problem of finding IDREF(S) attributes Section 6. An experimental validation of our algorithms is presented in Section 7. We conclude in Section 8.

2 Preliminaries

2.1 Data Model

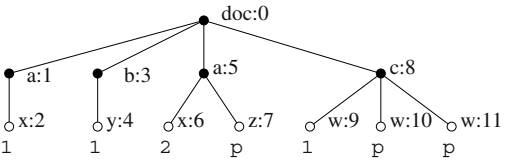
We represent XML documents as labeled trees with three kinds of nodes for representing elements, attributes and textual content. Each document D has a distinguished node called $root_D$. For clarity, we deal with a single document and drop the subscripts for identifying documents. Each node v in the tree has a label, denoted $label(v)$; a unique identifier (*id* for short), denoted $id(v)$; and a value, denoted $value(v)$. Without loss of generality, we ignore the ordering of nodes in the tree. Let $\mathcal{I}$ be set of node *id*s and let $\mathcal{V}$ be the set of all values in the document.

We say two nodes v_1 and v_2 are *node equal*, written $v_1 =_n v_2$, if $id(v_1) = id(v_2)$, and *value equal*, written $v_1 =_v v_2$, if $value(v_1) = value(v_2)$.

Running Example. We will use the simple XML document shown in Figure 1(a) to illustrate the various concepts throughout the paper. Figure 1(b) shows the tree representation of our example XML document. Each node v is annotated with a pair $label(v) : id(v)$. Multivalued attributes are represented as multiple attribute nodes, all with the same label. We also show the values of the attribute nodes in the document.

```
<doc>
    <a x='1'/>
    <b y='1'/>
    <a x='2' z='p'/>
    <c w='1 p p'/>
</doc>
```

(a) XML document.



(b) Tree representation.

Fig. 1. Example XML document and its corresponding tree representation.

2.2 Path Expressions

We use a subset of XPath, defined as follows. Let Σ^E and Σ^A be the sets of all element and, respectively, attribute labels that occur in the document; let $\Sigma = \Sigma^E \cup \Sigma^A$. A path expression is a string in the grammar $E ::= \varepsilon \mid a \mid \_ \mid E* \mid E.E$, where ε is a path expression that matches the empty path; $a \in \Sigma$; "_" is a path expression that matches any symbol in Σ; "*" is the usual Kleene star operator; and "." is the usual concatenation operation.

Let x be a node in the tree. $x[\![P]\!]$ denotes the set of nodes (i.e., node *id*s) reached by starting at node x and following paths that conform to P. We shall use $[\![P]\!]$ as an abbreviation for $root[\![P]\!]$.

2.3 Keys

Keys for XML documents were studied in [3]. A key specification consists of: a set of elements (called the *target set*) for which the key holds; and a set of *key paths* that specify the values on which the key is defined. In the following, Q defines a target set and $P_1, \ldots, P_k$ are key paths:

Definition 1 ([3]) *A node x satisfies a key specification $(Q, \{P_1, \ldots, P_k\})$ iff for any two nodes y_1, y_2 in $x[\![Q]\!]$, if for all $i, 1 \leq i \leq k$ there exist $z_1 \in y_1[\![P_i]\!]$ and $z_2 \in y_2[\![P_i]\!]$, such that $z_1 =_v z_2$, then $y_1 =_n y_2$.*

2.4 Attribute Mappings

Attributes in XML documents are not independent entities; rather, they are part of the elements where they are declared. The following definition ties together elements and attributes, and is central to the paper.

Definition 2 *For $x \in \Sigma^E$ and $y \in \Sigma^A$, we call the* attribute mapping *of y over x, denoted by M_x^y, the $\mathcal{I} \times \mathcal{I}$ relation defined by*

$$M_x^y = \{(z, w) : z \in [\![\_*.x]\!], w \in z[\![y]\!]\}$$

That is, the relation M_x^y contains edges in the tree that connect element nodes labeled x and attribute nodes labeled y.

We apply the standard relational projection operator to attribute mappings. For instance, $\pi_E(M_x^y)$ is the set of all unique *id*s for all elements in $[\![\_*.x]\!]$. Similarly for $\pi_A(M_x^y)$.

It is worth mentioning that, in [13], elements of the same label have the same *type*. Thus, we will use the term element type and element label interchangeably. We define the *type* of an attribute mapping, denoted by τ, as $\tau(M_x^y) = x$.

Example 1 The document in Figure 1(a) has the following non-empty attribute mappings: $M_a^x = \{(1,2),(5,6)\}$, $M_a^z = \{(5,7)\}$, $M_b^y = \{(3,4)\}$, and $M_c^w = \{(8,9),(8,10),(8,11)\}$.

As we discuss in the next section, the semantics of ID attributes relates element *types* to attribute *values*. Thus, we define the following.

Definition 3 *The* image *of attribute mapping m, denoted by $\iota(m)$ is the $\mathcal{I} \times \mathcal{V}$ relation defined as*

$$\iota(m) = \{z : z = value(w), w \in \pi_A(m)\}$$

Example 2 The images of the attribute mappings in Example 1 are: $\iota(M_a^x) = \{\mathtt{1},\mathtt{2}\}$, $\iota(M_b^y) = \{\mathtt{1}\}$, $\iota(M_a^z) = \{\mathtt{p}\}$, and $\iota(M_c^w) = \{\mathtt{1},\mathtt{p}\}$.

3 ID Attributes

The semantics of ID attributes in XML are defined as follows [13, Section 3.3.1][14, Section 3.3.8]:

- (ID1) Validity constraint: ID
  Values of type ID must be single-valued. A name must not appear more than once in an XML document as a value of this type; *i.e.*, ID values must uniquely identify the elements which bear them.
- (ID2) Validity constraint: One ID per Element Type
  No element type may have more than one ID attribute specified.

Rule (ID1) above can be split into two conditions: (ID1.a) the values of ID attributes must be singletons; (ID1.b) no two elements in the document may have the same value for their ID attribute. Note that an element may occur without its ID attribute, if the attribute is declared as *IMPLIED* instead of *REQUIRED*. [13, Section 3.3.2]

Definition 4 *An attribute mapping m is a* candidate ID mapping *if it is an injective function, that is,*

$$|m| = |\pi_E(m)| = |\pi_A(m)| = |\iota(m)|$$

Example 3 Among the attribute mappings in Example 1, M_a^x, M_b^y and M_a^z are candidate ID mappings.

Because of rule (ID1.b), we cannot say when a single attribute can be used as an ID attribute; we need to do it for *sets* of attributes, as follows.

Definition 5 (ID Set) *A set of attribute mappings $I = \{m_1, \ldots, m_n\}$ is an* ID set *for an XML document iff: each $m_i \in C$ is a candidate ID mapping,*

$$\bigcap_{m_i \in I} \tau(m_i) = \emptyset, \text{ and } \bigcap_{m_i \in I} \iota(m_i) = \emptyset$$

Example 4 Consider the attribute mappings in Example 1. The following are all ID sets: $I_1 = \{M_a^x\}$, $I_2 = \{M_b^y\}$, $I_3 = \{M_a^z\}$, and $I_4 = \{M_a^z, M_b^y\}$. Obviously, no ID set can contain both M_a^x and M_a^z. Also, no ID set can contain both M_a^x and M_b^y because the images of these mappings share the symbol `1`.

ID Attributes Are Not Unary Keys. Note that ID attributes are different from unary keys. Defining one unary key for each ID attribute in the DTD is not enough. To illustrate this, note that the document in Figure 1(a) satisfies keys $(a, \{x\})$ and $(b, \{y\})$. However, that document does not satisfy a DTD that defines x as ID attribute of a and y as ID attribute of b, because the value 1 appears in both attributes.

We could define a single unary key of the form $(\_*, \{x\})$, where x is some attribute label, and the target set is all elements in the document. This is suggested in [3]. However, doing so requires all ID attributes to have the same label (namely, x). Note that the document in Figure 1(a) satisfies a DTD that defines z as ID attribute for a and x as ID attribute for b; however, we cannot represent both ID attributes as a single unary key, unless we rename either x or z. Furthermore, a document satisfies a unary key of the form $(\_*, \{x\})$ only if all attributes labeled x are in fact ID attributes (e.g., one could not have an IDREF(S) attribute labeled x). Neither restriction is consistent with [13].

Finally, note that, unlike primary keys in the relational setting, that are restricted to be non-null, an ID attribute can be missing; i.e., if M_x^y is an ID attribute mapping, there can be elements of type x in the document that do not define y attributes.

3.1 IDREF and IDREFS Attributes

The semantics of IDREF and IDREFS attributes in XML are much simpler than the semantics of ID attributes [13, Section 3.3.1]:

- Validity constraint: IDREF
  Values of type IDREF *must* be single-valued, and values of type IDREFS *may* be multi-valued; each value must match the value of an ID attribute on some element in the XML document.

Given an ID set I, a mapping m that represents an IDREF attribute is such that $\iota(m) \subseteq \bigcup_{p \in I} \iota(p)$. If m contains multivalued attributes, then m represents an IDREFS attribute.

Evidently, the set of mappings that define IDREF(S) attributes depends on the ID set for the document:

Example 5 Consider the document in Figure 1(a) and the ID sets shown in Example 4. The only ID sets for which there are IDREF(S) attributes are $I_1 = \{M_a^x\}$ (M_b^y represents an IDREF attribute), and $I_4 = \{M_a^z, M_b^y\}$ (M_c^w represents an IDREFS attribute).

3.2 Finding "Good" ID Attributes

The example above shows that in general there will be multiple ID sets for a given document. How do we choose the "best" set? A simple approach is to select a set that contains as many ID attributes as possible; that is, an ID set of maximum cardinality. More generally, we allow attribute mappings to have weights, and choose ID sets of maximum total weight. The weight of a mapping m may depend both on the number of elements that are assigned assigns unique identifiers by m and on the number of reference edges that m induces in the XML graph.

4 The Complexity of Computing ID Sets

In this section we give the complexity of finding an ID set of maximum weight (for maximum cardinality ID sets, it suffices to fix the weights of all mappings to be unitary). For simplicity, and without loss of generality, we take as input a set of candidate ID mappings instead of a document; note that the mappings can be easily computed from the document. We consider initially the decision version of the problem, which we will call K-IDSET:

K-IDSET:
Instance: a set of candidate ID mappings $C = \{m_1, \ldots, m_n\}$ and an integer $K \leq n$. **Question**: is there $I \subseteq C$ such that I is an ID set and $|I| \geq K$?

Theorem 1 *K-IDSET is NP-Complete.*

Proof. The problem is trivially in NP. We show completeness by a reduction from INDEPENDENT SET (IS). Let $G = (V, E)$ be a simple graph with n vertices $v_1, \ldots, v_n$. In order to create the attribute mappings, define $\mathcal{I}$ and $\mathcal{V}$ as follows: let id_v, $v \in V$ denote a unique identifier assigned to vertex v (e.g., the discovery time of the vertex in a depth-first traversal of G); now define $\mathcal{I} = \{id_{v_1}, \ldots, id_{v_n}\}$ and $\mathcal{V} = \mathcal{I}$. Finally, for each $v_i \in V$ define $value(v_i) = id_{v_i}$.

Now define $C = \{m_1, \ldots, m_n\}$, where $m_i = \{(id_{v_i}, id_{v_i})\} \cup \{(id_{v_i}, id_u) : v_i u \in E\}$ and $\tau(m_i) = v_i$, $v_i \in V$. It is clear that G has an independent set of size K iff C has an ID set of size K. Also, C can be constructed in time polynomial on $|V| + |E|$.

The optimization version of the problem is a maximization problem defined as follows:

MAX-IDSET:
Instance: a set of candidate ID mappings $C = \{m_1, \ldots, m_n\}$ and a weight function $w : 2^{\mathcal{I}\times\mathcal{I}} \to \mathbb{Q}^+$. **Goal**: find $I \subseteq C$ such that I is an ID set and there is no $I' \subseteq C$ such that I' is an ID set and $\sum_{m_i \in I'} w(m_i) > \sum_{m_j \in I} w(m_j)$.

As Theorem 1 shows, K-IDSET contains IS as a subproblem. This is in fact bad news for the design of an algorithm for MAX-IDSET; the problem turns out to be complete for the class of NP-hard optimization problems, a notion called $\mathbf{FP}^{\mathbf{NP}}$-completeness [9].

Theorem 2 MAX-IDSET *is* $\mathbf{FP}^{\mathbf{NP}}$*-Complete. Furthermore, unless* $\mathbf{P} = \mathbf{NP}$, MAX-IDSET *has no constant factor approximation algorithm.*

Proof. The MAX INDEPENDENT SET (MIS) is the optimization version of IS; we prove completeness by reduction from MIS. The reduction is essentially the same given for the proof of Theorem 1, with the added restriction that $w(m_i) = w(v_i)$, $v_i \in V$. It is known that a constant factor approximation algorithm for MIS would yield a polynomial time algorithm for IS [4]. Since the reduction given in the proof above is in fact an L-reduction (see [9]), any constant factor approximation algorithm for MAX-IDSET is also a constant factor approximation algorithm for MIS.

The weight function in the maximization problem captures the notion of "goodness" of a candidate ID set and can be any function. As long as w can be computed efficiently (say, in polynomial time), the complexity of the problem does not change.

Theorem 2 essentially tells us that traditional techniques for obtaining approximability bounds (e.g., [12]) for this problem are of no use here. Instead, a heuristic that produces feasible solutions is, in a sense, as good as any other. For more details about MIS and related problems, we refer the reader to [4,9], and also to a recent survey [10].

5 A Heuristic Algorithm for Computing ID Sets

In this section we give a greedy heuristic for the MAX-IDSET problem. Let $\mathcal{M} = \{m_1, \ldots, m_n\}$ be the set of all non-empty attribute mappings in the document. The "goodness" of a mapping will be captured by the following two definitions.

Definition 6 *The* support *of attribute mapping m is defined as*

$$\phi(m) = |m| / \sum_{p \in \mathcal{M}} |p|$$

Input: XML document X

1. $\mathcal{M} \leftarrow$ all attribute mappings in X; $C \leftarrow$ all candidate ID mappings in $\mathcal{M}$ sorted by decreasing size;
2. Compute the weight $w(m)$ of each m in C
3. for t in Σ^E do :
4. Let m be a highest-weight mapping of type t in C
5. Remove from C all mappings of type t except m
6. end for
7. for m in C do :
8. $S \leftarrow$ all mappings in C whose images intersect $\iota(m)$
9. if $w(m) > \sum_{p \in S} w(p)$ then remove all $p \in S$ from C; otherwise remove m from C
10. end for
11. ID set $\leftarrow C$

Fig. 2. Algorithm for computing ID sets.

Definition 7 *The* coverage *of attribute mapping m is defined as*

$$\chi(m) = \left(\sum_{p \in \mathcal{M}, p \neq m} |\iota(m) \cap \iota(p)| \right) / \sum_{p \in \mathcal{M}} |\iota(p)|$$

Note that $\phi(m) \leq 1$, and $\chi(m) \leq 1$.

The support of attribute mapping M_x^y is the fraction of edges in the XML tree that connect x elements to y attributes. The coverage of an attribute mapping measures how much of the image of that mapping occurs elsewhere, as a fraction of all mappings images in the document. Intuitively, a candidate ID mapping with high support represents an attribute that identifies a high proportion of the occurrences of some element type, while a mapping with high coverage represents an attribute that many other attributes refer to; these properties capture different aspects of being a good candidate for an ID attribute.

Figure 2 shows our greedy algorithm for computing ID sets. Initially, we find all attribute mappings, and sort those that are candidate ID mappings by decreasing size. Next, we assign weights to all mappings. Recall that C is an ID set only if for any two mappings $m_1, m_2 \in C$, we have no *type conflict* (i.e., $\tau(m_1) = \tau(m_2)$) and no *image conflict* (i.e., $\iota(m_1) = \iota(m_2)$). Mappings pruned (i.e. removed from C) in an iteration of the algorithm are never considered in later iterations. We deal with type conflicts first (lines 3-6). From all mappings of type t, we choose one of highest weight (breaking ties arbitrarily) and prune all others; we repeat this for all element types. Image conflicts are dealt with next (lines 7-10). For each mapping m, we find the set of mappings that conflict with

its image (denoted by S), and prune either m or all mappings in S depending on the the weight of m and the aggregated weight of S. We use the weight function $w(m) = \alpha \cdot \chi(m) + \beta \cdot \phi(m)$, where α and β determine the relative priority of choosing mappings with high support (i.e., mappings that assign identifiers to more elements) over mappings with high coverage (i.e., mappings that yield more refences between elements).

Correctness. We now show that the output of the algorithm in Figure 2 is always an ID set. First, it is easy to see that lines 3-6 remove all type conflicts that exist in the document. And since the mappings pruned in that phase are not considered further, we need only to show that lines 7-10 remove all image conflicts left in C. Each iteration of this loop removes at least one mapping that causes conflicts in C. We can also show that in each iteration, no mapping in S has been visited in a previous iteration; it follows that all image conflicts are eventually eliminated.

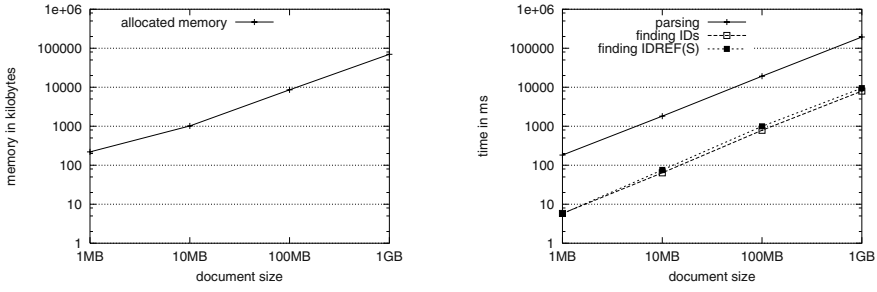
Complexity. We consider the complexity of building the data structures first. Recall that Definition 4 is based on the *sizes* of the relations involved. Also, only the *images* of the mappings are required for finding conflicts, and for computing coverages. Thus, for each $m \in \mathcal{M}$ we only materialize $\iota(m)$, which is kept sorted for faster processing; we also maintain the *values* of $|m|$, $|\pi_E(m)|$, $|\pi_A(m)|$, $|\iota(m)|$, and $w(m)$. Since $|\mathcal{M}| = O(n)$, where n is the size of the document, the space cost of the algorithm is $O(n)$. Note that all the data structures can be built in a single scan of the document.

Now we consider computing the weights of the mappings. Note that, for mapping m, $|m| = O(n)$. Computing $\chi(m)$ requires finding the intersection of $\iota(m)$ with all other mappings in $\mathcal{M}$ (note that mappings in $\mathcal{M}-C$ could represent IDREF attributes). The intersection of two mapping images can be found in $O(n \log n)$ time. Hence, the time to compute $\chi(m)$ is $O(n^2 \log n)$. Computing $\phi(m)$ can be done in $O(\log n)$; thus, line 2 of the algorithm requires $O(n^3 \log n)$ time.

Eliminating type conflicts (lines 3-6) requires finding all conflicting mappings for type t, which can be done in $O(n \log n)$ time; and finding one mapping of highest weight. Thus, lines 3-6 require $O(n^2 \log n)$ time. The complexity of eliminating image conflicts (lines 7-10) is as follows. Computing S is done by searching in C, and requires $O(n^2 \log n)$ time. We can compute the aggregate weight of S as we perform the search. Therefore, the total complexity of our algorithm is $O(n)$ space and $O(n^3 \log n)$ time.

5.1 Approximation Bound

Recall that a constant factor approximation algorithm for this problem is unlikely to exist. The algorithm in Figure 2 is adapted from a well-known heuristic for the MIS problem [9,10]. The following result can be shown. Let δ_m be the number of conflicts that exist for mapping m; let Δ be the largest δ_m in the document



(a) Memory requirements.

(b) Running times.

Fig. 3. Log-log plots of memory requirements and running times for XMark datasets of varying sizes. For the running times, each plot represents the average of of 30 trials.

(note that Δ has a standard meaning in the graph-theoretic formulation of IS). The approximation factor of the algorithm in Figure 2 is $1/\Delta$. Furthermore, all known better algorithms for MIS fail to improve substantially on this bound [10].

6 Finding IDREF(S) Attributes

Once we have an ID set, finding IDREF attributes is easy. Let I be an ID set and m be an attribute mapping not in I. If m is the mapping of an IDREF attribute, then (1) $\iota(m) \subseteq \cup_{m_i \in I} \iota(m_i)$, and (2) $|\pi_A(m)| = |\pi_E(m)|$. If condition (1) holds and $|\pi_A(m)| > |\pi_E(m)|$, then m is a mapping of an IDREFS attribute.

Testing condition (1) above requires computing the intersection of m and all the mappings in I, and, thus, requires time $O(n^2 \log n)$ in the worst case. One way of doing so would be to actually materialize $\cup_{m_i \in I} \iota(m_i)$ and compute all intersections using this new relation. However, although IDREF(S) attributes are untyped, computing the pairwise intersection of m with each individual mapping in I can give extra knowledge about the structure of the document. Finally, testing condition (2) can be done in $O(\log n)$ time using our data structures.

7 Experimental Validation

In this section we present some results that show how our algorithm behaves in practice. Our implementation was done in C++ and used the Xerces SAX parser[1] (in non-validating mode) for scanning the documents and building our data structures. All experiments were run on a 2.4 GHz Pentium 4 machine with Linux and 1GB of memory. Our implementation accepts three parameters: α and β in the weight function; and μ which is a lower bound on the cardinality of the mappings considered. For these experiments, we fix $\alpha = \beta = 1$.

[1] Available at `http://xml.apache.org`.

Table 1. Characteristics of the DTDs used in our experiments. The table shows the number of element declaration rules, ID,IDREF and IDREFS attributes declared in each DTD.

DTD	Element Rules	ID		IDREF		IDREFS	
		REQ.	IMPL.	REQ.	IMPL.	REQ.	IMPL.
XMark (4.2KB)	77	4	0	10	0	0	0
Mondial (4.3KB)	41	11	0	6	4	1	11
W3C DTD (47KB)	141	6	113	13	2	0	2
XDF (30KB)	142	0	11	0	19	1	0

7.1 Scalability

Our first experiment shows how the algorithm scales with document size. We used four documents for the XMark benchmark [11] of varying sizes. Figure 3(a) shows the amount of memory used for representing the data structures for each of the documents. As for running times, Figure 3(b) shows separately the times spent on parsing, computing the ID set, and finding the IDREF(S) attributes based on the ID set chosen, for each of the documents.

Several observations can be made from Figure 3. First, as expected, both the memory requirements and the time for parsing grow linearly with the document size. Second, as far as resources are concerned, the algorithm seems viable: it can process a 10MB document in less than 2 seconds using little memory, on a standard PC. Finally, Figure 3(b) shows that the running time of the algorithm is clearly dominated by the parsing: as one can see, the parsing time is always one order of magnitude higher than any other operation. Of course this is due to the I/O operations performed during the parsing.

7.2 Quality of the Recommendations

The previous experiment showed that the algorithm has reasonable performance. We now discuss its effectiveness. We considered 11 DTDs for real documents found on a crawl of the XML Web (see [8]), for which we were able to find several relatively large documents, and compared the recommendations of our algorithm to the specifications in those DTDs. Due to space limitations, we discuss here the results with 3 real DTDs that illustrate well the behavior of our algorithm with real data: Mondial[2], a geographic database; a DTD for the XML versions of W3C recommendations[3]; and NASA's eXtensible Data Format (XDF)[4], which defines a format for astronomical data sets. We also report the results on the synthetic DTD used in the XMark benchmark.

Recall attributes are specified in DTDs by rules `<!ATTLIST` $e\ a\ t\ p$`>`, where e is an element type, a is an attribute label, t is the type of the attribute,

[2] `http://www.informatik.uni-freiburg.de/~may/Mondial/florid-mondial.html`
[3] `http://www.w3.org/XML/1998/06/xmlspec-19990205.dtd`
[4] `http://xml.gsfc.nasa.gov/XDF/XDF_home.html`

and p is a participation constraint. Of course, we are interested in ID, IDREF and IDREFS types only; the participation constraint is either REQUIRED or IMPLIED. Table 1 shows the number of attributes of each type and participation constraint in the DTDs used in our experiments.

The DTDs for XDF and XML specifications are *generic*, in the sense that they are meant to capture a large class of widely varying documents. We were not able to find a single document that used all the rules in either DTD. The XMark and Mondial DTDs, on the other hand, describe specific documents.

Metrics. This section describes the metrics we use to compare the recommendations of our algorithm to the corresponding attribute declarations in the DTD. For participation constraints, if $\frac{|\pi_E(M_x^y)|}{|[\![\_{*}.x]\!]|} = 1$ we will say y is REQUIRED for x; otherwise, we say y is IMPLIED.

We consider two kinds of discrepancies between the recommendations made by the algorithm and the specifications in the DTDs: *misclassifications* and *artifacts*. A misclassification is a recommendation that does not agree with the DTD, and can occur for two reasons. First, there may be attributes described as ID or IDREF in the DTD but ignored by the algorithm. Second, there may be attributes specified as ID in the DTD but recommended as IDREF by the algorithm, or vice-versa.

A rule `<!ATTLIST` e a t p`>` is misclassified if the algorithm either does not recommend it or recommends it incorrectly, except if:

- e is declared optional and $[\![\_{*}.e]\!] = \emptyset$;
- a is declared IMPLIED and $\pi_A(M_e^a) = \emptyset$;
- a is an IDREFS attribute, $|\pi_E(M_e^a)| = |M_e^a|$, and our algorithm recommends it as IDREF.

Artifacts occur when the algorithm recommends attributes that do not appear in any rule in the DTD as either ID or IDREF. For example, an attribute that occurs only once in the document (e.g., at the root) might be recommended as an ID attribute. Artifacts may occur for a variety of reasons; it may be that an attribute serves as an ID for a particular document, but not for all, or that it was not included in the DTD because the user is not aware of or does not care about this attribute's properties.

Results. Table 2 compares the number of correct classifications to the number of misclassifications and artifacts produced for our test documents, all of which were valid according to the respective DTDs. For the W3C and XDF DTDs we ran the algorithm on several documents, and we report here representative results. For clarity, we report the measures and the accuracy scores for IDREF and IDREFS attributes together.

As one can see, our algorithm finds all ID and IDREF attributes for the Mondial DTD; also, no artifacts are produced for that document. The algorithm also performs very well for the XMark document. The misclassifications reported

Table 2. Analysis of our algorithm on different documents. The table shows, for each document and value for μ: the number of mappings considered; the number of ID/IDREF attributes that were correctly classified; the number of ID/IDREF attributes that were misclassified; and the number of artifacts that were recommended as ID/IDREF attributes. The accuracy is defined as (Correct)/(Correct+Misclassifications).

Document	μ	$\lvert\mathcal{M}\rvert$	Correct		Misclass.		Accuracy		Artifacts	
			ID	IDREF	ID	IDREF	ID	IDREF	ID	IDREF
XMark (10MB)	1	16	3	9	1	1	0.75	0.90	0	0
Mondial (1.2MB)	1	48	11	23	0	0	1.00	1.00	0	0
XML Schema Part 2 (479KB)	1	91	13	11	0	0	1.00	1.00	9	16
	2	69	12	10	1	1	0.92	0.91	5	10
	3	62	11	10	2	1	0.85	0.91	2	7
	4	61	11	10	2	1	0.85	0.91	2	7
	5	57	11	10	2	1	0.85	0.91	1	7
XDF document (38KB)	1	50	4	2	0	0	1.00	1.00	9	3
	2	40	3	1	1	0	0.75	0.50	6	0
	3	31	3	1	1	0	0.75	0.50	4	0

occur because there are two mappings with identical images in the document, and our algorithm picks the "wrong" one to be the ID attribute. We were not able to find all ID and IDREF attributes for the other two DTDs. However, this was expected, given that these DTDs are too broad and the instances we examined exercise only a fraction of the DTD rules. Note that the XDF DTD is roughly as large as the test document we used; in fact, most XDF documents we found were smaller than the XDF DTD.

Table 2 also shows a relatively high number of artifacts that are found by our algorithm, especially for the XDF DTD. Varying the minimum cardinality allowed for the mappings reduces the number of artifacts considerably; however, as expected, doing so also prunes some valid ID and IDREF mappings. It is interesting that some of the artifacts found appear to be natural candidates for being ID attributes. For example, one ID artifact for the XML Schema document contained email addresses of the authors of that document. Also, most of the IDREF artifacts refer to ID attributes that are correctly classified by our algorithm. For example, in the XML Schema document with $\mu = 2$, only 1 IDREF artifact refers to an ID artifact.

8 Conclusion

We discussed the problem of finding candidate ID and IDREF(S) attributes for schemaless XML documents. We showed this problem is complete for the class of NP-hard optimization problems, and that a constant rate approximation algorithm is unlikely to exist. We presented a greedy heuristic, and showed experimental results that indicate this heuristic performs well in practice.

We note that the algorithm presented here works in main memory, on a single document. This algorithm can be extended to deal with collections of documents by prefixing document identifiers to both element identifiers and attribute values. This would increase its resilience to artifacts and the confidence in the recommendations. Also, extending the algorithm to secondary memory should be straightforward.

As our experimental results show, our simple implementation can handle relatively large documents easily. Since the parsing of the documents dominates the running times, we believe that the algorithm could be used in an interactive tool which would perform the parsing once, and allow the user to try different ID sets (e.g., by requiring that certain attribute mappings be present/absent). This would allow the user to better understand the relationships in the document at hand.

Acknowledgments. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada and Bell University Laboratories. D. Barbosa was supported in part by an IBM PhD. Fellowship. This work was partly done while D. Barbosa was visiting the OGI School of Science and Engineering.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web*. Morgan Kauffman, 1999.
2. M. Arenas, W. Fan, and L. Libkin. On verifying consistency of XML specifications. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 259–270, 2002.
3. P. Buneman, S. Davidson, W. Fan, C. Hara, and W.-C. Tan. Keys for XML . In *Proceedings of the Tenth International Conference on the World Wide Web*, pages 201–210. ACM Press, 2001.
4. M. Garey and D. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. Freeman, 1979.
5. M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: A system for extracting document type descriptors from XML documents. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 165–176, Dallas, Texas, USA, May 16-18 2000.
6. G. Grahne and J. Zhu. Discovering approximate keys in XML data. In A. Press, editor, *Proceedings of the eleventh international conference on Information and knowledge management*, pages 453–460, McLean, Virginia, USA, November 4-9 2002.
7. H. Mannila and K.-J. Räihä. On the complexity of inferring functional dependencies. *Discrete Applied Mathematics*, 40(2):237–243, 1992.
8. L. Mignet, D. Barbosa, and P. Veltri. The XML Web: a first study. In *Proceedings of The Twelfth International World Wide Web Conference*, 2003. To appear.
9. C. Papadimitriou. *Computational Complexity*. Addison Wesley, 1995.
10. V. T. Paschos. A survey of approximately optimal solutions to some covering and packing problems. *ACM Computing Surveys*, 29(2):171–209, June 1997.

