Stéphane Bressan Stefano Ceri
Ela Hunt Zachary G. Ives
Zohra Bellahsène Michael Rys
Rainer Unland (Eds.)

Database and XML Technologies

Third International XML Database Symposium, XSym 2005
Trondheim, Norway, August 28-29, 2005
Proceedings

Springer

Volume Editors

Stéphane Bressan
National University of Singapore, Department of Computer Science
School of Computing
3 Science drive 2, 117543 Singapore, Republic of Singapore
E-mail: steph@nus.edu.sg

Stefano Ceri
Politecnico di Milano, Dipartimento di Elettronica e Informazione
Via Ponzio, 34/5, 20133 Milano, Italy
E-mail: ceri@elet.polimi.it

Ela Hunt
University of Glasgow, Department of Computing Science
Lilybank Gardens 8-17, Glasgow G12 8QQ, UK
E-mail: ela@dcs.gla.ac.uk

Zachary G. Ives
University of Pennsylvania, Computer and Information Science Department
3330 Walnut Street, Philadelphia, PA 19104-6389, USA
E-mail: zives@atcis.upenn.edu

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

Michael Rys
Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, USA
E-mail: mrys@mircosoft.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

Library of Congress Control Number: 2005931472
CR Subject Classification (1998): H.2, H.3, H.4, C.2.4

ISSN 0302-9743
ISBN-10 3-540-28583-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-28583-0 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 11547273 06/3142 5 4 3 2 1 0

Preface

This year marks an exciting time in the XML-database space: XQuery is moving closer to becoming a full W3C Recommendation, and the "Big 3" database vendors (IBM, Oracle, Microsoft) are expected to release XQuery support in their relational DBMSs, joining a number of existing open source and commercial products. Thus, we are very pleased to feature an industrial paper (describing the XML-specific features of Microsoft SQL Server) as well as 14 research papers. XSym's focus this year was on building XML repositories, and papers discussed the following topics: indexing support for the evaluation of XPath and XQuery; benchmarks and algorithms for XQuery and XPath evaluation; algorithms for constraint satisfaction checking, information extraction, and subtree matching; and applications of XML in information systems.

This year, XSym also coordinated its efforts with the Database and Programming Languages Symposium, DBPL 2005. The resulting program included not only presentations of the papers in this proceedings, but also a joint DBPL-XSym keynote talk by Giuseppe Castagna, developer of the CDuce language for XML processing, and a joint panel on open XML research problems and challenges.

The organizers would like to express their gratitude to the XSym Program Committee and external reviewers for their efforts in providing very thorough evaluations of the submitted papers under significant time constraints and to Microsoft for their sponsorship and for the use of the Microsoft Conference Management Toolkit. We would also like to thank Gavin Bierman and Christoph Koch, the organizers of DBPL, for their efforts and their willingness to coordinate with us.

These proceedings are dedicated to Alberto Mendelzon who sadly passed away this year. As a strong supporter of and active contributor to this symposium series he will always remain in our memory.

Singapore, Milan, Montpellier, Glasgow
Philadelphia, Essen, Redmond

July 2005

Stéphane Bressan
Stefano Ceri
Zohra Bellahsene
Ela Hunt
Zachary Ives
Rainer Unland
Michael Rys

General Chair

Stéphane Bressan, National University of Singapore (Singapore)

General Co-chair

Stefano Ceri, Politecnico di Milano (Italy)

Organizing Chair

Zohra Bellahsene, LIRMM (France)

Program Committee Chairs

Ela Hunt, University of Glasgow (UK)
Zachary Ives, University of Pennsylvania (USA)

Proceedings

Rainer Unland, University of Duisburg-Essen (Germany)

Sponsorship

Michael Rys, Microsoft (USA)

Communications

Akmal B. Chaudhri, IBM ISV & Developer Relations (USA)

International Programme Committee

External Reviewers

Andrei Arion	LRI – Université Paris XI (France)
Patrick Bosc	LRI – Université Paris XI (France)
Chavdar Botev	Cornell University (USA)
Giuseppe Castagna	LRI – Université Paris XI (France)
Laura Chiticariu	University of California, Santa Cruz (USA)
David DeHaan	University of Waterloo (Canada)
Maged El-Sayed	Worcester Polytechnic Institute (USA)
Fan Yang	Cornell University (USA)
Mirian Halfeld-Ferrari	LRI – Université Paris XI (France)
Ming Jiang	Worcester Polytechnic Institute (USA)
Ming Lee	Worcester Polytechnic Institute (USA)
Diego Milano	University of Roma (Italy)
	University of Edinburgh (UK)
Feng Shao	Cornell University (USA)
Frank Tompa	University of Waterloo (Canada)
Song Wang	Worcester Polytechnic Institute (USA)
Rui Yang	National University of Singapore (Singapore)
Ning Zhang	University of Waterloo (Canada)

Table of Contents

Patterns and Types for Querying XML Documents [⋆]

Giuseppe Castagna

CNRS, École Normale Supérieure de Paris, France

In order to manipulate XML data, a programming or query language should provide some primitives to deconstruct them, in particular to pinpoint and capture some subparts of the data.

Among various proposals for primitives for deconstructing XML data, two different and complementary approaches seem to clearly stem from practise: path expressions (usually XPath paths [7], but also the "dot" navigation of Cω [3]) and regular expression patterns [13].

Path expressions are navigational primitives that point out where to capture data substructures. They (and those of Cω, in particular) closely resemble the homonymous primitives used by OQL [9] in the contexts of OODB query languages with the difference that instead of sets of objects they return sets or sequences of elements: more precisely all elements that can be reached following the path at issue. These primitives are at the basis of standard languages such as XSLT [8] or XQuery [4].

More recently, a new kind of deconstructing primitives was proposed, regular expression patterns [13], which extend by regular expressions the pattern matching primitive as popularised by functional languages such as ML and Haskell. Regular expression patterns were first introduced in the XDuce [12] programming language and are becoming more and more popular, since they are being adopted by such quite different languages as ℂDuce [1] (a general purpose extension of the XDuce language) and its query language ℂQL [2], Xtatic [10] (an extension of C#), Scala [15] (a general purpose Java-like object-oriented language that compiles into Java bytecode), XHaskell [14] as well as the extension of Haskell proposed in [5].

The two kinds of primitives are not antagonists, but rather orthogonal and complementary. Path expressions implement a "vertical" exploration of data as they capture elements that may be at different depths, while patterns perform a "horizontal" exploration of data since they are able to perform finer grained decomposition on sequences of elements. The two kinds of primitives are quite useful and they mutually complement nicely. Therefore, it would seem natural to integrate both of them in a query or programming language for XML. Despite of that, we are aware of just two works in which both primitives are embedded (and, yet, loosely coupled): in ℂQL it is possible to write select-from-where expressions, where regular expression patterns are applied in the from clause to sequences that are returned by XPath-like expressions; Gapeyev and Pierce [11] show how it is possible to use regular expression patterns with an all match semantics to encode a subset of XPath and plan to use this encoding to add XPath to the Xtatic programming language.

The reason for the lack of study of the integration of these two primitives may be due to the fact that each of them is adopted by a different community: regular patterns

[⋆] Joint talk with DBPL 2005. Full version available in the Proc. of the *10th Intl. Symp. on Database Programming Languages*, G. Bierman and C. Koch eds., LNCS, Springer, 2005.

are almost confined to the programming language community while XPath expressions are pervasive in the database community.

The goal of this lecture is to give a brief presentation of the regular pattern expressions style together with the type system to which they are tightly connected, that is the semantic subtyping based type systems [6]. We are not promoting the use of these to the detriment of path expressions, since we think that the two approaches should be integrated in the same language and we see in that a great opportunity of collaboration between the database and the programming languages communities. Since the author belongs to latter, this lecture tries to describe the pattern approach addressing some points that should be of interest to the database community as well. In particular, after a general overview of regular expression patterns and types in which we show how to embed patterns in a select˙from˙where expression, we discuss several usages of these patterns/types, going from the classic use for partial correctness and schema specification to the definition of new data iterators, from the specification of efficient run-time to the definition of logical pattern-specific query optimisations.

References

1. V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-friendly general purpose language. In *ICFP '03, 8th ACM International Conference on Functional Programming*, pages 51–63, Uppsala, Sweden, 2003. ACM Press.
2. V. Benzaken, G. Castagna, and C. Miachon. A full pattern-based paradigm for XML query processing. In *PADL 05, 7th International Symposium on Practical Aspects of Declarative Languages*, number 3350 in LNCS, pages 235–252. Springer, January 2005.
3. Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in Cw. In *Proc. of ECOOP '2005, European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*. Springer, 2005.
4. S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. *XQuery 1.0: An XML Query Language*. W3C Working Draft, `http://www.w3.org/TR/xquery/`, May 2003.
5. Niklas Broberg, Andreas Farre, and Josef Svenningsson. Regular expression patterns. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 67–78, New York, NY, USA, 2004. ACM Press.
6. G. Castagna and A. Frisch. A gentle introduction to semantic subtyping. In Proceedings of *PPDP '05, the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, ACM Press (full paper) and *ICALP '05, 32nd International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science n. 3580, Springer (short abstract), Lisboa, Portugal, 2005. Joint ICALP-PPDP keynote talk.
7. J. Clark and S. DeRose. *XML Path Language (XPath)*. W3C Recommendation, `http://www.w3.org/TR/xpath/`, November 1999.
8. James Clark. *XSL Transformations (XSLT)*. W3C Recommendation, `http://www.w3.org/TR/xslt/`, November 1999.
9. Sophie Cluet. Designing OQL: allowing objects to be queried. *Inf. Syst.*, 23(5):279–305, 1998.
10. Vladimir Gapeyev and Benjamin C. Pierce. Regular object types. In *European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany*, 2003. A preliminary version was presented at FOOL '03.

11. Vladimir Gapeyev and Benjamin C. Pierce. Paths into patterns. Technical Report MS-CIS-04-25, University of Pennsylvania, October 2004.
12. Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. In *Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, 2000.
13. Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2001.
14. K. Zhuo Ming Lu and M. Sulzmann. An implementation of subtyping among regular expression types. In *Proc. of APLAS'04*, volume 3302 of LNCS, pages 57–73. Springer-Verlag, 2004.
15. Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical Report IC/2004/64, École Polytechnique Fédérale de Lausanne, 2004. Latest version at `http://scala.epfl.ch`.

Checking Functional Dependency Satisfaction in XML

Millist W. Vincent and Jixue Liu

School of Computer and Information Science
University of South Australia
{millist.vincent,jixue.liu}@unisa.edu.au

Abstract. Recently, the issue of functional dependencies in XML (XFDs) have been investigated. In this paper we consider the problem of checking the satisfaction of an XFD in an XML document. We present an efficient algorithm for the problem that is linear in the size of the XML document and linear in the number of XFDs to be checked. Also, our technique can be easily extended to efficiently incrementally check XFD satisfaction.

1 Introduction

The eXtensible Markup Language (XML) [5] has recently emerged as a standard for data representation and interchange on the Internet. 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 [7]. Several different classes of integrity constraints for XML have been defined including key constraints [6], path constraints [8], and inclusion constraints [10, 11] 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 [16] and which has been little investigated is how to extend the oldest and most well studied integrity constraint in relational databases, namely a *functional dependency* (FD), to XML and then how to develop a normalization theory for XML. This problem is not of just theoretical interest. The theory of FDs and normalization 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.

Recently, two approaches have been given for defining functional dependencies in XML (called XFDs). The first [1–3], proposed a definition based on the notion of a 'tree tuple' which in turn is based on the total unnesting of a relation [4]. More recently, we have proposed an alternative 'closest node' definition [14], which is based on paths and path instances that has similarity with the approach in [6] to defining keys in XML. This relationship between keys as defined in [6] and XFDs as defined in [14] extends further, as it was shown in [14] that in the case of simple paths, keys in XML are a special case of XFDs in the same way that keys in relational databases are a special case of FDs.

In general, the two approaches to defining XFDs are not comparable since they treat missing information in the XML document differently and the approach in [1–3] assumes the existence of a DTD whereas the approach in [14] does not. Howevever, we have recently shown that [15], in spite of the very different approaches used in [1–3] and [14], the two aproaches coincide for a large class of XML documents. In particular, we have shown that the definitions coincide for XML documents with no missing information conforming to a nonrecursve, disjunction free DTD. This class includes XML documents derived from complete relational databases using any 'non pathological' mapping. It has also been shown that in this situation, for mappings from a relation to an XML document defined by first mapping to a nested relation via an arbitrary sequence of nest and unnest operations, then followed my a direct mapping to XML, FDs in relations map to XFDs in XML. Hence there is a natural correspondence between FDs and XFDs.

In this paper we address the problem of developing an efficient algorithm for checking whether an XML document satisfies a set of XFDs as defined in [14]. We develop an algorithm which requires only one pass of the XML document and whose running time is linear in the size of the XML document and linear in the size of the number of XFDs. The algorithm uses an innovative method based on a multi level extension of extendible hashing. We also investigate the effect of the size on the number of paths on the l.h.s. of the XFD and show that the running time is both linear in the number of paths and also increases quite slowly with the number of paths.

Although the issue of developing checking the satisfaction of 'tree tuple' XFDs was not addressed in [1–3], testing satisfaction using the definitions in [1–3] directly is likely to be quite expensive. This is because there are three steps involved in the approach of [1–3]. The first is to generate a set of tuples from the total unnesting of an XML document. This set is likely to be much larger than the original XML document since unnesting generates all possible combinations amongst elements. The second step is to generate the set of tree tuples, since not all tuples generated from the total unnesting are 'tree tuples'. This is done by generating a special XML tree (document) from a tuple and checking if the document so generated is subsumed by the original XML tree (document). Once again this is likely to be an expensive procedure since it may require that the number of times the XML document is scanned is the same as the number of tuples in the total unnesting. In contrast, our method requires only one scan of the XML document. Finally, the definition in [1–3] requires scanning the set of tree tuples to check for satisfaction in a manner similar to ordinary FD satisfaction. This last step is common also to our approach.

The rest of this paper is organized as follows. Section 2 contains some preliminary definitions that we need before defining XFDs. We model an XML document as a tree as follows. In Section 3 the definition of an XFD is presented and the essential ideas of our algorithm are presented. Section 4 contains details of experiments that were performed to assess the efficiency of our approach and Section 5 contains concluding comments.

2 Preliminary Definitions

Definition 1. *Assume a countably infinite set* $\mathbf{E}$ *of element labels (tags), a countably 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:*

1. V *is a* finite *set of* nodes*;*
2. lab *is a total function from* V *to* $\mathbf{E} \cup \mathbf{A} \cup \{\mathcal{S}\}$*;*
3. ele *is a partial function from* V *to a sequence of nodes in* V *such that for any* $v \in V$*, if* $ele(v)$ *is defined then* $lab(v) \in \mathbf{E}$*;*
4. att *is a partial function from* $V \times \mathbf{A}$ *to* V *such that for any* $v \in V$ *and* $a \in \mathbf{A}$*, if* $att(v, a) = v_1$ *then* $lab(v) \in \mathbf{E}$ *and* $lab(v_1) = a$*;*
5. 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* $lab(v) \in \mathbf{A}$*;*
6. *We extend the definition of* val *to sets of nodes and if* $V_1 \subseteq V$*, then* $val(V_1)$ *is the set defined by* $val(V_1) = \{val(v) | v \in V_1\}$*;*
7. v_r *is a distinguished node in* V *called the root of* T*;*
8. *The parent-child edge relation on* V*,* $\{(v_1, v_2) | v_2$ *occurs in* $ele(v_1)$ *or* $v_2 = att(v_1, a)$ *for some* $a \in \mathbf{A}\}$ *is required to form a tree rooted at* v_r*;*

Also, the set of ancestors of a node $v \in V$ is denoted by $ancestor(v)$ and the parent of a node v by $parent(v)$.

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* $l_i \in \mathbf{E}$ *for* $1 <= i <= n-1$ *and* $l_n \in \mathbf{E} \cup \mathbf{A} \cup \{\mathcal{S}\}$ *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, Dept, Section, Emp`$\}$ and $\mathbf{A} = \{$`Project`$\}$ then `root, root.Dept` and `root.Dept.Section` are all paths.

Definition 3. *Let* p *denote the path* $l_1. \cdots .l_n$*. The function* $Parent(p)$ *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 instance, if $\mathbf{E} = \{$`root, Dept, Section, Emp`$\}$ and $\mathbf{A} = \{$`Project`$\}$ then `root.Dept` is a strict prefix of
`root.Dept.Section` and `root.Dept.Section.Emp` $\cap$
`root. Dept.Section.Project` $=$ `root.Dept.Section`.

Definition 4. *A* path instance *in an XML tree* $T = (V, lab, ele, att, val, v_r)$ *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 *finite* 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. In a sense we are assuming that XFDs and DTDs are orthogonal, in a similar fashion to that used in [6] where keys and DTDs are assumed to be orthogonal. We note that because of the restriction that P is finite, if P is derived from a DTD then the DTD must be non recursive. Next, we place the following restriction on the set of paths.

Definition 5. *A set* P *of paths is* downward closed *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 downward closed.

We now define the notion of an XML tree conforming to a set of paths P.

Definition 6. *Let* P *be a downward closed 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 a path in* P*.*

We note that if the set of paths is derived from a DTD, then requiring that the XML document conform to the set of paths is a much weaker condition than requiring that it conform to the DTD.

The next issue that arises in developing the machinery to define XFDs is the issue of missing information. This is addressed in [14] where missing nodes are considered and XFDs are defined using an extension of the strong satisfaction approach used in defining FD satisfaction in incomplete relations [4]. However, in this paper we take the simplifying assumption that there is no missing information in the XML tree. More precisely, we have the following definition.

Definition 7. *Let* P *be a downward closed set of paths, let* T *be an XML tree 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* $v'_1.\cdots.v'_m$*.*

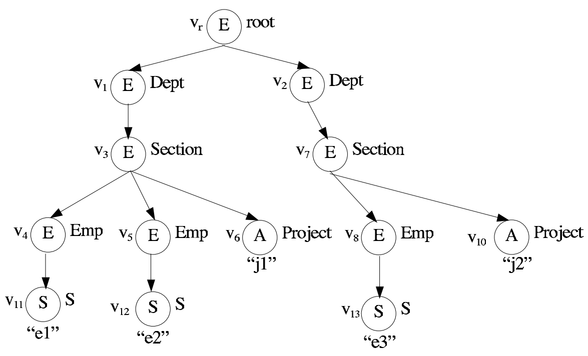


Fig. 1. A complete XML tree

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.