11. A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 974–985, Hong Kong, China, August 2002.
12. V. Vazirani. *Approximation Algorithms*. Springer Verlag, 2003.
13. Extensible markup language (XML) 1.0 - second edition. W3C Recommendation, October 6 2000. Available at: `http://www.w3.org/TR/2000/REC-xml-20001006`.
14. XML Schema part 1: Structures. W3C Recommendation, May 2 2001. Available at: `http://www.w3.org/TR/xmlschema-1/`.

XML Stream Processing Quality

Sven Schmidt, Rainer Gemulla, and Wolfgang Lehner

Dresden University of Technology, Germany
{ss54,rg654452,lehner}@inf.tu-dresden.de,
http://www.db.inf.tu-dresden.de

Abstract. Systems for selective dissemination of information (SDI) are used to efficiently filter, transform, and route incoming XML documents according to pre-registered XPath profiles to subscribers. Recent work focuses on the efficient implementation of the SDI core/filtering engine. Surprisingly, all systems are based on the best effort principle: The resulting XML document is delivered to the consumer as soon as the filtering engine has successfully finished. In this paper, we argue that a more specific Quality-of-Service consideration has to be applied to this scenario. We give a comprehensive motivation of quality of service in SDI-systems, discuss the two most critical factors of XML document size and shape and XPath structure and length, and finally outline our current prototype of a Quality-of-Service-based SDI-system implementation based on a real-time operating system and an extention of the XML toolkit.

1 Introduction

XML documents reflect the state-of-the-art for the exchange of electronic documents. The simplicity of the document structure in combination with comprehensive schema support are the main reason for this success story. A special kind of document exchange is performed in XML-based SDI systems (selective dissemination systems) following the publish/subscribe communication pattern between an information producer and information subscriber. On the one hand, XML documents are generated by a huge number and heterogeneous set of publishing components (publisher) and given to a (at least logically) central message broker. On the other hand, information consumers (subscriber) are registering subscriptions at the message broker usually using XPath or XQuery/XSLT expressions to denote the profile and delivery constraints. The message broker has to process incoming by filtering (in the case of XPath) or transforming (in the case of XQuery/XSLT) the original documents and deliver the result to the subscriber (figure 1).

Processing XML documents within this streaming XML document application is usually done on a best effort basis i.e. subscribers are allowed to specify only functionally oriented parameters within their profiles (like filtering expressions) but no parameters addressing the quality of the SDI service. Quality-of-Service in the context of XML-based SDI systems is absolutely necessary for example in application area of stock exchange, where trade-or-move messages

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 195–207, 2003.

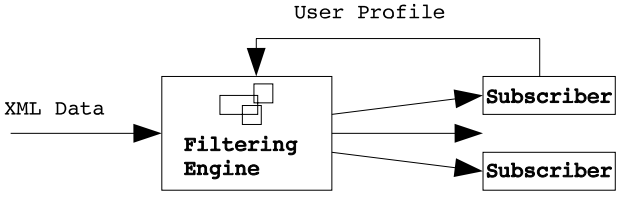


Fig. 1. basic logical architecture of SDI systems

have to be delivered to registered users within a specific time slot so that given deadlines can be met. Although a quality-of-service-based process scheduling of XML filtering operations yields typically less overall system throughput, the negotiated quality-of-service for the users can be guaranteed.

Contribution of the Paper: Scheduling and capacity planning in the context of XML documents and XPath expression evaluation is difficult but may be achieved within a certain framework. This topic is intensively discussed in the context of this paper. Specifically, the paper illustrates how the resource consumption of filtering, a typical operation in SDI systems, depends on the shape, size and complexity of the document, on the user profile specified as a filter expression, and on the efficiency of the processor which runs the filters against the documents. We finally sketch an XML-based SDI system environment which is based on a real-time operating system and is thus capable of providing Quality-of-Service for subscribers.

Structure of the Paper: The paper is organized as follows: In the next section, the current work in the area of XML-processing related to our approach is summarized. Section 3 considers Quality-of-Service perspectives for data processing in SDI systems and proposes a list of requirements regarding the predictability of XML data, filters and processors to consequently guarantee a user-defined quality of service. In section 4 the QoS parameters are used to obtain resource limits for QoS planning and in section 5 ideas about the architecture of a QoS-capable SDI system are given. Section 6 outlines the current state of our prototypical implementation based on the XML toolkit and on a real-time operating system. Section 7 finally concludes the paper with a short summary.

2 Related Work

The process of efficiently filtering and analyzing streaming data is intensively discussed in recent publications. Many mechanisms to evaluate continuous/standing queries against XML documents have been published. The work in this area ranges from pure processing efficiency to the handling of different data sources

[1], adoption of the query process by dynamic routing of tuples [5] and grouping of queries based on similarity including dynamic optimization of these query groups [12]. Surprisingly and to the best of our knowledge, no system incorporates the idea of quality of service for the filtering process in SDI systems as a first-class parameter. Since our techniques and parameter are based on previous work, we have to sketch the accompanying techniques:

One way to establish the filtering of XML documents with XPath expressions consists in using the standard DOM representation of the document. Unfortunately, using the DOM representation is not feasible for larger XML documents. The alternative way consists in relying on XML stream processing techniques [2,8,4,3] which particularly construct automatons based on the filter expressions or use special indexes on the streaming data. This class of XPath evaluations will be the base for our prototypical implementation outlined in section 6.

In [13] some basic XML metrics are used to characterize the document structure. Although their application area is completely different to Quality-of-Service in XML-based SDI systems, we exploit the idea of XML metrics as a base to estimate the resource consumption for the filtering process of a particular XML document.

3 XML–Based QoS–Perspectives

Before diving into detail, we have to outline the term "Quality-of-Service" in the context of SDI systems. In general a user is encouraged to specify QoS requirements regarding a job or a process a certain system has to perform. These requirements usually reflect the result of a negotiation between user and system. Once the system has accepted the user's QoS requirement, the system guarantees to meet these requirements. Simple examples of quality subjects are a certain precision of a result or meeting a deadline while performing the user's task.

The benefit for the user is predictability regarding the quality of the result or the maximal delay of receiving the result. This is helpful in a way that users are able to plan ahead other jobs in conjunction with the first one. As a consequence from the system perspective, adequate policies for handling QoS constraints have to be implemented. For example to guarantee that a job is able to consume a certain amount of memory during its execution, all the memory reservations have to be done in advance when assuring the quality (in this case the amount of available memory). In most cases even the deadline of the job execution is specified as a quality of service constraint. A job is known to require a certain amount of time or an amount of CPU slices to finish. Depending on concurrently running jobs in the system a specific resource manager is responsible for allocating the available CPU slices depending on the QoS specified time constraints. Most interesting from an SDI point of view is that every time a new job negotiates about available computation time or resources in general, an admission control has to either accept or reject the job according to the QoS requirements.

QoS management is well known for multimedia systems especially when dealing with time dependent media objects like audio and video streams. In such a

case the compliance to QoS requirements may result in video playback without dropping frames or in recording audio streams with an ensured sampling frequency.

Depending on the type of SDI system, deadlines in execution time or in data transmission are required from a user point of view. An example is the NASDAQ requirement regarding the response time to Trade-or-Move messages or (more generally) the message throughput in the stock exchange systems like Philadelphia Stock Exchange which are measured in nearly one hundred thousand messages (and therefore filtering processes) per second. To ensure quality of service for each single SDI subscriber job and fairness between all subscribers, SDI systems based on the best effort principle (i.e. process incoming messages as fast as the can without any further optimization and scheduling) are not sufficient for those critical applications. A solid basis should be systems with a guaranteed quality of its services.

Figure 2 shows the components which have to be considered when discussing quality of service in the context of XML-based SDI systems. The data part consists of XML messages which stream into the system. They are filtered by a XPath processor operating on top of a QoS capable operating system.

- processor: the algorithm of the filtering processor has to be evaluated with regard to predictability. This implies that non-deterministic algorithms can be considered only on a probability basis, while the runtime of deterministic algorithms can be precomputed for a given set of parameters.
- data: the shape and size of an XML document is one critical factor to determine the behavior of the algorithm. In our approach, we exploit metrics (special statistics) of individual XML documents to estimate the required capacity for the filtering process in order to meet the quality of service constraints.
- query: the second determining factor is the size and structure of the query to filter (in the case of XPath) or to transform (in the case of XQuery/XSLT) the incoming XML document. In our approach, we refer to the type and number of different location steps of XPath expressions denoting valid and individual subscriptions.
- QoS capable environment: the most critical point in building an SDI system considering QoS parameters is the existence of an adequate environment. As shown in section 6.1, we rely on a state-of-the-art real-time operating system which provides native streaming support with QoS. Ordinary best effort operating systems are usually not able to guarantee a certain amount of CPU time and/or data transfer rate to meet the subscription requirement.

4 Using Statistics

As outlined above, the shape and size of an XML document as well as the length and structure of the XPath expressions are the most critical factors estimating the overall resource consumption regarding a specific filter algorithm. The factors are described in the remainder of this section.

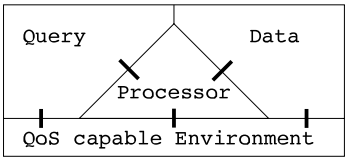


Fig. 2. QoS determining factors in XML-based subscription systems

4.1 Complexity of XML Data

In [13] different parameters for describing the structure of XML documents and schemes are outlined on a very abstract level. The so called *metrics* evolve from certain scheme properties and are based on the graphical representation of the XML scheme. The five identified metrics are:

- size: counted elements and attributes
- structure: number of recursions, IDREFs
- tree depth
- fan-in: number edges which leave a node
- fan-out: number of edges which point to a node

Obviously these metrics are related to the complexity of a document and strongly influence the resources needed to query these data. Depending on the requirements of the specific SDI system we may add some more parameters or we may only record metrics on a higher level (like the documents DTD). However, the question is how to obtain these statistics. We propose three different directions, outlined in the following:

- producer given statistics: We require the document statistics from the producer of an XML document. The statistics are gathered during the production process of the document and transferred to the filtering engine together with informational payload. This method, however, requires cooperative producers, fulfilling the requirements prescribed in a producer-filtering engine document transmission protocol. Examples are parameters like the DTD of a document (or of a collection of documents) and the document size (length) itself.
- generating statistics: We apply the method of gathering statistics in centralized systems to the SDI environment. This approach however implies that the stream of data will be broken because the incoming data has to be preprocessed and therefore stored temporarily. As soon as the preprocessing has completely finished, the actual filtering process may be initiated. Obviously, this pipeline breaking behavior of the naive statistic gathering method does not reflect a sound basis for efficiently operating SDI systems.

– cumulative statistics: as an alternative to the producer given statistics we start with default values. Then statistics of the first document are gathered during the filtering step. These statistical values are merged with the default values and used to estimate the overhead for the following document of the same producer. In general, the statistics of the i-th document are merged with the statistics of the documents 1 to i-1 and used to perform the capacity planning of the i+1-th document of the same producer. This method can be applied only in a "static" producer environment.

The assumption of creating data statistics at the data source is relatively strong but might improve the overall quality of the system tremendously. As a result of the above discussion the, set of document statistics to be used for QoS planning has to be chosen carefully in terms of resource consumption of the filtering engine. Section 5 gives further explanations on this.

4.2 Complexity of XPath Evaluation

In the context of XPath evaluation, the structure and the number of the XPath expressions are combined with the filter algorithm itself. It does not make sense to consider these two perspectives (i.e. XPath and processor) independently from each other because the requirements regarding the XPath expressions strongly vary in terms of the underlying evaluation engine.

Due to extensive main memory requirements, the well-known DOM based evaluation is not applicable for the purpose of SDI systems and will not be considered in this paper. Therefore we focus on the family of stream based XML filtering algorithms. One of the main ideas in this field is using an automaton which is constructed with regard to the set of given XPath expressions reflecting single subscriptions. Such an automaton has a number of different states which may become active while the processor is scanning through the XML document. The set of techniques may be classified according to the deterministic or non-deterministic behavior of the automaton.

Whereas for an NFA (non-deterministic finite automaton) the required amount of memory for representing the states determined by the number of states per automaton and the number of XPath expressions is known in advance, the processing time is indeterministic and difficult to estimate. In opposite to the NFA, the DFA (deterministic finite automaton) has no indeterminism regarding the state transitions but consumes more memory because of the amount of potential existing automaton states. In real application scenarios with thousands of registered XPath expressions it is not possible to construct all automaton states in main memory. The solution provided in [3] is to construct a state when it is needed the first time (data driven).

From the QoS point of view, XPath evaluation mechanisms with predictable resource consumption are of interest. It is necessary to consider worst-case and best-case scenarios as the basic resource limits. In the case of DFA worst case assumptions will not be sufficient, because the worst case is constructing *all*

states regardless of the XML document, so that more accurate approaches for estimating memory and CPU usage are required.

We make use of the XML toolkit implementation as a representative of deterministic automatons. For gathering the QoS parameters basically the memory consumption and the CPU usage are considered. In the XMLTK a lazy DFA is implemented. This means that a state is constructed on demand so that memory requirements may be reduced.

[3] proposes different methods for gathering the resource requirements of XML toolkit automaton. This is possible when making some restrictions regarding the data to be processed and regarding the filtering expressions. For example the XML documents have to follow a simple DTD[1] and the XPath expressions have to be linear and may only make use of a small set of location steps.

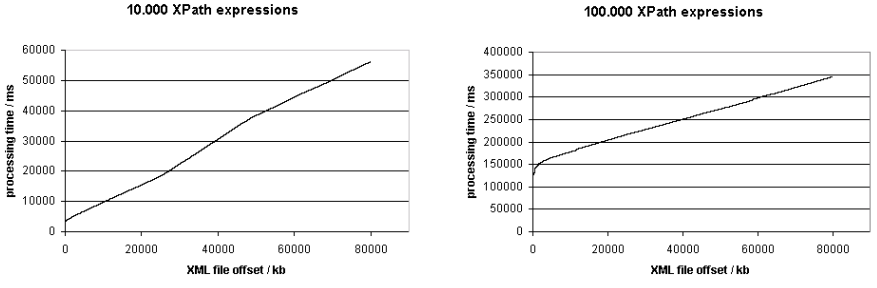


Fig. 3. CPU Usage with increasing number of XPath expressions

CPU Resource Utilization: Fortunately, one property of a lazy DFA is less overall memory consumption. The drawback are delays for state transitions in the warm-up phase. The cumulative time of state transitions and state creations is illustrated in figure 3. As long as not all states are constructed, the time needed for one state transition consists of the state creation time and the transition time itself. In terms of resource management the following approach may help: The time required for a certain number of state transitions may be calculated as follows:

$t(x) \leq x * t_s + t_{c-all}$

where x is the number of the steps initiating a state transition[2], t_s is the time required for a state transition (assumed to be constant for one automaton, independently of the number of registered XPath expressions) and t_{c-all} is the

[1] No cycles are allowed except to the own node.

[2] Every open and close tag causes a state transition. Therefore it should be possible to use statistics for estimating the number of state transitions regarding the XML file size.

time required for the creation of *all* states of the automaton (the time required for creating *one* state depends on the number of registered XPath expressions, so we use the cumulative value here).

The number of constructed states in the warm-up phase is obviously smaller then the number of all states required by the document. Using the time required to construct *all* states (t_{c-all}) in the formula will give an upper bound of the computation time. Assuming the warm-up phase to be shorter than the rest of the operating time, $t(x)$ is a reasonable parameter for resource planning.

Using the sketched approach of CPU utilization, the time for processing a single document may be scheduled just before the document arrives at the DFA. In section 5 an architecture for filtering subsequent XML documents is proposed.

Memory Consumption: Regarding to [3] there is an upper bound of the number of constructed states for a lazy DFA. This value depends on the structure of the registered XPath expressions as well as on the DTD of the XML documents. We assume that the memory requirements for each state can be calculated, so this upper bound may also be used for estimating an overall memory consumption better then worst-case. Having the estimated number of states and the memory used per state available, the required memory can be calculated. Hence for a static set of registered filter expressions and for a set of documents following *one* DTD the required memory is known and may be reserved in advance.

5 Architecture of a QoS-Capable SDI System

In SDI systems it is common that users subscribe to receive a certain kind of information they are interested and a (static) set of data sources register their services at the SDI system with a certain information profile.

This results in consecutive XML documents related to each other. These relationships may be used to optimize the statistic runs. Consider an example like stock exchange information received periodically from a registered data source (from a stock exchange). The consecutive documents reflect update operations in the sense that an update may exhibit the whole stock exchange document with partially modified exchange rates or it *only* consists of the updated exchange rates wrapped by an XML document. In summary, updates logically consist of:

- element content update
- update in document structure
- updated document with a new DTD or new XML scheme

All three kinds of update have to be accepted to preserve the flexibility of XML as a data exchange format.

Figure 4 sketches the idea of a QoS-capable SDI system. Starting with one data source disseminating a sequence of XML documents, some consecutive documents will follow the same DTD. The DTD is taken as a basis for the first set of QoS-determining parameters (I). On the subscriber side many XPath expressions are registered at the SDI system. The structure and length of these subscriptions

forms the second parameter set (II). Finally the XML document characteristics (in our case only the document length) is the third parameter (III). All parameters are used by the scheduler which is able to determine low-level resources like main memory and CPU consumption accordingly to the deployed filter mechanism. After negotiating with the operating systems resource manager, a new document filter job may be admitted or rejected (admission control). In the former case the scheduler reserves the resources at the resource manager and - as a consequence - the real-time operating system environment has to guarantee the required memory as well as the CPU time. The filtering job may then start and will successful finish after the predetermined amount of time.

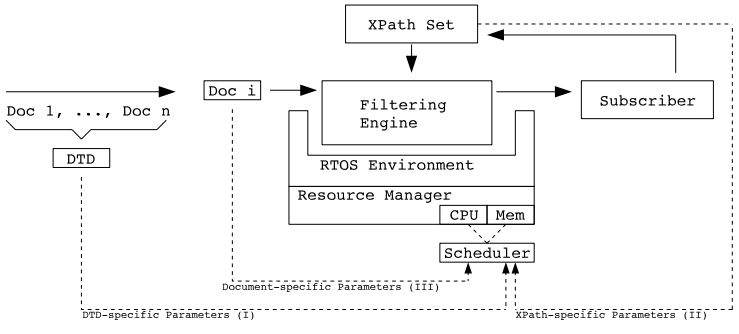


Fig. 4. data and information flow in a QoS SDI system

Periods: As soon as one of the factors (I, II, III) changes, the filter process must be re-scheduled. Due to the nature of an SDI system we assume the set of XPath expressions to be fixed for a long time (continuous/standing queries). The data will arrive at the SDI system depending on the frequency of dissemination of the data sources, so the shortest period will be the processing time of one single document followed by the period the DTD changes[3].

Unfortunately it is hard to create only *one* schedule for *a sequence* of documents because the documents parameter (III, the documents length) seems unpredictable in our case. As a result the sketched SDI system will initially perform an ad-hoc scheduling for each arriving XML document independently. A periodic behavior of the filtering system may be realized on a macro level (e.g. in cases of document updates while not changing the document structure and size) or on a micro level (a certain amount of CPU time is reserved for performing the different state transitions).

[3] In a worst case every new XML document follows another DTD. Hopefully lots of documents of the same data source will follow the same DTD.

6 Implementational Perspectives

As already outline in section 3 the use of a real-time operating system (RTOS) reflects a necessary precondition for the stated purpose of pushing QoS into SDI systems.

6.1 Real-Time Operating System Basis

A common property of existing RTOSes is the ability to reserve and to assure resources for user processes. Generally there are the two types of the soft and hard real-time systems. In hard real-time systems the resources, once assured to a process, have to be realized without any compromise. Examples are systems for controlling peripheral devices in critical application environments such as in medicine. Real-time multimedia systems for example are classified as soft real-time systems, because it is tolerable to drop a single frame during a video playback or to jitter in sampling frequency while recording an audio clip. In general, soft real-time systems only guarantee that a deadline is met with a certain probability (obviously as near as possible to 100 percent). Motivated by the XML document statistics, we propose that a soft real-time system is sufficient enough to provide a high standard quality-of-service in SDI systems. Probability measures gained through the XML document statistics in combination with finite-state automatons in order to process XPath filtering expressions can be directly mapped onto the operating system level probability model.

6.2 DROPS Environment

For our prototypical implementation, we chose the Dresden Real-Time Operating System (DROPS, [11]) for our efforts spent in integrating OoS strategies into an SDI system. DROPS is based on the L4-micro-kernel family and aims to provide Quality-of-Service guarantees for any kind of application. The DROPS Streaming Interface (DSI, [17]) supports a time-triggered communication for producer-consumer-relationships of applications which can be leveraged for connecting data sources and data destinations to the filtering engine. The packet size of the transfer units (XML chunks) are variable and may therefore depend on the structure of the XML stream. Memory and CPU reservation is performed by a QoS resource manager. Since the management of computing time is based on the model of periodic processes, DROPS is an ideal platform for processing streaming data. A single periodic process reflects either an entire XML document at a macro level (as a unit of capacity planning) or single node of the XML document and therefore a single transition in an XPath filtering automaton at a micro level (as a unit of resource consumption, e.g. CPU usage, memory usage). Moreover schedulable and real-time capable components like a file system or a network connection exist to model data sources and destinations.

Figure 5 outlines the components performing the stream processing at the operating level. The query processor is connected through the DROPS Streaming Interface (DSI) to other components for implementing the data streaming

constraint by quality of service parameters given at the start of the filtering process at the macro level, i.e. for each XML document. The query engine is a streaming XPath processor based on the XML toolkit ([3]).

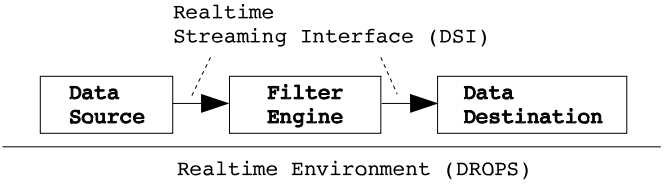


Fig. 5. connecting the involved components via the DROPS Streaming Interface

6.3 Adaption of the XML Toolkit

Due to the nature of SDI systems, stream based XPath processing techniques are more adequate for efficiently evaluating profiles against incoming XML documents because of the independence from document structure and document size. In our work we exploit the XML toolkit as a base for quality of service considerations. XMLTK implements XPath filtering on streaming data by constructing a deterministic finite automaton based on the registered XPath expressions. The prototypical implementation focuses on the core of XMLTK and ports the algorithms to the DROPS runtime environment (figure 6). Additionally the current implementation is extended to capture the following tasks:

- resource planning: Based on sample XML documents, the prototypical implementation plays the role of a proof-of-concept system with regard to the document statistics and the derivation of resource description at the operating system level.
- resource reservation: Based on the resource planning, the system performs resource reservation and is therefore able to decide whether to accept or reject a subscription with a specific quality-of-service requirement.
- filtering process scheduling: After the notification of an incoming XML document, the system provides the scheduling of the filtering process with the adequate parameters (especially memory and CPU reservation at the operating system level)
- monitoring filtering process: after scheduling according to the parameters, the system starts the filtering process, monitors the correct execution, and performs finalizing tasks like returning the allocated resources, etc.

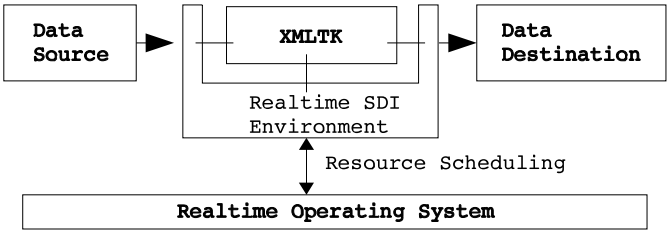


Fig. 6. XMLTK and the DROPS Environment

7 Summary and Conclusion

This paper introduces the concept of quality-of-service in the area of XML-based SDI systems. Currently discussed approaches in the SDI context focus on the efficiency of the filtering process but do not discuss detailed quality-of-service parameters. In this paper, we outline the motivation of Quality-of-Service in this application context, intensively discuss the two critical factors, XML document and XPath queries, to accurately estimate the resource consumption for a single XML document, and outline the requirements of the underlying real-time operating system. The current implementation is based on the DROPS operating system and extends the core components of the XML toolkit to parameterize the operating system. Although this paper sketches many points in the context of quality-of-service in XML-based subscription systems, we are fully aware that many issues are still open and therefore represent the subject of further research. However, filtering engines working on a best effort basis are definitely not the real answer to the challenge of a scalable and high-performing subscription system.

References

1. Altinel, M.; Aksoy, D.; Baby, T.; Franklin, M.; Shapiro, W.; Zdonik, S.: DBIS Toolkit: Adaptable Middleware for Large Scale Data Delivery in Proc. ACM SIGMOD Conference, Philadelphia, PA, pages 544–546, June 1999
2. Altinel, M.; Franklin, Michael J.: Efficient Filtering of XML Documents for Selective Dissemination of Information, in Proc. of the VLDB Conference, Cairo, Egypt, pages 53–64, September 2000
3. Avila-Campillo, I.; Green, T.J.; Gupta, A; Onizuka, M.; Raven, D.; Suciu, D. : XMLTK: An XML Toolkit for Scalable XML Stream Processing in Proc. of Programming Language Technologies for XML (PLAN-X) workshop, Pittsburgh, PA, October 2002
4. Chan, C.Y.; Felber, P.; Garofalakis, M.N.; Rastogi, R.: Efficient Filtering of XML Documents with XPath Expressions, in Proc. of the ICDE, San Jose, California, February 2002

5. Chandrasekaran, S.; Cooper, O.; Deshpande, A.; Franklin, M.J.; Hellerstein, J.M.; Hong, W.; Krishnamurthy, S.; Madden, S.; Raman, V.; Reiss, F.; Shah, M.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World, in Proc. of the CIDR Conference, Asilomar, CA, January 2003
6. Chaudhri B. A., Rashid A., Zicari, R.: XML Data Management – Native XML and XML-Enabled Database Systems Addison-Wesley, 2003
7. Chen, J.; DeWitt, David J.; Tian, F.; Wang, Y: NiagaraCQ: A Scalable Continuous Query System for Internet Databases, in Proc. of the: ACM SIGMOD Conference on Management of Data, Dallas, Texas, pages 379–390, May 2000
8. Diao, Y.; Fischer, P.; Franklin, Michael J.; To, R.: YFilter: Efficient and Scalable Filtering of XML Documents, in Proc. of the ICDE Conference, San Jose, California, pages 341–342, February 2002
9. Green, T.J.; Miklau, G.; Onizuka, M.; Suciu, D.: Processing XML Streams with Deterministic Automata, in Proc. of ICDT, Siena, Italy, pages 173–189, January 2003
10. Hamann, C.-J.; Märcz, A.; Meyer-Wegener, K.: Buffer Optimization in Realtime Media Servers using Jitter-contrained Periodic Streams, technical report, TU-Dresden, January 2001
11. Härtig, H.; Baumgartl, R.; Borris, M.; Hamann, C.-J.; Hohmuth, M.; Mehnert, F.; Reuther, L.; Schönberg, S.; Wolter, J.: DROPS - OS Support for Distributed Multimedia Applications, in Proc. of the ACM SIGOPS European Workshop, Sintra, Portugal, September 7–10, 1998
12. Ives, Zachary G.; Halevy, Alon Y.; Weld, S. Daniel: An XML Query Engine for Network-Bound Data, in: VLDB Journal 11(4), pages 380–402, 2002
13. Klettke, M., Meyer, H.: XML & Datenbanken dpunkt.verlag, 2003
14. Lehner, W.: Subskriptionssysteme – Marktplatz für omnipräsente Informationen, Teubner Texte zur Informatik, Band 36, B.G. Teubner Verlag Stuttgart/Leipzig/Wiesbaden, 2002
15. Lehner, W.: Datenbanktechnologie für Data-Warehouse-Systeme, dpunkt.verlag, Heidelberg, 2003
16. Lehner, W.; Irmert, F.: XPath-Aware Chunking of XML Documents, in Proc. of GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW) Leipzig, Germany, pages 108–126, February 2003
17. Löser, J.; Härtig, H.; Reuther, L.: A Streaming Interface for Real-Time Interprocess Communication, technical report, TU-Dresden, August 2001
18. Ludäscher, B.; Mukhopadhyay, P.; Papakonstantinou, Y.: A Transducer-Based XML Query Processor, in Proc. of the VLDB Conference, Hongkong, China, pages 227–238, August 2002
19. Mannino, M. V., Chu, P., Sager, T.: Statistical Profile Estimation in Database Systems in: ACM Computing Surveys, 20(3), 1988, pages 191–221

Representing Changes in XML Documents Using Dimensions

Manolis Gergatsoulis[1] and Yannis Stavrakas[2]

[1] Department of Archive and Library Sciences, Ionian University,
Palea Anaktora, Plateia Eleftherias, 49100 Corfu, Greece.
manolis@ionio.gr
http://www.ionio.gr/∼manolis/
[2] Knowledge & Database Systems Laboratory
Dept. of Electrical and Computing Engineering
National Technical University of Athens (NTUA), 15773 Athens, Greece.
ys@dblab.ntua.gr

Abstract. In this paper, we present a method for representing the history of XML documents using Multidimensional XML (MXML). We demonstrate how a set of basic change operations on XML documents can be represented in MXML, and show that temporal XML snapshots can be obtained from MXML representations of XML histories. We also argue that our model is capable to represent changes not only in an XML document but to the corresponding XML Schema document as well.