One important comment to make on completeness is that if the set of paths is derived from a DTD and if we consider trees that conform to the DTD, and not just to P, then complete trees correspond only to disjunction free DTDs as shown in [3].

The next function returns all the final nodes of the path instances of a path p in T.

Definition 8. *Let P be a downward closed 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 $N(p) = \{v | v_1. \cdots .v_n \in Paths(p) \wedge v = v_n\}$.*

For example, in Figure 1, $N($`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 downward closed 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 downward closed 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, $Nodes(v_1,$ `root.Dept.Section.Emp`$) = \{v_4, v_5\}$.

3 Checking XFDs

We firstly recall the definition of an XFD from [14], restricted to the situation where the XML document is complete.

Definition 11. *Let P be a set of downward closed paths and let T be a complete XML tree that conforms to P. An XML functional dependency (XFD) is a statement of the form: $p_1, \ldots, p_k \to q$, $k \geq 1$, where $p_1, \ldots, p_k$ and q are paths in P. T satisfies the XFD if there exists p_i, for some $i, 1 \leq i \leq k$, such that $p_i = q$ or whenever there exists two distinct path instances $v_1.\cdots.v_n$ and $v'_1.\cdots.v'_n$ defined over q in T such that $val(v_n) \neq val(v'_n)$, then $\exists i, 1 \leq i \leq k$, such that $val(Nodes(x_i, p_i)) \cap val(Nodes(y_i, p_i)) = \emptyset$, where: $x_i = \{v | v \in AAncestor(v_n) \wedge v \in N(p_i \cap q)\}$ and $y_i = \{v | v \in AAncestor(v'_n) \wedge v \in N(p_i \cap q)\}$.*

We now illustrate the definition by an example.

Example 1. Consider the XFD
`root.publication.publisher.S` $\to$ `root.publication.title` in Figure 2. Then $v_4 \in N($`root.publication.title`$)$ and $v_6 \in N($`root.publication.title`$)$ and $val(v_4) =$ `"t1"` $\neq val(v_6) =$ `"t2"`.
So `root.publication.title`$\cap$ `root.publication.publisher.S` $=$ `root.publication` and so $N($`root.publication`$) = \{v_1, v_2\}$. Thus $x_{1_1} = v_1$ and $y_{1_1} = v_2$ and so $Nodes(x_{1_1},$`root.publication.publisher.S`$) = v_{17}$ and thus $val(Nodes(x_{1_1},$`root.publication.publisher.S`$)) = \{$`"p1"`$\}$. Also, $Nodes(y_{1_1},$`root.publication.publisher.S`$) = v_{19}$ and so $val(Nodes(y_{1_1},$`root.publication.publisher.S`$)) = \{$`"p1"`$\}$ and so the XFD `root.publication.publisher.S` $\to$ `root.publication.title` is violated because $val(Nodes(x_{1_1},$`root.publication.publisher.S`$))$ $\cap$ $val(Nodes(y_{1_1},$`root.publication.publisher.S`$)) \neq \emptyset$. We note that if the val of node v_4 was changed to `"t2"` then the XFD would be satisfied.

Consider next the XFD `root.publication.title` $\to$ `root.publication.publisher.S`. The only nodes in $N($`root.publication.publisher.S`$)$ are v_{17} and v_{19} and $val(v_{17}) =$ `"p1"` and $val(v_{19}) =$ `"p1"` and so the XFD `root.publication.title` $\to$ `root.publication.publisher.S` is satisfied.

We now present an algorithm for checking whether an XML document satisfies a set of XFDs. The algorithm has two major steps. The first step is to produce what we call tuples. The second step is to hash the tuples to check if the document satisfies the XFDs.

3.1 Tuple Generation

We start with the definition of some terms.

Definition 12 (Relevant Path). Given a set Σ of FDs $\{f_1, \ldots, f_m\}$ we use $relev(\Sigma)$, called the set of *relevant paths*, to denote the list of *distinct* paths defined as the following:

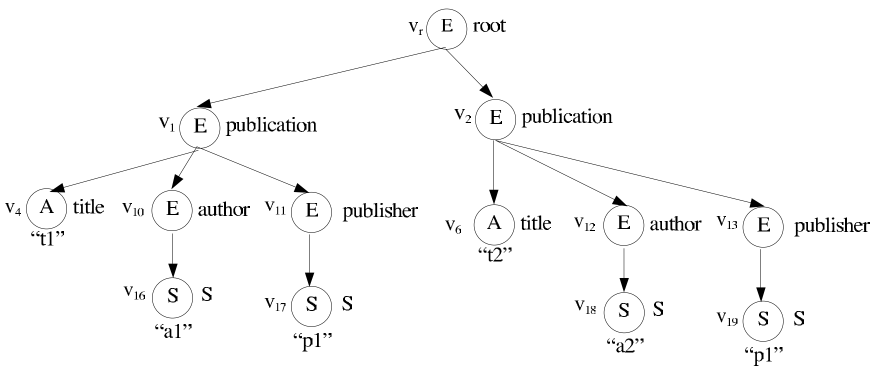


Fig. 2. An XML tree

- all paths involved in Σ, including those on the LHS of the XFDs and also those on the RHS;
- if $p_1, p_2 \in relev(\Sigma)$ then $p_1 \cap p_2 \in relev(\Sigma)$;
- the order of the paths and path intersections in the list agrees with the order of their appearances in documents.

Consider the example in Figure 3.
Let $\Sigma = \{root.A, root.A.B \rightarrow root.A.G.C,\ root.A.G.C, root.A.G.D \rightarrow root.A.B\}$. Then $relev(\Sigma) = [A, B, G, C, D]$. Note that for simplicity, we abbreviate paths by their end labels, which will not introduce confusion in the presentation.

We further use $pathroot(\Sigma)$, called the *path root*, to mean the shortest path in $relev(\Sigma)$. We call a subtree rooted at a node labelled by $pathroot(\Sigma)$ a *relevant* tree. We call the nodes in a relevant tree labelled by the end labels of the paths in $relev(\Sigma)$ *relevant nodes*. Given a relevant node v, $path(v)$ is the path on the path instance reaching v. Given a path $p \in relev(\Sigma)$, $posi(p)$ is the sequential number of p in $relev(p)$ and if p is the first element in $relev(\Sigma)$, then $posi(p)$ is 1.

In Figure 3, $pathroot(\Sigma) = root.A$. The subtree rooted at v_1 is relevant tree. All nodes labelled by A, B, C, D, G are relevant nodes. $posi(root.A.G.D) = 5$ and $posi(root.A) = 1$, $path(v_4) = root.A.G$.

The concept *tuple* defined following is an important construct used to model the result of document parsing.

Definition 13 (Tuple). Given a set Σ of XFDs and a relevant tree bT, a tuple t of bT over $relev(\Sigma)$ is defined as $t =< val_1, ..., val_n >$ where n is the number of paths in $relev(\Sigma)$ and for each i in $[1, ..., n]$, $p_i \in relev(\Sigma) \wedge val_i = val(v_u) \wedge v_u \in bT \wedge lab(v_u) = last(p_i)$.

We define the following terms to be used to indicate the directions of parsing in relation to the paths in $relev(\Sigma)$.

tuples

A	B	G	C	D
v_1	b1	v_4	c1	d1
v_1	b2	v_4	c1	d1
v_1	b1	v_4	c1	d2
v_1	b2	v_4	c1	d2
v_1	b1	v_5	c2	d3
v_1	b2	v_5	c2	d3

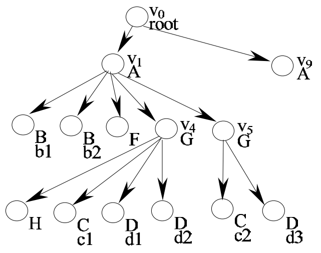


Fig. 3. An XML tree and its tuples

Definition 14. Let v_l be the last visited relevant node and v be the current visited node. Then:

- v is called a *down node* if $posi(path(v)) > posi(path(v_l))$;
- v is called a *up node* if $posi(path(v)) < posi(path(v_l))$;
- v is called a *across node* if $posi(path(v)) =$
  $posi(path(v_l))$.

Note that in this definition, the directions, *down*, *up*, and *across*, are defined relative to the order in $relev(\Sigma)$, not the directional positions in a tree. This is important because during parsing, we do not care about irrelevant nodes but only concentrate on relevant nodes.

We now propose the parsing algorithm. The algorithm reads text from a document and generates the tuples for a set of XFDs. After a line of text is read, the algorithm tokenizes the line into tokens of tags and text strings. If a token is a tag of a relevant path, then the parsing direction is determined. If the direction is downward, content of the element will be read and put into the current tuple. If it is across, new tuples are created. In the algorithm, there are two variables $openTuple$ and $oldOpenTuple$ used to deal with multiple occurrences of a node. For example in Figure 3, there are multiple B nodes. Multiple tuples need to be created so that each occurrence can be combined with values from other relevant nodes like C nodes and D nodes. In the algorithm, we discuss only elements but not attributes. Attributes are only specially cases of elements when parsed and the algorithm can be easily adapted to attributes.

```
Algorithm 1
INPUT: An XML document $T$ and $relev(\Sigma)$
OUTPUT: a set of tuples
Let $lastPosi = 1$, $curPosi = 1$,
    $openTuple = 1$, $lastOpenTuple = 1$
Let reading will read and tokenize input to one of the
following tokens: start tags, closing tags, and texts
Foreach $token$ in $T$ in order,
   if $token$ is text: set $token$ as value to the
      position $curPosi$ of the last $openTuple$
      of tuples
   let $curPosi = posi(tag)$
   if $curPosi = 0$ (NOT relevant): next token
   if $token$ is a closing tag
      if $current$ is the last in $relev(\Sigma)$
         $openTuple = oldOpenTuple = 1$
      next noken;
   if $curPosi > lastPosi$ (down)
      $lastOpenTuple = openTuple$
      $lastPosi = curPosi$, next token
   if $curPosi = lastPosi$ (across)
      create $oldOpenTuple$ new tuples
      copy the first $lastPosi - 1$ values from
      the previosu tuple to the new tuples
      $openTuple = openTuple + lastOpenTuple$
      next token
   if $curPosi < lastPosi$ (up)
      $lastPosi = curPosi$, next token
end foreach
```

Observation 1: The time for the above algorithm to generate tuples is linear in the size of the document.

3.2 Hashing and Adaption

Once we get the tuples, we use hashing to check if the XFDs are satisfied by the document which is now represented by the tuples. Hashing is done for each XFD. In other words, if there are m XFDs, m hash tables will be used. Let Tup be the tuples generated by Algorithm 1. We project tuples in Tup onto the paths of an XFD $f := \{p_1, ..., p_n\} \rightarrow q$ to get a projected bag of tuples denoted by $Tup(f)$. For each tuple t in $Tup(f)$, $f(p)$ denotes the projection $t[p_1, ..., p_n]$ and $f(q)$ denotes $t[q]$. Then a hash table is created for each XFD as follows.

The hash key of each hash table is $f(p)$ and the hash value is $f(q)$. When two tuples with the same $f(p)$ but different $f(q)$s are hashed into a bucket, the XFD is violated. This criteria corresponds exactly to the definition of an XFD but with the condition that there is no collision.

We define a collision to be the situation where two tuples get the same hash code which puts the two tuples in the same bucket. Based on the criteria above, this means that the two tuples make the XFD violated but in fact they do not. For example, if the two tuples for $< f(p), f(q) >$ are $< 10...0, 1 >$ and $< 20...0, 2 >$ where '...' represent 1000 zeros. If a hash function is the modular operator with the modular being 1 million indicating there are 1 million buckets in the hash table, then the two tuples will be put into the same bucket of the hash table which indicate that the XFD is not satisfied based on the criteria presented above. However, the tuples satisfy the XFD. With normal extendible hashing the traditional solution to this problem is to double the size of the hash table, but this means that memory space can be quickly doubled while the two tuple are still colliding. In fact with only two tuples that collide, we can exhaust memory, no matter how large, if there is no appropriate collision handling mechanism.

With our implementation, we use two types of collision handling mechanisms. The first one is doubling the size of the hash table. As discussed above, this only works for a limited number of cases. The second technique is used if the table size cannot be doubled. The second method involves the use of overflow buckets and is illustrated in Figure 4.

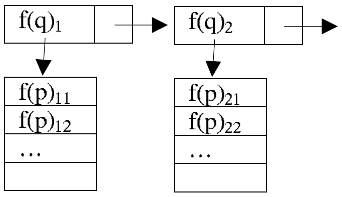


Fig. 4. Bucket structure of hash table

In the figure, a bucket has a section, denoted by $bask(q)$, to store $f(q)$ and a downward list, denoted by $bask(p)$, to store $f(p)$'s if there are multiple tuples having the same $f(q)$ but different $f(p)$'s because of a collision. It is also possible that multiple tuples having different $f(q)$'s come into the same bucket, as we discussed before, because of a collision. In this case, these tuples are stored, based on their $f(q)$ values, in the extended buckets which are also buckets connected to the main bucket horizontally.

With this extension, the following algorithm is used to check if the XFDs is satisfied.

The performance of the algorithm is basically linear. There is a cost to run the "bucket loop" in the algorithm. However, the cost really depends on the number of collisions. From our experiments, we observed that collision occurred, but the number of buckets involved in collisions is very low. At the same time, more collisions means a higher probability of violating the XFDs.

Algorithm 2
INPUT: A set $Tup(f)$ of tuple for XFD f
OUTPUT: true or false

```
Set the hash table size to the maximum allowed by
   the computer memory
Foreach $t$ in $Tup(f)$
  let $code = hashFunction(f(p))$
  set current bucket to bucket $code$
  bucket loop
    if $bask(q) = f(q)$, insert $f(p)$ in to
       $bask(p)$ else if it is not in it
       exit the bucket loop
    if $bask(q)! = f(q)$, check to see if
       $f(p)$ is in $bask(p)$,
       if yes, exit algorithm with false,
       if not, let the current bucket be
          the next extended bucket
          go to the beginning of the
          bucket loop
  end bucket loop
end foreach
return true
```

4 Experiments

In this section we report on experiments with the algorithms presented in the previous section. All the experiments were run on 1.5GHz Pentium 4 machine with 256MB memory. We used the DTD given in Example 5 and artificially generated XML documents of various sizes in which the XFD was satisfied, the worst case situation for running time as in such a case all the tuples need to be checked. When documents were generated, multiple occurrences of the nodes with the same labels at all levels were considered. Also, the XFDs were defined involving paths at many levels and at deep levels.

In the first experiment, we checked the satisfaction of one XFD and fixed the number of paths on the left hand side of the XFD to 3. We varied the size of the XML document and recorded the CPU time required to check the document for the satisfaction of one XFD. The results of this experiment are shown in Figure 6. These results indicate that the running time of the algorithm is essentially linear in the number of tuples. This is to be expected as the time to perform the checking of an XFD is basically the time required to read and parse the XML document once, which is linear in the size of the document and to hash the tuples into the hash table which again is linear.

In the second experiment, we limited ourselves to only one XFD, fixed the number of tuples in the XML document to 100,000 (and so the size of the document was also fixed), but varied the number of paths on the left hand side of the XFD. The results are shown in Figure 7. The figure shows that again the time is linear in relation to the number of paths. This is also to be expected

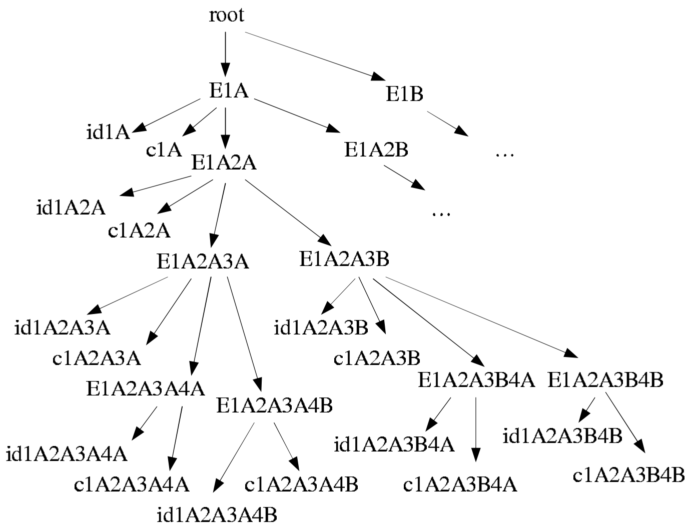


Fig. 5. Implementation DTD

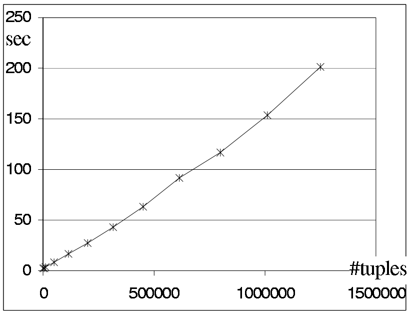


Fig. 6. The number of tuples vs checking time (in seconds)

because the number of paths in a XFD only increases the length of a tuple, but does not require any change to other control structures of the algorithm and therefore the times for reading, parsing, and checking are all kept linear. It is the increase of tuple length that caused the slight increase in processing time and this increase is slow and linear.

In the third experiment, we fixed the number of paths on the left hand side of a XFD to 3 and also fixed the file size and the number of tuples, but varied the number of XFDs to be checked. The result is shown in Figure 8. This result shows that the time is linear in the number of XFDs to be checked, but the increase is steeper than that of Figure 7. This is caused by the way we do the checking. In the previous section, we said that for each XFD, we create a hash table. However, for a very large number of XFDs this requires too much memory

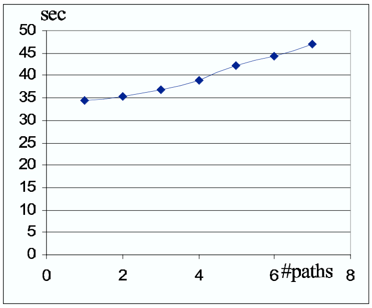


Fig. 7. Number of paths in the left and side of an XFD vs checking time

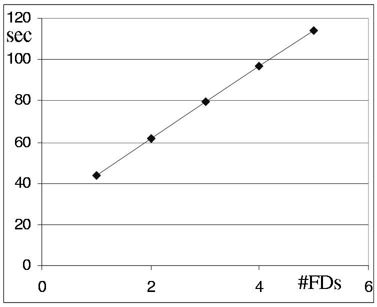


Fig. 8. Number of XFDs to be checked vs checking time

so in this experiment, we created one hash table, checked one XFD, and then used the same hash table to check the second XFD. Thus the time consumed is the addition of the times for checking these XFDs separately. The benefit of this algorithm is that parsing time is saved. Parsing time, based on our experience, is a little more than the time for hashing. Furthermore, the performance of the third experiment can be improved if a computer with bigger memory is used.

5 Conclusions

In this paper we have addressed the problem of developing an efficient algorithm for checking the satisfaction of XFDs, a new type of XML constraint that has recently been introduced [14]. We have developed a novel hash based algorithm that requires only one scan of the XML document and its running time is linear in the size of the XML document and linear in the number of XFDs. Also, our algorithms can be used to efficiently incrementally check an XML document.

There are several are other extensions to the work in this paper that we intend to conduct in the future. The first is to extend the algorithm to the case where there is missing information in the XML document as defined in [14]. The second is to extend the approach to the checking of multivalued dependencies in XML, another new XML constraint that has recently been introduced [12, 13].

References

1. M. Arenas and L. Libkin. A normal form for XML documents. In *Proc. ACM PODS Conference*, pages 85–96, 2002.
2. M. Arenas and L. Libkin. An information-theoretic approach to normal forms for relational and XML data. In *Proc. ACM PODS Conference*, pages 15–26, 2003.
3. M. Arenas and L. Libkin. A normal form for XML documents. *TODS*, 29(1):195 – 232, 2004.
4. P. Atzeni and V. DeAntonellis. *Foundations of databases*. Benjamin Cummings, 1993.
5. T. Bray, J. Paoli, and C.M. Sperberg-McQueen. Extensible markup language (XML) 1.0. Technical report, http://www.w3.org/Tr/1998/REC-XML-19980819, 1998.
6. P. Buneman, S. Davidson, W. Fan, C. Hara, and W. Tan. Reasoning about keys for XML. *Information Systems*, 28(8):1037–1063, 2003.
7. P. Buneman, W. Fan, J. Simeon, and S. Weinstein. Constraints for semistructured data and XML. *ACM SIGMOD Record*, 30(1):45–47, 2001.
8. P. Buneman, W. Fan, and S. Weinstein. Path constraints on structured and semistructured data. In *Proc. ACM PODS Conference*, pages 129 – 138, 1998.
9. Y. Chen, S. Davidson, and Y. Zheng. Xkvalidator: a constraint validator for xml. In *CIKM*, pages 446–452, 2002.
10. W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. *Journal of the ACM*, 49(3):368 – 406, 2002.
11. W. Fan and J. Simeon. Integrity constraints for XML. *Journal of Computer and System Sciences*, 66(1):254–291, 2003.
12. M.W. Vincent and J. Liu. Multivalued dependencies and a 4NF for XML. In *15th International Conference on Advanced Information Systems Engineering (CAISE)*, pages 14–29. Lecture Notes in Computer Science 2681 Springer, 2003.
13. M.W. Vincent, J. Liu, and C. Liu. Multivalued dependencies and a redundancy free 4NF for XML. In *International XML database symposium (Xsym)*, pages 254–266. Lecture Notes in Computer Science 2824 Springer, 2003.
14. M.W. Vincent, J. Liu, and C. Liu. Strong functional dependencies and their application to normal forms in XML. *ACM Transactions on Database Systems*, 29(3):445–462, 2004.
15. M.W. Vincent, J. Liu, C. Liu, and M.Mohania. On the definition of functional dependencies in XML. In *submitted to ACM Transactions on Internet Technology*, 2004.
16. J. Widom. Data management for XML - research directions. *IEEE data Engineering Bulletin*, 22(3):44–52, 1999.

A Theoretic Framework for Answering XPath Queries Using Views

Jian Tang[1] and Shuigeng Zhou[2]

[1] Department of Computer Science, Memorial University of Newfoundland
St. John's, NL, A1B 3X5, Canada
jian@cs.mun.ca

[2] Department of Computer Science, Fudan University
Shanghai, 200433, China
sgzhou@fudan.edu.cn

Abstract. Query rewriting has many applications, such as data caching, query optimization, schema integration, etc. This issue has been studied extensively for relational databases and, as a result, the technology is maturing. For XML data, however, it is still at the developing stage. Several works have studied this issue for XML documents recently. They are mostly application-specific, being that they address the issues of query rewriting in a specific domain, and develop methods to meet the specific requirements. In this paper, we study this issue in a general setting, and concentrate on the correctness requirement. Our approach is based on the concept of query containment for XPath queries, and address the question of how that concept can be adopted to develop solutions to query rewriting problem. We study various conditions under which the efficiencies and applicability can trade each other at different levels, and introduce algorithms accordingly.

1 Introduction

Query rewriting using views has many applications, such as data caching, query optimization, schema integration, etc. It can be simply described as follows. A user query wants to retrieve information from a given set of data. Instead of directly running the user query, we wish to rewrite it into another query by using a separate view query as a tool. The rewritten version then produces the output that can satisfy the users' needs. This issue has been studied extensively for relational databases [8, 9, 11, 13, 14]. As a result, the technology is maturing. For XML data, however, it is still at the developing stage. Several works have studied this issue for XML documents in specific contexts recently. In [3], the authors propose a system for answering XML queries using previously cached data. In [1, 5, 15], methods are suggested to rewrite queries using views to enhance the efficiencies. The work in [4, 17] study how the queries over the target schema can be answered using the views over the source data, and hence provide a way for schema integration. In [2, 7, 12], the authors study the query rewriting problem for semi-structured data. Since all these works are tailored to specific domains, they do not discuss how they can fit into a general setting. In this paper, we introduce a general framework for XPath query rewriting in a restricted context. Having such a framework has the following advantages. First, it characterizes the problem and the related solutions, and therefore provides us with insights into its theoretic nature. Second, it tells us how the two competing goals, completeness and efficiency, interplay and therefore suggests directions for further improvements over the existing solution.

Like the work in relational databases, most of the work on query rewriting using views for XML data are based on the concept of query containment of one kind or another. Recently, the research on query containment problem for XML queries has generated significant results. In [10], the authors study this problem in a limited class of XPath queries, which contains four kinds of symbols, /, //, [ ], *, and found that the problem of query containment is coNP-hard. In [6], the authors extend the results to the case where disjunctions, DTDs and some limited predicates are allowed.

There are two aspects for a general XPath query, *navigation script* and *tagging template* The navigation script guides the search for the required information while the tagging template provides the format for assembling the constructed document. Query rewritings for these two aspects are more or less orthogonal. In this paper, we study the framework for the rewriting for navigation script only, since this is the more vital and significant part of the two. We restrict our study also only to the subset of XPath queries that contains four kinds of symbols, /, //, [ ], *, as this can set up a foundation for further extension. We first provide a model for the problem, and then concentrate on the correctness problem. Our approach is based on the concept of query containment for XPath queries, and addresses the question of how that concept can be adopted to develop solutions to query rewriting problem. We study various conditions under which the efficiencies and applicability can trade each other at different levels, and introduce algorithms accordingly.

The rest of the paper is organized as follows. In Section 2, we first review some basic concepts, and then suggest a model for query rewriting problem. In Section 3, we present some solutions to query rewriting problem, and discuss the trade-offs between these solutions. We conclude the paper by summarizing the main results.

2 XPath Query and Rewriting

2.1 Pattern Trees and Input Trees

An XPath query can be denoted as a tree, called a *pattern tree*. Each node is attached with a label from an infinite alphabet, except for the root. The tree may contain branches, and can contain two kinds of edges, parent/child (denoted by single edges) and ancestor/descendent (denoted by double edges). If there is a child or descendant edge from n_1 to n_2, we say n_2 is a *C_child* or *D_child*, respectively, of n_1. Among all the nodes, there is a set of distinguished nodes, called *return nodes*. Although from the prescribed semantics, an XPath query should be considered to contain only a single return node, in this paper we do not restrict the number of return nodes to one. This will make the result applicable to more general query structures, such as those written in XQuery [18], where return nodes normally correspond to the last steps in path expressions, and accessing them (i.e., variable binding or referencing) triggers the creation of output nodes. (We use the convention that the root is always a return node.) Non-return nodes are called *transit nodes*.

XPath queries execute on XML document trees, which we will refer to as *input trees*. The execution proceeds by matching the nodes in the pattern tree to the nodes in the input tree. The following notations are from [10]. Let q be a pattern tree and t be an input tree. An *embedding* is a mapping e: nodes(q) $\rightarrow$ nodes(t) such that (1) for any node n $\in$ nodes(q), either label(n) = '*', or label(n) = label(e(n)) and (2) for any n_1, n_2 $\in$ nodes(q), if n_1 is a parent of n_2, then there is an edge from e(n_1) to e(n_2); if n_1 is an ancestor of n_2, then there is a path from e(n_1) to e(n_2). For each n $\in$ nodes(q), we say e matches n to e(n). Let return-nodes(q) = {n_1, …, n_k}. The set anws(q, t) = {{e(n_1), …, e(n_k)}: e is an embedding from nodes(q) to nodes(t)} is called the answer to q on t. We say that pattern trees *q is contained in* pattern tree *p* if anws(q, t) $\subseteq$ anws(p, t) for all t.

Let q be a pattern tree, and m be the number of descendant edges in q. Let $\vec{u}$ = <u_1, …, u_m>, where for all $1 \le i \le m$, u_i is a non-negative integer. The $\vec{u}$ - *extension of q with z* is an input tree, t_u, formed by modifying q as follows: replacing the label * with symbol z, and replacing the *i*th descendant edge, say ab, by a path $a\lambda_i b$ where λ_i contains u_i nodes, all labeled z. Note that λ_i is not originally in tree q. We call it the *ith guest path* in t_u. (Refer to Figure 2 for an example.) Thus except for the nodes in any guest path, all the nodes in a $\vec{u}$ -extension belong to the original q. To avoid ambiguity, we say that they are copies of those in q, and use symbol π for the mapping from the nodes in q to their copies in any of its $\vec{u}$ -extensions. If for all $1 \le i \le m$, $u_i = c$, i.e., all the guest paths contain equal number of nodes, c, then that $\vec{u}$ -extension is referred as a *c-extension*. For example, the tree in Figure 2.b is a 3-extension of the pattern tree in Figure 2.d. Call a path a *star-path* in a pattern tree if all its edges are child-edges and nodes are labeled *. For any tree or path t, we use |t| to denote the number of nodes it contains.

2.2 Query Rewriting

Regardless of the context, any technique for query rewriting uses substitution one way or another. The differences lie on the levels at which the substitution is made. For XML queries, most techniques apply the substitution implicitly at the pattern tree node level. The idea is, given pattern trees p and q, and an input tree to both, if the answer to be produced by p can be reproduced by q, then we can 'delegate' the task of p to q by rewriting the nodes of p in terms of the nodes of q [1]. The following two definitions formalize this idea.

Definition 1: Let p and q be pattern trees. A *rewriting of p by q* is a triplet *(p, q, h)* where *h*: return_nodes(p) $\rightarrow$ return_nodes(q) is called a *return node mapping* (RNM), p and q are respectively called *user* query and *view* query.

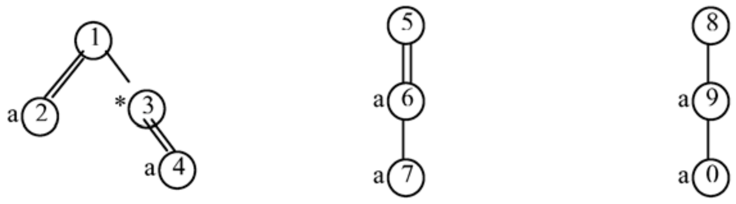
[1] Whether or not the answers reproduced should be complete is dictated by the correctness criteria for different applications. For example, if q is used to optimize the performance of p, then it must generate complete solutions. On the other hand, if q is used for the purpose of schema integration, it normally generates only partial answers.

In the above definition, the return node mapping is arbitrary. In particular, it does not have to be one-to-one or onto. This definition is applicable to any pair of pattern trees and RNMs. Our interest, however, is in a restricted class, as described below.

Definition 2: Rewriting (p, q, h) is *correct on an input tree t* if for all embedding e: nodes(q) $\rightarrow$ nodes(t), there is an embedding f: nodes(p) $\rightarrow$ nodes(t) such that f and $e \circ h$ are consistent on return-nodes(p) (i.e., for all n $\in$ return-nodes(p), f(n) = e(h(n))). If it is correct on all input trees, then we say it is *correct*.

Intuitively, a correct rewriting of p should produce the answers that are acceptable to p on any input. For example, consider the pattern trees in Figure 1.

Suppose return_nodes(p) = {1, 2, 4} and return_nodes(q) = {5, 6, 7}. Let the RNM h: return_nodes(p) $\rightarrow$ return_nodes(q) be defined as: h(1) = 5, h(2) = 6, h(4) = 7. We now show that the rewriting (p, q, h) is correct. Let t be any input tree, and e: nodes(q) $\rightarrow$ nodes(t) be an embedding.



a. pattern tree p b. pattern tree q c. input tree t_1

Fig. 1. An example for query rewriting

From the labels and the structure of q, we must have: e(5)) = root(t), label(e(6)) = a, label(e(7)) = a, there is a path λ from root(t) to e(6), and there is an edge ε from e(6) to e(7). Define g: nodes(p) $\rightarrow$ nodes(t) as: g(1) = root(t), g(2) = e(6), g(3) = child of root(t) in λ and g(4) = e(7). Clearly, g is an embedding, and g is consistent with $e \circ h$. This completes the proof.

Now we define another RNM as follows: $h_1(1) = 5$, $h_1(2) = 6$, $h_1(4) = 6$. It can be shown that the rewriting (p, q, h_1) is not correct. Indeed, consider the input tree in Figure c, and embedding e_1: nodes(q) $\rightarrow$ nodes(t_1) defined as $e_1(5) = 8$, $e_1(6) = 9$, $e_1(7) = 0$. Clearly, there does not exist an embedding that can match 2 to 9 and 4 to 9.

Note that for the same pair of queries, there may be more than one correct rewriting. For example, we can define h_2: return_nodes(p) $\rightarrow$ return_nodes(q) as: $h_2(1) = 5$, $h_2(2) = 7$, $h_2(4) = 7$. It can be easily shown that (p, q, h_2) is also a correct rewriting.

We now present some informal argument for the expressive power of our formulation. We argue that the condition in Definition 2 is the weakest one can assume conforming with the common notion used in practice for query rewriting, that is, substitution of view query for user query at the node level. First, note that, to be able to rewrite p using q, it is necessary that any result generated by q is acceptable by p. (This is a different way of saying that p contains q.) Although this assumption is weaker than ours, it nonetheless does not capture the idea of node substitution mentioned above. To capture that idea, additional component needs to be incorporated into the model to relate the view query nodes to the user query nodes in a manner that is independent of the input trees. This is way the RNM mapping is introduced in the above two definitions.

Now, there arise issues of how to determine efficiently if a given rewriting is correct and, given two pattern trees, how we find all the correct rewritings. We will study these issues in the subsequent sections.

3 Relating Query Containment to Query Rewriting

In the relational databases, query containment is the base for query rewriting. In this section, we study their relationship for XPath queries. To simplify the presentation, in this section when we mention rewriting (p, q, h) we assume implicitly that h is onto. This is because when the onto-condition is not met, we can always consider only the subset of the return nodes of q to which h is mapped. Then our results will follow without essential changes.

3.1 A Necessary and Sufficient Condition for Correct Rewriting

Query containment requires that any set of input nodes that are matched by the return nodes of the view query are also matched by the return nodes of the user query, while a correct query rewriting requires that this be true at the node level. This suggests that the latter is at least as strong a condition as the former. In the following theorem let L be the number of nodes in the longest star-path, and z be a label not used by any node in p.

Theorem 1[2]: A rewriting (p, q, h) is correct if and only if it is correct on the $\vec{u}$-extension of q for all $\vec{u} = <u_1, \ldots, u_m>$, where for all $1 \le i \le m$, $u_i \le L + 1$.

Proof: The 'only if' part is straightforward. We explain the idea for the proof of 'if' part. Let t be any input tree, and e: nodes(q) $\rightarrow$ nodes(t) be an embedding. Let $<a_i, b_i>$ be the *i*th descendant edge in q. It must be matched to a path $e(a_i)\lambda_i e(b_i)$ in t, where λ_i is a path in t with a length of possibly zero. Define $\vec{u} \equiv <v_1, \ldots, v_m>$, where for all $1 \le i \le m$, $v_i = |\lambda_i|$ when $|\lambda_i| \le L$, and $v_i = L+1$ when $|\lambda_i| \ge L+1$. Let this $\vec{u}$-extension of q with z be t_u. Denote by μ_i the ith guest path in t_u. Thus $|\mu_i| = v_i$ and all nodes in μ_i are labeled z. (Refer to Sec. 2.1.) By the assumption, (p, q, h) is a correct rewriting in t_u. Thus there is an embedding g: nodes(p) $\rightarrow$ nodes(t_u) such that for all n $\in$ return_nodes(p), g(n) = e(h(n)). We can define a mapping f: nodes(p) $\rightarrow$ nodes(t) in such a way that it is an embedding and for all n $\in$ return_nodes(p), f(n) = e(h(n)), as follows. If $g(n) \notin \mu_i$ for all $1 \le i \le m$, let $f(n) = e \circ \pi^{-1} \circ g(n)$. (Recall π maps the nodes in q to their copies in the $\vec{u}$-extension.) If $g(n) \in \mu_i$ for some $1 \le i \le m$, consider following two cases. (1) $|\mu_i| \le L$. In this case we let f(n) be the *j*th node in λ_i if g(n) is

[2] For this and the following theorems, we present only the informal argument. The formal proof is found in [16].

the *j*th node in μ_i. This is possible since $|\mu_i| = |\lambda_i|$. (2) $|\mu_i| = L+1$. In this case $|\mu_i| \le |\lambda_i|$. We have the following observations. First, since label(g(n)) = z, we have label(n) = *. Second, let α be the longest star-path in p such that $n \in \alpha$. By assumption, we have $|\alpha| \le L$. Thus at least one end node of α is incident with a descendant edge and is also matched to a node in μ_i, otherwise we would have $|\alpha| \ge |\mu_i| = L+1$, which is impossible. Because of this, we can let f match, one by one, all the nodes preceding the descendant edge that were previously matched to μ_i by g *onto a prefix* of λ_i, and all the nodes following the descendant edge that were previously matched to μ_i by g *onto a suffix* of λ_i. We call such a way of mapping 'prefix-suffix mapping'. Shown in Figure 2 is an example of prefix-suffix mapping. It can then be easily shown that the function f so defined is an embedding from p to t, and for all $n \in$ return_nodes(p), f(n) = e(h(n)). □

We have mentioned that the correct-rewriting problem is a subset of containment problem. A natural question is, how big is this subset? At this time, we do not have a definite answer. Rather than exploring the difference of the two, however, our interest is how we can solve the rewriting problem with the help of the solutions to the containment problem, and with what a price.