1 Introduction

The management of multiple versions of XML documents and semistructured data is an important problem for many applications and has recently attracted a lot of research interest [3,13,4,5,16,17]. One of the most recent approaches appearing in the literature [16,17] proposes the use of Multidimensional OEM (MOEM), a graph model designed for *multidimensional semistructured data (MSSD)*[15] as a formalism for representing the history of time-evolving semistructured data (SSD). MSSD are semistructured data that present different facets under different *contexts*. A context represents alternative *worlds*, and is expressed by assigning values to a set of user-defined variables called *dimensions*. The basic idea behind the approach proposed in [16,17] is to use MOEM with a time dimension whose values represent the time points under which an OEM object holds. In order to use MOEM to represent changes in OEM databases, a set of basic change operations for MOEM graphs as well as a mapping from changes in an OEM database to MOEM basic change operations has been defined. An interesting property of MOEM graphs constructed in this way is that they can give temporal snapshots of the OEM database for any time instance, by applying a process called *reduction*. Queries on the history of the changes can also be posed using MQL [18], a query language for MOEM.

Following the main ideas presented in [16,17], in this paper we address the problem of representing and querying histories of XML documents. We propose

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 208–222, 2003.

the use of *Multidimensional XML* (MXML) [7,8], an extension of XML which shares the same ideas as MSSD, in order to represent context-dependent information in XML. MXML is used as a formalism to represent and manipulate the histories of XML documents. The syntax particularities of XML require to adapt the MOEM approach described in [16,17] so that they are taken into account. The main contributions of the present work can be summarized as follows:

1. We consider four basic change operations on XML documents and show how the effect of these operations on (the elements and attributes of) XML documents, can be represented in Multidimensional MXML. We also show how our representation formalism can take into account attributes of type ID/IDREF(S) in the representation of the history of the XML document.
2. We demonstrate how we can obtain temporal snapshots that correspond to versions holding at a specific time, by applying a process called *reduction* to the MXML representation of the document's history.
3. We argue that our approach is powerful enough to represent not only the history of an XML document but also the history of the document's schema, expressed in XML Schema, which may also change over time. The temporal snapshots of the schema are also obtained by applying the reduction process.

2 Related Work

a) Representing and querying changes in semistructured data: The problem of representing and querying changes in semistructured data has also been studied in [3], where *Delta OEM* (DOEM in short) was proposed. DOEM is a graph model that extends OEM with annotations containing temporal information. Four basic change operations, namely *creNode*, *updNode*, *addArc*, and *remArc* are considered by the authors in order to modify an OEM graph. Those operations are mapped to four types of annotations, which are tags attached to a node or an edge, containing information that encodes the history of changes for that node or edge. To query DOEM databases, the query language *Chorel* is proposed. Chorel extends *Lorel* [1] with constructs called *annotation expressions*, which are placed in path expressions and are matched against annotations in the DOEM graph.

A special graph for modeling the dynamic aspects of semistructured data, called *semistructured temporal graph* is proposed in [13]. In this graph, every node and edge has a label that includes a part stating the valid interval for the node or edge. Modifications in the graph cause changes in the temporal part of labels of affected nodes and edges.

b) Approaches to represent time in XML: An approach for representing temporal XML documents is proposed in [2], where leaf data nodes can have alternative values, each holding under a time period. However, the model presented in [2], does not explicitly support facets with varying structure for non-leaf nodes. Other approaches to represent valid time in XML include [9,10]. In [6] the XPath data model is extended to support transaction time. The query language

of XPath is extended as well with transaction time axis to enable to access past and future states. Constructs that extract and compare times are also proposed. Finally, in [14] the XPath data model and query language is extended to include valid time, and XPath is extended with an axis to access valid time of nodes.
c) Schemes for multiversion XML documents: The problem of managing (storing, retrieving and querying) multiple versions of XML documents is examined in [4,5]. Most recently [20,19], the same authors proposed an approach of representing XML document versions by adding two extra attributes, namely `vstart` and `vend`, representing the time interval for which this elements version is valid. The authors also demonstrate how XQuery can be used to express queries in their representation scheme. The representation employed in [20,19] presents a lot of similarities with our approach which, however, is more general, and overcomes some limitations of the approach in [20,19].

3 Multidimensional XML

In a *multidimensional XML document* (*MXML document* in short), dimensions may be applied to elements and attributes. A *multidimensional element/attribute* is an element/attribute whose content depends on one or more dimensions.

3.1 Incorporating Dimensions in XML

The notion of *world* is fundamental in MXML. A world represents an environment under which data in a multidimensional document obtain a meaning. A world is determined by assigning values to a set of *dimensions*.

Definition 1. *Let $\mathcal{S}$ be a set of dimension names and for each $d \in \mathcal{S}$, let $\mathcal{D}_d$, with $\mathcal{D}_d \neq \emptyset$, be the domain of d. A* world *W is a set of pairs (d, u), where $d \in \mathcal{S}$ and $u \in \mathcal{D}_d$ such that for every dimension name in $\mathcal{S}$ there is exactly one element in W.*

MXML uses *context specifiers*, which are expressions that specify sets of worlds. Context specifiers qualify the variants (or *facets*) of multidimensional elements and attributes, called *context elements/attributes*, stating the sets of worlds under which each variant may hold. The context specifiers that qualify the facets of the same multidimensional element/attribute are considered to be *mutually exclusive* in the sense that they specify disjoint sets of worlds. An immediate consequence of this requirement is that every multidimensional element/attribute has at most one holding facet under each world. A multidimensional element is denoted by preceding the element's name with the special symbol "`@`", and encloses one or more *context elements*. Context elements have the same form as conventional XML elements. All context elements of a multidimensional element have the same name which is the name of the multidimensional element.

The syntax of XML is extended as follows in order to incorporate dimensions. In particular, a multidimensional element has the form:

```
<@element_name attribute_specification>
  [context_specifier_1]
    <element_name attribute_specification_1>
      element_content_1
    </element_name>
  [/]
  . . .
  [context_specifier_N]
    <element_name attribute_specification_N>
      element_content_N
    </element_name>
  [/]
</@element_name>
```

To declare a multidimensional attribute we use the following syntax:

```
attribute_name = [context_specifier_1] attribute_value_1 [/] . . .
                 [context_specifier_n] attribute_value_n [/]
```

For more details on MXML the reader may refer to [7].

As an example of MXML consider information about a book which exists in two different editions, an English and a Greek one. In Example 1, the element `book` has six subelements. The `isbn` and `publisher` are multidimensional elements and depend on the dimension `edition`. The elements `title` and `authors` remain the same under every possible world. The element `price` is a conventional element containing however a multidimensional attribute (the attribute `currency`) as well as the two multidimensional elements `value` and `discount`. Now two more dimensions appear, namely the dimensions `time` and `customer_type`. Notice that the value of the attribute `currency` depends on the dimensions `edition` and `time` (as to buy the English edition we have to pay in USD, while to buy the Greek edition we should pay in GRD before `2002-01-01` and in EURO after that date due to the change of the currency in EU countries). The element `value` depends on the same dimensions as the attribute `currency`, while the element `discount` depends on the dimensions `edition` and `customer_type`, as students are offered higher discount than libraries.

Example 1. Multidimensional Information about a book encoded in MXML.

```
<book>
  <@isbn>
     [edition = greek] <isbn>0-13-110370-9</isbn> [/]
     [edition = english] <isbn>0-13-110362-8</isbn> [/]
  </@isbn>
  <title>The C programming language</title>
  <authors>
     <author>Brian W. Kernighan</author>
     <author>Dennis M. Ritchie</author>
  </authors>
  <@publisher>
```

```
    [edition = english] <publisher>Prentice Hall</publisher> [/]
    [edition = greek] <publisher>Klidarithmos</publisher> [/]
  </@publisher>
  <@translator>
    [edition = greek] <translator>Thomas Moraitis</translator> [/]
  </@translator>
  <price currency = [edition = greek, time in {start .. 2001-12-31}]
                                                          GRD[/]
                    [edition = greek, time in {2002-01-01 .. now}]EURO[/]
                    [edition = english]USD[/]>
    <@value>
      [edition=greek,time in {start .. 2001-12-31}]
                                        <value>13.000</value>[/]
      [edition=greek,time in {2002-01-01 .. now}]<value>40</value>[/]
      [edition=english]<value>80</value>[/]
    </@value>
    <@discount>
      [edition = greek,customer_type = student]
                                            <discount>20</discount>[/]
      [edition = greek,customer_type = library]
                                            <discount>10</discount>[/]
    </@discount>
  </price>
</book>
```

3.2 Encoding MXML in XML

The introduction of contexts in XML documents can also be achieved by using standard XML syntax instead of the syntax proposed in the previous subsection. In fact a number of papers have appeared [2,9,10] in which time information (that we express here through a time dimension) is encoded either through additional attributes or by using elements with special meaning. Using similar ideas we can encode our MXML documents using the constructs offered by standard XML syntax. For example, our multidimensional elements could be encoded in standard XML by employing a construct of the form:

```
<mxml:group name = "p">
    <mxml:celem>
        <mxml:context> ... </mxml:context>
        <mxml:elem> <p> ... </p> </mxml:elem>
    </mxml:celem>
        ...
    <mxml:celem>
        <mxml:context> ... </mxml:context>
        <mxml:elem> <p> ... </p> </mxml:elem>
    </mxml:celem>
</mxml:group>
```

where a multidimensional element whose name is `p` is denoted by the special MXML element name `mxml:group`, and each element `mxml:celem` corresponds

to a context element together with its context specifier, belonging to that multidimensional element. In a similar way we could encode appropriately in standard XML the content of the element `mxml:context` (i.e. the context specifiers of MXML). We, however, keep using in the rest of the paper, the syntax that we have proposed for MXML, as it offers a clear distinction between context information and the corresponding facets of elements/attributes, resulting in documents that are more readable by humans. Moreover, MXML documents are shorter in size than their corresponding XML representation. We should, however, note that one could use the syntax that we propose for MXML and then transform it automatically into standard XML through a preprocessing phase.

3.3 Obtaining XML Instances from MXML

An important issue concerning the context specifiers of a multidimensional element/attribute is that they must be mutually exclusive, in other words, they must specify disjoint sets of worlds. This property makes it possible, given a specific world, to safely reduce an MXML document to an XML document holding under that world. Informally, the reduction of an MXML document D to an XML document D_w holding under the world w proceeds as follows:

Beginning from the document root to the leaf elements, each multidimensional element E is replaced by its context element E_w, which is the holding facet of E under the world w. If there is no such context element, then E along with its subelements is removed entirely. A multidimensional attribute A is transformed into a conventional attribute A_w whose name is the same as A and whose value is the holding one under w. If no such value exists then the attribute is removed entirely.

Example 2. For the world `w={(edition,greek), (customer_type,student), (time,2002-03-03)}`, the MXML document in Example 1 is reduced to the conventional XML document that follows:

```
<book>
  <isbn>0-13-110370-9</isbn>
  <title>The C programming language</title>
  <authors>
     <author>Brian W. Kernighan</author>
     <author>Dennis M. Ritchie</author>
  </authors>
  <publisher>Klidarithmos</publisher>
  <translator>Thomas Moraitis</translator>
  <price currency = EURO>
     <value>40</value> <discount>20</discount>
  </price>
</book>
```

The above process can be generalized for producing an MXML document for a set S of worlds. In this generalization, called *partial reduction*, the MXML document produced is a subdocument of the original MXML document, which encompasses all XML documents that hold under some world in S.

Example 3. For the worlds expressed by w={(edition,greek),(customer_type, student)}, the MXML document in Example 1 is partially reduced to an MXML document similar to the document of Example 2 except for the element price whose partially reduced version is given below:

```
<price currency = [time in {start .. 2001-12-31}]GRD[/]
                  [time in {2002-01-01 .. now}]EURO[/]>
   <@value>
      [time in {start .. 2001-12-31}]<value>13.000</value>[/]
      [time in {2002-01-01 .. now}]<value>40</value>[/]
   </@value>
   <discount>20</discount>
</price>
```

4 Representing Histories of XML Documents in MXML

In order to represent the changes in an XML document we encode this document as an MXML document in which a dimension named d is used to represent time. More specifically, instead of retaining multiple instances of the XML document, we retain a single MXML representation of all successive versions of the document. We assume that the time domain T of d is linear and discrete. As seen in Example 1, we also assume a) a reserved value *start*, such that $start < t$ for every $t \in T$, representing the beginning of time, and b) a reserved value *now*, such that $t < now$ for every $t \in T$, representing the current time.

The time period during which a context element/attribute is the holding facet of the corresponding element/attribute is denoted by qualifying that context element/attribute with a context specifier of the form [d in $\{t_1..t_2\}$]. In context specifiers the syntactic shorthand $v_1..v_n$ for discrete and totally ordered domains means all values v_i such that $v_1 \leq v_i \leq v_n$.

In the following three subsections we consider the representation of the histories of XML documents without attributes of type IDREF(S). The case of XML documents in which attributes of type IDREF(S) do appear is discussed in subsection 4.4.

4.1 Basic Change Operations on Elements

We consider three primitive change operations, namely *update*, *delete*, and *insert*, on XML documents and demonstrate how their effect on XML elements can be represented in MXML:

a) **Update:** Updating the value of an XML element can be seen as the replacement of the element with another element which has the same name but different content. The way that we represent the effect of update in the MXML representation of the history of the XML document is depicted in Figure 1. The value of the element r in the XML document on the left part of the table is updated at time t1 from v2 to the new value v4. The effect of this operation is shown on the right side of the table. The element r has now become a multidimensional

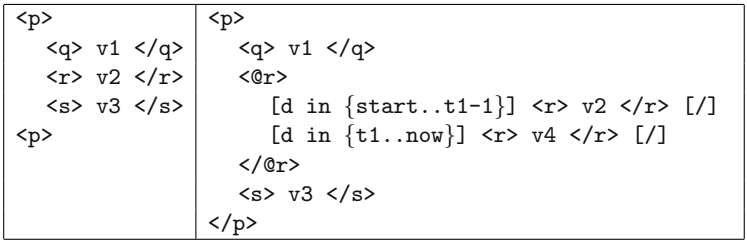


Fig. 1. Applying the operation **update** on the element `r` at time `t1`.

element with two facets. The first facet is valid in all time points of the interval `start..t1-1`, while the other is valid in all time points of the interval `t1..now`.

Note that a subsequent update of the same element `r` at a time point `t2` will be represented in the MXML document as follows: a) by simply adding a new facet in the multidimensional element `@r` holding in the interval `t2..now` and b) by changing the value of the dimension `d` of the most recent facet of `@r` from `t1..now` to `t1..t2-1`. The same process is also used to update complex elements.

b) **Delete:** The deletion of the element `r` at time `t1`, is represented in MXML by a multidimensional element with a single facet holding during the interval `start..t1-1`, as shown in Figure 2. In case, however, in which the element that we want to delete is already multidimensional, the deletion is representing by simply changing the end time point of the most recent facet of the element (the facet for which the end point of the value of `d` is `now`) from `now` to `t1-1`. The operation *delete* applies equally to both simple and complex elements. In the later case, the entire content of the element is considered as deleted without any need to transform the elements it contains to multidimensional elements. This is a consequence of the fact that the contexts of parent elements in MXML are inherited and combined with possible contexts of child elements [15].

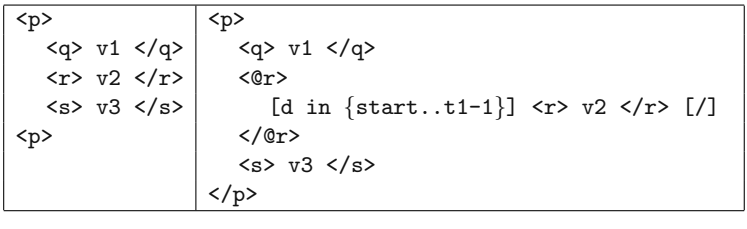


Fig. 2. Applying the operation **delete** on the element `r` at time `t1`.

c) **Insert:** As depicted in Figure 3, the new element `r` inserted at the time point `t1`, is added as a multidimensional element with a single context element holding during the interval `t1..now`, in the MXML representation of the document.

Notice that the inserted element may be a complex one (as shown in Figure 3). In this case, all the descendants of the inserted element inherit the same context.

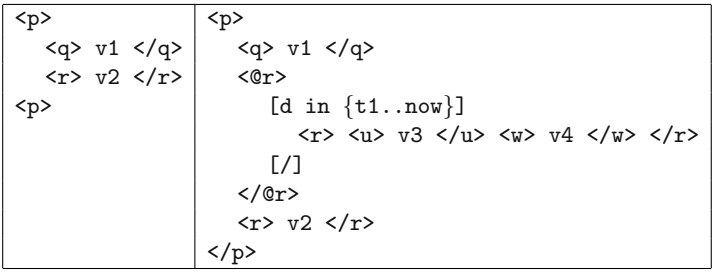


Fig. 3. Insert the element `<r> <u> v3 </u> <w> v4 </w> </r>` after the element `q` at time `t1`.

Notice also there may exist other subelements with the same element name (`r` in our example) without any confusion.

4.2 Basic Change Operations on Attributes

The basic change operations on attributes are similar to those on elements:
a) **Update**. Consider the element:

```
<p a1 = "9"> v1 </p>
```

and suppose that we update the value of the attribute `a1` to the new value `8` at time point `t1`. Then we obtain the following MXML element:

```
<p a1 = [d in {start..t1-1}]"9"[/][d in {t1..now}]"8"[/]> v1 </q>
```

b) **Delete**. Consider the element:

```
<p a1 = "9"> v1 </p>
```

and suppose that we want to delete the attribute `a1` of `p` at time point `t1`. Then we get the following element in MXML:

```
<p a1 = [d in {start..t1-1}]"9"[/]> v1 </p>
```

c) **Insert**. Consider the element:

```
<p a1 = "9"> v1 </p>
```

and suppose that we want to add at time point `t1` a new attribute whose name is `a2` and whose value is `3`. Then we get the following element in MXML:

```
<p a1 = "9" a2 = [d in {t1..now}]"3"[/]> v1 </p>
```

4.3 A More Complex Example

Consider six successive versions (a)-(f) of an XML document shown in Figure 4. Here, each version is obtained from the previous version by applying a basic change operation as follows: (b) is obtained from (a) by inserting the subelement `<q> v2 </q>` at time `t1`. Version (c) is obtained from (b) by inserting, at time `t2`, the attribute `a = "1"` to the first occurrence of the element `q`. Version (d) is obtained from (c) by updating the value of the second occurrence of `q` at time `t3`. Version (e) is obtained by deleting `<s> v4 </s>` at time `t4`. Finally, version (f) is obtained from (e) by updating the value of the element `u` at time `t5`.

One can easily verify that all versions of the XML document of Figure 4 are represented by the MXML document in Figure 5.

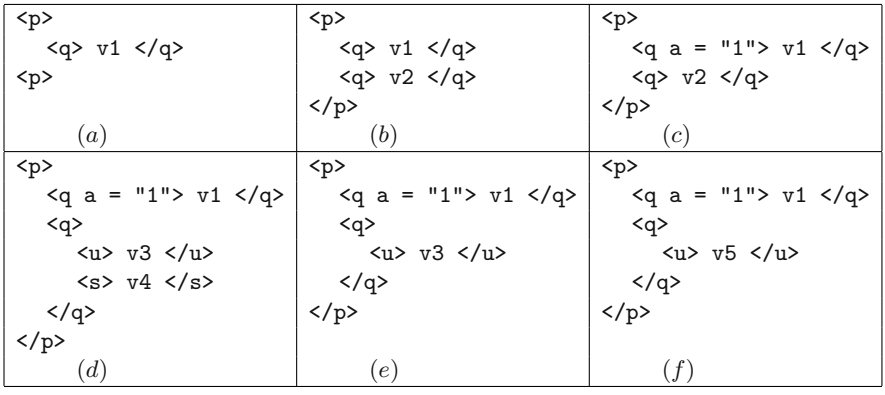


Fig. 4. Six successive versions of an XML document.

```
<p>
  <q a = [d in {t2..now}] "1" [/]> v1 </q>
  <@q>
    [d in {t1..t3-1}] <q> v2 </q> [/]
    [d in {t3..now}]
      <q>
        <@u>
          [d in {start..t5-1}] <u> v3 </u> [/]
          [d in {t5..now}] <u> v5 </u> [/]
        </@u>
        <@s> [d in {start..t4-1}] <s> v4 </s> [/] </@s>
      </q>
    [/]
  </@q>
</p>
```

Fig. 5. MXML document representing the XML document versions of Figure 4.

4.4 Representing the Basic Change Operations in the Presence of Attributes of Type IDREF(S)

So far we have assumed that attributes of type IDREF(S) do not appear in the XML document. However, attributes of this type are often used in XML documents as they offer a means to avoid unnecessary repetition of information. In this subsection, we adapt our method of representing changes of XML documents so as to take into account the presence of attributes of type IDREF(S). This adaptation is based on the following rules:

– When a conventional element E, which has an attribute A of type ID, is transformed into a multidimensional element (as a result of a basic change operation), then A becomes an attribute of the multidimensional element.

- The basic change operations are not allowed to modify the values of attributes of type ID (they are, however, allowed to modify attributes of type IDREF(S)).
- Context elements do not have attributes of type ID. All the facets of a multidimensional element share the same ID attribute of that multidimensional element (if there is such an attribute).

The manipulation of an ID attribute during an update operation is shown in Figure 6. The element `r`, whose content is updated at time `t1`, has an attribute named `id` of type ID for which there is a reference from the element `q` (through its attribute `ref` of type IDREF). In order to represent the effect of this update, a multidimensional element `@r` is created and the two `r` elements (the old one and the new one) become facets of that multidimensional element. Notice however that the attribute `id` of the old element `r` has been removed from the facet `r` and becomes now an attribute of the multidimensional element `@r`. When we apply the reduction process to get the temporal snapshot of the document holding at a time point `t`, this attribute is copied to the element facet holding at `t`.

```
<p>                                  <p>
  <q ref="id1"> ... </q>               <q ref="id1"> ... </q>
    ...                                  ...
    <r id="id1"> v1 </r>                 <@r id="id1">
    ...                                    [d in {start..t1-1}] <r> v1 </r> [/]
</p>                                       [d in {t1..now}] <r> v2 </r> [/]
                                       </@r>
                                       ...
                                   </p>
```

Fig. 6. Updating an element which has an attribute of type ID.

The deletion of an element which is referred from another element through an attribute of type IDREF, is problematic since it results in a reference not showing to an existing element at every time point. However, this is a problem related to the deletion operation itself and not to its representation in MXML.

5 Querying the History of an XML Document

Supporting effective query answering on the history of an XML document is vital in the evaluation of a versioning model. In [4], a taxonomy of the most important queries on versions is given. This taxonomy consists of four categories, namely: 1) *Temporal selection*, which refer to the retrieval of either a particular version or a sequence of successive versions of the document. 2) *Document evolution & historical queries*, which focus on querying the changes between successive versions. 3)*Structural projection*, in which the user requests certain parts (elements) of the document. 4) *Content-based selection*, which refers in retrieving all versions (of the whole document or part of it) that fulfill some conditions.

Temporal selection queries are answered in our approach through the processes of reduction. Reduction to XML is used to retrieve a particular version for a given time instance. One can easily verify that each version of the XML document in Figure 4 can be obtained by applying the reduction to XML on the MXML in Figure 5. Partial reduction is used to retrieve an MXML subdocument encompassing a set of XML versions that correspond to a set of time instances.

We are currently working in incorporating context in XPath and XQuery, in order to come up with a context-aware query language for MXML. Previous work on a context-aware query language for MOEM led to *Multidimensional Query Language* (*MQL*) [18], which extends Lorel [1] and incorporates context specifiers. In [17], we demonstrate how MQL can be used to formulate queries on the history of OEM databases that cover all the above categories. Our current work follows a similar direction, and aims to extend XPath with context qualifiers that express conditions on the contexts of the corresponding path elements. Moreover, we plan to extend XQuery with additional clauses that use *context variables* to pose complex conditions on contexts.

6 Representing the History of the Schema of an XML Document

Changes in an XML document often require corresponding changes in the document's schema, as the preservation of the same schema might be too restrictive concerning the set of changes that we want to permit. The representation of the history of the document's schema in case that we allow its evolution is as important as the representation of the document's history. In this section we show that our model is powerful enough to represent also the history of the document's schema encoded as XML Schema.

Consider for example the XML document in the left side of Figure 2. The schema for this document may be encoded in XML Schema as follows:

```
<xs:element name="p">
  <xs:complexType>
    <xs:sequence>
        <xs:element name="q" type="xs:string"/>
        <xs:element name="r" type="xs:string"/>
        <xs:element name="s" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

It is easy to see that by deleting the element `<r>v2</r>` (as described in Figure 2), we get an XML document which is not valid with respect to the above schema (in which the existence of the element `r` between the elements `q` and `s` is mandatory). For this reason, it is necessary to modify the document's schema if we want the XML document resulting by applying the deletion to become valid. The new schema might declare that the content of the element `p`

is now an element `q` followed by an element `s`. This change can be represented in our formalism by turning the element `sequence` of the above XML Schema into a multidimensional element with two facets:

```
<xs:element name="p">
  <xs:complexType>
    <@xs:sequence>
      [d in {start..t1-1}]
         <xs:sequence>
           <xs:element name="q" type="xs:string"/>
           <xs:element name="r" type="xs:string"/>
           <xs:element name="s" type="xs:string"/>
         </xs:sequence>
      [/]
      [d in {t1..now}]
         <xs:sequence>
           <xs:element name="q" type="xs:string"/>
           <xs:element name="s" type="xs:string"/>
         </xs:sequence>
      [/]
    </@xs:sequence>
  </xs:complexType>
</xs:element>
```

It is easy to see that the resulting multidimensional schema records the history of the document's schema. Moreover, the snapshots of this schema that refer to specific versions of the XML document (at specific time points), can again be obtained by applying the reduction process.

Another schema that could also validate the document might retain the element `r` as optional. This could also be encoded in our multidimensional formalism by simply replacing the line:

```
<xs:element name="r" type="xs:string"/>
```

in the initial XML schema description by:

```
<xs:element name="r" type="xs:string"
            minOccurs= [d in {t1..now}]"0"[/]/>
```

where a multidimensional version of the attribute `minOccurs` has been added, through which it is declared that the attribute `minOccurs` with value `0` is present in the elements declaration during the time interval `t1..now`.

7 Conclusions and Future Work

In this paper we proposed the use of MXML as a model for representing the history of XML documents as well as the evolution of their schema. We demonstrated how the effect of the basic change operations on XML documents can be represented in MXML and showed that we can easily obtain temporal snapshots of the XML document from its MXML representation.

The method presented in this paper is based on the ideas proposed by the same authors in [16,17]. Other works closely related to ours include [20,19], where the authors represent the interval through which an element holds by adding two extra attributes, namely `vstart` and `vend` whose values are the start and the end points of this interval respectively. However, the approach in [20,19]: a) does not address the problem of IDREF(S), and b) versions of an element are not explicitly associated as being facets of the same (multidimensional) element. As shown in [17], grouping facets together allows the formulation of cross-world queries, that relate facets that hold under different worlds.

We believe that our approach is more general than other approaches as we allow the treatment of multiple dimensions in a uniform manner. Consequently, our approach may be proved powerful enough to represent multiple versioning not only with respect to time but also to other context parameters such as language, degree of detail, geographic region etc. For instance, incorporating dimensions in semistructured data and XML gives new possibilities for designing Web based applications that deal with context-dependent data. In [11,12] a web-publishing platform is presented that supports context-dependent delivery, through a model for context which is based on application-specific sets of *characteristics* which are in fact the same as the dimensions of our model. Moreover, a method of constructing multidimensional web pages is presented in [8].

The investigation of efficient methods for storing and retrieving MXML documents and the incorporation of contexts in XPath and XQuery is in our immediate plans for future work.

References

1. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.
2. T. Amagasa, M. Yoshikawa, and S. Uemura. A Data Model for Temporal XML Documents. In M. T. Ibrahim, J. Kung, and N. Revell, editors, *Database and Expert Systems Applications, 11th International Conference (DEXA 2000), London, UK, Sept. 4-8, Proceedings*, Lecture Notes in Computer Science (LNCS) 1873, pages 334–344. Springer-Verlag, Sept. 2000.
3. S. S. Chawathe, S. Abiteboul, and J. Widom. Managing Historical Semistructured Data. *Theory and Practice of Object Systems*, 24(4):1–20, 1999.
4. S.-Y. Chien, V. Tsotras, and C. Zaniolo. Efficient Schemes for Managing Multiversion XML Documents. *The VLDB Journal*, 11(4):332–353, 2002.
5. S.-Y. Chien, V. J. Tsotras, C. Zaniolo, and D. Zhang. Efficient Complex Query Support for Multiversion XML Documents. In *Advances in Database Technology - EDBT 2002, Proceedings of the 8th Conference on Extending Database Technology*, Lecture Notes in Computer Science (LNCS) Vol 2287, pages 161–178. Springer-Verlag, 2002.
6. Curtis E. Dyreson. Observing Transaction-time Semantics with TTXPath. In *Proceedings of the 2nd International Conference on Web Information Systems Engineering (WISE 2001), Kyoto, Japan, Dec. 2001*, pages 193–202. IEEE Computer Press, 2001.

7. M. Gergatsoulis, Y. Stavrakas, and D. Karteris. Incorporating Dimensions to XML and DTD. In H. C. Mayr, J. Lanzanski, G. Quirchmayr, and P. Vogel, editors, Database and Expert Systems Applications (DEXA' 01), Munich, Germany, September 2001, Proceedings, Lecture Notes in Computer Science (LNCS), Vol. 2113, pages 646–656. Springer-Verlag, 2001.
8. M. Gergatsoulis, Y. Stavrakas, D. Karteris, A. Mouzaki, and D. Sterpis. A Web-based System for Handling Multidimensional Information through MXML. In A. Kaplinskas and J. Eder, editors, Advances in Databases and Information Systems (ADBIS' 01), Proceedings, Lecture Notes in Computer Science (LNCS), Vol. 2151, pages 352–365. Springer-Verlag, 2001.
9. F. Grandi and F. Mandreoli. The Valid Web: an XML/XSL Infrastructure for Temporal Management of Web Documents. In T. M. Yakhno, editor, *Advances in Information Systems. First International Conference (ADVIS'02), Izmir, Turkey, 25-27 October*, pages 294–303, 2000.
10. Fabio Grandi. XML Representation and Management of Temporal Information for the Web-Based Cultural Heritage Applications. *Data Science Journal*, 1(1):68–83, 2002.
11. M. C. Norrie and A. Palinginis. Empowering Databases for Context-Dependent Information Delivery. In *Proc. of UMICS' 03*, 2003. (to appear).
12. M. C. Norrie and A. Palinginis. From State to Structure: An XML Web Publishing Framework. In *Proc. of the 15th Conference on Advanced Information Systems Engineering (CAiSE' 03)*, June 2003. (to appear).
13. B. Oliboni, E. Quintarelli, and L. Tanca. Temporal Aspects of Semistructured Data. In *Proc. of the 8th International Symposium on Temporal Representation and Reasoning (TIME-01)*, pages 119–127, 2001.
14. S. Shang and C. E. Dyreson. Adding Valid Time to XPath. In *Database and Network Information Systems, Proceedings of DNIS 2002*, Lecture Notes in Computer Science (LNCS), Vol. 2544, pages 29–42. Springer-Verlag, 2002.
15. Y. Stavrakas and M. Gergatsoulis. Multidimensional Semistructured Data: Representing Context-Dependent Information on the Web. In A. B. Pidduck, J. Mylopoulos, C. Woo, and T. Oszu, editors, *Advanced Information Systems Engineering, 14th International Conference (CAiSE'02), Toronto, Ontario, Canada, May 2002. Proceedings.*, Lecture Notes in Computer Science (LNCS), Vol. 2348, pages 183–199. Springer-Verlag, 2002.
16. Y. Stavrakas, M. Gergatsoulis, C. Doulkeridis, and V. Zafeiris. Accomodating Changes in Semistructured Databases Using Multidimensional OEM. In Y. Manolopoulos and P. Navat, editors, Advances in Databases and Information Systems (ADBIS' 02), Proceedings, Lecture Notes in Computer Science (LNCS), Vol. 2435, pages 360–373. Springer-Verlag, 2002.
17. Y. Stavrakas, M. Gergatsoulis, C. Doulkeridis, and V. Zafeiris. Representing and Querying Histories of Semistructured Databases Using Multidimensional OEM. *Information Systems*, ??(??), 2003. (to appear).
18. Y. Stavrakas, K. Pristouris, A. Efantis, and T. Sellis. Implementing a Query Language for Context-dependent Semistructured Data. Submitted for publication, 2003.
19. F. Wang and C. Zaniolo. Representing and Quering the Evolution of Databases and their Schemas in XML. In *Proc. of SEKE'03*, July 2003. (to appear).
20. F. Wang and C. Zaniolo. Temporal Queries in XML Document Archives and Web Warehouses. In *Proc. of TIME-ICTL' 03*, July 2003. (to appear).

Updating XQuery Views Published over Relational Data: A Round-Trip Case Study

Ling Wang, Mukesh Mulchandani, and Elke A. Rundensteiner

Department of Computer Science
Worcester Polytechnic Institute Worcester, MA 01609
{lingw, mukesh, rundenst}@cs.wpi.edu

Abstract. Managing XML data using relational database systems, including query processing over virtual XML views that wrap relational sources, has been heavily studied in the last few years. Updating such virtual XML views, however, is not well studied, although it is essential for building a viable full-featured XML data management systems. XML view update is a challenging problem because of having to address the mismatch between the two rather different data models and distinct query paradigms. In this paper, we tackle the XQuery view update problem, in particular, we focus on the *round-trip XML view update subproblem*. This case, characterized by a pair of loading and extraction mappings that load XML data into the relational store and extract appropriate XML views, is very common in practice, as many applications utilize a relational engine for XML document storage. We discuss and prove the updatability of such views. We also present a decomposition-based update translation strategy for solving this problem. As evidence of feasibility, we have implemented the proposed strategies within the *Rainbow* XQuery system. Experimental studies are also given to assess the performance characteristics of our update system in different scenarios.

1 Introduction

Motivation. XML [5] has become the standard for interchanging data between web applications because of its modeling flexibility. The database community has focused on combining the strengths of the XML data model with the maturity of relational database technology to provide both reliable persistent storage as well as flexible query processing and publishing. Examples of such XML management systems include EXPERANTO [6], SilkRoute [10] and Rainbow [22], which typically offer support for XML view creation over relational data and for querying against such XML wrapper views to bridge relational databases with XML applications. However, in order for such systems to become viable XML data management systems, they must also support updates, not just queries of (virtual) XML views.

This view-update problem is a long-standing issue that has been studied in the context of the relational data model. Much work has been done on defining

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 223–237, 2003.

what a correct translation entails [9] and how to eliminate ambiguity in translation [7,2]. However, update operations have not been given too much attention yet in the XML context. [18] studies the performance of translated updates executed on the relational store, assuming that the view update is indeed translatable. Updating of virtual XQuery views comes with new challenges beyond those of relational views since we have to address the mismatch between the two data models (the flexible hierarchical XML view model and the flat relational base model) and between the two query languages (XQuery versus SQL queries).

In this paper, we characterize a common sub-case of the *XQuery view update* problem which we call the **Round-trip XML View Update Problem (RXU)**. This is an important case since many XML applications use relational technology to store, query and update XML documents. Such systems require typically a two-way mapping to first load and then to extract XML out of the relational database. Hence, we refer to this as the "round-trip" case. In this paper, we show that the view update operations in this case are always translatable.

We present our framework named *Rainfall* for update translation of this round-trip problem. Due to there not yet being any standard update language, we have extended the XQuery grammar to support XML updates similar to [18]. We have implemented the proposed strategies for update decomposition, translation and propagation within the *Rainbow* XML data management system [22]. Experiments are also presented to compare update translation with the alternative, which would be the re-loading of the updated XML into the relational data store. We also assess various performance characteristics of our update solution.

Contributions. In summary, we make the following contributions in this paper:

- We characterize a subproblem of the general XML view update called *round-trip XML view update problem (RXU)*, which is a common case in practice.
- We formally describe the view updatability for the RXU case and prove its correctness.
- We provide a decomposition-based update translation solution called *Rainfall* to translate XQuery updates on XML virtual views into a set of SQL-level updates.
- We implement our update solution within the *Rainbow* XML data management system to support the view update extension.
- We present a performance study conducted to assess our update translation strategy.

Outline. This paper is structured as follows. We briefly introduce the XML data model and XQuery update extension in Section 2. Section 3 characterizes the *Round-trip XML view update problem*, and discusses the view updatability in this case. We describe our decomposition-based update strategy and system implementation in Section 5 and evaluate these techniques in Section 6. Section 7 reviews related work while Section 8 concludes our work.

2 Background

XQuery Views of Relational Data. XML (Extensible Markup Language) [5] is used both for defining document markup and for data exchange. XML Schema [19] is a standardized syntax used to represent the structure of XML documents. Figures 1 and 2 respectively show our running example of an XML schema and document representing a book list from an online book store application.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema>
  <xs:elementname="bib">
    <xs:complexType>
      <xs:sequence>
        <xs:elementname="book" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:elementname="bookid" type="xs:string" nillable="false"/>
              <xs:elementname="title" type="xs:string" nillable="false"/>
              <xs:elementname="author">
                <xs:complexType>
                  <xs:sequence>
                    <xs:elementname="aname" type="xs:string" maxOccurs="unbounded"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:elementname="publisher">
                <xs:complexType>
                  <xs:sequence>
                    <xs:elementname="pname" type="xs:string"/>
                    <xs:elementname="location" type="xs:string"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:elementname="review" type="xs:string" nillable="true"/>
            </xs:sequence>
            <xs:attributename="year" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Fig. 1. Example XML schema

```
<bib>
  <book year="1994">
    <bookid>98001</bookid>
    <title>TCP/IP Illustrated</title>
    <author>
      <aname>W. Stevens</aname>
    </author>
    <publisher>
      <pname>Addison-Wesley</pname>
      <location>San Francisco</location>
    </publisher>
    <review>
      One of the best books on TCP/IP.
    </review>
  </book>
  <book year="1992">
    <bookid>98002</bookid>
    <title>Programming in Unix</title>
    <author>
      <aname>Bram Stoker</aname>
    </author>
    <publisher>
      <pname>Addison-Wesley</pname>
      <location>Boston</location>
    </publisher>
    <review>
      A clear and detailed discussion of UNIX programming.
    </review>
  </book>
  ... ...
</bib>
```

Fig. 2. Example XML data

Many XML applications use a relational data store by applying loading strategy such as [17,8]. Figures 3 and 4 show an example relational database generated from the XML schema and data of our running example using a shared inlining loading strategy [17]. The basic XML view, called *Default XML View*, is a one-to-one mapping to bridge the gap between the two heterogeneous data models, that is the XML (nested) data model and relational (flat) data model. Each table in the relational database is represented as one XML element and each of its tuples as subelements of this table element. Figure 5 depicts the default XML view of the database (Figure 3).

A default XML view explicitly exposes the tables and their structure to the end users. However, end users often want to deal with an application specific view of the data. For this reason, XML data management systems provide a facility to define user-specific view capabilities on top of this default XML view, called a *virtual view*. Such a *virtual view* can be specified by an XQuery expression, then called a *view query*. Several recent systems such as XPERANTO [6], SilkRoute [10] and Rainbow [22] follow this approach of XML-to-Relational mapping via

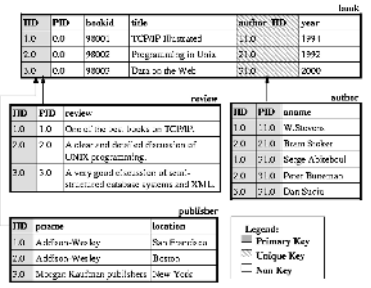


Fig. 3. Relations in database

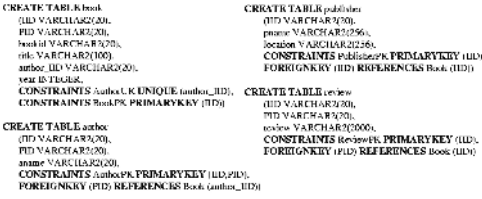


Fig. 4. Database schema of Figure 3

defining XML views over relational data. An XML query language, such as XQuery proposed by World Wide Web Consortium (W3C), can be used both to define such views and also to query them. Figure 6 shows the view query defining a virtual view identical to the originally loaded XML document in Figure 2.

Fig. 5. Default XML view of database shown in Figure 3

Fig. 6. Virtual XQuery view over default XML view shown in Figure 5 producing the XML data in Figure 2

XQuery Updates. Although W3C is adding update capabilities to the XQuery standard [20], currently no update language for XML has yet been standardized. For our work, we thus adopt an extension of the XQuery language syntax with update operations that follows [18]. XQuery is extended with a *FLWU* expression composed of *FOR...LET...WHERE...UPDATE* clauses (Figure 7). Figure

8 shows an example *Insert* update, which inserts a new *book* element into the (virtual) view defined in Figure 6.

```
FOR $binding1 IN Xpath-expr,...
LET $binding := Xpath-expr, ...
WHERE predicate1, ...
updateOp, ...

Where updateOp is defined in EBNF as :

UPDATE $binding {subOp {, subOp}* } and subOp is:

DELETE $child |
RENAME $child To new_name |
INSERT ( $bind [BEFORE | AFTER $child]
        | new_attribute(name, value)
        | new_ref(name, value)
        | content [BEFORE | AFTER $child]  ) |
REPLACE $child WITH  ( new_attribute(name, value)
                     | new_ref(name, value)
                     | content ) |
FOR $sub_binding IN Xpath-subexpr, ...
WHERE predicate1, ... updateOp.
```

Fig. 7. Update language as extension of XQuery

```
FOR $root in document("view.xml")
UPDATE $root {
  INSERT
  <book year="1995">
    <bookid>98004</bookid>,
    <title>"Languages and Machines"</title>,
    <author>
      <aname>"Thomas A. Sudkamp"</aname>
    </author>,
    <publisher>
      <pname>"Addison Wesley Longman, Inc."</pname>,
      <location>"Boston"</location>
    </publisher>,
    <review>
      "An Introduction to the theory of Computer Science"
    </review>
  </book>
}
```

Fig. 8. Insert update on XQuery view shown in Figure 6

3 Round-Trip XML View Update Problem

3.1 Definition of the Round-Trip XML View Update Problem

The general *XQuery view update problem* can be characterized as follows. Given a relational database and an XQuery view definition over it, can the system decide if an update against the view can be translated into corresponding updates against the underlying relational database without violating any consistency. And, if it translatable, then how would this translation be done.

Given the general problem definition as above, we now focus on one important case which we name the **round-trip XML view update problem**. Given an XML schema and a valid XML document, by using a suitable loading algorithm, such as inlining [17], edge or universal [8], accompanied with a constraint-preserving mapping such as described in [14], assume we built a relational database. We call it a **structured database**. Further we specify an XML view query on this *structured database* using an XQuery expression, which constructs an XML view with the content identical to the XML document that had just been supplied as input to the loading mapping. We call this special-purpose view query an *extraction query*. We then can extract a view schema by analyzing the extraction query semantics and the relational database schema. Thus the view has the same content and schema as the original XML document which had just been captured by the relational database. We call this special view a *twin-view*. The problem of updating the database through this *twin-view* is referred to as the *round-trip XML view update problem* (Figure 9).

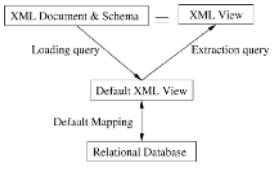


Fig. 9. Round-Trip Update Problem

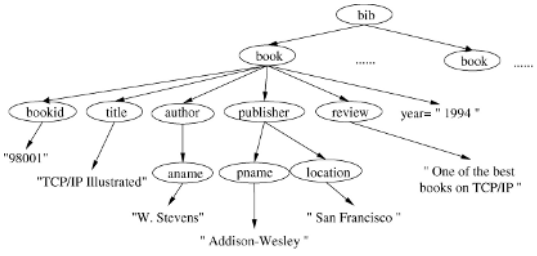


Fig. 10. Tree representation for XML document shown in Figure 2

3.2 Characterization of the XML Loading

As defined above, RXU is closely related with the loading procedure of the XML document and schema into the relational database. To address the influence of the loading strategy on the view updatability, we hence now study the loading strategy characteristics for the RXU case. Many XML loading strategies have been presented in the literature [14,17,8]. Not only the XML document, but also the XML schema is typically captured in this procedure, which are called data and constraint information respectively.

Data Loading Completeness. The XML (nested) data structure is distinct from the relational (flat) data model. Thus the loading procedure must translate from one model (structure) to the other. The completeness of data loading is important in RXU since the *twin-view* requires exactly the same content as the original document , independent on whatever we may do to the structure.

Definition 1. *Given an XML document D_x, a loading L generates a resulting relational database instance D_r, denoted by $D_x \xrightarrow{L} D_r$. L is a* **lossless data loading** *iff $\exists L'$ such that $D_r \xrightarrow{L'} D_x$ holds true.*

Figure 10 is a tree structured representation of the XML document in Figure 2, while Figure 3 is a structured database resulting from applying the inlining loading to that XML document. The extraction query in Figure 6 will generate the *twin-view* from the *structured database* of Figure 3. Thus this loading is a lossless data loading by Definition 1.

A lossless data loading guarantees to capture all leaves in the XML tree-structured representation (Figure 10). Leaves represent actual data instead of document structure. Hence we will be able to reconstruct the XML document. While a lossy data loading may not have loaded some of leaves, hence is not sufficient for reconstruction. Most loading strategies presented in the literature, such as Inlining [17] and Edge [8], are all lossless data loadings.

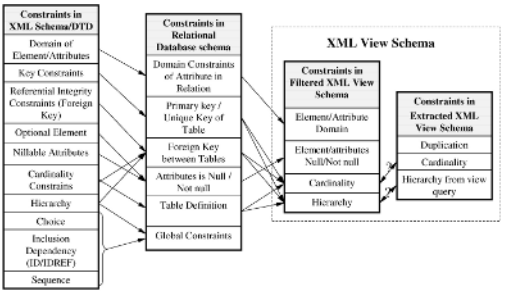


Fig. 11. Comparison of constraints of XML schema, relational database schema and XML view schema

```
<bib>
FOR $book in document("default.xml")/book/row
RETURN{
  <book year=$book/year/text()>
    <bookid>$book/bookid/text()</bookid>,
    <title>$book/title/text()</title>,
      <author>
      FOR $aname in document("default.xml")/author/row
      WHERE $book/author_IID = $aname/PID
      RETURN{
            <aname>$aname/aname/text()</aname>}
    </author>
  </book>
}
</bib>
```

Fig. 12. XQuery example

Constraint Loading Completeness. Given a relational database schema S_r and a view query Q, we define the constraints implied by the XML view as *XML View Schema*, which can be extracted by a mapping named *constraint extraction mapping* denoted by $\hat{e}$. As shown in Figure 11, an *XML View Schema* is a combination of a *Filtered XML View Schema (FSchema)* FS_v and an *Extracted XML View Schema (ESchema)* ES_v, thus denoted as $S_v(ES_v, FS_v)$. An *FSchema*, modeling the constraints extracted from the schema of any relation referenced by Q, is inferred by analyzing the relational database schema and filtering this schema using the view definition, hence denoted as $FS_v = \hat{e}(S_r)$. These constraints may include domain constraints, cardinality constraints, null constraints and hierarchical constraints. An *ESchema* consisting of constraints that can be inferred from the *view query* semantics is extracted by analyzing the view query expression, hence represented as $ES_v = \hat{e}(Q)$. They include cardinality constraints, hierarchical constraints and duplication constraints. The constraints implied in the view definition may not be consistent with the constraints imposed by the relational database schema. Hence the ESchema may conflict with the FSchema. This mismatch may cause some problem in the later update translation step. However, in RXU, we assume that the *view schema* is exactly the same as the original XML schema. Hence this mismatch problem will not arise. This assumption relies on the idea of *constraint loading completeness* and *extraction query.*

Definition 2. *Given an XML schema S_x, a loading L generates a structured database with schema S_r, denoted by $S_x \xrightarrow{L} S_r$. L is a* **lossless constraint loading** *iff $\exists Q$ be an extraction query generating an XML view with schema $S_v = (\hat{e}(S_r), \hat{e}(Q))$, such that $S_v = S_x$ holds true.*

An XML to relational database loading is a **lossless loading** iff it is both a lossless data loading as defined by Definition 1 and a lossless constraint loading as defined by Definition 2. Obviously the loading in RXU must be a lossless loading. Most loadings proposed in the literature are all lossless data loading strategies,

however few of them are also lossless constraint loading strategies. For example, Edge [8] is a lossless data loading, while it is not a lossless constraint loading. In order for such loading strategies to be usable for RXU, it must accompany a constraint preserving loading such as proposed in [14].

4 On the View Updatability in RXU

Basic Concepts. We first review the relational data model and view definition framework. The notation used is shown in Table 1. A *relational database* is a combination of a set of relations and a set of integrity constraints. A *database state*, denoted by s, is an assignment of data values to relations such that the integrity constraints are satisfied. The database *status*, denoted by S, is the set of all possible database states. An *data update* of a relational database with status S is a mapping from S into S, denoted as $\hat{u} : S \rightarrow S$. A *view* V of a given relational database with status S is defined by a set of relations and a mapping f that associates with each database state $s \in S$ a view state $f(s)$. In our case the mapping f is the *view definition mapping* expressed in an XQuery Q. The set $f(S) = \{f(s)|s \in S\}$ is the *view status*. The set of view definition mappings on S is denoted as $M(S)$. A *valid view update* u on view state is an update that satisfies all the constraints of view schema.

Table 1. Notation table

S	database status	s	current database state
f	view definition mapping	$M(S)$	a set of view definition mappings on S
$f(S)$	view status	$f(s)$	view state associated with database state s
U^r	set of all database updates	U^v	set of all valid view updates

Translation Criteria. We now discuss what is the criteria of translating an XML view update. By the **Correctness Criteria,** only the desired update is performed on the view, that is, it is consistent and has no view side effects. Given $u \in U^v$, $\exists u' \in U^r$ such that (a) $u(f(s)) = f(u'(s))$, (b) $\forall s \in S$, $uf(s) = f(s) \Rightarrow u'(s) = s$. In order to permit all possible changes but only in their simplest forms, **the Simplicity Criteria** requires that all candidate update translations satisfy the following rules [11]: (a) *One step changes.* Each database tuple is affected by at most one step of the translation for any single view update request. (b) *Minimal changes.* There is no valid translation that implements the request by performing only a proper subset of database requests. (c) *Replacement cannot be simplified.* That is, we always pick the simplest replace operation, e.g, a database replacement that does not involve changing the key is simpler than one where the key changes. (d) *No insert-delete pairs.* We do not allow candidate translations to include both deletions and insertions on the same tuple of the same relation. Instead they must be converted into replacements, which we consider simpler.

Definition 3. *Given an update $u \in U^v$ on view state $f(s) \in f(S)$, if $\exists u' \in U^r$ that satisfies the correctness criteria defined above, u is called* **translatable** *for $f(s)$ (also can be called f-translatable). $f(s)$ is called* **updatable** *by u. u' is named a* **correct translation** *for u. Further, if u' also satisfies the simplicity criteria defined above, we say u' is an* **optimized translation**.

Updatability of RXU views. We now study the updatability of views in the RXU space. The view complement theory in [2] proposes that if a complementary view, which includes information not "visible" in the view, is chosen and is held constant, then there is at most one translation of any given view update. Although as described in [13], translators based on complements do not necessarily translate all translatable updates. It still provides us with a conservative computation for the set of translatable updates. This fits our RXU case well, since here the complement view always corresponds to a constant. We hence use the view complement theory to prove that any update on a *twin-view* is always translatable.

The complementary theory proposed in [2] can be explained as below.

Definition 4. *Let $f, g \in M(S)$. We say that f is greater than g or that f determines g, denoted by $f \geq g$, iff $\forall s \in S, \forall s' \in S, f(s) = f(s') \Rightarrow g(s) = g(s')$.*

Definition 5. *Let $f, g \in M(S)$. We say that f and g are equivalent, denoted by $f \equiv g$, iff $f \geq g$ and $g \geq f$.*

Definition 6. *Let $f, g \in M(S)$. The product of f and g, denoted by $f \times g$, is defined by $f \times g(s) = (f(s), g(s)), \forall s \in S$.*

Definition 7. *Let $f \in M(S)$. A view $g \in M(S)$ is called a* **complement** *of f, iff $f \times g \equiv 1$. Further, g is the* **minimal complement** *of f iff (i) g is a complement of f, and (ii) if h is a complement of f and $h \leq g$, then $h \equiv g$.*

Definition 4 can be interpreted as $f \geq g$ iff whenever we know the view state $f(s)$, then we also can compute the view state $g(s)$. Definition 6 implies that the product $f \times g$ "adds" to f the information in g. We denote the identity mapping on S as $\mathbf{1}$ and a constant mapping on S as $\mathbf{0}$. In our case, the mapping query used to define the default XML view is mapping $\mathbf{1}$. And a XQuery such as $< bib >< /bib >$ is a constant mapping. According to Definition 7, if $f \times g \equiv 1$, then f, g contain sufficient information for computing the database, and the complementary view g contains the information not "visible" within the view f. For example, assuming the query in Figure 6 define a mapping f, the query in Figure 12 defines a mapping g, then $f \geq g$ and $g \times f \equiv 1$. f is complement of g.

Lemma 1. *Given a complement g of f and a view update $u \in U^v$, u is g-translatable iff $\forall s \in S, \exists s' \in S$ so that $f(s') = uf(s)$ and $g(s') = g(s)$.*

This lemma is the complement theory, which implies that given a complement g of the view f and a view update $u \in U^v$, the translation of u that leaves g invariant is the desired translation satisfying our correctness criteria defined above. This is first presented in [9] as the "absence of side effects" feature. For the proof of this lemma, please refer to [2]. We now use this theory to prove that any update on the view of RXU is always translatable, as described below.

Observation 1 *Within the RXU case, given an XQuery view definition f defined over the relational state s, $\forall u \in U^v$, u is translatable by Definition 3.*

Proof. (i) Since the mapping query defining the default XML view is $\mathbf{1}$, according to Definition 5, in RXU, $\forall f$, $f \equiv \mathbf{1}$. This is because we can always compute the default XML view from the view state $f(s)$ by using the loading mapping, that is $f \geq \mathbf{1}$, while $\mathbf{1} \geq f$ always holds true. (ii) Since $\mathbf{0}$ is the complement of $\mathbf{1}$, while $f \equiv \mathbf{1}$, then $\mathbf{0}$ is the complement view of f. (iii) $\forall u \in U^v$, let $f(s') = uf(s)$, then $\mathbf{0}(s') = \mathbf{0}(s)$. Thus, by Lemma 1, u is always translatable.

5 Rainfall — Our Approach for XQuery View Update

Updating through an XML view can be broken into three separate but consecutive processes:

- *Information Preparation.* This process analyzes the XQuery view definition to provide us with a prior knowledge about the relationship of the view with the relational database, that is, extracting the view schema. It also performs pre-checking of updates issued on the view to reject invalid updates using the view schema.
- *Update Decomposition.* This is the key process of the XML update translation to bridge the XQuery model and the relational query model. The given XML update request is decomposed into a set of valid database operations, with each being applied to a single relation.
- *Global Integrity Maintenance.* Because of the structural model of the relational database with its integrity constraints, the database operations resulting from the *decomposition* process may need to be propagated globally throughout the base relations to assure the consistency of the relational database.

We hence call our strategy a *decomposition-based update strategy.* Our update strategy will generate an optimized update translation which follows the simplicity criteria defined in Section 4. For details on our update translation strategy, please refer to [21].

5.1 System Framework

Figure 13 depicts the architecture of our *Rainfall* update system, which is an extension of the base XML query engine *Rainbow* [22]. *Rainbow* is an XML data

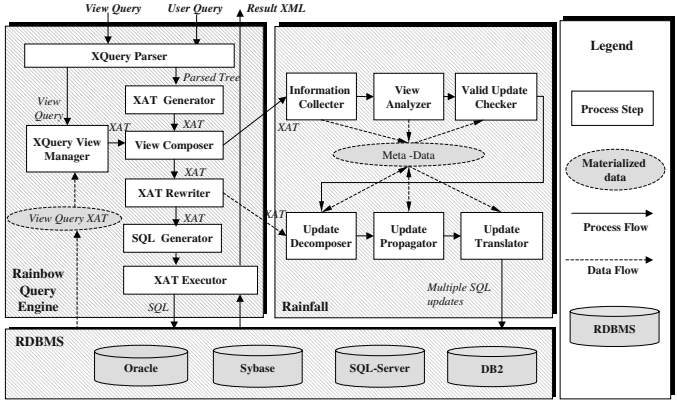


Fig. 13. Architecture of Rainbow query engine with update extension

management system designed to support XQuery processing and optimization based on an XML algebra with the underlying data store being relational.

An XML view or update query is first passed to the *XQuery parser* for syntax checking. We have extended the *Kweelt XQuery parser* [16] to support the update grammar (Figure 7). The *XAT generator* and *View composer* of the Rainbow query engine generates an algebraic representation of XQuery called XML algebra tree (XAT). For a description of XQuery processing in *Rainbow* refer to [23].

The *Rainfall* update system takes the above XAT from the *Rainbow query engine*. An *Infomation Collector* first identifies all the relations related to the view and their relationships. It then collects their schemas and integrity constraints. This information is stored in a metadata structure, which will serve as *view schema* as defined in Section 3. After that, a *View Analyzer* studies the key features of the XQuery view definition to prepare a translation policy. The *Valid Update Checker* examines if the user update query is valid or not. Invalid updates are rejected, while valid updates will be prepared for further processing. Then, the *Update Decomposer* will decompose the translatable update query into several smaller-granularity update queries, each defined on a single relational table. The *Update Propagator* then analyzes what type of propagated update should be generated to keep the integrity constraints of the relational database satisfied. It also records the propagated updates into a metadata structure for the next translation step. Finally, the *Update Translator* translates the update information in the metadata structure into SQL update statements. These statements will be submitted to the relational database engine for execution.

6 Experiments

We conducted several experiments on our *Rainfall* update system to assess the performance of our update translation strategy in RXU case. We first show

the update translation over XML re-loading to claim that update translation is indeed viable in practice. We then show our update translation strategy is pretty stable and efficient in different update scenarios. In the last experiment, we describe the performance of each translation step. If not stated otherwise, all experiments use the XML schema from our running example in Figure 1. The XML data is randomly being generated. The test system is Intel(R) Celeron(TM) 733MHz processor, 384M memory, running Windows2000.

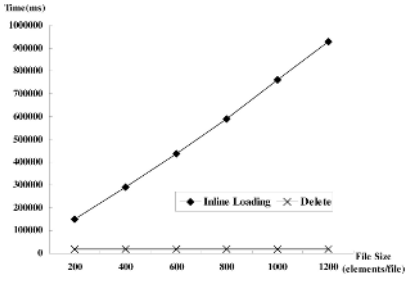


Fig. 14. Performance comparison of re-loading and update translation, shared inlining loading, delete update.

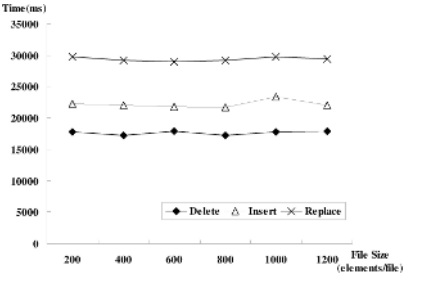


Fig. 15. Performance comparison for different update types, file-size = 800 elements/file.

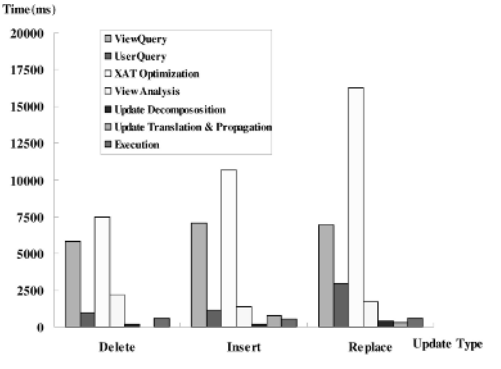


Fig. 16. Comparison of translation steps for different update types

(1) Experiment on Updating vs. Loading. We evaluate the cost of the translation of an XML update specified against an XML virtual view (Figure 6) into a relational update that then is executed against the relational database. We compare it against re-loading the XML data into relational data store after

directly being applied the update on the original XML document (Figure 14). We use a delete update on the view defined in Figure 6. The loading strategy used is shared inlining [17]. We observe that as the XML file size increases in the number of elements in the XML document, the re-loading time increases linearly in the size of the file. The update translation remains fairly steady. Thus the update translation is an efficient mechanism and indeed appears to be viable in practice.

(2) Experiment on Translation for Different Update Types. The performance of different update types is compared in Figure 15. The underlying relational database and view query are the same as in experiment 1. Update operations considered are delete, insert and replace on the view. We find that all three types of update costs are fairly stable even for increasing file sizes. Delete is the cheapest operation. While replace is the most expensive, it is still cheaper than performing a combination of a delete followed by an insert. This is the reason for the forth simplicity criteria described in Section 4. Given that only the last step of execution actually touches the relational data, the database size does not have much impact on the overall performance.

(3) Experiment on Translation Steps for Different Update Types. For the same experimental setup as in experiment 2, we now break down the costs for the different steps of update translation for the three update types. The result is shown in Figure 16. The result shows that the XAT optimization takes more time compared to XAT generation and update translation. We merge the user XAT and mapping XAT query trees, and optimize the merged XAT before we start the other update translation steps. The reason for this step is to simplify the XAT. Another costly step is the view analysis which analyzes the view structure, finds the related *relations* and their relationships and thus prepares for update translation.

7 Related Work

The view update problem has been studied in depth for relational databases. [9] is one of the first works dealing with view updates for relational databases. It stipulated a notion of *correct translation*, and described several conditions for the existence of such translation in the case of a *clean source*, that is updating a clean source will not generate any view side-effect. An abstract formulation of the update translation problem is given by the *view complementary theorem* in [7,2] which uses the complement of a view to resolve the ambiguity in mapping between old and new database states. Finally, [11,12,1] study the view update problem for SPJ queries on relations that are in Boyce-Codd Normal Form. Our work follows [7] to prove the correct translatability. However, it is more complex than the pure relational view update problem, since not only do all the problems in the relational context still exist in the XML semantics context, but in addition we have to address the mismatch coming from the two distinct data models. Our constraint mapping in Figure 11 takes this mismatch into account, thus addressing some of the key issues in the XML context.

Closely related to the work of [9], in [3], view update translation algorithms of [11] have been further extended for object-based views. However, an XML model has features in its schema and query language distinct from those in the OO model. The algebraic framework and the update decomposition strategy used in our system bridges the nested XQuery with SQL model gap. They thus provide us with a clear solution for view update translation.

The XML view update problem has not yet been much addressed by the database community. [15] introduces the XML view update in SQL-Server2000, based on a specific *annotated schema* and update language called *updategrams*. Different with their work, our update system explores in general the XML view update problem instead of a system-specific solution, though we have also implemented our ideas to check their feasibility. One of the recent work [18] presents an XQuery update grammar, and studies the performance of updates assuming that the view is indeed translatable and has in fact already been translated using a fixed shredding technique, that is inlining [17]. Instead of assuming update always translatable, our work addresses how the updatability infected by XML nested structure. The proposed solution is not limited in specific loading strategy. The most recent work [4] studies the updatability of XML view using nested relational algebra. By assuming the algebra representation of view do not include unnest operator, while nest operator occur last, and won't affect the view udpatability. However, by using XQuery to define the view, the unnested operator is unavoidable. Also, since the order of nest operator will decide the hierarchy of XML view, it will affect the view updatability. Compared to their work, updating XQuery view problem tackled in our paper is more complex.

8 Conclusions

In this paper, we have characterized the round-trip based XQuery view update problem in the context of XML views being published over relational databases. We prove that the updates issued on the view within this problem space are always translatable. A decomposition-based update translation approach is described for generating optimized update plans. A system framework for implementing this approach is also presented. Its performance is studied in various scenarios. Although we base our discussion and have implemented the update strategy in the context of the *Rainbow* XML management system, both the concepts and the algorithms can easily be applied to other systems.

References

1. A. M. Keller. The Role of Semantics in Translating View Updates. *IEEE Transactions on Computers*, 19(1):63–73, 1986.
2. F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. In *ACM Transactions on Database Systems*, pages 557–575, Dec 1981.
3. T. Barsalou, N. Siambela, A. M. Keller, and G. Wiederhold. Updating Relational Databases through Object-Based Views. In *10th ACM SIGACT-SIGMOD*, pages 248–257, 1991.

4. V. P. Braganholo, S. B. Davidson, and C. A. Heuser. On the Updatability of XML Views over Relational Databases. In *WEBDB*, 2003.
5. E. T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible Markup Language (XML), 1997. http://www.w3.org/TR/PR-xml-971208.
6. M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *The VLDB Journal*, pages 646–648, 2000.
7. S. S. Cosmadakis and C. H. Papadimitriou. Updates of Relational Views. *Journal of the Association for Computing Machinery*, pages 742–760, Oct 1984.
8. F. Daniela and K. Donald. Storing and Querying XML Data Using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.
9. U. Dayal and P. A. Bernstein. On the Correct Translation of Update Operations on Relational Views. In *ACM Transactions on Database Systems*, volume 3(3), pages 381–416, Sept 1982.
10. M. F. Fernandez, A. Morishima, D. Suciu, and W. C. Tan. Publishing Relational Data in XML: the SilkRoute Approach. *IEEE Data Engineering Bulletin*, 24(2):12–19, 2001.
11. A. M. Keller. Algorithms for Translating View Updates to Database Updates for View Involving Selections, Projections and Joins. In *Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 154–163, 1985.
12. A. M. Keller. Choosing a View Update Translator by Dialog at View Definition Time. In *VLDB*, pages 467–474, 1986.
13. A. M. Keller. Comments on Bancilhon and Spyratos' "update semantics and relational views". *ACM Transactions on Database Systems*, 12(3):521–523, 1987.
14. D. Lee and W. W. Chu. Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema. In *ER*, pages 323–338, Oct 2000.
15. M. Rys. Bringing the Internet to Your Database: Using SQL Server 2000 and XML to Build Loosely-Coupled Systems. In *VLDB*, pages 465–472, 2001.
16. A. Sahuguet and L. Dupont. Querying xml in the new millennium, 2002.
17. J. Shanmugasundaram, G. He, K. Tufte, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*, pages 302–314, September 1999.
18. I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *Proceedings of the ACM SIGMOD International Conference*, pages 413–424, May 2001.
19. W3C. XML Schema. http://www.w3.org/XML/Schema.
20. W3C. XQuery: A Query Language for XML. http://www.w3.org/TR/xquery/, February 2001.
21. L. Wang, M. Mulchandani, and E. A. Rundensteiner. Updating XQuery Views Published over Relational Data. Technical Report WPI-CS-TR-03-23, Computer Science Department, WPI, 2003.
22. X. Zhang, K. Dimitrova, L. Wang, M. EL-Sayed, B. Murphy, L. Ding, and E. A. Rundensteiner. RainbowII: Multi-XQuery Optimization Using Materialized XML Views. In *Demo Session Proceedings of SIGMOD*, 2003.
23. X. Zhang and E. Rundensteiner. XAT: XML Algebra for Rainbow System. Technical Report WPI-CS-TR-02-24, Computer Science Department, WPI, July 2002.

Repairs and Consistent Answers for XML Data with Functional Dependencies

S. Flesca, F. Furfaro, S. Greco, and E. Zumpano

D.E.I.S. – Università della Calabria
Via P. Bucci
87036 – Rende (CS)
ITALY
{flesca, furfaro, greco, zumpano}@si.deis.unical.it

Abstract. In this paper we consider the problem of XML data which may be inconsistent with respect to a set of functional dependencies. We propose a technique for computing repairs (minimal sets of update operations making data consistent) and consistent answers. More specifically, our repairs are based on i) the replacing of values associated with attributes and elements, and ii) the introduction of a function stating if the node information is reliable.

1 Introduction

The World-Wide-Web is of strategic importance as a global repository for information and a means of communicating and sharing knowledge. Its explosive growth has caused deep changes in all the aspects of human life, has been a driving force for the development of modern applications (e.g., Web portals, digital libraries, wrapper generators, etc.) and has greatly simplified the access to existing sources of information, ranging from traditional DBMS to semi-structured Web repositories. The adoption by the WWW consortium (W3C) of XML (eXtensible Markup Language) as the new standard for information exchange among web applications has let researchers to investigate classical problems in the new environment of repositories containing large amounts of data in XML format.

Great attention has been recently also devoted to the introduction of integrity constraints and the definition of normal forms for XML [4,6,10,13]. XML allows a simple form of constraints to describe references obtained through ID/IDREF, but it does not actually provides a general mechanism for expressing semantic constraints like those commonly used in relational databases. The need of enriching the semantics of XML is so deep as a large amount of XML data originates in object-oriented and relational databases, where different forms of integrity constraints are used to add semantics to the collected information.

This work stems from the need of enriching the semantics of XML documents. This need is attested by new several works which introduce different forms of constraints to XML documents [4,5,6,10,13]. Most of them introduce a simple

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 238–253, 2003.

form of constraints such as keys and foreign keys, whereas some others attempt to extend the class of integrity constraints associated to XML documents.

Obviously, reasoning about constraints in the presence of an incomplete knowledge of the data structure is rather complex, so that some of these attempts are likely of being a purely theoretical exercise. In fact, their practical applicability follows the solution of non trivial problems such as the implication and the interaction among constraints which are far from being solved.

In the presence of constraints on data, an XML document may result to be inconsistent, i.e. it does not respect some constraint. The following example shows the case of an inconsistent XML document.

Example 1. Consider the following XML document representing a collection of books

```
<bib>
    <book isbn="0-451-16194-7">
       <title> A First Course in Database Systems </title>
       <author> Ullman </author>
       <author> Widom  </author>
       <publisher> Prentice-Hall </publisher>
    </book>
    <book isbn="0-451-16194-7">
       <title> Principles of Database and Knowledge-Base Systems </title>
       <author> Ullman </author>
       <publisher> Computer Science Press </publisher>
    </book>
</bib>
```

and the functional dependency `bib.book.@isbn` $\rightarrow$ `bib.book.title.S` stating that two books with the same isbn must have the same title. [1]

The above document does not satisfy this functional dependency, as the first and the second book have the same `isbn` attribute, but different titles. □

The above example shows that, generally, the satisfaction of constraints cannot be guaranteed, thus in the presence of an XML document which must satisfy a set of constraints we have to manage potential inconsistencies of data. This problem has been recently investigated for relational databases and several techniques based on the computation of repairs (minimal sets of insert/delete operations) and consistent answers have been proposed for such a context [3, 11]. However, these techniques cannot be easily extended to XML data because of the different structure of data and the different nature of constraints. For instance, the document of the previous example can be repaired by performing the following minimal sets of update operations:

- replace the string `A First Course in Database Systems` with the title `Principles of Database and Knowledge − Base Systems`,
- replace the string `Principles of Database and Knowledge − Base Systems` with the title `A First Course in Database Systems`,

[1] The symbol `S` is used to extract the text content from an element.

– assign a new different value to one of the two `isbn` attributes, so that there are no two books with the same isbn.

Note that the document can be made consistent by replacing one of the two values `"0-451-16194-7"` with any value in the domain, a part from those introducing inconsistencies. To this end we shall use the unknown value $\perp$ in order to replace inconsistent data. Moreover, when inconsistencies cannot be repaired by assigning different values to attributes or changing some element content, we consider an alternative strategy which uses a boolean function specifying the *reliability* of elements.

Generally, more than one strategy can be used to repair a document, thus generating several repaired documents. Concerning the issue of querying an XML document with functional dependencies, we shall consider as *certain information* only the information contained in all possible repaired documents.

The violation of a functional dependency suggests a set of possible update operations in order to ensure its satisfiability, yielding a consistent scenario of the information. In repairing documents we prefer the repairs performing minimal sets of changes to the original document, in the same way as well known approaches proposed for relational database repairing.

Example 2. Consider the XML document of the previous Example where the element `title` in the first book is missing. In this case, the update action consisting in assigning the value `Principles of Database and Knowledge-Base Systems` to the title of the first book is reliable.

Consider again the XML document of the previous example with the functional dependency `bib.book.@isbn` $\rightarrow$ `bib.book` stating that two books having the same isbn coincide. In this case we could consider two repairs which make the isbn value unreliable, and two repairs which make the (node) book unreliable. However, as the unreliability of a book implies the unreliability of all its (sub-)elements, we consider as feasible only the two repairs updating the isbn value. □

2 Preliminaries

XML Trees and DTDs

A *tree* T is a tuple $(r_T, N_T, E_T, \lambda_T)$, where $N_T \subseteq \mathbb{N}$ is the set of nodes, $\lambda_T : N_T \rightarrow \Sigma$ is a node labelling function, $r_T \in N_T$ is the distinguished root of t, and $E_T \subseteq N_T \times N_T$ is an (acyclic) set of edges such that starting from any node $n_i \in N_T$ it is possible to reach any other node $n_j \in N_T$, walking through a sequence of edges $e_1, \ldots, e_k$. The set of leaf nodes of a tree T will be denoted as $Leaves(T)$.

Given a tree $T = (r_T, N_T, E_T, \lambda_T)$, we say that a tree $T' = (r_{T'}, N_{T'}, E_{T'}, \lambda_{T'})$ is a *subtree* of T if the following conditions hold:

1. $N_{T'} \subseteq N_T$;

2. the edge (n_i, n_j) belongs to $E_{T'}$ iff $n_i \in N_{T'}$, $n_j \in N_{T'}$ and $(n_i, n_j) \in E_T$.

The set of trees defined on the alphabet of node labels Σ will be denoted as T_Σ.

Given a tag alphabet τ, an attribute name alphabet α, a string alphabet Str and a symbol S not belonging to $\tau \cup \alpha$, an *XML tree* is a pair $XT = \langle T, \delta \rangle$, where:

- $T = (r, N, E, \lambda)$ is a tree in $T_{\tau \cup \alpha \cup \{S\}}$;
- given a node n of T, $\lambda(n) \in \alpha \cup \{\mathtt{S}\} \Leftrightarrow n \in Leaves(T)$;
- $\delta : Leaves(T) \rightarrow Str$ is a function associating a (string) value to every leaf of T.

The symbol S is used to represent the #PCDATA content of elements.

A DTD is a tuple $D = (\tau, \alpha, P, R, rt)$ where: i) P is the set of *element type definitions*; ii) R is the set of *attribute lists*; iii) $rt \in \tau$ is the tag of the document root element.

Example 3. The following XML document (conforming the DTD reported on the right-hand side of the document) represents a collection of books, and is graphically represented by the XML tree in Fig. 1.

```
<bib>                                   <!ELEMENT bib (book+)>
  <book>                                <!ELEMENT book (written_by, title,
    <written_by>                                    pub, year?)>
      <author ano="A1">                 <!ELEMENT written_by (author+)>
         <name>Ullman</name>            <!ELEMENT author (name)>
      </author>                           <!ATTLIST author ano CDATA>
      <author ano="A2">                 <!ELEMENT name PCDATA>
         <name>Widom</name>             <!ELEMENT title PCDATA>
      </author>                         <!ELEMENT pub PCDATA>
    </written_by>                       <!ELEMENT year PCDATA>
    <title> A First Course in
       Database Systems </title>
    <publisher> Prentice-Hall </publisher>
  </book>
  <book>
    <written_by>
      <author ano="A1">
         <name>Ullman</name>
      </author>
    </written_by>
    <title> Principles of Database
        and Knowledge-Base Systems
    </title>
    <publisher> CS Press </publisher>
  </book>
</bib>
```

The internal nodes of the XML tree have a unique label, denoting the tag name of the corresponding element. The leaf nodes correspond to either an attribute or the textual content of an element, and are labelled with two strings. The first one denotes the attribute name (in the case that the node represents

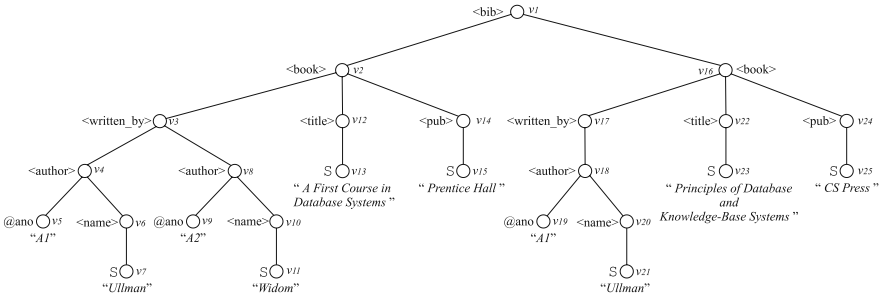


Fig. 1. An XML Tree

an attribute) or is equal to the symbol S (in the case that the node represents an element content). The second label denotes either the value of the attribute or the string contained inside the element corresponding to the node. □

A path p on a DTD $D = (\tau, \alpha, P, R, rt)$ is a sequence $p = s_1, \ldots, s_m$ of symbols in $\tau \cup \alpha \cup \{\mathtt{S}\}$ such that:

1. $s_1 = rt$;
2. for each i in $2..m-1$, $s_i \in \tau$ and s_i appears in the element type definition of s_{i-1};
3. $s_m \in \alpha \Rightarrow s_m$ appears in the attribute list of s_{m-1};
4. $s_m \in \tau \cup \{S\} \Rightarrow s_m$ appears in the element type definition of s_{m-1}.

The set of paths which can be defined on a DTD D will be denoted as $paths(D)$. In particular, $paths(D)$ is partitioned into two disjoint sets: 1) $EPaths(D)$, which contains all the paths $p = s_1, \ldots, s_m$ where $s_m \in \tau$ (i.e. the paths whose last symbol denotes an element); 2) $StrPaths(D)$ contains the paths whose last symbol denotes either the textual content of an element or an attribute.

Example 4. Consider the DTD D of Example 3. The set of paths defined on D is partitioned into the following sets:

$EPaths(D) = \{$ bib, bib.book, bib.book.written_by,
bib.book.written_by.author,
bib.book.written_by.author.name,
bib.book.title, bib.book.pub, bib.book.year $\}$

$StrPaths(D) = \{$ bib.book.written_by.author.@ano,
bib.book.written_by.author.name.S, bib.book.title.S,
bib.book.pub.S, bib.book.year.S $\}$

□

Given an XML tree $XT = \langle T, \delta \rangle$ conforming a DTD D, a path $p \in paths(D)$ identifies the set of nodes which can be reached, starting from the root of XT, by going through a sequence of nodes "spelling" p. More formally, $p = s_1, \ldots, s_m$ identifies the set of nodes $\{n_1, \ldots, n_k\}$ of XT such that, for each $i \in 1..k$, there exists a sequence of nodes $n_{i_1}, \ldots, n_{i_m}$ with the following properties:

1. $n_{i_1} = r_T$ and $n_{i_m} = n_i$;
2. for each $j \in 1..m-1$, $n_{i_{j+1}}$ is a child of n_{i_j};
3. for each $j \in 1..m$, $\lambda(n_{i_j}) = s_j$.

The set of nodes of XT identified by p will be denoted as $p(XT)$. Moreover, we denote with $XT.p$ the *answer* of the path p applied on XT, that is:

- if $p \in EPath(D)$, then $XT.p = p(XT)$;
- if $p \in StrPath(D)$, then $XT.p = \{\delta_T(x) | x \in p(XT)\}$.

Thus, the answer of a path p applied on XT is either a set of node identifiers, or a set of (string) values, depending on whether the last symbol s_m in p belongs to τ (i.e. s_m is a tag name) or to $\alpha \cup \{\mathtt{S}\}$ (i.e. s_m is either an attribute name or the symbol S).

Example 5. Let XT be the XML tree of Fig. 1. In the following table we report the answers of different paths (defined over the DTD associated to XT) applied on XT.

path p	XT.p
bib.book.title	$\{v_{12}, v_{22}\}$
bib.book.title.S	{ "A First Course ..." , "Principles of Database ..." }
bib.book.written_by.author	$\{v_4, v_8, v_{18}\}$
bib.book.written_by.author.@ano	{ "A1" , "A2" }
bib.book.year	$\emptyset$
bib.book.year.S	$\emptyset$

The answers to both the paths *bib.book.year* and *bib.book.year*.S are empty sets, as there is no node in XT associated to an element `year`. □

3 XML and Functional Dependencies

In this Section, we recall the notion of functional dependency in the XML setting proposed in [4,6] [2]. A functional dependency $A \rightarrow B$ in a relational database D models the correspondence between A and B values in the tuples of D. However, there is no standard tuple concept for XML. Thus, before introducing functional dependencies for XML, we provide the concept of tree tuples, corresponding to the concept of tuples in relational databases.

Informally, a tree tuple groups together nodes of the document which are semantically correlated, according to the structure of the tree. For instance, a tree tuple of the XML tree XT of Fig. 1 consists of a sub-tree which contains information about a book. Observe that each book is possibly described by more than one tree tuple, as each tree tuple contains the information of only one author (see Example 6).

[2] An alternative definition has been proposed in [13]

Definition 1 (Tree Tuple). *Given an XML tree XT conforming the DTD D, a tree tuple t of XT is a maximal sub-tree of XT such that, for every path $p \in paths(D)$, $t.p$ contains at most one element.* □

Example 6. Consider the XML tree XT of Fig. 1. The subtrees of XT shown in Fig. 2(a) and Fig. 2(b) are tree tuples, whereas the subtrees in Fig. 3(a) and Fig. 3(b) are not tree tuples.

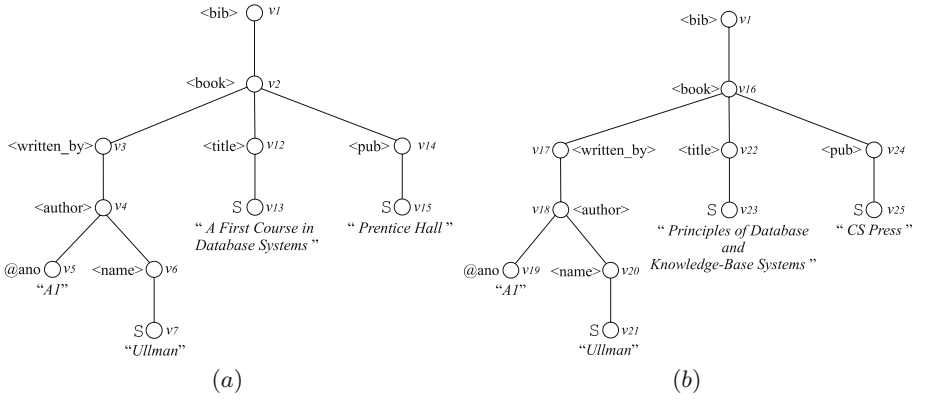


Fig. 2. Two tree tuples of the XML tree of Fig. 1

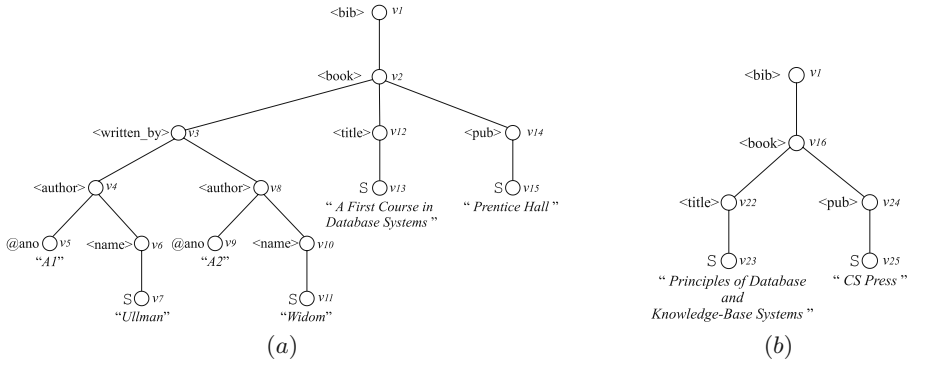


Fig. 3. Two subtrees of the XML tree of Fig. 1 which are not tree tuples

The subtree of Fig. 3(a) is not a tree tuple as there are two distinct nodes (i.e. v_4 and v_8) which correspond to the same path *bib.book.written_by.author*. This means that each book stored in XT can correspond to more than one tree tuple: each tree tuple corresponds to one of the book authors.

The subtree of Fig. 3(b) is not a tree tuple as it is not maximal: it is a subtree of the tree tuple of Fig. 2(b). □

Given a XML tree XT, a pair of tree tuples t_1, t_2 of XT, and a set $S \subseteq paths(D)$, $t_1.S = t_2.S$ means that $t_1.p = t_2.p$ for each path $p \in S$. Moreover we say that $t_1.S \neq \emptyset$ if $t_1.p \neq \emptyset$ for each $p \in S$.

Definition 2 (Functional Dependency). Given a DTD D, a functional dependency on D is an expression of the form $S \rightarrow p$, where S is a finite non empty subset of $paths(D)$ and p is an element of $paths(D)$. □

Given an XML tree XT conforming a DTD D and a functional dependency $F : S_1 \rightarrow S_2$, we say that XT *satisfies* F ($XT \models F$) iff for each pair of tree tuples t_1, t_2 of XT, $t_1.S_1 = t_2.S_1 \wedge t_1.S_1 \neq \emptyset \Rightarrow t_1.S_2 = t_2.S_2$. Given a set of functional dependencies $\mathcal{FD} = \{F_1, \ldots, F_n\}$ over D, we say that XT satisfies $\mathcal{FD}$ if it satisfies F_i for every $i \in 1..n$.

Example 7. Consider the XML tree XT of Fig. 1. The constraint that the attribute @`ano` identifies univocally the (value of the) name of every author can be expressed with the following functional dependency:

$$bib.book.written\_by.author.@ano \rightarrow bib.book.written\_by.author.name.\texttt{S}$$

To say that two distinct authors of the same book cannot have the same value of the attribute `ano` we can use the following FD:

$$\{bib.book, bib.book.written\_by.author.@ano\} \rightarrow bib.book.written\_by.author$$

□

A set of functional dependencies $\mathcal{FD}$ over a DTD D is satisfiable if there exists an XML tree XT conforming D such that $XT \models \mathcal{FD}$.

4 Repairing and Querying Inconsistent XML Databases

In this Section we present an approach to the problem of repairing XML documents which are inconsistent w.r.t. a given set of functional dependencies. A possibly inconsistent XML document can be repaired by taking two different kind of actions: 1) by changing the value of an attribute or the content of an element, 2) by marking some of the attributes or elements of the document as "unreliable".

Example 8. Consider the following XML document conforming the DTD reported on its right-hand side:

```
<cars>                                  <!ELEMENT cars (car+)>
  <car cno="c1">                        <!ELEMENT car (policy?, garage+)>
    <policy pno="p1"/>                    <!ATTLIST car cno CDATA>
    <garage>                            <!ELEMENT policy EMPTY>
      <name> Olympo </name>               <!ATTLIST policy pno CDATA>
      <city> Boston </city>             <!ELEMENT garage (name, city)>
    </garage>                           <!ELEMENT name PCDATA>
    <garage>                            <!ELEMENT city PCDATA>
      <name> Johnson </name>
      <city> Cambridge </city>
    </garage>
  </car>
</cars>
```

and the functional dependency $\{\texttt{cars.car.policy}\} \rightarrow \texttt{cars.car.garage}$ saying that, if a car has a policy, then it can be repaired by only one garage. Otherwise, if no policy is associated to the car, then it can be repaired in more than one garage. □

The above document does not satisfy the functional dependency, as the car with $@\texttt{cno} = \texttt{c1}$ has a policy, but is associated with two garages. This inconsistency may have one of the following causes: 1) the `policy` element is incorrect; 2) one of the two `author` elements is incorrect.

The above functional dependency involves only node identifiers, so that it is not possible to repair the document by changing some of its element values. A possible repair strategy consists of considering *unreliable* either the `policy` element or one of the `author` elements.

We point out that marking a node as unreliable is a more preserving mechanism than simply deleting it. Indeed, a simple deletion of a whole `garage` element would produce undesired side-effects. For instance, if we delete one of the two `garage` elements and then ask whether the car can be repaired in only one garage, the answer would be *"yes"*. On the contrary, by marking one of the two `garage` elements as "unreliable", we will consider the *"yes"* answer as not reliable.

Example 9. Consider the XML tree XT of Fig. 4, conforming the DTD D of Example 3 and suppose that we are given the following functional dependency: $\{\texttt{bib.book}, \texttt{bib.book.written\_by.author.@ano}\} \rightarrow \texttt{bib.book.written\_by.author}$.

The XML tree XT does not satisfy the above FD, as the two `author` elements, contained in the same book, have the same value of the attribute `@ano`, whereas the above FD requires that, for each book, there is only one author having a given `@ano` value. □

The constraint in the above example may not be satisfied for two possible reasons: 1) one of the two `@ano` values is incorrect; 2) one of the two `author` elements is incorrect.

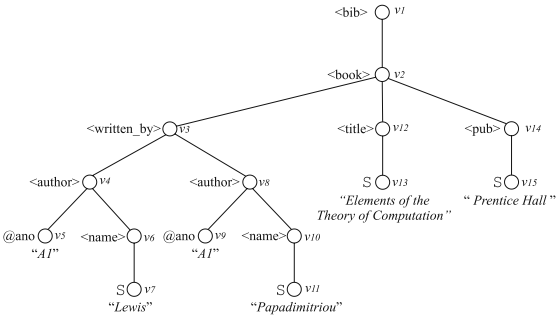


Fig. 4. An XML tree

Therefore, two repairing strategies are possible. If we assume that the former of the two errors occurs, we are induced to change the @`ano` value of one of the authors. That is, we can make XT consistent w.r.t. the given FD by assigning a new value (denoted as $\perp_1$) to the attribute @`ano` of any of the `author` elements (see Fig. 5(a)).

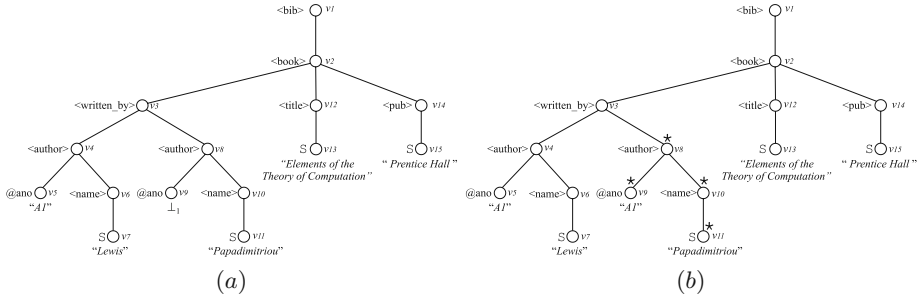


Fig. 5. Two repairs of the XML tree of Fig. 4

Otherwise, if we assume that the latter error occurs (i.e. one of the two `author` elements is incorrect), we choose to mark one of the two authors having the same @`ano` as unreliable (see Fig. 5(b), where unreliable nodes are marked with the symbol $\star$).

However, the latter strategy changes a larger portion of the document, since it marks a whole `author` element as unreliable, whereas the first strategy only changes its @`ano`. Repair strategies performing smaller changes to the original document will be preferred, in the same way as in well-known approaches to relational database repairing [3,11].

Thus, we propose two different kinds of actions which can be performed for repairing inconsistent XML documents: 1) updating element values and 2) marking elements as unreliable. Observe that we prefer marking a node as unreliable

rather than deleting it, since removing elements from an XML document leads to two undesired side effects: it causes incorrect answers to queries, like in example 8, and does not always suffice to remove inconsistency. In fact, deleting a node can lead to a new document not conforming the given DTD.

4.1 R-XML Tree

Given an XML tree XT, the reliability of the nodes of XT is given by providing a boolean function that assigns "*true*" to every reliable node and "*false*" to every unreliable node. More formally:

Definition 3 (R-XML tree). A *R-XML tree* is a triplet $RXT = \langle T, \delta, \varrho \rangle$ where $\langle T, \delta \rangle$ is an XML tree and ϱ is a reliability function from N_T to $\{\texttt{true}, \texttt{false}\}$, such that, for each pair of nodes $n_1, n_2 \in N_T$ with n_2 descendent of n_1, it holds that $\varrho(n_1) = false \Rightarrow \varrho(n_2) = false$. □

An XML tree XT is an R-XML tree such that ϱ returns true for all nodes in XT. Thus, a R-XML tree can be thought of as an XML tree where each node is marked with a boolean value (*true* if the node is reliable, and *false* otherwise). We now introduce the concept of satisfiability of functional dependencies over R-XML trees.

Definition 4 (Weak satisfiability). Let $RXT = \langle T, \delta, \varrho \rangle$ be an R-XML tree conforming a DTD D, and $f : S \rightarrow p$ be a functional dependency. We say that RXT *weakly satisfies* f ($RXT \models_w f$) if one of the following conditions holds:

1. $\langle T, \delta \rangle \models f$;
2. for each pair of tuples t_1, t_2 of RXT one of the following holds:
   a. there exists a path $p_i \in S$ such that:
      $(\varrho(p_i(t_1)) = false) \ \vee \ (\varrho(p_i(t_2)) = false)$;
   b. $(\varrho(p(t_1)) = false) \ \vee \ (\varrho(p(t_2)) = false)$. □

It is worth noting that for XML-trees the weak satisfiability reduces to the standard notion of satisfiability. Basically, the weak satisfiability does not consider unsatisfied functional dependencies over paths containing unreliable nodes.

Given a set of functional dependencies $\mathcal{FD} = \{F_1, \ldots, F_n\}$ over D, we say that RXT weakly satisfies $\mathcal{FD}$ ($D \models_w \mathcal{FD}$) if it weakly satisfies F_i for every $i \in 1..n$.

Before presenting our repairing technique we need some preliminary notations. The composition of two reliability functions ϱ_1 and ϱ_2 is $\varrho_1 \cdot \varrho_2(n) = min(\varrho_1(n), \varrho_2(n))$. The composition of two functions δ_1 and δ_2 associating values to leaf nodes is

$$\delta_1 \cdot \delta_2(n) = \begin{cases} \delta_1(n) \text{ if } \delta_1(n) \text{ is defined over } n, \\ \delta_2(n) \text{ otherwise (i.e. } \delta_1(n) \text{ is not defined over } n). \end{cases}$$

The composition of functions is useful to update node values (strings assigned to leaf nodes and reliability values). Moreover, by composing two reliability functions, the value of a node cannot be increased (i.e. reliable nodes can be made unreliable, but unreliable nodes cannot be made reliable).

In the following, for a given R-XML tree $RXT = \langle T, \delta_T, \varrho_T \rangle$ and reliability function ϱ (resp. function assigning leaf values δ), we denote with $\varrho(RXT) = \langle T, \delta_T, \varrho \cdot \varrho_T \rangle$ (resp. $\delta(RXT) = \langle T, \delta \cdot \delta_T, \varrho_T \rangle$) the application of ϱ (resp. δ) to RXT.

Definition 5 (Weak repair). Let $RXT = \langle T, \delta, \varrho \rangle$ be an R-XML tree conforming a DTD D and $\mathcal{FD}$ a set of functional dependencies. A (weak) repair for RXT is a pair of functions δ' and ϱ' such that $RXT' = \langle T, \delta' \cdot \delta, \varrho' \cdot \varrho \rangle$ weakly satisfies FD ($RXT \models_w \mathcal{FD}$). □

Example 10. Consider the XML document of Example 3, graphically represented in Fig. 1, and the functional dependency `bib.book.written_by.author.@ano` → `bib.book.written_by.author`.

The document is not consistent as there are two authors with the same value for the attribute `@ano`. Possible repairs are: $R_1 = \langle \{\delta(\texttt{v5}) = \bot\}, \varrho_{\{\}}(\texttt{v}) \rangle$, $R_2 = \langle \{\delta(\texttt{v9}) = \bot\}, \varrho_{\{\}}(\texttt{v}) \rangle$, $R_3 = \langle \{\}, \varrho_{\{\texttt{v4,v5,v6,v7}\}}(\texttt{v}) \rangle$ and $R_4 = \langle \{\}, \varrho_{\{\texttt{v8,v9,v10,v11}\}}(\texttt{v}) \rangle$, where the function $\varrho_S(v)$ states that $v \in S$ is defined *false* and $v \notin S$ is defined *true* by ϱ. □

As we have assumed that the reliability value of a node cannot be greater than the reliability value of its ancestors, we often do not specify the reliability value of descendants of unreliable nodes. For instance, regarding the reliability function of the repair R_3, we shall denote R_3 as $\langle \{\}, \varrho_{\{\texttt{v4}\}} \rangle$, as the nodes $v5, v6$ and $v7$ are descendant of the node $v4$,.

The set of weak repairs for a possibly inconsistent R-XML tree RXT, with respect to a set of functional dependencies $\mathcal{FD}$, will by denoted by $\mathbf{R}(RXT, \mathcal{FD})$.

Given a set of of labelled nodes N and a reliability function ϱ defined on N, we denote with $True_\varrho(N) = \{n \in N | \varrho(n) = true\}$ and with $False_\varrho(N) = \{n \in N | \varrho(n) = false\}$. Analogously, we denote with $Updated_\delta(N)$ the set of (leaf) nodes on which δ is defined, i.e. the set of nodes modified by δ. With a little abuse of notation we apply the functions $True_\varrho$, (resp. $False_\varrho$, $Updated_\delta$) to trees as well. When these functions are applied to a R-XML tree $RXT = \langle T, \delta, \varrho \rangle$, their results consist of the subtree of RXT only containing the nodes in $True_\varrho(N_T)$ (resp. $False_\varrho(N_T)$, $Updated_\delta(N_T)$).

Definition 6 (Minimal Repair). Let $XT = \langle T, \delta \rangle$ be an XML Tree conforming a DTD D, $\mathcal{FD}$ a set of functional dependencies and $R_1 = \langle \delta_1, \varrho_1 \rangle$, $R_2 = \langle \delta_2, \varrho_2 \rangle$ two repairs for XT. We say that R_1 is *smaller than* R_2 ($R_1 \preceq R_2$) if $Updated_{\delta_1}(N_T) \cup False_{\varrho_1}(N_T) \subseteq Updated_{\delta_2}(N_T) \cup False_{\varrho_2}(N_T)$ and $False_{\varrho_1}(N_T) \subseteq False_{\varrho_2}(N_T)$.

Moreover, we say that a repair R is minimal if there is no repair $R' \neq R$ such that $R' \preceq R$. □

We also use the notation $R_1 \prec R_2$ if $R_1 \neq R_2$ and $R_1 \preceq R_2$.

Example 11. Consider the repairs of Example 10. As $R_1 \prec R_3$ and $R_2 \prec R_4$, R_1 and R_2 are minimal repairs. □

Minimal repairs give preference to smaller sets. However, as a repair can be obtained by either changing the value of a node or making it unreliable, minimal repairs give preference to value updates. The set of weak repairs for a possibly inconsistent XML tree RXT with respect to a set of functional dependencies $\mathcal{FD}$ will by denoted by $\mathbf{MR}(RXT, \mathcal{FD})$.

Definition 7 (Weak answer). Let $RXT = \langle T, \delta, \varrho \rangle$ be an R-XML tree conforming a DTD D, $\mathcal{FD}$ a set of functional dependencies and p a path over D. The (weak) answer of the path p over RXT, denoted by $RXT.p$ is the pair $(XT.p, \varrho')$ where $XT = \langle T, \delta \rangle$ and ϱ' is the function ϱ defined only for the nodes in $XT.p$. □

Definition 8 (Possible and certain answers). Let $RXT = \langle T, \delta, \varrho \rangle$ be an R-XML tree conforming a DTD D, $\mathcal{FD}$ a set of functional dependencies and p a path over D.

- The possible answer of the path p over RXT, denoted by $RXT.p^{\exists}$, is

$$\bigcup_{(\delta', \varrho') \in \mathbf{MR}(RXT, \mathcal{FD})} True_{\varrho' \cdot \varrho}(\langle T, \delta' \cdot \delta, \varrho' \cdot \varrho \rangle).p$$

- The certain answer of the path p over RXT, denoted by $RXT.p^{\forall}$, is

$$\bigcap_{(\delta', \varrho') \in \mathbf{MR}(RXT, \mathcal{FD})} True_{\varrho' \cdot \varrho}(\langle T, \delta' \cdot \delta, \varrho' \cdot \varrho \rangle).p$$

□

As an XML tree is a special case of a R-XML tree, the possible and certain answers can be, obviously, also defined for XML trees.

Example 12. Consider the XML tree of Example 9 pictured in Fig 4, with the functional dependency from `@ano` to `author`. For the path query `bib.book.title.S`, both the possible and certain answers consist of the set { `"Elements of the Theory of Computation"` }. Moreover, for the path query `bib.book.author.name.S`, the possible answer is the set { `"Lewis"`, `"Papadimitriou"` }, whereas the certain answer is the empty set. □

5 A Technique for XML Repairs

We now present an algorithm computing certain queries.

Algorithm 1 first uses the function `computeRepairs`, which is described below, to compute the set of all the possible repairs for RXT w.r.t. $\mathcal{FD}$ (steps 2-4).

Algorithm 1
INPUT:
$RXT = \langle T, \delta, \varrho \rangle$*: R-XML tree conforming a DTD* D
$\mathcal{FD} = \{F_1, \ldots, F_m\}$*: Set of functional dependencies*
OUTPUT:
a unique repaired R-XML tree for computing certain answers
VAR
S*: Set of repairs*
begin
1) $S = \emptyset$
2) ***for each*** $(F : S \rightarrow p) \in \mathcal{FD}$ *s.t.* $RXT \not\models_w F$
3) ***for each*** t_1, t_2 *tuples of* RXT *s.t.* t_1, t_2 *do not weakly satisfy* F
4) $S = S \cup \text{computeRepairs}(F, t_1, t_2, RXT)$
5) $S = \text{removeNonMinimal}(S, RXT)$*;*
6) $\langle \delta', \varrho' \rangle = \text{mergeRepairs}(S)$
7) ***return*** $\langle T, \delta' \cdot \delta, \varrho' \cdot \varrho \rangle$
end

Function *computeRepairs*(F, t_1, t_2, RXT)
INPUT:
$RXT = \langle T, \delta, \varrho \rangle$: R-XML tree conforming a DTD D
$F : X \rightarrow p$ functional dependency
t_1, t_2 tuples of RXT
RETURNS:
S: Set of repairs
begin
1) $S = \emptyset$
2) **if** $p \in StrPaths(D)$ **then**
3) $S = S \cup \{\langle \{\delta(p(t_1)) = t_2.p\}, \varrho \rangle\} \cup \{\langle \{\delta(p(t_2)) = t_1.p\}, \varrho \rangle\}$
4) **else** $S = S \cup \{\langle \emptyset, \varrho_{\{t_1.p\}} \cdot \varrho \rangle\} \cup \{\langle \emptyset, \varrho_{\{t_2.p\}} \cdot \varrho \rangle\}$
5) **for each** $p_i \in X$ **do**
6) **if** $p_i \in StrPaths(D)$ **then**
7) $S = S \cup \{\langle \{\delta(p_i(t_1)) = \perp_1\}, \varrho \rangle\} \cup \{\langle \{\delta(p_i(t_2)) = \perp_2\}, \varrho \rangle\}$
8) **else** $S = S \cup \{\langle \emptyset, \varrho_{\{t_1.p_i\}} \cdot \varrho \rangle\} \cup \{\langle \emptyset, \varrho_{\{t_2.p_i\}} \cdot \varrho \rangle\}$
end

Fig. 6. Function ComputeRepairs

Then, non minimal repairs are removed from this set (step 5). Finally, all the repairs in this set are joined together, using the function `mergeRepairs`. This function returns an R-XML tree where all the possibly unreliable nodes (i.e. nodes that are unreliable in at least one repair, or nodes having different values in two distinct repairs) are marked (steps 6-7).

The function *ComputeRepairs* computes the set of repairs considering a functional dependency $F : X \rightarrow p$ and only two tree tuples over the input R-XML tree. The function build the following (alternative) repairs:

- if p defines a string, then one of the two terminal values $t_1.p$ and $t_2.p$ is changed, so that they become equal (step 3);
- if p defines a node, then either the node $t_1.p$ or the node $t_2.p$ is marked as unreliable (step 4);
- For each path p_i in X
  - if p_i defines a string, then one of the two terminal values $t_1.p_i$ and $t_2.p_i$ is changed to $\perp$ (step 7);
  - if p_i defines a node, then either the node $t_1.p_i$ or the node $t_2.p_i$ is marked as unreliable (step 8).

Given an R-XML tree $RXT = \langle T, \delta, \varrho \rangle$ and a set of repairs S, the function `mergeRepairs` computes a repair $\langle \delta', \varrho' \rangle$ defined as follows:

1. $\delta'(n) = v$ iff $\delta''(n) = v$ for all the repairs $\langle \delta'', \varrho'' \rangle \in S$ such that $\delta''(n)$ is defined;
2. $\varrho'(n) = false$ iff either there exists a repair $\langle \delta'', \varrho'' \rangle \in S$ such that $\varrho''(n) = false$, or there exist two repairs $\langle \delta_1, \varrho_1 \rangle$, $\langle \delta_2, \varrho_2 \rangle \in S$ such that $\delta_1(n)$ and $\delta_2(n)$ are both defined and $\delta_1(n) \neq \delta_2(n)$.

The following results characterize the complexity of Algorithm 1, and state that it can be correctly used to compute certain answer.

Theorem 1. *Algorithm 1 is sound and complete, and works in polynomial time.* □

Corollary 1. *Let $XT = \langle T, \delta \rangle$ be an XML Tree conforming a DTD D, $\mathcal{FD}$ a set of functional dependencies and p a path. The computation of the certain answer of p over XT ($XT.p^{\forall}$) can be done in polynomial time.* □

References

1. Abiteboul, S., Hull, R., Vianu, V., *Foundations of Databases*, Addison-Wesley, 1994.
2. Abiteboul, S., Segoufin, L., Vianu, V., Representing and Querying XML with Incomplete Information, *Proc. of Symposium on Principles of Database Systems (PODS)*, Santa Barbara, CA, USA, 2001.
3. Arenas, M., Bertossi, L., Chomicki, J., Consistent Query Answers in Inconsistent Databases, *Proc. of Symposium on Principles of Database Systems (PODS)*, Philadephia, PA, USA, 1999.
4. Arenas, M., Libkin, L., A Normal Form for XML Documents, *Proc. of Symposium on Principles of Database Systems (PODS)*, Madison, WI, USA, 2002.
5. Arenas, M., Fan, W., Libkin, L., On Verifying Consistency of XML Specifications, *Proc. of Symposium on Principles of Database Systems (PODS)*, Madison, WI, USA, 2002.
6. Arenas, M., Fan, W., Libkin, L., What's Hard about XML Schema Constraints? *Proc. of 13th Int. Conf. on Database and Expert Systems Applications (DEXA)*, Aix en Provence, France, 2002.

7. Atzeni, P., Chan, E. P. F., Independent Database Schemes under Functional and Inclusion Dependencies, *Proc. of 13th Int. Conf. on Very Large Data Bases (VLDB)*, Brighton, England, 1987.
8. Buneman, P., Davidson, S. B., Fan, W., Hara, C. S., Tan, W. C., Keys for XML, *Computer Networks*, Vol. 39(5), 2002.
9. Buneman, P., Fan, W., Weinstein, S., Path Constraints in Semistructured and Structured Databases, *Proc. of Symposium on Principles of Database Systems (PODS)*, Seattle, WA, USA, 1998.
10. Fan, W., Libkin, L., On XML integrity constraints in the presence of DTDs, *Journal of the ACM*, Vol. 49(3), 2002.
11. Greco, S., and Zumpano E., Querying Inconsistent Databases, *Proc. of 7th Int. Conf. on Logic for Programming and Automated Reasoning (LPAR)*, Reunion Island, France, 2000.
12. Suciu, D., Semistructured Data and XML, *Proc. of 5th Int. Conf. on Foundations of Data Organization and Algorithms (FODO)*, Kobe, Japan, 1998.
13. Vincent, M. W., Liu, J., Functional Dependencies for XML. *Proc. of 5th Asia Pacific Web Conference (APWeb)*, 2003.
14. Yang, X., Yu, G., Wang G., Efficiently Mapping Integrity Constraints from Relational Database to XML Document, *Proc. of 5th East European Conf. on Advances in Databases and Information Systems (ADBIS)*, Vilnius, Lithuania, 2001.

A Redundancy Free 4NF for XML

Millist W. Vincent, Jixue Liu, and Chengfei Liu

School of Computer and Information Science
University of South Australia
{millist.vincent, jixue.liu, chengfei.liu}@unisa.edu.au

Abstract. While providing syntactic flexibility, XML provides little semantic content and so the study of integrity constraints in XML plays an important role in helping to improve the semantic expressiveness of XML. Functional dependencies (FDs) and multivalued dependencies (MVDs) play a fundamental role in relational databases where they provide semantics for the data and at the same time are the foundation for database design. In some previous work, we defined the notion of multivalued dependencies in XML (called XMVDs) and defined a normal form for a restricted class of XMVDs, called hierarchical XMVDs. In this paper we generalise this previous work and define a normal form for arbitrary XMVDs. We then justify our definition by proving that it guarantees the elimination of redundancy in XML documents.

1 Introduction

XML has recently emerged as a standard for data representation and interchange on the Internet [18,1]. While providing syntactic flexibility, XML provides little semantic content and as a result several papers have addressed the topic of how to improve the semantic expressiveness of XML. Among the most important of these approaches has been that of defining integrity constraints in XML [3]. Several different classes of integrity constraints for XML have been defined including key constraints [3,4], path constraints [6], and inclusion constraints [7] and properties such as axiomatization and satisfiability have been investigated for these constraints. However, one topic that has been identified as an open problem in XML research [18] and which has been little investigated is how to extended the traditional integrity constraints in relational databases, namely *functional dependencies* (FDs) and *multivalued dependencies* (MVDs), to XML and then how to develop a normalisation theory for XML. This problem is not of just theoretical interest. The theory of normalisation forms the cornerstone of practical relational database design and the development of a similar theory for XML will similarly lay the foundation for understanding how to design XML documents. In addition, the study of FDs and MVDs in XML is important because of the close connection between XML and relational databases. With current technology, the source of XML data is typically a relational database [1] and relational databases are also normally used to store XML data [9]. Hence, given that FDs and MVDs are the most important constraints in relational databases, the study

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 254–266, 2003.

of these constraints in XML assumes heightened importance over other types of constraints which are unique to XML [5].

In this paper we extend some previous work [16,15] and consider the problem of defining multivalued dependencies and normal forms in XML documents. Multivalued dependencies in XML (called XMVDs) were first defined in [16]. In that paper we extended the approach used in [13,14] to define functional dependencies and defined XMVDs in XML documents. We then formally justified our definition by proving that, for a very general class of mappings from relations to XML, a relation satisfies a multivalued dependency (MVD) if and only if the corresponding XML document satisfies the corresponding XMVD. The class of mappings considered was those defined by converting a flat relation to a nested relation by an arbitrary sequences of nest operators, and then mapping the nested relation to an XML document in the obvious manner. Thus our definition of a XMVD in an XML document is a natural extension of the definition of a MVD in relations. In [15] the issue of defining normal forms in the presence of XMVDs was addressed. In that paper we defined a normal form for a restricted class of XMVDs, namely what we termed *hierarchical XMVDs*. Also, extending some of our previous work on formally defining redundancy in flat relations ([11, 12,8]) and in XML ([13]), we formally defined redundancy in [15] and showed that the normal form that we defined guaranteed the elimination of redundancy in the presence of XMVDs.

The main contribution of this paper is to extend the results obtained in [15]. As just mentioned, in [15] we considered only a restricted class of XMVDs called hierarchical XMVDs. Essentially, an XMVD is hierarchical if the paths on the r.h.s. of an XMVD are descendants of the path on the l.h.s. of the XMVD. In this paper we define a normal form for arbitrary XMVDs, i.e. no retriction is placed on the relationships between the paths in the XMVD. We then formally justify our definition by proving that it guarantees the elimination of redundancy.

The rest of this paper is organised as follows. Section 2 contains some preliminary definitions. Section 3 contains the definition of an XMVD. In Section 4 we define a 4NF for XML and prove that it eliminates redundancy. Finally, Section 5 contains some concluding comments.

2 Preliminary Definitions

In this section we present some preliminary definitions that we need before defining XFDs. We model an XML document as a tree as follows.

Definition 1. *Assume a countably infinite set* $\mathbf{E}$ *of element labels (tags), a countable infinite set* $\mathbf{A}$ *of attribute names and a symbol* $\mathcal{S}$ *indicating text. An* XML tree *is defined to be* $T = (V, lab, ele, att, val, v_r)$ *where* V *is a* finite *set of nodes in* T*;* lab *is a function from* V *to* $\mathbf{E} \cup \mathbf{A} \cup \{\mathcal{S}\}$*;* ele *is a partial function from* V *to a sequence of* V *nodes such that for any* $v \in V$*, if* $ele(v)$ *is defined then* $lab(v) \in \mathbf{E}$*;* att *is a partial function from* $V \times \mathbf{A}$ *to* V *such that for any* $v \in V$ *and* $l \in \mathbf{A}$*, if* $att(v, l) = v_1$ *then* $lab(v) \in \mathbf{E}$ *and* $lab(v_1) = l$*;* val *is a*

function such that for any node in $v \in V, val(v) = v$ *if* $lab(v) \in \mathbf{E}$ *and* $val(v)$ *is a string if either* $lab(v) = \mathcal{S}$ *or* $lab(v) \in \mathbf{A}$*;* v_r *is a distinguished node in* V *called the root of* T *and we define* $lab(v_r) = root$*. Since node identifiers are unique, a consequence of the definition of val is that if* $v_1 \in \mathbf{E}$ *and* $v_2 \in \mathbf{E}$ *and* $v_1 \neq v_2$ *then* $val(v_1) \neq val(v_2)$*. We also extend the definition of val to sets of nodes and if* $V_1 \subseteq V$*, then* $val(V_1)$ *is the set defined by* $val(V_1) = \{val(v) | v \in V_1\}$*.*

For any $v \in V$*, if* $ele(v)$ *is defined then the nodes in* $ele(v)$ *are called* subelements *of* v*. For any* $l \in \mathbf{A}$*, if* $att(v, l) = v_1$ *then* v_1 *is called an* attribute *of* v*. Note that an XML tree* T *must be a tree. Since* T *is a tree the set of ancestors of a node* v*, is denoted by* $Ancestor(v)$*. The children of a node* v *are also defined as in Definition 1 and we denote the parent of a node* v *by* $Parent(v)$*.*

We note that our definition of *val* differs slightly from that in [4] since we have extended the definition of the *val* function so that it is also defined on element nodes. The reason for this is that we want to include in our definition paths that do not end at leaf nodes, and when we do this we want to compare element nodes by node identity, i.e. node equality, but when we compare attribute or text nodes we want to compare them by their contents, i.e. value equality. This point will become clearer in the examples and definitions that follow.

We now give some preliminary definitions related to paths.

Definition 2. *A* path *is an expression of the form* $l_1.\cdots.l_n$*,* $n \geq 1$*, where* $l_i \in \mathbf{E} \cup \mathbf{A} \cup \{\mathcal{S}\}$ *for all* $i, 1 \leq i \leq n$ *and* $l_1 = root$*. If* p *is the path* $l_1.\cdots.l_n$ *then* $Last(p) = l_n$*.*

For instance, if $\mathbf{E}$ = {`root, Division, Employee`} and $\mathbf{A}$ = {`D#, Emp#`} then `root, root.Division, root.Division.D#,`
`root.Division.Employee.Emp#.S` are all paths.

Definition 3. *Let* p *denote the path* $l_1.\cdots.l_n$*. The function* $Parnt(p)$ *is the path* $l_1.\cdots.l_{n-1}$*. Let* p *denote the path* $l_1.\cdots.l_n$ *and let* q *denote the path* $q_1.\cdots.q_m$*. The path* p *is said to be a* prefix *of the path* q*, denoted by* $p \subseteq q$*, if* $n \leq m$ *and* $l_1 = q_1, \ldots, l_n = q_n$*. Two paths* p *and* q *are equal, denoted by* $p = q$*, if* p *is a prefix of* q *and* q *is a prefix of* p*. The path* p *is said to be a* strict prefix *of* q*, denoted by* $p \subset q$*, if* p *is a prefix of* q *and* $p \neq q$*. We also define the intersection of two paths* p_1 *and* p_2*, denoted but* $p_1 \cap p_2$*, to be the maximal common prefix of both paths. It is clear that the intersection of two paths is also a path.*

For example, if $\mathbf{E}$ = {`root, Division, Employee`} and $\mathbf{A}$ = {`D#, Emp#`} then `root.Division` is a strict prefix of `root.Division.Employee` and `root.Division.D#` $\cap$ `root.Division.Employee.Emp#.S` = `root.Division`.

Definition 4. *A* path instance *in an XML tree* T *is a sequence* $v_1.\cdots.v_n$ *such that* $v_1 = v_r$ *and for all* $v_i, 1 < i \leq n, v_i \in V$ *and* v_i *is a child of* v_{i-1}*. A path instance* $v_1.\cdots.v_n$ *is said to be* defined over the path $l_1.\cdots.l_n$ *if for all* $v_i, 1 \leq i \leq n$*,* $lab(v_i) = l_i$*. Two path instances* $v_1.\cdots.v_n$ *and* $v'_1.\cdots.v'_n$ *are said to be* distinct *if* $v_i \neq v'_i$ *for some* i*,* $1 \leq i \leq n$*. The path instance* $v_1.\cdots.v_n$ *is*

said to be a prefix *of* $v'_1.\cdots.v'_m$ *if* $n \leq m$ *and* $v_i = v'_i$ *for all* $i, 1 \leq i \leq n$. *The path instance* $v_1.\cdots.v_n$ *is said to be a* strict prefix *of* $v'_1.\cdots.v'_m$ *if* $n < m$ *and* $v_i = v'_i$ *for all* $i, 1 \leq i \leq n$. *The set of path instances over a path* p *in a tree* T *is denoted by* $Paths(p)$

For example, in Figure 1, $v_r.v_1.v_3$ is a path instance defined over the path `root.Dept.Section` and $v_r.v_1.v_3$ is a strict prefix of $v_r.v_1.v_3.v_4$

We now assume the existence of a set of legal paths P for an XML application. Essentially, P defines the semantics of an XML application in the same way that a set of relational schema define the semantics of a relational application. P may be derived from the DTD, if one exists, or P be derived from some other source which understands the semantics of the application if no DTD exists. The advantage of assuming the existence of a set of paths, rather than a DTD, is that it allows for a greater degree of generality since having an XML tree conforming to a set of paths is much less restrictive than having it conform to a DTD. Firstly we place the following restriction on the set of paths.

Definition 5. *A set* P *of paths is* consistent *if for any path* $p \in P$, *if* $p_1 \subset p$ *then* $p_1 \in P$.

This is natural restriction on the set of paths and any set of paths that is generated from a DTD will be consistent.

We now define the notion of an XML tree conforming to a set of paths P.

Definition 6. *Let* P *be a consistent set of paths and let* T *be an XML tree. Then* T *is said to* conform *to* P *if every path instance in* T *is a path instance over some path in* P.

The next issue that arises in developing the machinery to define XFDs is the issue is that of missing information. This is addressed in [13] but in this we take the simplifying assumption that there is no missing information in XML trees. More formally, we have the following definition.

Definition 7. *Let* P *be a consistent set of paths, let* T *be an XML that conforms to* P. *Then* T *is defined to be* complete *if whenever there exist paths* p_1 *and* p_2 *in* P *such that* $p_1 \subset p_2$ *and there exists a path instance* $v_1.\cdots.v_n$ *defined over* p_1, *in* T, *then there exists a path instance* $v'_1.\cdots.v'_m$ *defined over* p_2 *in* T *such that* $v_1.\cdots.v_n$ *is a prefix of the instance* $v'_1.\cdots.v'_m$.

For example, if we take P to be {`root, root.Dept, root.Dept.Section, root.Dept.Section.Emp, root.Dept.Section.Emp.S, root.Dept.Section.Project`} then the tree in Figure 1 conforms to P and is complete.

The next function returns all the final nodes of the path instances of a path p in T.

Definition 8. *Let* P *be a consistent set of paths, let* T *be an XML tree that conforms to* P . *The function* $N(p)$, *where* $p \in P$, *is the set of nodes defined by* $N(p) = \{v | v_1.\cdots.v_n \in Paths(p) \wedge v = v_n\}$.

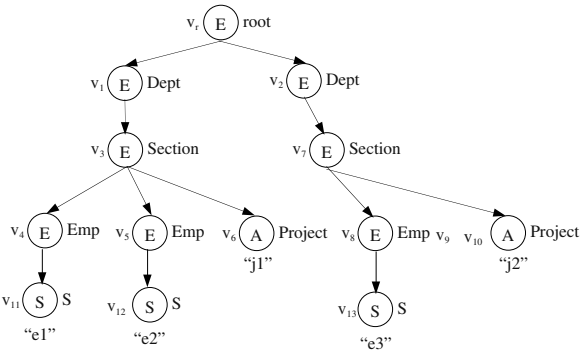


Fig. 1. A complete XML tree.

For example, in Figure 1, $N(\texttt{root.Dept}) = \{v_1, v_2\}$.

We now need to define a function that returns a node and its ancestors.

Definition 9. *Let P be a consistent set of paths, let T be an XML tree that conforms to P. The function $AAncestor(v)$, where $v \in V$, is the set of nodes in T defined by $AAncestor(v) = v \cup Ancestor(v)$.*

For example in Figure 1, $AAncestor(v_3) = \{v_r, v_1, v_3\}$. The next function returns all nodes that are the final nodes of path instances of p and are descendants of v.

Definition 10. *Let P be a consistent set of paths, let T be an XML tree that conforms to P. The function $Nodes(v, p)$, where $v \in V$ and $p \in P$, is the set of nodes in T defined by $Nodes(v, p) = \{x | x \in N(p) \wedge v \in AAncestor(x)\}$*

For example, in Figure 1 , $Nodes(v_1, \texttt{root.Dept.Section.Emp}) = \{v_4, v_5\}$. We also define a partial ordering on the set of nodes as follows.

Definition 11. *The partial ordering $>$ on the set of nodes V in an XML tree T is defined by $v_1 > v_2$ iff $v_2 \in Ancestor(v_1)$.*

3 XMVDs in XML

Before presenting the main definition of the paper, we present an example to illustrate the thinking behind the definition. Consider the relation shown in Figure 2. It satisfies the MVD $\texttt{Course} \rightarrow\rightarrow \texttt{Teacher|Text}$. The XML tree shown in Figure 3 is then a XML representation of the data in Figure 2. The tree has the following property. There exists two path instances of `root.Id.Id.Id.Text`, namely $v_r.v_{13}.v_{17}.v_{21}.v_9$ and $v_r.v_{16}.v_{20}.v_{24}.v_{12}$ such that $val(v_9) \neq val(v_{12})$. Also, these two paths have the property that for the closest `Teacher` node to v_9, namely v_5, and the closest `Teacher` node to v_{12}, namely v_8, then $val(v_5) \neq val(v_8)$ and for the closest `Course` node to both v_9 and v_5, namely v_1, and for the closest

`Course` node to both v_{12} and v_8, namely v_4, we have that $val(v_1) = val(v_4)$. Then the existence of the two path instances $v_r.v_{13}.v_{17}.v_{21}.v_9$ and $v_r.v_{16}.v_{20}.v_{24}.v_{12}$ with these properties and the fact that `Course` $\rightarrow\!\!\rightarrow$ `Teacher|Text` is satisfied in the relation in Figure 2 implies that there exists two path instances of `root.Id.Id.Id.Text`, namely $v_r.v_{15}.v_{19}.v_{23}.v_{11}$ and $v_r.v_{14}.v_{18}.v_{22}.v_{10}$, with the following properties. $val(v_{11}) = val(v_9)$ and for the closest `Teacher` node to v_{11}, v_7, $val(v_7) = val(v_8)$ and for the closest `Course` node to v_{11} and v_7, namely v_3, $val(v_3) = val(v_1)$. Also, $val(v_{10}) = val(v_{12})$ and the closest `Teacher` node to v_{10}, v_6, $val(v_6) = val(v_5)$ and for the closest `Course` node to v_{10} and v_6, namely v_2, $val(v_2) = val(v_4)$. This type of constraint is an XMVD. We note however that there are many other ways that the relation in Figure 2 could be represented in an XML tree. For instance we could also represent the relation by Figure 4 and this XML tree also satisfies the XMVD. In comparing the two representations, it is clear that the representation in Figure 4 is a more compact representation than that in Figure 3 and we shall see later that the example in Figure 4 is normalised whereas the example in Figure 3 is not.

Course	Teacher	Text
Algorithms	Fred	Text A
Algorithms	Mary	Text B
Algorithms	Fred	Text B
Algorithms	Mary	Text A

Fig. 2. A flat relation satisfying a MVD.

This leads us to the main definition of our paper. In this paper we consider the simplest case where there are only single paths on the l.h.s. and r.h.s. of the XMVD and all paths end in an attribute or text node.

Definition 12. *Let P be a consistent set of paths and let T be an XML tree that conforms to P and is complete. An XMVD is a statement of the form $p \rightarrow\!\!\rightarrow q|r$ where p, q and r are paths in P. T satisfies $p \rightarrow\!\!\rightarrow q|r$ if whenever there exists two distinct paths path instances $v_1.\cdots.v_n$ and $w_1.\cdots.w_n$ in $Paths(q)$ such that:*

(i) $val(v_n) \neq val(w_n)$;

(ii) there exists two nodes z_1, z_2, where $z_1 \in Nodes(x_{1_1}, r)$ and $z_2 \in Nodes(y_{1_1}, r)$ such that $val(z_1) \neq val(z_2)$;

(iii) there exists two nodes z_3 and z_4, where $z_3 \in Nodes(x_{1_{1_1}}, p)$ and $z_4 \in Nodes(y_{1_{1_1}}, p)$, such that $val(z_3) = val(z_4)$;

then:

(a) there exists a path $v'_1.\cdots.v'_n$ in $Paths(q)$ such that $val(v'_n) = val(v_n)$ and there exists a node z'_1 in $Nodes(x'_{1_1}, r)$ such that $val(z'_1) = val(z_2)$ and there exists a node z'_3 in $Nodes(x'_{1_{1_1}}, p)$ such that $val(z'_3) = val(z_3)$;