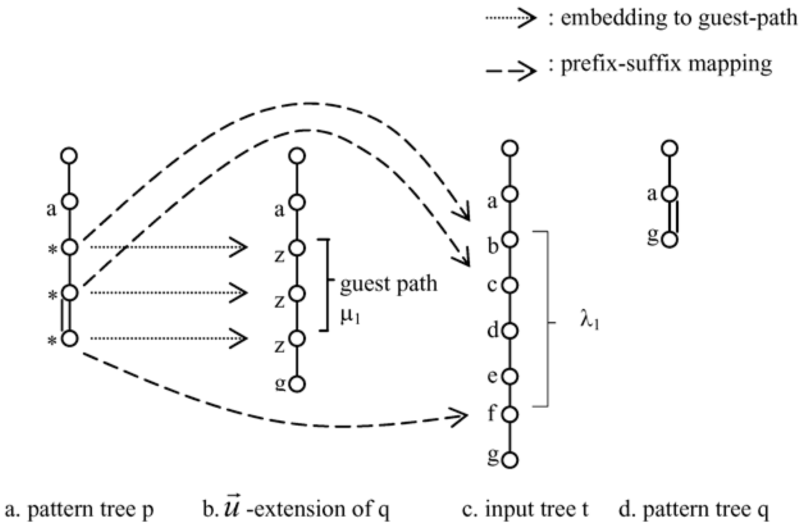


Fig. 2. Prefix-suffix mapping from p to λ_1, L = 2, $|\mu_1| = 3$, $|\lambda_1| = 5$

It is worth mentioning here that the RNM for a correct rewriting is not equivalent to homomorphism introduced in [10]. A homomorphism from p to q requires explicitly every node and edge in p to follow some structural pattern, depending on those in q, while a correct rewriting requires only some mapping from the return nodes in p to those in q that can meet the correctness criterion, without imposing structural constraints. It can be shown that homomorphism implies correct rewriting, but the reverse is not true [16].

Using Theorem 1, we can develop a sound and complete algorithm to determine if (p, q, h) is a correct rewriting. However, checking whether or not the rewriting is correct on all the $\vec{u}$-extensions of q specified in the theorem is intractable: there are $(L+1)^r$ $\vec{u}$-extensions to consider in total, where r is the number of descendant edges in q. In the subsequence sections, we will present alternative methods that can have better performance, at the price of stronger assumptions.

3.2 An Efficient Method

We observe that the main cost of the algorithm mentioned above results from a large number of $\vec{u}$-extensions that must be considered. By adding a little strong condition, we can reduce this number to one, as described by the following theorem.

Theorem 2: Assume any transit node of p labeled '*' is not incident with a descendant edge, L is the number of nodes in the longest star-path in p, and z is a label not used in p. Then the following three statements are equivalent:

1. p contains q.
2. There exists an embedding e: nodes(p) $\rightarrow$ nodes(t_{L+1}) such that e(return_nodes(p)) = π(return_nodes(q)), where t_{L+1} is the (L+1)-extension of q with z, and π maps the nodes in q to their copies in t_{L+1}.
3. There exists an RNM h: return_nodes(p) $\rightarrow$ return_nodes(q) such that (p, q, h) is a correct rewriting.

Proof:

1 $\Rightarrow$ 2: Clearly, π: nodes(q) $\rightarrow$ nodes(t_{L+1}) is an embedding. Since p contains q, by definition, there is an embedding e: nodes(p) $\rightarrow$ nodes(t_{L+1}) such that e(return_nodes(p)) = π(return_nodes(q)).

2 $\Rightarrow$ 3: Let h = $\pi^{-1} \circ$ e: nodes(p) $\rightarrow$ nodes(q). We will show that (p, q, h) is a correct rewriting. (In the triplet, h should be understood as restricted on return_nodes(p).) First note that for any node x $\in$ nodes(q), π(x) does not belong to any guest-path t_{L+1}. In particular, π(return_nodes(q)) is disjoint with any guest-path in t_{L+1}. We now prove the claim that for all n $\in$ nodes(p), e(n) does not belong to any guest-path in t_{L+1}. If n $\in$ return_nodes(p), then by assumption, e(n) $\in$ π(return_nodes(q)). The claim follows. Now assume n is a transit node. Assume the contrary, i.e., e(n) belongs to some guest-path, r. Let s be the longest star-path containing n whose nodes are *all* matched to r by e. Thus $|s| \leq L$. On the other hand, from the above arguments, all the nodes in s are transit nodes. By assumption they are incident only with child edges. Note that neither the node preceding s nor the node following s is matched to r. Thus we must have $|s| = |r| = L + 1$. This is impossible, implying e(n) cannot belong to any guest-path. Our claim follows. Now, let t be any input tree. Let g: nodes(q) $\rightarrow$ nodes(t) be an embedding. Consider mapping g$\circ$h: nodes(p) $\rightarrow$ nodes(t). Let o $\in$ nodes(p) be an arbitrary node. If label(o) $\neq$ '*', then label(o) = label(e(o)) $\neq$ 'z', implying label(e(o)) = label(π^{-1}(e(o))) $\neq$ '*'. Since g is an embedding, label(π^{-1}(e(o))) = label(g(π^{-1}(e(o)))). Thus label(g$\circ$h(o)) = label(g(π^{-1}(e(o)))) = label(o). Now let o_1, o_2 $\in$ nodes(p) be arbitrary

nodes. First assume there is a child edge from o_1 to o_2. Then there is an edge from $e(o_1)$ to $e(o_2)$. Since neither $e(o_1)$ nor $e(o_2)$ belongs to a guest-path, there is a child edge from $\pi^{-1}(e(o_1))$ to $\pi^{-1}(e(o_2))$, implying there is an edge from $g(\pi^{-1}(e(o_1)))$ to $g(\pi^{-1}(e(o_2)))$. Second, assume there is a descendant edge from o_1 to o_2. Then there is a path from $e(o_1)$ to $e(o_2)$. Again, since $e(o_1)$ and $e(o_2)$ are not in guest-paths, there must be a path from $\pi^{-1}(e(o_1))$ to $\pi^{-1}(e(o_2))$ (which may contain some guest-path as sub-path). This implies that there is a path from $g(\pi^{-1}(e(o_1)))$ to $g(\pi^{-1}(e(o_2)))$. We have proven that the mapping $g \circ h$ is an embedding. Note $\pi^{-1}(\pi(\text{return\_nodes}(q))) = \text{return\_nodes}(q)$. Since by assumption, $e(\text{return\_nodes}(p)) = \pi(\text{return\_nodes}(q))$, we have $\pi^{-1}(e(\text{return\_nodes}(p))) = \text{return\_nodes}(q)$, or $h(\text{return\_nodes}(p)) = \text{return\_nodes}(q)$. Let $n \in \text{return\_nodes}(p)$. We have $(g \circ h)(n) = g(h(n))$. Therefore $h = \pi^{-1} \circ e$ is the desired RNM.

$3 \Rightarrow 1$: This follows from that given any input tree t, (p, q, h) being correct on t implies $p \supseteq q$ on t. □

The assumption in Theorem 2 somewhat constrains the cases where the theorem can be used. It nonetheless covers many common cases. From the proof of the theorem, if statement 2 is true, and we know π and e, then we can construct a correct rewriting, i.e., $(p, q, \pi^{-1} \circ e)$. To determine π, we first construct t_{L+1}. This can be done in a single scan of the nodes and edges in q. For each node scanned, we create its correspondence in t_{L+1} consistent with π. Each time when a child edge is scanned in q, we create an edge in t_{L+1}, and when a descendant edge is scanned, we create a guest-path of length L+1. When we finish scanning q, the construction of t_{L+1} is completed.

We now look into the question of how to search for embeddings from p to t_{L+1} efficiently. We shall now present an algorithm that searches for embeddings in the general case. We first need a data structure to store the embeddings with the matching between return nodes annotated. We use the term 'sub-graph tree' to refer to a tree that is a sub-graph. (A sub-graph tree is therefore not necessarily a subtree.) We use the term *embedding-tree* to refer to a tree that stores embeddings. An Embedding tree consists of two kinds of nodes, P_nodes (for pattern tree) and I_nodes (for input tree). All the nodes are labeled the ids of the corresponding tree nodes. Parents and children must be of different kinds. The root is a P_node. If a P_node is labeled x and it has an I_node child labeled y, then there is an embedding that matches node x to node y. Each embedding is represented by a subgraph-tree that contains *all* the P_nodes and, for each P_node, *exactly one* I_node child. Shown on the left side of the double arrow in Figure 3 is the embedding tree that contains all the embeddings for the pattern tree and the input tree respectively in figures 1.a and 1.c.

The circles denote P_nodes and the rectangles denote I_nodes. On the left side of the double arrow is the embedding tree, which contains two embeddings, represented by the sub-graph trees on the right side of the double arrow.

Algorithm 1 below constructs an embedding-tree for all the embeddings from a given pattern tree to an input tree[3]. For simplicity, we will use the phrase 'return

[3] This is an extension of the one in [10], which generates only binary answers.

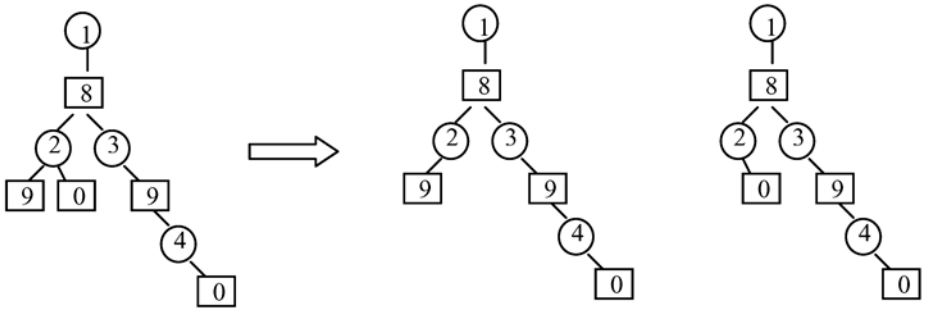


Fig. 3. An Embedding-tree

nodes' also to refer to those nodes in any $\vec{u}$ -extension of q that are copies of the return nodes in q.

Algorithm 1
Input: pattern tree p with root r_1, and a set R_1 of return nodes
Input tree t with root r_2, and a set R_2 of return nodes
Two dimensional arrays C and D, where each array entry takes as value a set of I_nodes, or U (i.e., undefined).
Output: the embedding tree containing the set of all embeddings: nodes(p) $\rightarrow$ nodes(t) that match r_1 to r_2

```
1  set every entry of C and D to U
2  insert as the root of the embedding tree a P_node $n_1$ for $r_1$
3  C_Build($r_1$, $r_2$, C[$r_1$, $r_2$])
4  if C[$r_1$, $r_2$] = $\Phi$ then
5      remove $n_1$
6      return              // no embedding found
7  create as the single child of $n_1$ the I_node C[$r_1$, $r_2$]
```

C_Build(PatternTreeNode x, InputTreeNode y, SetofInodes C[x, y])

```
1   if (label(x) $\neq$ * and label(x) $\neq$ label(y)) then C[x, y] $\leftarrow$ $\Phi$; return
2   if (x $\in$ return_nodes(p) and y $\notin$ return_nodes(t)) then C[x, y] $\leftarrow$ $\Phi$; return
3   for each C_child x' of x   //test if the sub-pattern rooted at x' can match a sub-tree
                               //rooted at a child of y
4       for each child y' of y
5           if C[x', y'] = U then C_Build(x', y', C[x', y']) //make the call only if no earlier
                                                             //call made on the same pair
6       if C[x', y'] = $\Phi$ for every child y' of y, then C[x, y] $\leftarrow$ $\Phi$; return   //no match
7   for each D_child x' of x   //test if the sub-pattern rooted at x' can match a sub-tree
                               //rooted at a descendant of y
8       for each child y' of y
9           if D[x', y'] = U then D_Build(x', y', D[x', y'])
10          if D[x', y'] = $\Phi$ for every child y' of y, then C[x, y] $\leftarrow$ $\Phi$; return   //no match
11  create an I_node $n_1$ labeled y            //can match, so store it in embedding tree
```

```
12  for each C_child x' of x                    //also store all the matches for each child
13    create a P_node $n_2$ as a new child of $n_1$
14    for each child y' of y
15      if C[x', y'] $\neq \Phi$ then let C[x', y'] be a new child of $n_2$ //C[x',y'] has a single member
16  for each D_child x' of x
17    create a P_node $n_3$ as a new child of $n_1$
18    for each child y' of y
19      if D[x', y'] $\neq \Phi$ then let D[x', y'] be a new set of children of $n_3$ //D[x',y'] may have
                                                                              //multiple members
20  C[x, y] = {$n_1$}; return
```

C_Build(x, y, C[x, y]) stores into entry C[x, y] an I_node for y, or Φ, depending on whether or not the sub-pattern rooted at x can match the subtree rooted at y in such a way that all the return nodes in the sub-pattern are matched to the return nodes in the subtree. If the match is successful, it also creates the P_node children for the I_node, one for each child of x (lines 13 and 17). These P_nodes in turn have their I_node children (lines 15 and 19), storing the matching for the children of x. Line 5 checks and see if C[x', y'] is set by some earlier calls. This may happen due to the presence of the loop in line 22 in D_Build( ), whose pseudo-code is shown below.

D_Build(PatternTreeNode x, InputTreeNode y, SetofINodes D[x, y])

```
1   L $\leftarrow \Phi$
2   if (label(x) $\neq$ * and label(x) $\neq$ label(y)) then go to 22    //cannot match y, let's turn
                                                                  // to its descendants
3   if (x $\in$ return_nodes(p) and y $\notin$ return_nodes(t)) then go to 22
4     for each C_child x' of x
5       for each child y' of y
6         if C[x', y'] = U then C_Build(x', y', C[x', y']))
7         if C[x', y'] = $\Phi$ for every child y' of y,  then go to 22
8       for each D_child x' of x
9         for each child y' of y
10          if D[x', y'] = U then D_Build(x', y', D[x', y'] )
11          if D[x', y'] = $\Phi$ for every child y' of y, then go to 22
12      create an I_node $n_1$ labeled y
13    for each C_child x' of x
14      create a P_node $n_2$ as a new child of $n_1$
15      for each child y' of y
16        if C[x', y'] $\neq \Phi$ then let C[x', y'] be a new child of $n_2$
17    for each D_child x' of x
18      create a P_node $n_3$ as a new child of $n_1$
19      for each child y' of y
20        if D[x', y'] $\neq \Phi$ then let D[x', y'] be a new set of children of $n_3$
21    L $\leftarrow$ {$n_1$}
22  for each z $\in$ children(y)  //recursively determine if x can match the descendants of y
23    if D[x, z] = U then D_Build(x, z, D[x, z])
24    L $\leftarrow$ L $\cup$ D[x, z]
25  D[x, y] $\leftarrow$ L; return
```

D_Build(x, y, D[x, y]) stores into entry D[x, y] a set of I_nodes if the sub-pattern rooted at x can match the subtree rooted at y and/or y's descendants, such that the return nodes are matched to the return nodes, and Φ otherwise. The way it stores the matching information for the children of x is identical to that in C_Build( ).

For the time complexity of the algorithm, notice that for any pair of a node in p and a node in t, once the corresponding entry in C or D array is set by a call, then no later calls will be made on the same pair. This means all the calls are made on different pairs. Thus the total number of calls is at most $|p|\bullet|t|$, implying a time complexity of $O(|p|\bullet|t|)$. (Note that once an embedding is returned, we need to check further if it maps the return nodes of p *onto* the return nodes of q. This can be done by simply comparing |e(return_nodes(p)| and |return_nodes(q)|, and incurs only a linear time complexity.)

3.3 A Weaker Condition

Correct rewriting imposes a fixed relationship between the return nodes of p and their RNM correspondences in q. This suggests that the paths delimited by the return nodes in p and those by their RNM correspondences in q may need to follow some patterns. In this section, we will look at this in detail. In the following, for any pattern tree or input tree, we use the notation $<a_1, \ldots, a_m>$ to refer to a path that starts with node a_1 and ends at node a_m. Note that this notation does not contain information about the kind of edges connecting the adjacent nodes when the path is in a pattern tree. Also, for any two paths λ and μ, and mapping g: nodes(λ) $\rightarrow$ nodes(μ), we use the notation $g(\lambda) = \mu$ to indicate that g maps respectively the beginning and the end nodes of λ to the beginning and the end nodes of μ, and preserves the order of any two nodes in λ when it maps them to different nodes in μ.

Definition 3: Let $\lambda = <x_1, \ldots, x_m>$ and $\mu = <y_1, \ldots, y_n>$. We say e: nodes(λ) $\rightarrow$ nodes(μ) is a *relay* from λ to μ, denoted as $e_{relay}(\lambda) = \mu$ (or simply $e_{relay}(\lambda)$), if $e(\lambda) = \mu$, and the following conditions hold true:

1. $m = 1$, $n = 1$ and (label(x_1) = * or label(x_1) = label(y_1)), or
2. $m = 2$, $n = 2$, $e_{relay}(x_1) = y_1$, $e_{relay}(x_2) = y_2$, there is a child edge from x_1 to x_2, and there is a child edge from y_1 to y_2, or
3. $2 \le m \le n$, $e_{relay}(x_1) = y_1$, $e_{relay}(x_m) = y_n$, each node in sub-path $<x_2, \ldots, x_{m-1}>$ is labeled *, does not branch, and path $<x_1, \ldots, x_m>$ contains at least one descendant edge, or
4. there is $1 \le i \le m$, $1 \le j \le n$, such that $e_{relay}(<x_1,\ldots,x_i>)=<y_1,\ldots,y_j>$, and $e_{relay}(<x_i,\ldots, x_m>)=<y_j,\ldots,y_n>$

The idea is that for any embedding g from μ to an input path, we can compose e_{relay} and g to form an embedding from λ to the same input path. This is clearly the case when conditions 1 and 2 are met. When condition 3 is met, we retain the mapping for the two end nodes of λ, but apply the prefix-suffix mapping introduced in Section 3.1 for the inner nodes if necessary. Condition 4 simply makes the definition recursive. Consider Figure 4.

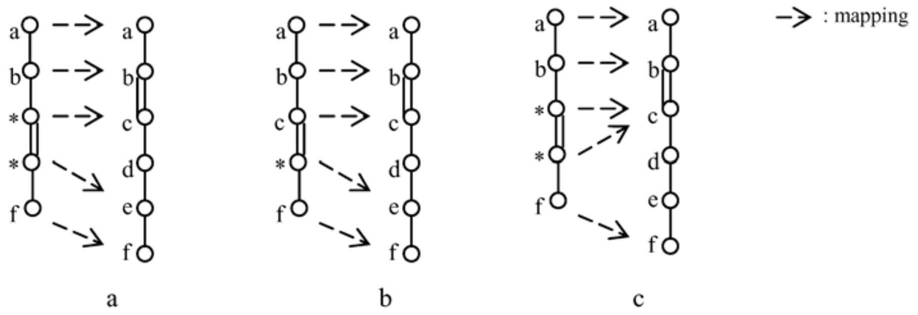


Fig. 4. a and c: relay, b: not relay

In Figure 4.a, the mapping is a relay, since the path on the left can be decomposed into two sub-paths, <a, b> and <b, f>. The former meets condition 2, and the latter meets condition 3. Then by applying condition 4 the claim follows. To see how condition 3 meets our intuition in this instance, suppose an embedding g matches path <a, f> on the right to an input path γ, and matches descendant edge <b, c> to the sub-path <b, c> of γ. Then the mapping shown in the figure as is can be composed with g to form an embedding from the path on the left to γ. On the other hand, if g matches <b, c> to sub-path <b, x, c> of γ, we directly match the upper node labeled * on the left to node x, and keep the rest of the composition unchanged. This still results in an embedding from the path on the left to γ. In general, no matter what the input path is, we can form an embedding from the path on the left to it, as long as there is an embedding from the path on the right to it. This is the idea behind the notion of relay.

Similarly, the mapping in Figure 4.c is also a relay. The mapping in Figure 4.b, however, is not a relay. This is because the path on the left cannot be decomposed in accordance with condition 4. Thus, for example, if the right path is matched to input path γ but the descendant edge <b, c> in it is matched to sub-path <b, x, c> of γ, it is not possible to form an embedding from the left path to γ.

In the general case, we have the following lemma.

Lemma 1. Let λ and μ be two pattern tree paths, and g(λ) = μ be an arbitrary relay. Let t be an input path and e: nodes(μ)→nodes(t) be an embedding such that e(start-node(μ)) = start-node(t) and e(end-node(μ)) = end-node(t). Then there is an embedding f: nodes(λ)→nodes(t) such that f(start-node(λ)) = start-node(t) and f(end-node(λ)) = end-node(t).

Idea of Proof: If g meets conditions 1 or 2, we simply let f = e ∘ g. If e meets condition 3, then we can perform prefix-suffix mapping from the inner nodes of λ to the inner nodes of t, resulting in an embedding. If we have to decompose λ according to condition 4, then we can apply the above procedure to the resultant sub-paths. □

Based on the above lemma, we have the following

Theorem 3: Let p and q be pattern trees. If there is a mapping g: nodes(p) → nodes(q) satisfying the following conditions: (1) g(return_nodes(p)) = return_nodes(q), (2) for all path λ in p in which the beginning and ending nodes are return or leaf nodes, and the remaining are transit nodes, g is a relay, then (p, q, g) is a correct rewriting.

Proof: By condition 1, g can match return nodes in p only to return nodes in q. Let t be an input tree, and e: nodes(q) $\rightarrow$ nodes(t) be an embedding . Let λ be an arbitrary return-or-leaf-node-delimited path in p and $g(\lambda) = \mu$ where μ is a path in q. Let n be whichever delimiting node of λ that is a return node. By condition 1 in the theorem we have g(n) is also a return and delimiting node of μ. Applying lemma 1 to λ, we obtain an embedding that matches λ to $e(\mu)$, and n to e(g(n)). Note that since any inner node in λ does not branch, which is required by condition 3 in Definition 3, the possible prefix-suffix mapping performed for the inner nodes of λ will not affect the embeddings for other paths in p. Thus the embedding obtained collectively for all such paths is indeed an embedding from p to t. □

Is Theorem 3 weaker than Theorem 2? The answer is yes. The following is why. Suppose Condition 2 in Theorem 2 is true. Let $h = \pi^{-1} \circ e$: nodes(p) $\rightarrow$ nodes(q). The arguments in the proof of Theorem 2 have shown that for all o $\in$ nodes(p), label(o) $\neq$ * $\Rightarrow$ label(o) = label(h(o)), and for all $n_1, n_2 \in$ nodes(p), if there is a child edge from n_1 to n_2, then there is a child edge from $h(n_1)$ to $h(n_2)$, Thus the first two conditions in Definition 3 are true. This means h is a relay for any path in p. The proof for Theorem 2 also shows h(return_nodes(p)) = return_nodes(q). Together, these imply the two conditions in Theorem 3 (with h substituting for g). On the other hand, the conditions in Theorem 3 do not imply those in Theorem 2: they do not require that transit nodes labeled * not be incident with descendant edges. This means the assumption in Theorem 3 is strictly weaker than that in Theorem 2.

Theorem 3 gives an approach to searching for correct rewriting, i.e., searching for relays. In reality, we can narrow the search space by discarding “replicas”. Note that it is possible that multiple relays are identical when they are restricted to return nodes. When this happens, we need to consider only one of them.

Theorem 4: Let λ and μ be two pattern tree paths, and $g(\lambda) = \mu$ be a relay. Then there is a relay $f(\lambda) = \mu$ such that $\pi \circ f$: nodes(λ)$\rightarrow$ nodes(t_0) is an embedding, where t_0 is the 0-extension of μ and π maps each node in μ to its copy in t_0. (Refer to Sec. 2.1.)

Idea of Proof: If g meets condition 1 or 2, let f = g. In this case t_0 is identical to μ, since μ does not contain descendant edges. Thus $\pi \circ f$ is an embedding. If g meets condition 3, we obtain f by performing prefix-suffix mapping from the inner nodes of λ to the inner nodes of μ. In this case t_0 is identical to μ., except that in the place of each descendant edge in μ is an edge of t_0. Since prefix-suffix mapping always maps a child edge in λ to an edge in μ, which is also an edge in t_0, $\pi \circ f$ is surely an embedding. Note that f is still a relay from λ to μ, since prefix-suffix mapping does not alter condition 3. In case we need to decompose λ according to condition 4, we apply the above procedure to the resultant sub-paths of λ. □

Note that in Theorem 4, if n is an end node of λ, then f(n) = g(n). Thus if we want to find a relay that satisfies the conditions in Theorem 3, we can consider f only. For this purpose, according to Theorem 4 also, we can first find all the embeddings from λ to t_0, then determine those that can be transformed to relays from λ to μ. Searching for embeddings can be done by applying Algorithm 1. The transformation is done simply by applying π^{-1} to the embeddings. (In the following algorithm this is done implicitly.)

In the following algorithm, we use the term 'D_edge' to refer to an edge in t_0 that corresponds to the descendant edge in q. We term a path *segment* that is delimited by return or leaf nodes in p.

Algorithm 2

Input: pattern tree p; input tree t_0, with a set of D_edges; an embedding e: nodes(p) $\rightarrow$ nodes(t_0)

Output: a decision of whether or not e is a relay for all the segments in p

```
1   s ← next segment
2   if s = NULL return 'yes'
3   if ¬Relay(s, e) return 'no'
4   go to 1
```

Boolean Relay(Segment s, Embedding e)

```
1   <a, b> ← first child edge ε in s such that e(ε) is a D_edge;
2   if <a, b> = NULL then return 'yes'
3   if label(b) ≠ * then return 'no'
4   c ← first successor of b in s that is incident with a descendant edge and delimits
        a star-path with no branching
5   if c = NULL, then return 'no'
6   d ← child of c
7   if Relay(<d, …, end-node(s)>, e) then return 'yes'
8   return 'no'
```

The idea should be clear. Each child edge in the prefix <start-node(s), a> is not matched to a D_edge, thus we have e_{relay}(<start(s), a>) is true by Condition 2 and 4 in Definition 3. If the test in line 5 evaluates to true, then there does not exist a sub-path starting from *a* that meets any condition in Definition 3, else path <a, …, d> meets Condition 3, e_{relay}(<a,…,d>) is true. Thus e_{relay}(start(s), d) is true by Condition 4. This means e_{relay}(s) is true iff e_{relay}(<d, …, end(s)>) is true. For example, when we apply the algorithm to the left path in Figure 4.a, two recursive calls will be made. The top call is on the entire path, and the nested call is on the path containing the bottom two nodes.

For time complexity, it is easy to see that on any path, Relay( ) returns in O(m) time where m is the number of nodes in the path, implying that the algorithm runs in O(n) where n is the number of nodes in p. Thus, to find all the correct rewriting based on Theorem 3, in addition to the cost of running Algorithm 1 on p and t_0, we need to pay an extra cost of O(w•n), where w is the number of embeddings from nodes(p) to nodes(t_0). In the general case, w is much smaller than n. Therefore, this extra cost is close to linear. Note that, if Algorithm 2 returns 'no' for all the embeddings, this does not necessarily mean that there does not exist a correct rewriting for p and q. This is because the conditions in Theorem 3 are not necessary conditions. When that happens, we may need to resort to the method based on Theorem 1 for the final judgment. (Refer to the discussion in the next section.)

4 Conclusion

We study the issue of query rewriting using views for XPath queries in a general setting. Several issues are studied, including conditions for correct query rewriting, search methods, and trade-offs between efficiency and applicability. Our solution can be used as a basis for developing solutions suited to special requirement. For example, for small queries, the method based on Theorem 1 should be used, as it is both sound and complete. If a query is large, but has no transit node with a wildcard label that is associated with a descendant edge, then the method based on Theorem 2 is the best, since it is both sound and complete (under that condition), as well as efficient. These methods can also be used in a hybrid manner. For example, if the condition in Theorem 2 is not met, we use the method based on Theorem 3. If it returns ‘no’, we then use Theorem 1.

There are several related issues. Suppose there does not exist a correct rewriting for p and q. (This is the case, for example, when p does not contain q.) How do we search efficiently for another query $q' \subset q$ such that there exists a correct rewriting for p and q’? Another issue is the extension of the model to incorporate more features of XPath queries. These issues can be viewed as immediate follow ups of the work in this paper, and deserve further study.

References

1. F. Ozcan, K. Beyer and R. Cochrane, “A Framework for Using Materialized XPath Views in XML Query Processing”, In 30th VLDB Conf., 2004, pp 60 – 71.
2. D. Calvanese, G. Giacomo, M. Lenzerini and M. Vardi, “Answering Regular Path Queries Using Views”, In 16th Intl. Conf. On Data Engg., 2000, pp 389 - 398.
3. L. Chen and E. Rundensteiner, “ACE-XQ: A Cache-Aware XQuery Answering System”, In WebDB, pp 31 – 36.
4. V. Cristophides, S. Cluet and J. Simeon, “On Wrapping Query Languages and Efficient XML Integration”, In SIGMOD Conf., 2000, pp141 – 152.
5. A. Deutsch and V. Tannen, “Reformulation of XML Queries and Constraints”, In 9th Intl. Conf. on Database Theory, 2003, pp 225 – 241.
6. F. Neven and T. Schwentick, “XPath Containment in the Presence of Disjunction, DTDs and Variables”, In 9th Intl. Conf. on Database Theory, 2003, pp 315 – 329.
7. G. Grahne and A. Thomo, “Query Containment and Rewriting Using Views for Regular Path Queries Under Constraints, In 22nd PODS, 2003, pp 111 – 121.
8. T. Kirk, A. Levy, Y. Sagiv and D. Srivastava, “The Information Manifold”, AAAI Spring Sym. on Information Gathering from Heterogeneous and Distributed Environments, 1995.
9. A. Levy, A. Mendelzon, Y. Sagiv and D. Srivastava, “Answering Queries Using Views”, In 14th PODS, 1995, pp 95 – 104.
10. G. Miklau and D. Suciu, “Containment and Equivalence for an Xpath Fragment”, In 21st PODS, 2002, pp 65 - 76
11. P. Mitra, “An Algorithm for Answering Queries Efficiently Using Views”, In 12th Australian Database Conf., 2001, pp 99 – 106.
12. Y. Papakonstantinou and V. Vassalos, “Query Rewriting Using Semistructured Views”, In SIGMOD Conf., 1999, pp 455 – 466.

13. R. Pottinger and A. Levy, “A Scalable Algorithm for Answering Queries Using Views”, VLDB Journal, 10(2-3), 2001, pp 182 - 198.
14. X. Qian, “Query Folding”, In 12th Intl. Conf. On Data Engg., 1996, 48 – 55.
15. J. Shanmungasundaram, J. Kiernan, E. Shekita, C. Fan and J. Funderburk, “Querying XML Views of Relational Data”, In 27th VLDB Conf., 2001, pp 261 – 270.
16. J. Tang and S. Zhou, “Rewriting Queries Using Views for XML Documents”, TR-04, MUN, 2004.
17. Yu and L. Popa, “Constraint-Based XML Query Rewriting for Data Integration”, In SIGMOD Conf., 2004, pp 371 – 382.
18. XQuery: A Query Language for XML, http://www.W3.org/TR/xquery, 2003.

A Path-Based Labeling Scheme for Efficient Structural Join

Hanyu Li, Mong Li Lee, and Wynne Hsu

School of Computing, National University of Singapore
3 Science Drive 2, Singapore 117543
{lihanyu,leeml,whsu}@comp.nus.edu.sg

Abstract. The structural join has become a core operation in XML query processing. This work examines how path information in XML can be utilized to speed up the structural join operation. We introduce a novel approach to pre-filter path expressions and identify a minimal set of candidate elements for the structural join. The proposed solution comprises of a path-based node labeling scheme and a path join algorithm. The former associates every node in an XML document with its path type, while the latter greatly reduces the cost of subsequent element node join by filtering out elements with irrelevant path types. Comparative experiments with the state-of-the-art holistic join algorithm clearly demonstrate that the proposed approach is efficient and scalable for queries ranging from simple paths to complex branch queries.

1 Introduction

Standard XML query languages such as XQuery and XPath support queries that specify element structure patterns and value predicates imposed on these elements. For example, the query "//dept[/name='CS']//professor" retrieves all the professors from the CS department. It comprises of a value predicate "name='CS' " and two structural relations "dept//professor" and "dept/name" where '/' and "//" denote the child and descendant relationships respectively.

Traditional relational database access methods such as the B^+-tree can be easily extended to process value predicates in XML queries. Hence, the support of tree structured relationships becomes the key to efficient XML query processing.

Various node labeling schemes have been developed to allow the containment relationship between any two XML elements to be determined quickly by examining their labels or *node ids*. [4] identifies the structural join as a core operation for XML query pattern matching and develops a structural join algorithm called *Stack-Tree* which utilizes the interval-based node labeling scheme to evaluate the containment relationship of XML elements. Index-based methods such as B^+-tree [7], XR-tree [11], and XB-tree [5] speed up the structural join operation by reducing the number of elements involved in the node join.

In this work, we design a novel path-based approach to further expedite the structural join operation. The idea is to associate path information with the element nodes in an XML document so that we can filter out nodes that clearly do not match the query, and identify a minimal set of nodes for the regular node join. The proposed approach has the following unique features and contributions:

1. *Path Labeling Scheme.* We design a path-based labeling scheme that assigns a *path id* to every element to indicate the type of path on which a node occurs. The scheme is compact, and the path ids have a much smaller size requirement compared to the node ids (see Section 5 on space requirement).
2. *Containment of Path Ids.* The well-known node containment concept allows the structural relationship between any two nodes in an XML document to be determined by their node labels. Here, we introduce the notion of *path id containment* and show how the path labeling scheme makes it easy to distinguish between parent-child and ancestor-descendant relationships.
3. *Path Join.* We design a path join algorithm as a preprocessing step before regular node join to filter out irrelevant paths. The path join algorithm associates a set of relevant path ids to every node in the query pattern, thus identifying the candidate elements for the subsequent node join. Experimental results indicate that the relatively inexpensive path join can greatly reduce the number of elements involved in the node join.

The rest of this paper is organized as follows. Section 2 briefly reviews the related work. Section 3 presents the path-based labeling scheme. Section 4 describes the query evaluation. Finally, Section 5 presents the experimental results, and we conclude in Section 6.

2 Related Work

There has been a long stream of research in XML query evaluation. Early works develop various path index solutions such as DataGuides [10] and 1-Index [14] to summarize the raw paths starting from the root node in an XML document. These index structures do not support branch queries and XML queries involving wildcards and ancestor-descendant relationships efficiently. Index Fabric [9] utilizes the index structure Patricia Trie to organize all the raw paths, and provides support for the "refined paths" which frequently occur in the query workload.

The work in BLAS [6] also utilizes path information (p-labeling) to prefilter out unnecessary elements. BLAS employs integer intervals to represent all the possible paths, regardless of whether or not they occur in the XML dataset. Hence, BLAS will perform best for suffix queries, i.e., queries that start with optional descendant axis followed by child axes. In contrast, the proposed path encoding scheme utilizes bit sequences to denote the paths that actually occur in the XML datasets. Therefore, the proposed solution will yield optimal performance for simple queries, which are a superset of suffix queries.

The structural join is a core operation in many XML query optimization methods [4, 5, 7, 11–13]. [13] uses a sort-merge or a nested-loop approach to process the structural join. This method may scan the same element sets multiple times if the XML elements are recursive. *Stack-Tree* [4] solves this problem by applying an internal stack to store the subset of elements that is likely to be used later. Index-based binary structural join solutions such as B^+-tree [7], one dimensional R-tree [7], XB-tree [5] and XR-tree [11] employ different ways to "skip" elements involved in the query pattern without missing any matches. Holistic twig join methods such as XB-tree based TwigStack [5] and XR-tree based TSGeneric [12] are designed to process XML queries involving more than two nodes.

Sequence based approaches such as ViST [16] and PRIX [15] apply different ways to transform both the XML documents and queries into sequences. Query evaluation is carried out using sub-sequence matching. However, ViST may produce false alarms in the results, and PRIX requires a substantial amount of post-processing to guarantee the accuracy of the query results.