(b) there exists a path $w'_1.\cdots.w'_n$ in $Paths(q)$ such that $val(w'_n) = val(w_n)$ and there exists a node z'_2 in $Nodes(x'_{1_1}, r)$ such that $val(z'_2) = val(z_1)$ and there exists a node z'_4 in $Nodes(x'_{1_{1_1}}, p_l)$ such that $val(z'_4) = val(z_4)$;

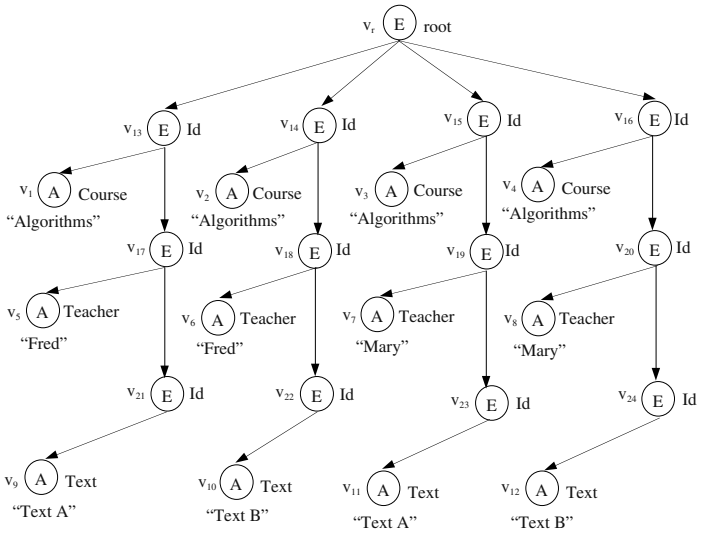


Fig. 3. An XML tree

where $x_{1_1} = \{v | v \in \{v_1, \cdots, v_n\} \wedge v \in N(r \cap q)\}$ *and* $y_{1_1} = \{v | v \in \{w_1, \cdots, w_n\} \wedge v \in N(r \cap q)\}$ *and* $x_{1_{1_1}} = \{v | v \in \{v_1, \cdots, v_n\} \wedge v \in N(p \cap r \cap q)\}$ *and* $y_{1_{1_1}} = \{v | v \in \{w_1, \cdots, w_n\} \wedge v \in N(p \cap r \cap q)\}$

and $x'_{1_1} = \{v | v \in \{v'_1, \cdots, v'_n\} \wedge v \in N(r \cap q)\}$ *and* $y'_{1_1} = \{v | v \in \{w'_1, \cdots, v'_n\} \wedge v \in N(r \cap q)\}$ *and* $x'_{1_{1_1}} = \{v | v \in \{v'_1, \cdots, v'_n\} \wedge v \in N(p \cap r \cap q)\}$ *and* $y'_{1_{1_1}} = \{v | v \in \{w'_1, \cdots, w'_n\} \wedge v \in N(p \cap r \cap q)\}$.

We note that since the path $r \cap q$ is a prefix of q, there exists only one node in $v_1.\cdots.v_n$ that is also in $N(r \cap q)$ and so x_1 is always defined and is a single node. Similarly for $y_1, x_{1_{1_1}}, y_{1_{1_1}}, x'_{1_1}, y'_{1_1}, x'_{1_{1_1}}, y'_{1_{1_1}}$. We now illustrate the definition by some examples.

Example 1. Consider the XML tree shown in Figure 4 and the XMVD `root.Id.Course` $\twoheadrightarrow$ `root.Id.Id.Teacher|root.Id.Id.Text`. Let $v_1.\cdots.v_n$ be the path instance $v_r.v_8.v_2.v_4$ and let $w_1.\cdots.w_n$ be the path instance $v_r.v_8.v_2.v_5$. Both path instances are in $Paths($`root.Id.Id.Teacher`$)$ and $val(v_4) \neq val(v_5)$. Moreover, $x_{1_1} = v_8$, $y_{1_1} = v_8$, $x_{1_{1_1}} = v_8$ and $y_{1_{1_1}} = v_8$. So if we let $z_1 = v_6$ and $z_2 = v_7$ then $z_1 \in Nodes(x_{1_1},$`root.Id.Id.Text`$)$ and

$z_2 \in Nodes(y_{1_1},$`root.Id.Id.Text`$)$. Also if we let $z_3 = v_1$ and $z_4 = v_1$ then $z_3 \in Nodes(x_{1_{1_1}},$`root.Id.Course`$)$ and $z_4 \in Nodes(y_{1_{1_1}},$`root.Id.Course`$)$ then $val(z_3) = val(z_4)$. Hence conditions (i), (ii) and (iii) of the definition of an XMVD are satisfied.

If we let $v'^i_1.\cdots.v'^i_n$ be the path $v_r.v_8.v_2.v_4$ we firstly have that $val(v'^i_n) = val(v^i_n)$ as required. Also, since the path instances are the same we have that $x_{1_1} = x'_{1_1}$ and $x_{1_{1_1}} = x'_{1_{1_1}}$. So if we let $z'_1 = v_7$ then

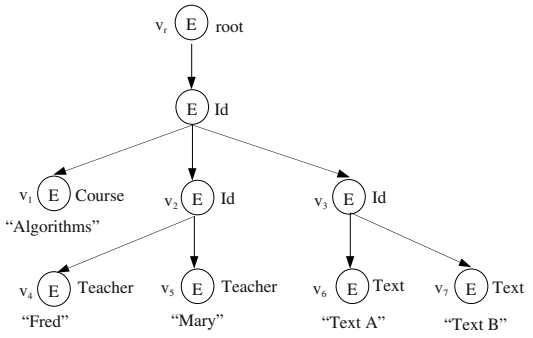


Fig. 4. An XML tree

$z'_1 \in Nodes(x'_{1_1}, \texttt{root.Id.Id.Text})$ and $val(z'_1) = val(z_2)$ and if we let $z'_3 = v_1$ then $z'_3 \in Nodes(x'_{1_{1_l}}, \texttt{root.Id.Course})$ and $val(z'_3) = val(z_3)$. So part (a) of the definition of an XMVD is satisfied. Next if we let $w'^i_1.\cdots.w'^i_n$ be the path $v_r.v_8.v_2.v_5$ then we firstly have that $val(w'^i_n) = val(w^i_n)$ since the paths are the same . Also, since the paths are the same we have that $y_{1_1} = y'_{1_1}$ and $y_{1_{1_1}} = y'_{1_{1_1}}$. So if we let $z'_2 = v_6$ then $z'_2 \in Nodes(y'_{1_1}, \texttt{root.Id.Id.Text})$ and $val(z'_2) = val(z_1)$ and if we let $z'_4 = v_1$ then $z'_4 \in Nodes(x'_{1_{1_l}}, \texttt{root.Id.Course})$ and $val(z'_4) = val(z_4)$. Hence part (b) on the definition of an XMVD is satisfied and so T satisfies $\texttt{root.Id.Course} \rightarrow\rightarrow \texttt{root.Id.Id.Teacher|root.Id.Id.Text}$.

As explained earlier, the tree in Figure 4 also satisfies
$\texttt{root.Id.Course} \rightarrow\rightarrow \texttt{root.Id.Id.Teacher|root.Id.Id.Text}$.

Example 2. Consider the XML tree shown in Figure 5 and the XMVD $\texttt{root.Project.P#} \rightarrow\rightarrow \texttt{Root.Project.Person.Name|root.Project.Part.Pid}$. For the path instances $v_r.v_1.v_5.v_{13}$ and $v_r.v_2.v_8.v_{16}$ in

$Paths(\texttt{Root.Project.Person.Name})$ we have that $val(v_{13}) \neq val(v_{16})$. Moreover, $x_{1_1} = v_1$, $y_{1_1} = v_2$, $x_{1_{1_1}} = v_1$ and $y_{1_{1_1}} = v_2$. So if we let $z_1 = v_{15}$ and $z_2 = v_{18}$ then $z_1 \in Nodes(x_{1_1}, \texttt{root.Project.Part.Pid})$ and $z_2 \in Nodes(y_{1_1},$

$\texttt{root.Project.Part.Pid})$. Also if we let $z_3 = v_4$ and $z_4 = v_7$ then $z_3 \in Nodes(x_{1_{1_1}}, \texttt{root.Project.P\#})$ and $z_4 \in Nodes(y_{1_{1_1}}, \texttt{root.Project.P\#}\ )$ and $val(z_3) = val(z_4)$. Hence conditions (i), (ii) and (iii) of the definition of an XMVD are satisfied. However, for the only other path in

$Paths(\texttt{Root.Project.Person.Name})$, namely $v_r.v_3.v_{11}.v_{19}$ we have that $x'_{1_1} = v_3$ and so $Nodes(x'_{1_1}, \texttt{root.Project.part.Pid}) = v_{21}$ and since $val(v_{21}) \neq val(z_2)$ and so it does not satisfy condition (a) and thus
$\texttt{root.Project.P#} \rightarrow\rightarrow \texttt{Root.Project.Person.Name|root.Project.part.Pid}$ is violated in T.

Consider then the XMVD $\texttt{root.Project.Person.Name}$

$\rightarrow\rightarrow$ $\texttt{Root.Project.Person.Skill}$ $\texttt{|root.Project.P\#}$ in the same XML tree. For the path instances $v_r.v_1.v_5.v_{14}$ and $v_r.v_3.v_{11}.v_{20}$ in

$Paths$(`Root.Project. Person.Skill`) we have that $val(v_{14}) \neq val(v_{20})$. Moreover, $x_{1_1} = v_1$, $y_{1_1} = v_3$, $x_{1_{1_1}} = v_1$ and $y_{1_{1_1}} = v_3$. So if we let $z_1 = v_4$ and $z_2 = v_{10}$ then $z_1 \in Nodes(x_{1_1}$,`root.Project.P#`) and $z_2 \in Nodes(y_{1_1}$, `root.Project.P#`). Also if we let $z_3 = v_{13}$ and $z_4 = v_{19}$ then $z_3 \in Nodes(x_{1_{1_1}}$, `root.Project.Person.Name`) and $z_4 \in Nodes(y_{1_{1_1}}$, `root.Project.Person.Name`) and $val(z_3) = val(z_4)$. Hence the conditions of (i), (ii) and (iii) of the definition of an XMVD are satisfied. However there does not exist another path instance in
$Paths$(`Root.Project.Person.Skill`) such that val of the last node in the path is equal to val of node v_{14} and so part (a) of the definition of an XMVD is violated.

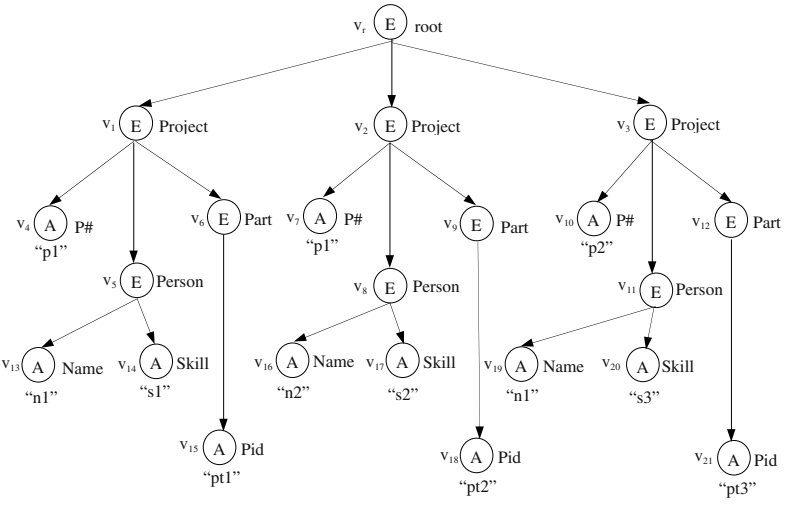


Fig. 5. An XML tree

4 A Redundancy Free 4NF for XML Documents

In this section we propose a 4NF for XML documents. We also provide a formal justification for the normal form by showing that it ensures the elimination of redundancy in the presence of XMVDs. This approach to justifying the definition of a normal form is an extension of the approach adopted by one of the authors in some other research which investigated the issue of providing justification for the normal forms defined in standard relational databases [10,11,12,8].

The approach that we use to justifying our normal form is to formalise the notion of redundancy, the most intuitive approach to justifying normal forms, and then to try to ensure that our normal form ensures there is no redundancy.

However, defining redundancy is not quite so straightforward as might first appear. The most obvious approach is, given a relation r and a FD $A \rightarrow B$ and two tuples t_1 and t_2, to define a value $t_1[B]$ to be redundant if $t_1[B] = t_2[B]$ and $t_1[A] = t_2[A]$. While this definition is fine for FDs in relations, it doesn't generalise in an obvious way to other classes of relational integrity constraints, such as *multi-valued dependencies* (MVDs) or *join dependencies* (JDs) or *inclusion dependencies* (INDs), nor to other data models. The key to finding the appropriate generalisation is based on the observation that if a value $t_1[B]$ is redundant in the sense just defined then every change of $t_1[B]$ to a new value results in the violation of $A \rightarrow B$. One can then define a data value to be redundant if every change of it to a new value results in the violation of the set of constraints (whatever they may be). This is essentially the definition proposed in [12] where it was shown that BCNF is a necessary and sufficient condition for there to be no redundancy in the case of FD constraints and fourth normal form (4NF) is a necessary and sufficient condition for there to be no redundancy in the case of FD and MVD constraints.

The definition we propose is the following which is an extension of the definition given in [12].

Definition 13. *Let T be an XML tree and let v be a node in T. Then the change from v to v', resulting in a new tree T', is said to be a* valid change *if $v \neq v'$ and $val(v) \neq val(v')$.*

We note that the second condition in the definition, $val(v) \neq val(v')$, is automatically satisfied if the first condition is satisfied when $lab(v) \in \mathbf{E}$.

Definition 14. *Let P be a consistent set of paths and let Σ be a set of XMVDs such that every path appearing in an XMVD in Σ is in P. Then Σ is said to* cause redundancy *if there exists a complete XML tree T which conforms to P and satisfies Σ and a node v in T such that* every *valid change from v to v', resulting in a new XML tree T', causes Σ to be violated.*

The essential idea is that if a value is redundant, then it is implied by the other data values and the set of constraints and so any change to the value causes a violation of the constraints. For example, consider Figure 3 and the set Σ of XMVDs

$\{$`root.Id.Course` $\twoheadrightarrow$ `root.Id.Id.Teacher|root.Id.Id.Id.Text`$\}$. Then Σ causes redundancy because the tree shown in Figure 3 satisfies Σ yet every valid change to any of the `Text` nodes (or `Teacher` nodes) results in the violation of Σ.

Next, we define the notion of a key.

Definition 15. *Let P be a consistent set of paths, let T be an XML tree that conforms to P and is complete and let $p \in P$. Then T satisfies the key constraint p if whenever there exists two nodes v_1 and v_2 in $N(p)$ in T such that $val(v_1) = val(v_2)$ then $v_1 = v_2$.*

We note that since node indentifiers in XML trees are unique, it follows that if $Last(p) \in \mathbf{E}$ then p is automatically a key. Next, we define a normal form for XML.

Definition 16. *Let Σ be a set XMVDs and key constraints. Then Σ is in* XML fourth normal form *(4XNF) if for every XMVD $p \rightarrow\rightarrow q|r \in \Sigma$, at least one of the following conditions holds:*

(A) q and r are both keys;

(B) p is a key and $q \cap r = p$;

(C) p is a key and $q \cap r$ is a strict prefix of p;

(D) $q \cap r = root$;

(E) there exists an XMVD $s \rightarrow\rightarrow t|u \in \Sigma$ such that $s \cap p = root$ and $t \geq q \cap r$ and t is a key and $u \geq q \cap r$ and u is a key;

(F) there exists an XMVD $a \rightarrow\rightarrow b|c \in \Sigma$ such that $a \cap p = root$ and b is not a key and c is a key and $b \cap c \cap p \neq root$ and there exists $b \rightarrow\rightarrow d|e$ such that d is a key and e is a key and $d \geq q \cap r$ and $e \geq q \cap r$;

(G) q is not a key and r is not a key and there exists $p \rightarrow\rightarrow q|k$ and there exists $p \rightarrow\rightarrow k|r$ such that $k \geq q \cap r$;

(H) p is a key, q is a key and r is not a key and $q \cap r \neq p$ and $q \cap r$ is not a strict prefix of p and there exists $x \rightarrow\rightarrow q|k$ such that $x < p$ and $k > p \cap q \cap r$ and $k \cap q \cap r \neq p \cap q \cap r$;

(I) p is a key, q is not a key and r is a key and $q \cap r \neq p$ and $q \cap r$ is not a strict prefix of p and $\exists x \rightarrow\rightarrow k|r$ such that $x < p$ and $k > p \cap q \cap r$ and $k \cap q \cap r \neq p \cap q \cap r$.

We now illustrate the definition by an example.

Example 3. Consider the tree T in Figure 3 and assume that the only constraint is the XMVD

`root.Id.Course` $\rightarrow\rightarrow$ `root.Id.Id.Teacher|root.Id.Id.Id.Text`. T satisfies Σ and is complete. However Σ is not in 4XNF since `root.Id.Course` is not a key and `root.Id.Id.Teacher` is not a key and `root.Id.Id.Id.Text` is not a key. Consider then the tree shown in Figure 4 and assume that the only XMVD is `root.Id.Course` $\rightarrow\rightarrow$ `root.Id.Teacher|root.Id.Text`. If `root.Id.Course` is a key, which would be the case if `Course` was specified as type `ID` in the full DTD, then Σ is in 4XNF since `root.Id.Course` is a key and `root.Id.Teacher` $\cap$ `root.Id.Id.Id.Text` = `root.Id` and so (C) of the condition for 4XNF is satisfied since `root.Id` is a strict prefix of `root.Id.Course`.

This leads to the main result of the paper.

Theorem 1. *Let Σ be a set of XMVDS and key constraints. If Σ is in 4XNF then it does not cause redundancy.*

5 Conclusions

In this paper we have investigated the issues of XMVDs and 4NF in XML. We proposed a normal form for XML documents in the presence of XMVDs and justified it by showing that it ensures the elimination of redundancy. This extended the results in [15] which defined a normal form for a restricted class of XMVDs called hierarchical XMVDs.

There are several other issues related to the ones addressed in this paper that warrant further investigation. The first is the need to generalise the main result of this paper. We need to show that 4XNF we proposed is also a necessary condition for the elimination of redundancy. Secondly, we need to investigate the problem of developing an axiom system for reasoning about the implication of XMVDs. In [13] an axiom system for reasoning about the implication of XFDs was developed and the system was shown to be sound for arbitrary XFDs. Later [17], the implication problem for XFDs was shown to be decidable and the axiom system presented in [13] was shown to be complete for unary XFDs and a polynomial time algorithm was developed for determining if a unary XFD is implied by a set of unary XFDs. Similarly, we need to develop an axiom system and algorithm for the implication problem for XMVDs. Thirdly, we need to develop algorithms for converting unnormalised XML documents to normalised ones. In the relational case, the equivalent procedure is performed using a decomposition algorithm based on the projection operator. However, at the moment there has been no commonly agreed upon algebra defined for XML, let alone a projection operator, so the development of procedures for normalising XML documents is likely to be more complex than in the relational case. Fourthly, it is necessary to consider the case where both XFDs and XMVDs exist in a document. It is interesting to note that unlike the situation for the relational case, 4XNF is not a straightforward generalisation of the normal form for XFDs (XNF). This means that, in contrast to the relational case where 4NF implies BCNF in the case where both MVDs and FDs are present, in XML a different normal form from 4XNF is needed when the constraints on an XML document contain both XFDs and XMVDs. The situation is further complicated by the fact that XMVDs and XFDs interact, in the sense that there are XMVDs and XFDs implied by a combined set of XMVDs and XFDs which are not implied by either the XMVDs or XFDs considered alone. This situation parallels that of relational databases [2].

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web.* Morgan Kauffman, 2000.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases.* Addison WAesley, 1996.
3. P. Buneman, S. Davidson, W. Fan, and C. Hara. Reasoning about keys for xml. In *International Workshop on Database Programming Languages*, 2001.
4. P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Keys for xml. *Computer Networks*, 39(5):473–487, 2002.

5. P. Buneman, W. Fan, J. Simeon, and S. Weinstein. Constraints for semistructured data and xml. *ACM SIGMOD Record*, 30(1):45–47, 2001.
6. P. Buneman, W. Fan, and S. Weinstein. Path constraints on structured and semistructured data. In *Proc. ACM PODS Conference*, pages 129–138, 1998.
7. W. Fan and J. Simeon. Integrity constraints for xml. In *Proc. ACM PODS Conference*, pages 23–34, 2000.
8. M. Levene and M. W. Vincent. Justification for inclusion dependency normal form. *IEEE Transactions on Knowledge and Data Engineering*, 12:281–291, 2000.
9. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton: Relational databases for querying xml documents: Limitations and opportunities. In *VLDB Conference*, pages 302–314, 1999.
10. M. W. Vincent. A corrected 5nf definition for relational database design. *Theoretical Computer Science*, 185:379–391, 1997.
11. M. W. Vincent. A new redundancy free normal form for relational database design. In B. Thalheim and L. Libkin, editors, *Database Semantics*, pages 247 –264. Springer Verlag, 1998.
12. M. W. Vincent. Semantic foundations of 4nf in relational database design. *Acta Informatica*, 36:1–41, 1999.
13. M.W. Vincent and J. Liu. Strong functional dependencies and a redundancy free normal form for xml. Submitted to ACM Transactions on Database Systems, 2002.
14. M.W. Vincent and J. Liu. Functional dependencies for xml. In *Fifth Asian Pacific Web Conference*, 2003.
15. M.W. Vincent and J. Liu. Multivalued dependencies and a 4nf for xml. In *15th International Conference on Advanced Information Systems Engineering (CAISE)*, 2003.
16. M.W. Vincent and J. Liu. Multivalued dependencies in xml. In *20th British National Conference on Databases (BNCOD)*, 2003.
17. M.W. Vincent, J. Liu, and C. Liu. The implication problem for unary functional dependencies in xml. In *submitted for publication*, 2003.
18. J. Widom. Data management for xml - research directions. *IEEE data Engineering Bulletin*, 22(3):44–52, 1999.

Supporting XML Security Models Using Relational Databases: A Vision

Dongwon Lee, Wang-Chien Lee, and Peng Liu

Penn State University
dongwon@psu.edu, wlee@cse.psu.edu, pliu@ist.psu.edu

Abstract. As the *secure* distribution and sharing of information over the World Wide Web becomes increasingly important, the needs for flexible and efficient support of access control systems naturally arise. Since the eXtensible Markup Language (XML) is emerging as the format of the Internet era for storing and exchanging information, there have been, recently, many proposals to extend the XML model to incorporate security aspects. To the lesser or greater extent, however, such proposals neglect the fact that the data for XML documents will most likely reside in relational databases, and consequently do not utilize various security models proposed for and implemented in relational databases.
In this paper, we take a rather different approach. We explore how to support security models for XML documents by leveraging on techniques developed for relational databases. More specifically, in our approach, (1) Users make XML queries against the given XML view/schema, (2) Access controls for XML data are also specified in the XML model, but (3) Data are stored in relational databases, and (4) Security check and query evaluation are also done in relational databases. Instead of re-inventing wheels, we take two representative methods in both XML security model and XML to relational conversion problems, and show how to glue them together in a seamless manner to efficiently support access controls for the XML model using relational databases.

1 Introduction

Since the eXtensible Markup Language (XML) was invented for exchanging and storing information over the World Wide Web (Web) [4], its usage has exploded significantly. As more information is exchanged and processed over the Web, the issues of security become increasingly important. Such issues are diverse, spanning from data level security using cryptography to network transport level security to high-level access controls. In this paper, our focus is on how to support high-level *access controls* for XML documents.

Table 1 illustrates the current development of XML security models. First row refers to (research-oriented) security models developed for XML and relational models, respectively, while second row refers to database products for each model. In general, not all features proposed by research in the first row are implemented in the real implementations in the second row yet. For instance,

Z. Bellahsène et al. (Eds.): XSym 2003, LNCS 2824, pp. 267–281, 2003.

Table 1. The overview of XML and Relational security model supports.

XML	**Relational**	
XML Security Models ([6], [2], etc)	Relational Security Models ([13], etc)	**Models**
XML Databases (Xindice, Tamino, etc)	Relational Databases (Oracle, DB2, SQL Server, etc)	**Products**

most XML database products currently do not have any support for access controls. Similarly, commercial relational products have implemented only minimal features of access controls via authorizations such as `GRANT` and `REVOKE`.

Recently, many access control methods extending the XML model to incorporate security aspects have been proposed (e.g., XACML [11], [6], [2], [23]). To the lesser or greater extent, however, such proposals neglect the fact that the most data for XML documents still reside in relational databases behind the scenes, and consequently do not utilize various security models that have been proposed for and implemented in relational databases. We believe that current XML security research (i.e., upper-left column of Table 1) is re-inventing wheels without utilizing existing relational security models (i.e., upper-right column of Table 1) or security features that are already implemented and being used in relational products (i.e., lower-right column of Table 1).

Therefore, our goal in this research is to study how to support XML security models by utilizing existing security support of relational security models or relational products. More specifically, we assume that

- XML documents are converted into and stored in relational databases.
- Users are given an XML view/schema against which they issue XML queries.
- Access controls are specified by security administrators in the XML schema and documents.
- Security check and query evaluation are done by relational databases (or by a middleware on top of relational databases), and only valid answers are returned to users in the XML format.

In this paper, as a preliminary work, we explore various research issues and a few possible sketches of solutions to achieve the vision of supporting XML security models using relational databases. Furthermore, we present an illustrative example that shows a complete steps of the vision as a proof of concept. We hope to draw more research interests and efforts onto the direction that we are proposing in this paper.

2 Related Work

Since our research relates to two seemingly unrelated works, we first survey those works in two separate categories, and then discuss a few works that are overall similar/dissimilar to our proposal.

2.1 XML and Relational Security Models

XML access control models. Several authorization-based XML access control models are proposed. In [19], authorizations are specified on portions of a HTML document, however, no semantic context similar to that provided by XML can be supported. In [7], a specific authorization sheet is associated with each XML document/DTD expressing the authorizations on the document. In [6], the model proposed in [7] is extended by enriching the authorization types supported by the model, providing a complete description of the specification and enforcement mechanism. Among comparable proposals, in [2], an access control environment for XML documents and some techniques to deal with authorization priorities and conflict resolution issues are proposed. Finally, the use of authorization priorities with propagation and overriding, which is an important aspect of XML access control may recall approaches in the context of object-oriented databases, like [9] and [18]. Although our proposal is based on existing XML authorization models such as [6], we focus on how to use relational databases to help enforce XML authorization models, and none of the above XML authorization models address the interaction between XML and relational access controls.

Relational access control models. Relational access control models can be classified into two categories: *multilevel security models* [15,24,20] and *discretionary security models*. Multilevel security models assign each data object (e.g., a tuple) as well as each subject (e.g., a user) a security *level* (or class), and enforce the following two specific access control rules: (1) a level L_i subject can never read a level L_j data object unless $L_i \geq L_j$; (2) a level L_i subject can never write a level L_j data object unless $L_j \geq L_i$. Although multilevel security models are widely used in military applications, they are seldom used in commercial applications for their restrictive nature. By contrast, discretionary security models allow the creator of a data object x to own all the privileges associated with x and to grant some of the privileges to other users in such a way that a variety of access control policies could be enforced. Discretionary security models are dominant in commercial data management. Although several more expressive and flexible discretionary security models are proposed [13, 14], most real world database systems implement a discretionary access control model similar to the one implemented in System R [12], where (1) access control is supported by the GRANT and REVOKE commands; (2) only table or column level authorizations are directly supported; (3) views are used to indirectly support some data dependent access control. Nevertheless, role-based access control [21] is not implemented in System R but implemented by most existing DBMSs such as Oracle. It is clear that our XML-relational access control scheme cannot be directly supported by table or column level authorizations, since each XML path is usually stored in a set of tuples. Although views can be used to support XPath-oriented access control, they are expensive and difficult to manage.

2.2 Conversion Methods between XML and Relational Models

Toward conversion between XML and relational models, an array of research has addressed the particular issues lately. On the commercial side, database vendors are busily extending their databases to adopt XML types. Typically, they can handle XML data using BLOB/CLOB formats along with a limited keyword searching or using some object-relational features [5,1], but not many details have been revealed. On the research side, various proposals have been made recently. Here, we only survey two kinds of works related to XML-to-relational conversion – *structure-based* and *data-based* conversions. The former generates a target relational schema from the given XML schema as a source, while the latter uses XML documents directly to generate relational tuples. Since the data-based conversion methods do not require an XML schema as input, the methods work for arbitrary XML schema. However, it cannot capture semantics that appear in XML schema, but hidden in XML documents.

Structure-based conversions. Work done in STORED [8] is one of the first significant and concrete attempts to this end and deals with non-valid XML documents. STORED uses a data mining technique to find a representative DTD whose support exceeds the pre-defined threshold and convert XML documents to relational format using the DTD. [3] discusses template language-based conversion from DTD to relational schema which requires human experts to write an XML-based conversion rule. [22] presents three inlining algorithms that focus on the table level of the schema conversions. On the contrary, [16] proposes a method where the hidden semantic constraints in DTD are systematically found and translated into relational formats. Since the method is orthogonal to the structure-oriented conversion methods, it can be used along with algorithms [8, 3,22,10] with little change. [17] proposes an algorithm for mapping a DTD to the Entity-Relationship (ER) model (and thus the relational model) and examine some of the issues in loading XML data into the generated model.

Data-based conversions. [10] studies different performance issues among eight algorithms that focus on the attribute and value level of the schema. [25] proposes a DTD-independent, path-based mapping algorithm. While ignoring specific characteristics hidden in each DTD, [25] decomposes XML documents into element, attribute, text and path tables, so that the changes of DTDs of the XML documents do not necessarily result in invalid mapping as found in examples [8,22].

2.3 Supporting XML Security Models Using Relational Databases

To our best knowledge, the only work that is directly related to our proposal is [23]. [23] also proposes an idea of using RDBMS to handle access controls for XML documents, in a rather limited setting. According to our taxonomy of Table 8, [23] roughly uses the following methods: model-to-RDBMS conversion, schema-level XML security model, structure-based data conversion, and external

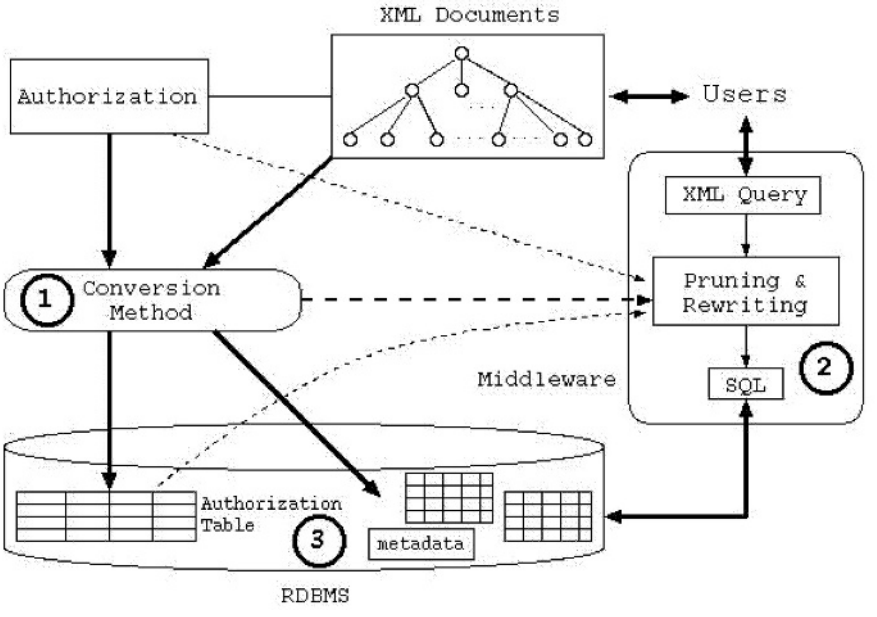


Fig. 1. The framework for supporting XML security models using relational databases.

and pre-pruning security check. In this research, however, we aim at conducting a much more extensive and systematic research than [23].

3 Framework and Research Issues

In this section, we discuss various research issues that arise in supporting XML security models using relational databases. Figure 1 illustrates an overall framework for our vision. From the users' perspective, they are accessing data in a given set of XML documents defined by associated DTDs or XML schema. The access controls of the data are governed by authorization rules specified in accordance with a given XML security model.

Behind the scene, the XML documents and the associated authorization rules are actually preprocessed and converted into relational tables, metadata, and authorization tables in relational databases. This process is illustrated by circle (1) in the figure. To access the XML data, a user submits an XML query against the known XML schema. Based on the authorization rules, a user's query is first pruned and rewritten into SQL. This process is illustrated by circle (2) in the figure. Please note that, in our proposal, the pruning process here can be adapted to generate various execution plans. Thus, access control can be enforced purely based on security pruning, internal security mechanism of relational databases, or a mixture of these two. Thus, when the SQL query is processed in the relational

databases, authorization table may be checked (as illustrated in (3)) before the valid answers are returned to users in the XML format.

3.1 XML to Relational Conversion

As shown in Figure 1, two of the initial but critical tasks for secure storage and access of XML data in relational databases are to (1) map XML authorization rules into the existing access control mechanism in relational databases; and (2) map XML documents into tables in relational databases. In the following, we discuss some of the issues.

1. **Theoretical study of XML and Relational security models.** To fully realize our vision, a thorough study on the expressive power of XML and relational security models must be done. Since there is not a unified security model for both models, nor a single standard agreed upon in community, one must first understand the pros and cons of different security models (e.g., security propagation, conflict resolution, etc) and their relationships among others. Furthermore, a finding of theoretical mapping (i.e., complete and sound algorithms) from the source XML security model to the target relational security model would be challenging tasks.

2. **Schema-level vs. Instance-level.** When XML security authorizations are specified based on XML schema, it is called a *schema-level* access control. When an XML document element or attribute can carry additional tag, specifying security information that can overwrite access control specified in the XML schema, it is called as *instance-level* access control. Depending on which scheme is used in the XML security model, how to convert such authorization rules into relational format becomes challenging. This issue also affects security evaluation strategy significantly.

3. **Which XML-to-relational conversion method is appropriate to use?** Recently many conversion algorithms have been proposed (e.g., [8,22, 10,25]), each of which has different pros and cons for different applications. When the schema-level access control is specified, for instance, we believe the path-based conversion methods such as XRel [25] are good candidates. This is because in XML model, using XPath to specify the scope of the objects is quite natural. For instance, a statement "*manager* has `read` and `write` accesses for `//employees/salary`" indicates that a subject (i.e., manager) can read and write all objects (i.e., elements) as long as the objects are `salary` under `employee` element. When path-based XML-to-relational conversion methods are used, such XPath-based security scope can be easily captured into single authorization table in relational databases. If other structure-oriented conversion methods such as hybrid inlining [22] are used for conversion, then authorization information in the XML model would be scattered into several table in relational databases, making difficult or inefficient to do security evaluation. However, in general, a question of which XML-to-relational conversion algorithms suits best for the given application is non-trivial to answer.

3.2 Security Evaluation

In this context, we explore three dimensions – *where*, *how*, and *when* to evaluate and enforce security information.

1. **Where to evaluate?** Incoming queries are in XML format such as XPath, while query processing is done by relational databases. Therefore, at some point, input queries must be rewritten to SQL format. One extreme to support security evaluation is to do all the necessary security check outside of relational databases, while the other extreme is to push all security check down to database engine, utilizing built-in features in relational databases. Let us consider three strategies for instance. (1) In the *external evaluation*, users' XML query is compared against authorization rules from both XML and relational models and pruned such that only query nodes that are valid against the given authorization rules remain at the end. Since pruning stage guarantees only valid data access, relational database can do normal SQL query processing, without worrying about insecure access. (2) In the *internal evaluation*, authorization rules are first converted and stored in relational databases via some conversion method, and security check is conducted inside relational databases. This approach explores features such as GRANT, REVOKE or view implemented in relational databases. (3) For some cases, it might be better to combine two extremes (i.e., external and internal security evaluation approaches) to strike a balance, thus *hybrid evaluation*. It is not clear how to split such a task – what part of security check is best done externally, and what is best done inside of relational databases?

2. **How to evaluate?** Second dimension of security evaluation is *how* to enforce the security. In the *strict evaluation* approach, no data protected by authorization rules can be accessed or returned. However, in the *relaxed evaluation* approach, during query processing, any data can be accessed, but only secure data must be returned to users. Interesting question is, then, when would be such a relaxed evaluation useful? Some XML data might be tagged as "accessible during query processing", but not "returnable". In such a situation, a question of if one can utilize such a relaxed evaluation for faster and secure query processing is interesting.

3. **When to evaluate?** Last dimension of security evaluation is *when* to evaluate. In the most primitive *post-processing* approach, XML query is processed like a normal query, then at the end, a portion of answers that violate security check is pruned and the remains are returned to user. However, one can think of other approaches such as *pre-pruning* (e.g., [6]) or *interleaving*. Investigating when one approach is better than another and, if so, in what situation would be a challenging task.

3.3 Authorization in Relational Databases

In representing authorizations in relational databases, the typical method is to use the *authorization table* that essentially contains a tuples of subject/object

pairs and its allowed actions. Although varied, most authorization table schemes restrict the granularity of access control for relational model to either table-level or column-level. On the contrary, in XML security model, the finer access control is possible (e.g., nodel-level). Therefore, due to this differences of granularities of two models, problem occurs. Consider the following DTD, for instance:

```
<!ELEMENT A (B+, C, X)>
<!ELEMENT B (D, E)>
<!ELEMENT C (#PCDATA)>
<!ELEMENT D (#PCDATA)>
<!ELEMENT E (#PCDATA)>
<!ELEMENT X (#PCDATA)>
```

If one used the hybrid inlining method [22] for XML-to-relational conversion, one essentially would have the following two tables generated, `A(C,X)`, and `B(D,E,fk_A)`, where `fk_A` is the foreign key referencing the primary key of the table `A`. Now, consider the following four authorization rules:

```
A1:  (Admin, /A/B, read, +)
A2:  (Admin, /A/C, read, +)
A3:  (Admin, /A/B[./D>5], read, +)
A4:  (Admin, /A/B[./D>5]/E, read, +)
```

For instance, the first rule `A1` states that the subject (i.e., Admin) can read all nodes `B` under `A`, and the fourth rule `A4` states that the subject can read all nodes `E` under `B` under `A` as long as `B` has a child node `D` whose value is greater than 5. Figure 2 illustrates pictorial representation of the scope of objects covered by each authorization rule. For instance, the rule `A1` covers all the nodes `B` of XML document. According to the hybrid inlining conversion, all the information related to the nodes `B` would be stored into the table `B(D,E,fk_A)`. Therefore, by applying the *table-level* SQL GRANT statement shown below, one can achieve the same security enforcement as dictated in the `A1`.

```
GRANT SELECT TO USER Admin ON B;
```

Similarly, the rule `A2` can be enforced by *column-level* GRANT statement as follows:

```
GRANT SELECT TO USER Admin ON A(C);
```

However, the rules `A3` and `A4` cannot be enforced using GRANT statement since most relational databases do not have *tuple-level* or *cell-level* authorization yet. One may support such an authorization by first creating a view and then issue a GRANT statement to the created view. For instance, the following statements enforce the rule `A3`.

```
CREATE VIEW tmp AS SELECT * FROM B WHERE D>5;
GRANT SELECT TO USER Admin ON tmp;
```

Fig. 2. Granularity discrepancy. Dark area refers to the objects being covered by authorization rules. (a) `A1` covers the complete information of the table `B`, (b) `A2` covers the attribute `C` of the table `A`, (c) `A3` covers all the attributes but only tuples satisfying `D>5`, and (d) `A4` covers the attribute `E` of the table `B` but only tuples satisfying `D>5`.

The rule `A4` can be supported similarly. Although finer-grained access controls of XML security models (e.g., tuple-level or cell-level) can be supported using views in relational databases, this scheme shares the same limitations of handling a large number of views – difficulty or inefficiency of maintenance and update. Therefore, it is not entirely clear how to efficiently support such finer-grained access controls in relational databases.

4 Illustrative Example

In this section, we will go over an example of XML document with authorization rules and illustrate how that might be supported using one of the conversion methods. Tables 2 and 3 are the DTD for `AllDepts` and an XML document conforming to the DTD. Furthermore, Table 4 contains several authorization rules in the XML security model proposed by [6]. Note that the scope of the authorization rules is specified by XPath expressions, and thus node-level fine-grained access control is allowed in that model. In that table, sign "-" and "+" mean the action is "prohibited" and "allowed", respectively, and type "R" and "L" mean the propagation of the rule is "recursive" and "local", respectively. That is, the recursive propagation implies the rule applies to nodes N specified by the XPath expression as well as all the descendants of N. In the local propagation, the authorization rule applies to only nodes specified by the XPath expression. For instance, the rule `R1` states that "No public read/write is allowed for the attribute `dname` under the element `AllDepts`, as a child or descendant." Also, the rule `R4` states that "Manager can read/write the `Budget` element under the `Proj` element when the `Proj` has a `private` as a value for the attribute `type`."

Now, suppose the XML document in Table 3 is converted and stored in relational databases using XRel method [25] as shown in Table 5. Note that what is shown in Table 5 is a simplified version of the original XRel algorithm for simplicity. In XRel, all root-to-leaf paths of XML document, where leaf is either attribute or element, are assigned a unique ID (i.e., pathID) and stored in the path table (d). Then, each node (i.e., text string, attribute value, and element) of the XML document is captured in the appropriate table separately (i.e., text table (b), attribute table(a), and element table (c)).. For instance,

Table 2. XML side: A DTD for `AllDepts`.

```
<!DOCTYPE AllDepts [
  <!ELEMENT Dept     (Manager,Staff+,Proj*)>
  <!ATTLIST Dept      dname    ID           #REQUIRED>
  <!ELEMENT Manager  (Name,Addr,Salary)>
  <!ATTLIST Manager   eid      ID           #REQUIRED>
  <!ELEMENT Staff    (Name,Addr,Salary)>
  <!ATTLIST Staff     eid      ID           #REQUIRED>
  <!ELEMENT Proj     (Year,Budget)>
  <!ATTLIST Proj      pname    ID           #REQUIRED
                      type   (public|private) #IMPLIED>
  <!ELEMENT Year     (#PCDATA)>
  <!ELEMENT Budget   (#PCDATA)>
]>
```

Table 3. XML side: An XML document for `AllDepts`. Note that `[XX]` in front of elements, attributes, or text string below is not part of the XML document, but a node ID number added for the discussion of this paper.

```
<[1]AllDepts>
  <[2]Dept [3]dname='CS'>
    <[4]Manager [5]eid='m10'>
      <[6]Name>[7]Tom</Name> <[8]Addr>[9]110 Foster Ave.</Addr>
      <[10]Salary>[11]70K</Salary>
    </Manager>
    <[12]Staff [13]eid='e10'>
      <[14]Name>[15]Jane</Name> <[16]Addr>[17]54 Union St.</Addr>
      <[18]Salary>[19]45K</Salary>
    </Staff>
    <[20]Proj [21]pname='XML' [22]type='public'>
      <[23]Year>[24]2003</Year>  <[25]Budget>[26]100K</Budget>
    </Proj>
    <[27]Proj [28]pname='Stream' [29]type='private'>
      <[30]Year>[31]2002</Year>  <[32]Budget>[33]300K</Budget>
    </Proj>
  </Dept>
</AllDepts>
```

Table 4. XML side: Authorization rules for `AllDepts`.

No.	Subject	Object	Action	Sign	Type
R1	Public	`/AllDepts/*/@dname`	read,write	-	L
R2	Public	`//Dept/*/Name`	read	+	L
R3	Manager	`//Dept/Staff`	read	+	R
R4	Manager	`//Dept/Proj[./@type='private']/Budget`	read,write	+	L
R5	Staff,Manager	`//Dept/Proj[./@type='public']/Budget`	read	+	L

the node with nodeID=4 (i.e., `Manager`) is captured in the third tuple of the element table (c) using the `/AllDepts/Dept/Manager` path. Similarly, the node with nodeID=17 (i.e., `54 Union St.`) is captured in the fifth tuple of the text table (b) using the `/AllDepts/Dept/Staff/Addr` path.

Note that no authorization information of Table 4 is captured by the XRel conversion. Therefore, to support access controls of XML model using XRel method, one needs to *somehow* carry over the authorization rules of Table 4 into relational forms, too.

Table 5. RDBMS side: Four (i.e., attribute, text, element, and path) tables generated from the XML document of Table 3 by XRel.

(a) Attribute table

pathID	value	nodeID
3	CS	3
5	m10	5
10	e10	13
15	XML	21
16	public	22
15	Stream	28
16	private	29

(b) Text table

pathID	value	nodeID
6	Tom	7
7	110 Foster Ave.	9
8	70K	11
11	Jane	15
12	54 Union St.	17
13	45K	19
17	2003	24
18	100K	26
17	2002	31
18	300K	33

(c) Element table

pathID	index	rindex	nodeID
1	1	1	1
2	1	1	2
4	1	1	4
6	1	1	6
7	1	1	8
8	1	1	10
9	1	1	12
11	1	1	14
12	1	1	16
13	1	1	18
14	1	2	20
17	1	1	23
18	1	1	25
14	2	1	27
17	1	1	30
18	1	1	32

(d) Path table

pathID	pathExpr
1	`/AllDepts`
2	`/AllDepts/Dept`
3	`/AllDepts/Dept/@dname`
4	`/AllDepts/Dept/Manager`
5	`/AllDepts/Dept/Manager/@eid`
6	`/AllDepts/Dept/Manager/Name`
7	`/AllDepts/Dept/Manager/Addr`
8	`/AllDepts/Dept/Manager/Salary`
9	`/AllDepts/Dept/Staff`
10	`/AllDepts/Dept/Staff/@eid`
11	`/AllDepts/Dept/Staff/Name`
12	`/AllDepts/Dept/Staff/Addr`
13	`/AllDepts/Dept/Staff/Salary`
14	`/AllDepts/Dept/Proj`
15	`/AllDepts/Dept/Proj/@pname`
16	`/AllDepts/Dept/Proj/@type`
17	`/AllDepts/Dept/Proj/Year`
18	`/AllDepts/Dept/Proj/Budget`

Among many possible approaches, one extreme is *not* to store any accessibility information and instead directly use Table 4 in evaluating access controls. That is, this process of security evaluation is entirely conducted outside of relational databases (i.e., external security evaluation) and has to first evaluate the XPath expressions of Table 4 to find out all XML nodes being enforced. Another variation is to store a list of XML nodes for each authorization rule in an auxiliary table. For instance, for the rule `R2` of Table 4, the XPath expression `//Dept/*/Name` is evaluated first. Then, two path expressions (i.e., pathID is 3 and 11) stored in the path table (d) of Table 5 are identified as matches and stored instead in a table. An example auxiliary table is shown in Table 6.

Table 6. RDBMS side: A pathID-based auxiliary table for `AllDepts` of Table 4.

Subject	pathID	Action	Sign	Type
Public	3	SELECT,UPDATE	-	L
Public	6, 11	SELECT	+	L
Manager	9,10,11,12,13	SELECT	+	R

One limitation of this approach is that this cannot handle the so-called *twig query* such as rules `R4` and `R5` of Table 4 that have the filtering condition. That is, in the `R4`, the XPath expression `//Dept/Proj[./@type='private']/Budget` is essentially a tree with two branches, `//Dept/Proj/Budget` and `//Dept/Proj/@type='private'`, with the node `Budget` being projected at the end. Since the path table (d) of Table 5 has only root-to-leaf paths, it is not straightforward to handle such a case. Therefore, to support such a twig case of authorization rules, one can instead store node IDs in the auxiliary table as shown in Table 7, where two rules `R4` and `R5` of Table 4 are captured with proper node IDs. This approach is essentially similar to the *materialization* method (e.g., [26]) where each user (or role) separately keeps a list of XML nodes that he/she is allowed to access, according to access controls. This approach becomes problematic when the number of rules in Table 4 is huge, although it can be alleviated using a space-efficient method like CAM of [26]. The more serious problem of this approach is that after the original XML data are stored in a relational database, the XML nodes to materialze are scattered among tables, making it difficult to efficiently keeping track of.

Note that neither of two presented schemes can be implemented in the basic relational security features, since these require *value-based* or *content-based* security constraint. Therefore, it would be challenging to investigate how to support such schemes using table-level or column-level relational security constraint.

Table 7. RDBMS side: A nodeID-based auxiliary table for `AllDepts` of Table 4.

Subject	nodeID	Action	Sign	Type
Manager	33	SELECT,UPDATE	+	L
Staff,Manager	26	SELECT	+	L

Suppose a manager "Tom" wants to retrieve department names as follows:

```
/AllDepts/Dept//Name
```

Since the `Name` element is accessible to public, no security check is needed and the input XPath is translated to the following SQL according to XRel algorithm:

```
SELECT e.nodeID
FROM   Element e, Path p
WHERE  p.pathExpr LIKE '/AllDepts/Dept%/Name' AND
       e.pathID = p.pathID
```

Table 8. Main issues and choices used in the example throughout Section 4. Choices made are [boxed] .

Issue	Choice
Target model	model-to-model vs. [model-to-RDBMS]
XML security model	[schema-level] vs. instance-level vs. both
XML-to-Relational conversion	structure-based vs. [data-based]
Security check location	external vs. [internal] vs. hybrid
Security check time	post-pruning vs. [pre-pruning] vs. intermixed

Secondly, suppose a regular user "Jane" wants to retrieve salary information of staffs as follows:

```
/AllDepts/Dept/Staff/Salary
```

When an auxiliary table such as Table 6 is given, then, this XPath could be re-written to the following SQL query, where the last WHERE condition "`a.pathID = p.pathID`" ensures valid access control.

```
SELECT e.nodeID
FROM   Element e, Path p, Auxilary a
WHERE  p.pathExpr LIKE '/AllDepts/Dept/Staff/Salary' AND
       e.pathID = p.pathID AND a.pathID = p.pathID
```

The presented example so far and its choices made from the main issues discussed in Section 3 can be summarized as shown in Table 8. As illustrated, choices that we pick in this example is only one of many possible approaches, and more study is needed to understand detailed pros/cons and behaviors of them.

5 Conclusion

In this paper, we explore the research issues on how to support access controls of XML data by leveraging existing techniques in relational databases. We envisage an XML data management system in which (1) users make XML queries against a given XML schema; (2) access controls for XML data are also specified in a XML security model; and (3) Data are stored in relational databases. In such a system, while users view and access data based on an XML data model, the fact that the data is stored and processed in a relational database system is made transparent to the users. We present a framework for processing XML queries in the above system and examine various research issues appeared in the framework. In this paper, we discuss several important problems in terms of converting XML data to relational representation and storage, converting XML security models to relational access control mechanisms and metadata, and security evaluations. We sketch how to resolve the above problems and glue various components in

our framework to efficiently support access controls for the XML model using relational databases. We also use examples to illustrate the various issues we discussed in the paper. While this paper, as a preliminary work, points out a vitally important research direction, we believe that more in-depth studies are needed to realize our vision.

References

1. S. Banerjee, V. Krishnamurthy, M. Krishnaprasad, and R. Murthy. "Oracle8i – The XML Enabled Data Management System.". In *IEEE ICDE*, San Diego, CA, Feb. 2000.
2. E. Bertino and E. Ferrari. "Secure and Selective Dissemination of XML Documents". *IEEE Trans. on Information and System Security (TISSEC)*, 5(3):290–331, Aug. 2002.
3. R. Bourret. "XML and Databases". Web page, Sep. 1999. http://www.rpbourret.com/xml/XMLAndDatabases.htm.
4. T. Bray, J. Paoli, and C. M. Sperberg-McQueen (Eds). "Extensible Markup Language (XML) 1.0 (2nd Edition)". W3C Recommendation, Oct. 2000. http://www.w3.org/TR/2000/REC-xml-20001006.
5. J. M. Cheng and J. Xu. "XML and DB2". In *IEEE ICDE*, San Diego, CA, Feb. 2000.
6. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. "A Fine-Grained Access Control System for XML Documents". *IEEE Trans. on Information and System Security (TISSEC)*, 5(2):169–202, May 2002.
7. E. Damiani, S. De Capitani Di Vimercati, S. Paraboschi, and P. Samarati. "Design and Implementation of an Access Control Processor for XML Documents". *Computer Networks*, 33(6):59–75, 2000.
8. A. Deutsch, M. F. Fernandez, and D. Suciu. "Storing Semistructured Data with STORED". In *ACM SIGMOD*, Philadephia, PA, Jun. 1998.
9. E. Fernandez, E. Gudes, and H. Song. "A Model of Evaluation and Administration of Security in Object-Oriented Databases". *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 6(2):275–292, 1994.
10. D. Florescu and D. Kossmann. "Storing and Querying XML Data Using an RDBMS". *IEEE Data Eng. Bulletin*, 22(3):27–34, Sep. 1999.
11. S. Godik and T. Moses (Eds). "eXtensible Access Control Markup Language (XACML) Version 1.0". OASIS Specification Set, Feb. 2003. http://www.oasis-open.org/committees/xacml/repository/.
12. P. P. Griffiths and B. W. Wade. "An Authorization Mechanism for a Relational Database System". *ACM Trans. on Database Systems (TODS)*, 1(3):242–255, Sep. 1976.
13. S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. "Flexible Support for Multiple Access Control Policies". *ACM Trans. on Database Systems (TODS)*, 26(2):214–260, Jun. 2001.
14. S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. "A Unified Framework for Enforcing Multiple Access Control Policies". In *ACM SIGMOD*, pages 474–485, May 1997.
15. S. Jajodia and R. Sandhu. "Toward a Multilevel Secure Relational Data Model". In *ACM SIGMOD*, May 1990.

16. D. Lee and W. W. Chu. "Constraints-preserving Transformation from XML Document Type Definition to Relational Schema". In *Int'l Conf. on Conceptual Modeling (ER)*, pages 323–338, Salt Lake City, UT, Oct. 2000.
17. W.-C. Lee, G. Mitchell, and X. Zhang. "Integrating XML Data with Relational Databases". In *IEEE Int'l Workshop on Knowledge Discovery and Data Mining in World Wide Web*, Taipei, Taiwan, Apr. 2000.
18. F. Rabitti, E. Bertino, and G. Ahn. "A Model of Authorization for Next-Generation Database Systems". *ACM Trans. on Database Systems (TODS)*, 16(1):89–131, 1991.
19. P. Samarati, E. Bertino, and S. Jajodia. "An Authorization Model for a Distributed Hypertext System". *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 8(4):555–562, 1996.
20. R. Sandhu and F. Chen. "The Multilevel Relational (MLR) Data Model". *IEEE Trans. on Information and System Security (TISSEC)*, 1(1), 1998.
21. R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. "Role-Based Access Control Models". *IEEE Computer*, 29(2), 1996.
22. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. "Relational Databases for Querying XML Documents: Limitations and Opportunities". In *VLDB*, Edinburgh, Scotland, Sep. 1999.
23. K.-L. Tan, M. L. Lee, and Y. Wang. "Access Control of XML Documents in Relational Database Systems". In *Int'l Conf. on Internet Computing (IC)*, Las Vegas, NV, Jun. 2001.
24. M. Winslett, K. Smith, and X. Qian. "Formal Query Languages for Secure Relational Databases". *ACM Trans. on Database Systems (TODS)*, 19(4):626–662, 1994.
25. M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. "XRel: A Path-Based Approach to Storage and Retrieval of XML Documents using Relational Databases". *ACM Trans. on Internet Technology (TOIT)*, 1(2):110–141, Nov. 2001.
26. T. Yu, D. Srivastava, L. V.S. Lakshmanan, and H. V. Jagadish. "Compressed Accessibility Map: Efficient Access Control for XML". In *VLDB*, Hong Kong, China, 2002.

Author Index