3 Path-Based Labeling Scheme

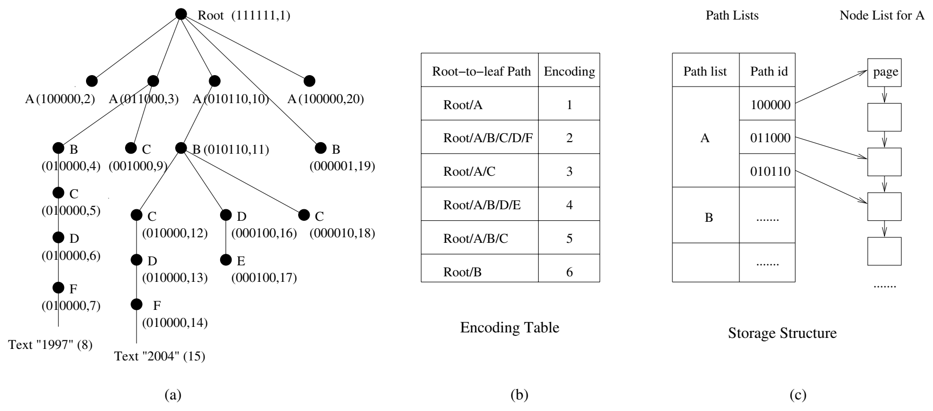
We design a path-based labeling scheme that assigns a *path id* to every element node in an XML document to indicate the type of path on which the node occurs. Each element node is now identified by a pair of (path id, node id). The node id can be assigned using any existing node labeling scheme, e.g., interval-based [13], prefix [8], prime number [17]. Text nodes are labeled with node ids only.

A path id is composed of a sequence of bits. We first omit the text nodes from an XML document. Then we find distinct root-to-leaf paths in the XML document by considering only the tag names of the elements on the paths. We use an integer to encode each distinct root-to-leaf path in an XML document. The number of bits in the path id is given by the number of the distinct root-to-leaf element sequences of the tag names that occur in the XML document. Path ids are assigned to element nodes in a bottom-up manner as follows.

1. After omitting the text nodes in an XML document, let the number of distinct root-to-leaf paths in the XML document be k. Then the path id of an element node has k bits. These bits are initially set to 0.
2. The path id of every leaf element node is given by setting the *ith* bit (from the left) to 1, where i denotes the encoding of the root-to-leaf path on which the leaf node occurs.
3. The path id of every non-leaf element node is given by a bit-or operation on the path ids of all its element child nodes.

Consider the XML instance in Figure 1(a) where the node ids have been assigned using a pre-order traversal. Figure 1(b) shows the integer encodings of each root-to-leaf paths in the XML instance. Since there are 6 unique root-to-leaf paths, 6 bits are used for the path ids.

The path id of the element leaf node F (node id = 7) is 010000 since the encoding of the path $Root/A/B/C/D/F$ on which F occurs is 2. The path id of the non-leaf node A (node id = 3) is obtained by a bit-or operation on the path ids of its child nodes B and C, whose path ids are 010000 and 001000 respectively. Therefore, the path id of A is 011000. Note that each text node is only labeled by a node id.



Root–to–leaf Path	Encoding
Root/A	1
Root/A/B/C/D/F	2
Root/A/C	3
Root/A/B/D/E	4
Root/A/B/C	5
Root/B	6

Fig. 1. Path-Based Labeling Scheme and Its Storage Structure

3.1 Storage Structure

In order to facilitate the direct retrieval of elements with a specified path id, we design the following storage structure:

1. All the path ids of one element tag comprise the path id list of this element.
2. All the node ids of one element tag comprise the node id list of this element. The node list is first clustered by element path ids, and then sorted on the node ids.
3. Each path id in the path id list points to the first element with this path id.

Figure 1(c) shows how the example XML document in Figure 1(a) is stored. Tag A has four occurrences with three distinct path ids. There are two occurrences corresponding to the first path id (100000) and one each corresponding to the other two (011000 and 010110).

3.2 Containment of Path IDs

In this section, we introduce the notion of path id containment that is based on the proposed path labeling scheme and examine the relationship of path id containment with node containment.

Definition (Path ID Containment): Let Pid_A and Pid_B be the path ids of nodes A and B respectively. If all the bits with value 1 in Pid_A cover all the bits with value 1 at corresponding positions in Pid_B, then we say Pid_A contains Pid_B.

The containment relationship between the path ids can simply be determined with a bit-and operation. That is, if $(Pid_A \ \& \ Pid_B) = Pid_B$ where & denotes the "bit-and" operation, then Pid_A contains Pid_B. For example, in Figure 1, the path id of A(010110,10) contains the path id of C(010000,12).

Definition (Strict Path ID Containment): Let Pid_A and Pid_B be the path ids of nodes A and B respectively. If Pid_A contains Pid_B and $Pid_A \neq Pid_B$, then we say Pid_A strictly contains Pid_B.

In Figure 1, the path id of A (010110,10) strictly contains the path id of C (010000,12).

The node containment between nodes can be deduced from path id containment relationship. Theorem 1 introduces this.

Theorem 1: Let Pid_A and Pid_D be the path ids for elements with tags A and D respectively. If Pid_A strictly contains Pid_D, then each A with Pid_A must have at least one descendant D with Pid_D.

Proof: Since Pid_A strictly contains Pid_D, then all the bits with value 1 in Pid_D must occur in Pid_A at the same positions. Further, Pid_A will have at least one bit with value 1 such that the corresponding bit (at the same position) in Pid_D is 0. Consequently, elements with tag A will occur in the same root-to-leaf paths as the elements with tag D, and there will exist at least one root-to-leaf path such that elements with tag A occur, and elements with tag D do not occur. As a result, all the elements with tag A must have elements with tag D as descendants. □

Consider again Figure 1. The path id 010110 for node B strictly contains the path id 010000 for node C. Therefore, each node B (node id=11) with path id 010110 must be the ancestor of at least one node C (node id=12) with path id 010000.

In the case where there are two sets of nodes with the same path ids, we need to check their corresponding root-to-leaf paths to determine their structural relationship. For example, the nodes A and B (node ids are 10 and 11) in Figure 1 have the same path id 010110. We can decompose the path id 010110 into 3 root-to-leaf paths with the encodings 2, 4 and 5 since the bits in the corresponding positions are 1. Thus, by looking up any of these paths (in the encoding table) where nodes A and B occur, we know that all the nodes A with path id 010110 have B descendants with this path id.

Similarly, the encoding table can help us to determine the exact containment relationship between any two sets of nodes, that is, parent-child, grandparent-grandchild, etc. For example, given the path id 010110 for nodes B and the path id 010000 for nodes C, we know that all the B nodes have C descendants. Further, from the root-to-leaf path with value 2, we also know that the B nodes are parents of the corresponding C nodes.

In other words, if Pid_A and Pid_D are the path ids of two sets of nodes A and D respectively, then we can determine the exact relationship (parent-child, grandparent-grandchild..) between these two sets of nodes from the encoding table, provided that a tag name occurs no more than once in any path. For example, suppose nodes A and D have the same path ids, and their corresponding root-to-leaf path is "A/D/A/D". In this case, the structural relationship between A and D can only be determined by examining their node labels (node ids).

4 Evaluation of Structural Join

The structural join operation evaluates the containment relationship between nodes in given XML queries. Our path-based approach processes structural join in two steps: (1) path join, and (2) node join. The algorithms for carrying out these two steps are called *PJoin* and *NJoin* respectively.

4.1 *PJoin*

The *PJoin* algorithm (Algorithm 1) aims to eliminate as many unnecessary path types as possible, thus minimizing the elements involved in the subsequent *NJoin*.

Given an XML query modelled using a tree structure T, *PJoin* will retrieve the set of path ids for every element node in T. Starting from element leaf nodes in T, *PJoin* will perform a binary path join between each pair of parent-child nodes. This process is carried out in a bottom-up manner until the root node is reached. After that, a top-down binary path join is performed to further remove unnecessary path ids.

A binary path join takes as input two lists of path ids, one for the parent node and the other for the child node. A nested loop is used to find the matching pairs of path ids based on the path id containment property. Any path id that does not satisfy the path id containment relationship is removed from the lists of path ids of both parent and child nodes.

```
Algorithm 1 PJoin (T)
  Input: T - An XML Query.
  Output: Path ids for the nodes in T.

  1. Associate every node in T with its path ids.
  2. Perform a bottom-up binary path join on T.
  3. Perform a top-down binary path join on T.
```

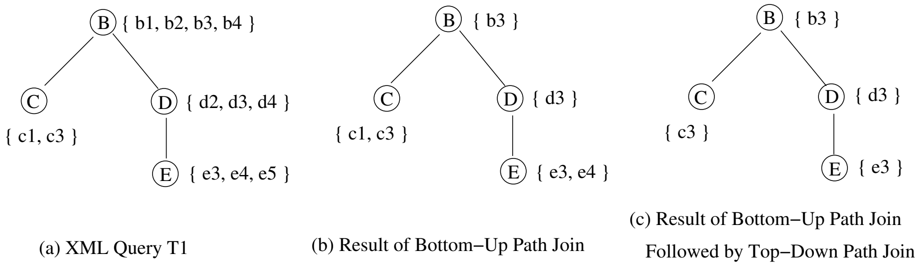


Fig. 2. Example of *PJoin*

Consider the XML query $T1$ in Figure 2(a) where the lists of path ids have been associated with the corresponding nodes. We assume that the path ids with the same subscripts satisfy the path id containment relationship, that is, b_1 contains c_1, b_3 contains d_3 and e_3, etc.

The *PJoin* algorithm evaluates the query $T1$ by first joining the path ids of node B with that of node C. The path id c_1 and c_3 are contained in the path id b_1 and b_3 respectively. Thus, we remove b_2 and b_4 from the set of path ids of B.

Next, the algorithm joins the set of path ids of D with that of E. This is followed by a join between the sets of path ids of B and D. The result of the bottom-up path join is shown in Figure 2(b).

Finally, the algorithm carries out a top-down path join on $T1$ starting from the root node B. Figure 2(c) shows the final sets of path ids that are associated with each node in $T1$. Compared to the initial set of path ids associated with each node in Figure 2(a), the $PJoin$ algorithm has greatly reduced the number of elements involved in the query. The subsequent node join is now almost optimal.

Note that omitting either the bottom-up or top-down tree traversal will not be able to achieve this optimal result. This is because a single tree traversal cannot project the result of each binary path join to the nodes which have been processed earlier. In Figure 2(b), elements C and E contains unnecessary path ids $c1$ and $e4$ compared to the final optimal results.

4.2 *NJoin*

The output of $PJoin$ algorithm is a set of path ids for the element nodes in a given query tree. Elements with these path ids are retrieved for a node join to obtain the result of the query. Algorithm 2 shows the details of $NJoin$.

We modify the holistic structural join developed in [5] to perform the node join. The element nodes are retrieved according to the path ids obtained from the $PJoin$, while all the value (text) nodes are retrieved directly. Finally, a holistic structural join is carried out on all the lists obtained.

Algorithm 2 $NJoin(T)$

```
Input: $T$ - An XML Query.
Output: All occurrences of nodes in $T$.

1. Retrieve the elements according to the path ids associated with nodes in $T$
2. Retrieve the values imposed on the element nodes.
3. Perform holistic structural join on $T$.
```

We observe that the structural join in Line 3 of Algorithm 2 requires that the input stream for every node in the query must be an ordered list of node ids. However, Line 1 of Algorithm 2 produces a set of ordered sublists, each of which is associated with a path id obtained in the $PJoin$. Therefore, when performing the structural join, we will need to examine these multiple sublists of node ids for an element tag to find the smallest node id to be processed next.

4.3 Discussion

The path join is designed to reduce the number of elements involved in the subsequent node join. In this section, we analyze the effectiveness of the proposed path join.

Definition (Exact Pid Set): Let P be a set of path ids obtained for a node n in an XML query T. P is an *exact Pid set* with respect to T and n if the following conditions hold:

1. for each path id $p_i \in P$, the element with tag n and path id p_i is a result for T, and
2. for each path id $p_j \notin P$, the element with tag n and path id p_j is not a result for T.

Definition (Super Pid Set): Let P be a set of path ids obtained for a node n in an XML query T. P is a *super Pid set* with respect to T and n if each element with a tag n in the final result (after node join) is associated with a path id p_i such that $p_i \in P$.

Clearly, each element node is associated with its super Pid set after the path join. The result is optimal when these super Pid sets are also the exact Pid sets.

Next, we examine the situations where path join will yield exact Pid sets. We assume that the XML elements are non-recursive.

Simple Path Queries. Suppose query T is a simple path query without value predicates. Then each node in T will have an exact Pid set after the path join. This is because all the path ids that satisfy the path id containment property are reserved in the adjacent nodes of T. Since this containment property is transitive, all the path ids of a node n in T will contain the path ids of its descendant nodes, and vice versa. Moreover, the encoding table for the paths can identify the exact containment relationship (parent-child or ancestor-descendant) between the nodes in T. Therefore, given a simple path XML query without value predicates, the path join will eliminate all the elements (path ids) that do not appear in the final result sets.

Branch Queries. If a query T is a branch query, then we cannot guarantee that the nodes on the branch path have exact Pid sets because of the manner in which the path ids are assigned to the elements. In other words, the path id is designed to capture the containment relationship, but not the relationship between sibling nodes.

Consider the query in Figure 3(a) which is issued on the XML instance in Figure 1. After the path join, node F will be associated with a path id 010000. However, we see that only F(010000,14) is an answer to this query while F(010000,7) is not. This is because we can only detect 010000 (path id of F) is contained by 010110 (path id of B), but do not know whether an F element with path id 010000 will have sibling E. Finally, note that the path id set of B in Figure 3(a) is guaranteed to be an exact Pid set since B has no sibling nodes.

Queries with Value Predicates. Figure 3(b) shows an XML query with value predicates that will lead to super Pid sets after a path join. The node D (010000,13) and F (010000,14) in Figure 1 are not answers to the query although the path id 010000 occurs in the path id sets of D and F respectively after the path join. This is because we do not assign path information to value

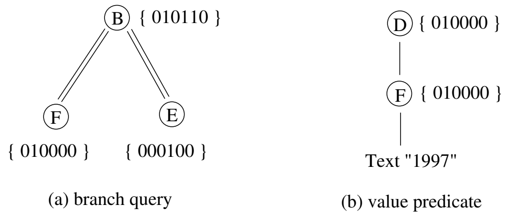


Fig. 3. Examples of Super Pid Set

nodes. Therefore, the element nodes with the matching path id can only satisfy the structural relationship, and not the value constraints. As a result, if an XML query has value predicates, the path id set of each node in the query pattern may not be the exact *Pid* set.

To summarize, for non-recursive XML data, the exact *Pid* sets will be associated with the element nodes in simple XML query patterns after the path join, while only the super *Pid* sets (which are much less than the full path id sets of elements as shown in our experimental section) can be guaranteed for the nodes in branch queries and queries with value predicates.

5 Experiments

This section presents the results of experiments to evaluate the performance of proposed path-based approach. We compare the path-based approach with the state-of-the-art XB-tree based TwigStack [5]. Both solutions are implemented in C++. All experiments are carried out on a Pentium IV 2.4 GHz CPU with 1 GB RAM. The operating system is Linux 2.4. The page size is set to be 4 KB.

Table 1 shows the characteristics of the experimental datasets which include Shakespeare's Plays (SSPlays) [1], DBLP [2] and XMark benchmark [3]. Attributes are omitted for simplicity.

Table 1. Characteristics of Datasets

Datasets	Size	♯(Distinct Elements)	♯(Elements)
SSPlays	7.5 MB	21	179,690
DBLP	60.7 MB	32	1,534,453
XMark	61.4 MB	74	959,495

5.1 Storage Requirements

We first compare the space requirement of the path-based approach with the XB-tree [5]. Our implementation of the XB-tree bulkloads the data and keeps every node half full except for the root node. The page occupancy for the node lists in the path-based approach is also kept at 50%. To be consistent with the XB-tree, the path-based solution also utilizes the interval-based node labeling scheme to assign the node ids. The storage requirements are shown in Table 2.

It can be observed that the sizes of encoding tables are very small (0.24K, 0.38K and 2.9K respectively), and hence we load them into memory in our experiments. The space required by the path lists is determined by the degree of regularity of the structures of the XML documents (see Table 3). The real-world datasets typically have a regular structure, and thus have fewer distinct paths (40 distinct paths in SSPlays and 69 in DBLP) compared to the 344 distinct paths in the synthetic XMark dataset. Since the number of bits in the path id is given by the number of distinct paths, the path ids for the SSPlays and DBLP are only 5 and 9 bytes respectively. In contrast, the irregular structure in XMark needs 43 bytes for the path id.

Table 2. Space Requirements

Datasets	XB	Path		
		Encoding Table	Path Lists	Node Lists
SSPlays	8.0MB	0.24KB	5.9KB	6.5MB
DBLP	69.6MB	0.38KB	9.1 KB	57.2MB
XMark	40.4MB	2.90KB	884.2KB	32.3MB

Table 3. Storage for Path Ids

Datasets	♯(Distinct Path)	Path Id Size(Bytes)
SSPlays	40	5
DBLP	69	9
XMark	344	43

We also observe that the size of the path lists is relatively small compared with that of node lists. Even for the most irregular structure dataset XMark, the size of path lists takes only 2.7% of its node lists size (884K and 32M respectively). This feature fundamentally guarantees the low cost of path join.

5.2 Query Performance

Next, we investigate the query performance of the path-based approach and compare it with the XB-tree based holistic join [5]. Table 4 shows the query workload for the various datasets. The query workload comprises short simple queries, long path queries and branch queries (Q1-Q8). To examine the effect of parent-child relationship, we replace some ancestor-descendant edges in queries Q1, Q3 and Q8 with parent-child relationship. Moreover, value constraints are imposed on queries Q1, Q2, Q5 and Q6 respectively to test the influence of value predicates.

Effectiveness of Path Join. In this set of experiments, we demonstrate the effectiveness of the path join algorithm in filtering out elements that are not relevant for the subsequent node join.

A metric called "Filtering Efficiency" is first defined to measure the filtering ability of path join. This metric gives the ratio of the number of nodes after a

Table 4. Query Workload

	Query	Dataset	♯ Nodes in Result
Q1	//PLAY//TITLE	SSPlays	1068
Q2	//PLAY/ACT/SCENE/SPEECH/LINE/STAGEDIR	SSPlays	2259
Q3	//SCENE//STAGEDIR	SSPlays	6974
Q4	//proceedings/booktitle	DBLP	3314
Q5	//proceedings[/url]/year	DBLP	5526
Q6	//people/person/profile[/age]/education	XMark	7933
Q7	//closed_auction/annotation[//emph]//keyword	XMark	13759
Q8	//regions/australia/item//keyword[//bold]//emph	XMark	74
Q1pc	//PLAY/TITLE	SSPlays	74
Q3pc	//SCENE/STAGEDIR	SSPlays	5010
Q8pc	//regions/australia/item//keyword[/bold]/emph	XMark	74
Q1v	//PLAY//TITLE="ACT II"	SSPlays	111
Q2v	//PLAY/ACT/SCENE/SPEECH/LINE/STAGEDIR="Aside"	SSPlays	1044
Q5v	//proceedings[/url]//year="1995"	DBLP	432
Q6v	//people/person[/age="18"]/profile/education	XMark	2336

path join over the total number of nodes involved in the query. That is, given a query Q, we have

$$Filtering\ \ Efficiency = \frac{\sum |N_i^p|}{\sum |N_i|}$$

where $|N_i^p|$ denotes the number of instances for node N_i after a path join and $|N_i|$ denotes the total number of instances for N_i in the query.

We also define "Query Selectivity" to reflect the percentage of nodes in the result set compared to the original number of nodes involved in the query. Given a query Q, we have

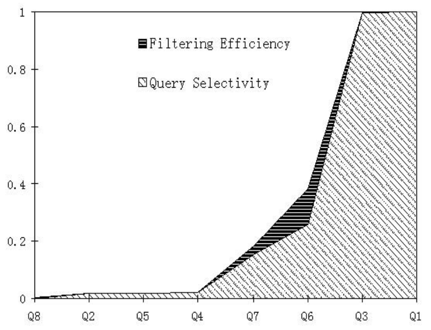
$$Query\ \ Selectivity = \frac{\sum |N_i^n|}{\sum |N_i|}$$

where $|N_i^n|$ denotes the number of instances for node N_i in the result set after a node join and $|N_i|$ is the same as above.

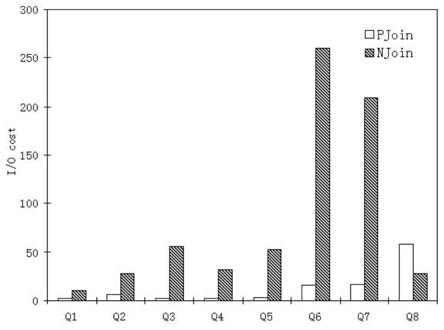
The effectiveness of path join can be measured by comparing the values of its Filtering Efficiency and Query Selectivity. Based on the definitions of these two metrics, we can see the closer the two values are, the more effective the path join is for the query. The optimal case is achieved when the Filtering Efficiency is equivalent to the Query Selectivity, indicating that path join has effectively filtered out all irrelevant elements for the subsequent node join.

Figures 4(a) compares the Filtering Efficiency with Query Selectivity for queries Q1 to Q8 whose Query Selectivity values are in ascending order. Except for queries Q6, Q7 and Q8, the rest queries have the same values for two metrics. This shows that path join has effectively removed all irrelevant elements for the node join.

Queries Q6, Q7 and Q8 have higher Filtering Efficiency values compared to their Query Selectivity. This indicates that the path join algorithm does not produce exact *Pid* sets for the subsequent node join for these queries. As we



(a) Filtering Efficiency Vs Query Selectivity



(b) PJoin and NJoin (I/O cost)

Fig. 4. Effectiveness of Path Join

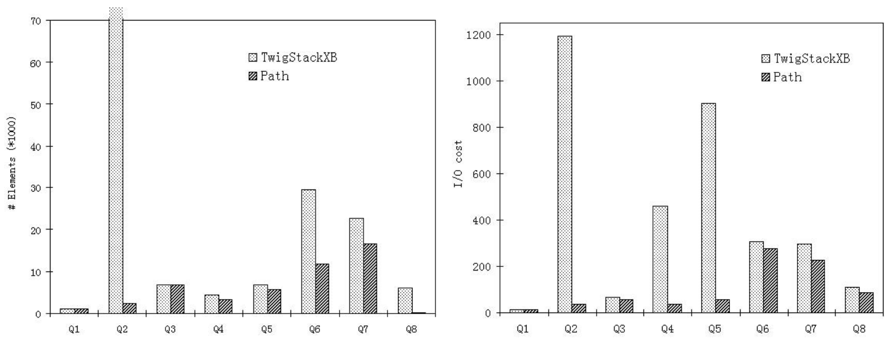
have analyzed in Section 4.3, the *Pid* sets associated with nodes after the path join may not be the exact *Pid* sets for branch queries. Since Q6, Q7 and Q8 are all branch queries, this result is expected. Note that path join still remains efficient in eliminating unnecessary path types even for branch queries, which can be seen from the close values of filtering efficiency and query selectivity of queries Q5, Q6, Q7 and Q8 (all are branch queries, and the two values are same for Q5).

Figure 4(b) compares the I/O cost of path join and node join. The graph shows that the cost of path join is very marginal for the majority of the queries compared to that of node join. This is because the size of path lists involved in the query is much smaller than that of node lists (recall Table 2).

The costs of path join for queries Q1 to Q5 are negligible because of the regular structures of SSPlays and DBLP. The path join is more expensive for the queries over XMark dataset (Q6 to Q8) due to its irregular structure, which results in a larger number of path types and longer path ids. Among these queries on synthetic dataset (Q6 to Q8), query Q8 is the only one where the cost of path join is greater than the node join. This can be explained by the low selectivity of Q8 (74 nodes in result, Table 4), which directly contributes to the low cost of node join. Finally, the result in Figure 4(a) clearly demonstrates that the path join remains effective in filtering out a large number of elements for queries even with the influence of irregularity in synthetic dataset.

Efficiency of Approach. In this set of experiments, we compare the performance of path-based approach with XB-tree based holistic join [5]. The metrics used are the total number of elements accessed and I/O cost. Figure 5 shows that the path-based approach performs significantly better than the XB-tree based holistic join. This is because path join is able to greatly reduce the actual number of elements retrieved.

We observe that the underlying data storage structure of path-based approach has an direct effect on the query performance. For queries Q4 and Q5, the I/O costs are smaller than the number of elements accessed in path-based

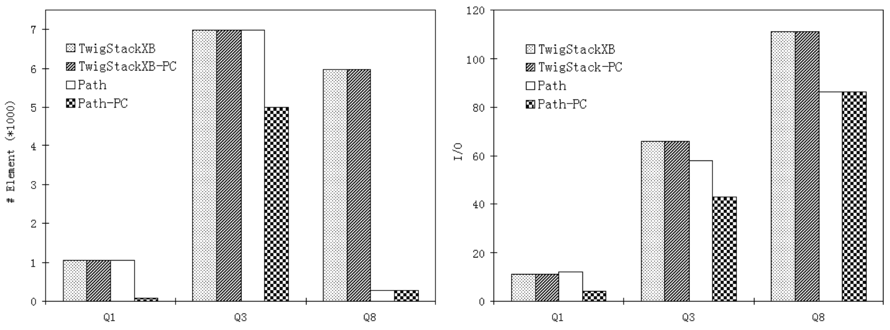


(a) Elements Accessed (b) I/O Cost

Fig. 5. Queries with Structural Patterns Only

approach (see Figure5(a) and (b)). This is because the path-based approach clusters the node records based on their paths. This further reduces the I/O cost during data retrieval. In contrast, the I/O cost for XB-tree is determined by the storage distribution of matching data. In the worst case, the elements to be accessed are scattered over the entire list, leading to high I/O costs.

Effect of Parent-Child Relationships. We examine the effect of parent-child relationship on query performance by replacing some ancestor-descendant edges in queries Q1, Q3 and Q8 with parent-child edges. Figure 6 shows the results.



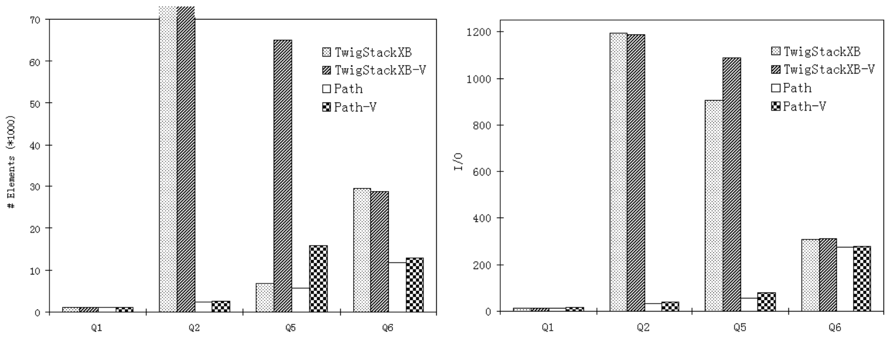
(a) Elements Accessed (b) I/O Cost

Fig. 6. Parent-Child Queries

The XB-tree based holistic join utilizes the same method to evaluate the parent-child queries and ancestor-descendant queries. Therefore, XB-tree based holistic join has the same evaluation performance for parent-child and ancestor-descendant queries. To avoid incorrect result, each parent-child edge is (inexpensively) verified before it is output.

In contrast, the proposed path-based approach checks for parent-child edges during the path join. This task is achieved by looking up the encoding table (see Figure 1(b)). In the case where the results of parent-child queries are subsets of the ancestor-descendant counterparts, the cost to evaluate queries may be further reduced since fewer elements are involved in the node join. For example, queries Q1pc and Q3pc have smaller result sets compared to Q1 and Q3 (see Table 4) respectively. Thus Q1pc and Q3pc show better performance in Figure 6.

Effect of Value Predicates. Finally, we investigate how the proposed approach and XB-tree perform for queries involving value predicates. We add value constraints on queries Q1, Q2, Q5 and Q6 respectively. The results are shown in Figure 7.



(a) Elements Accessed (b) I/O Cost

Fig. 7. Queries with Value Predicates

When evaluating XML queries involving value predicates, the path-based solution first carries out a path join to process the structural aspects of the queries. To determine the final set of results, the subsequent node join will retrieve the value nodes and element nodes obtained by path join. Therefore, the path-based solution needs to access more nodes to evaluate the value predicates in the queries compared to the corresponding queries without value predicates. This can be observed in Figure 7.

The XB-tree based holistic join solution treats value nodes the same way as element nodes. The additional value predicates will incur more costs during the retrieval of nodes. However, the value constraints may reduce the total number of element nodes accessed. This is because the XB-tree approach employs the XB-tree index to search for the matching nodes. Thus, it may skip some element nodes that match the structural query pattern but not the value predicates. Figure 7 shows that the addition of value predicates have different effects on performances of Q5 and Q6.

Overall, the evaluation of structural patterns still dominates the query performance even for queries involving value predicates. This is shown clearly in Figure 7.

6 Conclusion

In this paper, we have presented a new paradigm for processing structural join. The proposed solution includes a path-based labeling scheme and a path join algorithm that is able to compute the minimal sets of elements required for the subsequent node join. Experimental results clearly show that the proposed approach outperforms existing structural join methods for the following reasons:

1. The path join filters out nodes with path types that are not relevant to the subsequent node join;
2. The cost of path join is marginal compared to node join in the majority of the queries;
3. The element records are clustered according to the path types, which further reduces the I/O cost during element retrieval.

References

1. http://www.ibiblio.org/xml/examples/shakespeare.
2. http://www.informatik.uni-trier.de/~ley/db/.
3. http://monetdb.cwi.nl/.
4. S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of ICDE, USA*, 2002.
5. N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of SIGMOD, USA*, 2002.
6. Y. Chen, S. B. Davidson, and Y. Zheng. BLAS: An Efficient XPath Processing System. In *Proceedings of SIGMOD, France*, 2004.
7. S-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proceedings of VLDB, China*, 2002.
8. E. Cohen, H. Kaplan, and T. Milo. Labelling Dynamic XML Tree. In *Proceedings of PODS, USA*, 2002.
9. B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A Fast Index for Semistructured Data. In *Proceedings of VLDB, Italy*, 2001.
10. R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of VLDB, Greece*, 1997.
11. H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML Data for Efficient Structural Joins. In *Proceedings of ICDE, India*, 2003.
12. H. Jiang, W. Wang, and H. Lu. Holistic Twig Joins on Indexed XML Documents. In *Proceedings of VLDB, Germany*, 2003.
13. Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of VLDB, Italy*, 2001.
14. T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proceedings of ICDT, Israel*, 1999.
15. P. Rao and B. Moon. PRIX: Indexing and Querying XML Using Prüfer Sequences. In *Proceedings of ICDE, USA*, 2004.
16. H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proceedings of SIGMOD, USA*, 2003.
17. X. Wu, M. Lee, and W. Hsu. A Prime Number Labelling Scheme for Dynamic Ordered XML Trees. In *Proceedings of ICDE, USA*, 2004.

The BIRD Numbering Scheme for XML and Tree Databases – Deciding and Reconstructing Tree Relations Using Efficient Arithmetic Operations

Felix Weigel[1], Klaus U. Schulz[1], and Holger Meuss[2]

[1] Centre for Information and Language Processing, University of Munich, Germany
{weigel,schulz}@cis.uni-muenchen.de
[2] European Southern Observatory, Garching, Germany
hmeuss@eso.org

Abstract. This paper introduces the BIRD family of numbering schemes for tree databases, which is based on a structural summary such as the DataGuide. Given the BIRD IDs of two database nodes and the corresponding nodes in the structural summary we *decide* the extended XPath relations *Child*, *Child*$^+$, *Child**, *Following*, *NextSibling*, *NextSibling*$^+$, *NextSibling** for the nodes without access to the database. Similarly we can *reconstruct* the parent node and neighbouring siblings of a given node. All decision and reconstruction steps are based on simple arithmetic operations. The BIRD scheme offers high expressivity and efficiency paired with modest storage demands. Compared to other identification schemes with similar expressivity, BIRD performs best in terms of both storage consumption and execution time, with experiments underlining the crucial role of ID reconstruction in query evaluation. A very attractive feature of the BIRD scheme is that all extended XPath relations can be decided and reconstructed in constant time, i.e., independent of tree position and distance of the nodes involved. All results are shown to scale up to the multi-Gigabyte level.

1 Introduction

Query formalisms for tree databases and XML help to process data on the web, to extract and integrate data from distinct repositories and sites [1], to organize the exchange of commercial and scientific data, to access user-specified corporate resources in LDAP directories, and to consult linguistic databases [2]. Query formalisms for XML, representing a combination of information retrieval and database techniques, are paramount in the future development of search engines for the web and for digital libraries. In the meantime, an impressive number of query formalisms for tree databases and XML have been proposed [3–5] and many systems have been developed that offer distinct functionalities for querying trees and XML [6–12].

In most of these cases, the underlying evaluation algorithms use a characteristic set of fundamental operational steps that may be described as *decision* or *reconstruction* of tree relations:

– *Decision.* Given two database nodes and a binary tree relation, decide if the relation holds between the nodes.
– *Reconstruction.* Given a database node and a functional[1] tree relation R, compute the R-image of the node.

Unlike decision, where potential ancestors, siblings etc. to be checked are already known, reconstruction starts from a given node and reproduces those nodes in its tree neighbourhood having a specific relation to that node. Standard relations for describing unranked ordered trees are the "generalized XPath axes" [4, 13]: *Child*, *NextSibling*, their inverses *Parent*, *PreviousSibling*, the (reflexive-)transitive closures of these relations, as well as *Following* and its inverse. *Parent*, *NextSibling* and *PreviousSibling* are functional. Examples and details explaining the use of decision and reconstruction for evaluating generalized XPath axes are given in Section 2.

Since most query formalisms and systems have to deal with large data sets, the efficiency of the underlying evaluation algorithms and their basic operations is a major concern. We focus on the question how special conventions for assigning unique identifiers to the nodes of a tree database (also called *node identification*, *tree encoding* or *numbering schemes*) can help to solve decision and reconstruction problems efficiently for the above generalized XPath axes without access to the tree database, thus avoiding I/O-operations. Node identification schemes are largely complementary to other optimization techniques for tree queries such as special-purpose index structures and join algorithms. Hence the latter can benefit from intelligent identification schemes [9, 14–18].

For judging the quality of an identification scheme, four properties are essential:

– *Expressivity.* Which decision and reconstruction problems are supported by the scheme, i.e., can be solved without database access for given node IDs?
– *Runtime performance.* How long does it take to solve the decision and reconstruction problems that are supported? Are there any dependencies on properties of the nodes involved, e.g., their depth or distance in the document tree?
– *Storage consumption.* Which storage capabilities are needed for realizing the identification scheme, given a large tree database?
– *Robustness.* Is it possible to add new nodes to the database without spoiling the IDs already assigned to the existing parts of the documents?

Main Contribution. In this paper we suggest a new node identification scheme for tree databases. Node identifiers are integers, called *BIRD* numbers (for *Balanced Index-based numbering scheme for Reconstruction and Decision*). BIRD

[1] A binary tree relation R is *functional* iff for every node n there exists at most one node m such that $R(n, m)$ holds. The node m is called the *R-image* of n.

numbering is compatible with document order in the sense that nodes visited later in a pre-left traversal of the document tree have larger BIRD numbers, and sparse in the sense that not all possible numbers are used in the BIRD scheme. In addition, each node has an integer *weight*. Deciding (reconstructing) tree relations boils down to trivial arithmetic tests (calculations) based on BIRD numbers and weights.

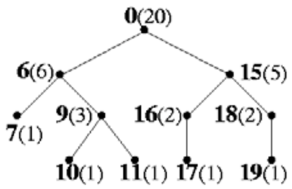


Fig. 1. BIRD numbers and weights (in brackets)

As an illustration, consider the tree in Figure 1. Each node n is annotated with its BIRD number $Id(n)$ (in bold) and with its weight $w(n)$ (in brackets). For any descendant n' of a node n we have $Id(n) < Id(n') < Id(n) + w(n)$. *Decision* problems for any XPath axis can be solved based on the following observations: node n' is a descendant of a given node n iff $Id(n) = Id(n') - (Id(n') \bmod w(n))$ [2]. To check if node n' is a following sibling of n we test if $Id(n) < Id(n')$ and n, n' have the same parent (parents are reconstructed, s.b.). Node n' follows n (in the sense of XPath's `following` relation) iff $Id(n') \geq Id(n) + w(n')$. Furthermore, once we know the weight b of the unknown parent of a given node n, we can *reconstruct* the BIRD number of the parent, which is $Id(n) - (Id(n) \bmod b)$. This briefly indicates how BIRD may be used to decide generalized XPath axes for two given nodes and to partially reconstruct the tree neighbourhood (here: the parent of a node). Reconstruction along other functional axes is discussed below.

We shall see that the BIRD scheme supports decision (reconstruction) of *all* (functional) generalized XPath axes. In this sense, BIRD offers "maximal" expressivity. The triviality of the above arithmetic operations guarantees high efficiency provided we have fast access to weights. To this end, BIRD weights are stored in a tree-formed structural summary or index (e.g., the DataGuide [19] of the database) which is held in main memory. Experiments (see Section 9) show that BIRD outperforms other node identification schemes for tree databases: using BIRD, basic decision and reconstruction steps are solved faster than with other schemes. At the same time, storage requirements of BIRD are modest.

Further Contributions.

- We provide an *abstract view on query formalisms for XML and tree databases* that helps to explain the role of decision and reconstruction operations for tree relations. We show how to place existing systems and techniques in this picture.
- We review various node identification schemes for tree databases known from the literature and *classify* these schemes in terms of the decision and reconstruction steps that are supported, taking into account a list of important tree relations.
- We present the results of an *extensive evaluation experiment*, where various node identification schemes have been applied to three XML databases ranging from medium (157 Megabyte) to big (8.6 Gigabyte) size. The com-

[2] For integers k, l ($l \neq 0$), let $k \bmod l$ denote the unique number $m \equiv k$ modulo l s.th. $0 \leq m < l$.

putation time with BIRD is almost always faster than with other schemes, up to two orders of magnitude. We also evaluate storage requirements.

The structure of the paper is as follows. Section 2 briefly explains the use of decision and reconstruction operations in query formalisms for tree databases and XML. Section 3 provides some formal background. Section 4 introduces the family of BIRD numbering schemes. In Sections 5 and 6 we show that BIRD numbering supports reconstruction and decision of all generalized XPath axes. Section 7 reviews and analyzes other identification schemes for tree databases suggested in the literature. Sections 8-11 present various experimental evaluation results and compare BIRD and other schemes on this basis, with special attention paid to the aforementioned four quality criteria. Due to space limitations, minor formal issues are omitted and evaluation results are described in a condensed way. See [20] for a full exposition and more detailed discussion.

2 Motivating Decision and Reconstruction

Looking at the core functionalities and abstracting from specific details, queries against tree databases and XML are typically built using unary predicates (e.g., labeling conditions, name tests in XPath) and binary tree relations such as *Child*, *Child*$^+$, *NextSibling* etc.[3] In what follows, we assume that the reader is familiar with the above relations and other standard relations on trees such as, e.g., *Parent*, *PreviousSibling*, *Following* etc. [4, 13]. See [20] for details. Query plans for evaluating such queries cover a spectrum of strategies with the following two extreme positions:

1. We may use the unary conditions to fetch a set of candidate image nodes for every single query node. In a second step, pairs of candidates from distinct sets are combined using joins over the binary relations, which amounts to solving a decision problem for the respective generalized XPath axis.
2. Since candidate sets for unselective unary predicates may be very large, we may alternatively fetch only the candidate sets for highly restricted query nodes (e.g., query leaves with selective keywords). From these nodes, candidates for other query nodes are obtained via reconstruction.

Reconstruction steps are particularly interesting along binary relations R that are functional (*Parent*, *PreviousSibling*, *NextSibling*, *i-th-Child* for $i \geq 1$, or any composition of these relations) or selective in the sense that database nodes typically have a small set of possible R-successors (transitive or reflexive-transitive closures of *Parent*, *PreviousSibling*, *NextSibling*). Given a query containing a condition $R(x, y)$ for such a relation R, if we already have a small candidate set for x, then reconstruction along R efficiently computes all relevant candidates for y. By contrast, if the unary conditions for y are weak, obtaining a consistent candidate set for x and y via decision might be costly.

[3] As usual, R^+ and R^* respectively denote the transitive and reflexive-transitive closure of the binary relation R. Similarly R^i denotes the i-fold composition $R \circ \cdots \circ R$.

Different query plans which are more or less close to either of the above positions are explained in [21]. The evaluation strategies described in [9, 16–18, 22] follow the first paradigm, whereas [11, 12, 15, 23] adhere to the second paradigm.

3 Formal Background

By a *database*, we mean a finite ordered rooted and labeled tree DB with non-empty set of nodes, N (see [20] for a formal definition). In what follows, DB denotes a database with set of nodes N and root n_r. Σ denotes the *alphabet of labels*. $L : N \to \Sigma$ assigns a unique label $L(n) \in \Sigma$ to each node $n \in N$.

Definition 1. *A* structural summary *for DB is a finite (not necessarily ordered) rooted tree Ind with set of nodes M, together with a surjective mapping $\Phi : N \to M$ preserving the root and Child relationship in the obvious sense. Φ is called the* index mapping*. For $m \in M$, the set $\Phi^{-1}(m)$ is called the* set of database nodes with index node *m.*

A structural summary can be considered as a special kind of index structure. In what follows, by an index, we always mean a structural summary. The *DataGuide* [19] (or *1-Index* [24], being equivalent to the DataGuide for tree databases) will serve as our standard example of a structural summary. To introduce the DataGuide, the following notions are needed.

Definition 2. *A string $\pi \in \Sigma^+$ is called a* label path *of DB iff there exists a sequence of nodes $n_0, n_1, \ldots, n_k \in N$ $(k \geq 0)$ such that $n_0 = n_r$, $\langle n_i, n_{i+1}\rangle \in$ Child for $0 \leq i < k$ and $\pi = L(n_0)L(n_1)\cdots L(n_k)$. In this situation, π is called the* label path of *n_k, we write $\pi = lp(n_k)$. The* length *of π is k. A label path π of DB is* maximal *iff π is not a proper prefix of any label path ϱ of DB. The* height *of DB is the maximal length of a label path of DB.*

Note that each label path π is non-empty and starts with the root label $L(n_r)$. $LP(DB)$ denotes the set of all label paths of the database DB.

Definition 3. *The* DataGuide *of DB is the finite rooted unordered node-labeled tree DG(DB) with set of nodes $LP(DB)$ where $L(n_r)$ is the root of DG(DB), $\varrho \in LP(DB)$ is a child of $\pi \in LP(DB)$ iff there exists a label $l \in \Sigma$ such that $\varrho = \pi l$, and the label of $\pi \in LP(DB)$ is the last symbol of π.*

DG(DB) represents a structural summary for DB with index mapping lp in the sense of Definition 1.

Example 1. Figure 2 shows a database DB and its DataGuide *DG(DB)*. Nodes of DB and *DG(DB)* are labeled with numeric information for the child-balanced numbering scheme that is introduced below.

Definition 4. *A function $f : N \to I\!N$ is* compatible *with the document order (i.e., preorder) $<_{pre}$ on DB iff $m <_{pre} n$ implies that $f(m) < f(n)$, for all $m, n \in N$.*

4 The Family of BIRD Numbering Schemes

BIRD numbering schemes for the nodes of a database *DB* as introduced below are always compatible with the document order in the sense of Definition 4. When enumerating the nodes, for each node $n \in N$ of the database we will need a certain interval size, or *weight*, to number all nodes in the subtree with root n. Our numbering schemes are based on a structural summary *Ind* of *DB* with index mapping Φ (see Definition 1). We unify all weights needed for database nodes with the same index node m, selecting the maximal interval size among all members of the equivalence class $\Phi^{-1}(m)$. This unified weight is attached to the associated node m of the structural summary. When enumerating the nodes of the database, we reserve this interval size for all subtrees rooted at any of the nodes in $\Phi^{-1}(m)$. Since in general not all these subtrees are of the same size, some numbers remain unused in the enumeration[4]. Because of space limitations, we only consider "balanced" variants of the BIRD scheme. Here the weights for index nodes are further unified among all children (or grand-children, etc.) of a given index node. Unbalanced variants have lower storage requirements, but are less expressive.

4.1 Balanced Weights of Index Nodes

Let n denote a node of the database *DB* with root n_r, let $s \geq 1$. By the *s-step ancestor* of n, we mean the ancestor of n that is reached in exactly s parent steps. As a matter of fact, the s-step ancestor of n is defined if and only if n is a node in depth $s' \geq s$, using the standard notion of the depth of a node in a tree. Since balanced weights are based on maximal interval sizes of siblings, cousins, grand-cousins, etc. in a tree, we need the following definition of s-equivalent nodes. Basically, two nodes are 1-equivalent iff they are siblings, 2-equivalent iff they are siblings or cousins (i.e. share the same grandparent) etc.

Definition 5. *The equivalence relations $\sim_s$ ($s \geq 1$) on the set of nodes N of a given tree are inductively defined as follows:*

1. *for all $n, n' \in N$: $n \sim_1 n'$ iff the 1-step ancestors (i.e., parents) of n and n' are defined and coincide.*
2. *Let $s \geq 1$. For all $n, n' \in N$: $n \sim_{s+1} n'$ iff $n \sim_s n'$, or the $s+1$-step ancestors of n and n' are defined and coincide.*

If $n \sim_s n'$, we say that n and n' are s-equivalent*. By $[n]_s$ we mean the equivalence class of node n w.r.t. $\sim_s$.*

Definition 6. *Let Ind denote a structural summary for the database DB. Let $s \geq 1$. The* s-balanced pre-weight $w'_s(m)$ *and the* s-balanced weight $w_s(m)$ *of an index node m are recursively defined in a bottom-up manner as follows:*

4 Unused numbers may also be reserved deliberately for future node insertions into *DB*.

$$w'_s(m) := \begin{cases} w_s(m_1) \cdot \mathit{max}\{\mathit{childCount}(n) + 1 \mid n \in \Phi^{-1}(m)\} \\ \qquad \textit{iff } m \textit{ has any child } m_1, \\ 1 \textit{ otherwise,} \end{cases}$$

$$w_s(m) := \mathit{max}\{w'_s(m') \mid m \sim_s m'\}.$$

Here childCount(n) *denotes the number of children of the database node* n.

Note that the maximum operation in the definition of $w_s(m)$ leads to unified weights for s-equivalent nodes (balancing). It also guarantees well-definedness of pre-weights $w'_s(m)$ since any two children m_1 and m_2 of m have the same s-balanced weights $w_s(m_1) = w_s(m_2)$. 1-balanced weights are also called *child-balanced* weights. If $s = h$ denotes the height of the database *DB*, then $w_s(m)$ is called the *totally balanced weight* of the index node m.

Example 2. Consider the database *DB* with the DataGuide *DG(DB)* shown in Figures 2 *(a)* and *(b)*, respectively. Each node m of the DataGuide is annotated with its child-balanced weight $w_1(m)$ and, for convenience, the pre-weight $w'_1(m)$ (in brackets). If m has children, we also depict the sequences of child labels of all $n \in \Phi^{-1}(m)$ (boxes). The maximal number of children in such a sequence determines the number $\mathit{max}\{\mathit{childCount}(n) + 1 \mid n \in \Phi^{-1}(m)\}$ used in Definition 6 (written next to each box). Note that only the weights $w_1(m)$ are stored physically in the DataGuide.

To understand how the depicted pre-weights and weights in the DataGuide are computed, consider the left-most path *racbc* in Figure 2 *(b)*. The procedure runs bottom-up and begins with the leaves *racbc* and *racbb*, whose pre-weight is fixed to $w'_1(\mathit{racbc}) = w'_1(\mathit{racbb}) = 1$ by Definition 6. Taking the maximal pre-weight among the two siblings we obtain $w_1(\mathit{racbc}) = w_1(\mathit{racbb}) = 1$. We next consider index node *racb*, which is associated via Φ^{-1} with database nodes 111, 114, and 282 (larger numbers[5] in Figure 2 *(a)*). Nodes 111 and 282 have no children, but *childCount*$(114) = 2$ (length of the sequence *cb* in the bottom left box). Therefore the child weight is multiplied by a factor $2 + 1 = 3$ according to Definition 6. The children of *racb* have weight 1, so the resulting pre-weight is $w'_1(\mathit{racb}) = 3$. In the next step, the weight $w_1(\mathit{racb})$ is computed: The bottom-up algorithm has already computed the pre-weights for the siblings *racc* and *racd*, which is 1 for leaves. The weight of each of the three siblings *racb*, *racc*, and *racd* is the maximum of their pre-weights, i.e., 3. On the higher levels, pre-weights and weights are computed in exactly the same way until we reach the root with weight $w(r) = 450$.

In the remainder of the paper, Ind_{w_s} denotes the variant of the structural summary *Ind* for the database *DB* where the s-balanced weight $w_s(m)$ is attached to each index node m, as illustrated in Figure 2 *(b)* for the DataGuide. Φ denotes the index mapping.

[5] For convenience, the example refers to database nodes by their BIRD IDs, although the computation of the IDs is explained later, in the next section.

4.2 Balanced Enumeration of Database Nodes

The *s-balanced numbering scheme* assigns an integer $Id_s(n)$ to each node n of *DB*, given the weight-annotated index Ind_{w_s}. In the special case where $s = h$ represents the height of *DB*, the scheme is called the *totally balanced numbering scheme.*

Definition 7. *Let $s \geq 1$. The BIRD number $Id_s(n_r)$ for the root n_r is any multiple of $w_s(\Phi(n_r))$. Let n denote an arbitrary node of DB. Let $n_1, \ldots, n_k$ ($k \geq 1$) denote the sequence of all children of n in the canonical left-to-right ordering. Given the BIRD number $Id_s(n)$ for the parent node n and the balanced child weight $w = w_s(\Phi(n_1)) = \ldots = w_s(\Phi(n_k))$, the BIRD number $Id_s(n_1)$ for the first child n_1 is the smallest multiple of w larger than $Id_s(n)$. The BIRD number for the i-th child n_i for $2 \leq i \leq k$ is $Id_s(n_i) := Id_s(n_1) + (i-1) \cdot w$.*

Example 3. In Figure 2 *(a)*, each database node n is annotated with $Id_1(n)$ (large number). The enumeration started with 0 for the root node, and went top-down through the tree in the manner described above. Note that weights for index nodes and identifiers of database node are defined in a way that all node identifiers in the subtree of a node n are guaranteed to fall into the interval $[Id_s(n), Id_s(n) + w_s(\Phi(n))[$. This important relation between weights of index nodes and database node identifiers is established in Proposition 2. The upper bound of the interval of each node is denoted with the small numbers in brackets in Figure 2 *(a)*. Note that only IDs are stored physically in the database.

Proposition 1. *Let $s \geq 1$. The mapping Id_s is injective and compatible with the document order. Regardless of the initial assignment $Id_s(n_r)$ for the root node n_r, for each node $n \in N$ we have $Id_s(n) \bmod w_s(\Phi(n)) = 0$.*

Proposition 2. *Let $s \geq 1$. Let n be a node of DB, let $n_1, \ldots, n_k$ denote the sequence of all children of n in the canonical left-to-right ordering. Let $w := w_s(\Phi(n_1)) = \ldots = w_s(\Phi(n_k))$. Then $Id_s(n) < Id_s(n_1) < \ldots < Id_s(n_k) < Id_s(n_k) + w \leq Id_s(n) + w_s(\Phi(n))$.*

5 Reconstructing the Tree Structure

We discuss how parts of the tree structure of the database can be reconstructed without accessing the database, given the number of a node and the corresponding index node with its weight. As before, *DB* denotes a database and *Ind* denotes a structural summary for *DB* with index mapping Φ.

Lemma 1. *[Reconstruction of parents/ancestors, i-th child, i-th left/right sibling] Let $s, i \geq 1$. Assume that for some database node n we are given its BIRD number $Id_s(n)$ and the index node $m := \Phi(n)$. Then, using Ind_{w_s} we may solve the following tasks without access to DB:*

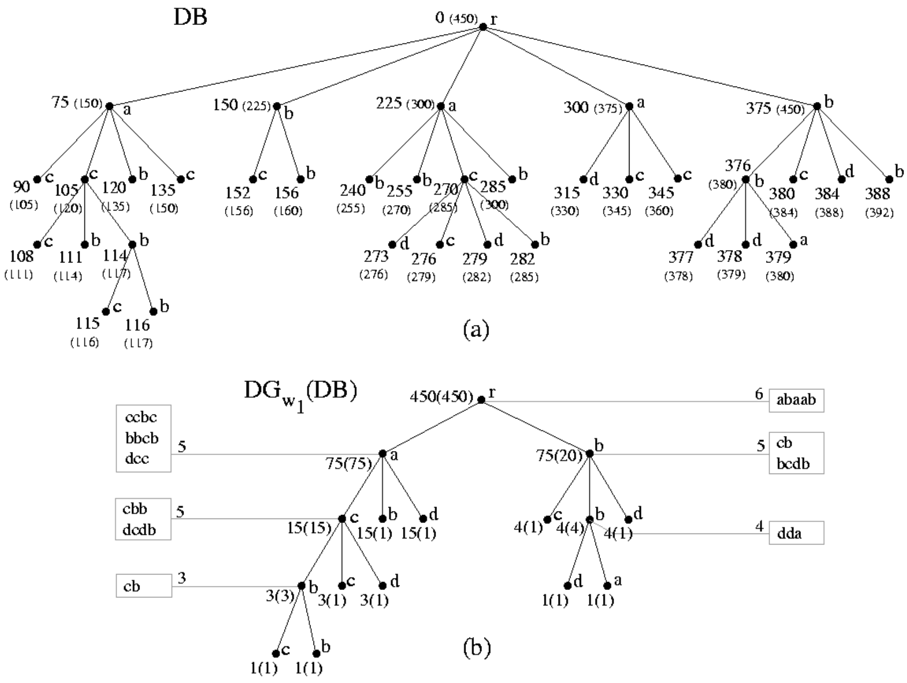


Fig. 2. Child-balanced numbering scheme. *(a)* Database *DB*. Large numbers denote child-balanced BIRD IDs; small numbers in brackets denote upper interval bounds. *(b)* DataGuide for *DB*. Large numbers denote child-balanced weights; small numbers in brackets denote pre-weights. For each non-leaf node m of *DG(DB)*, the number $max\{childCount(n) + 1 \mid n \in \Phi^{-1}(m)\}$ is indicated (cf. Definition 6), next to the child labels of all $n \in \Phi^{-1}(m)$ (boxes). Only IDs and weights are stored physically

1. *Decide if there exists an ancestor n' of n that is reached from n with exactly (at least) i parent steps. In the affirmative case, compute the number $Id_s(n')$ and the index node $m' := \Phi(n')$ corresponding to n'.*
2. *Compute the number $Id_s(n_i)$ of the i-th child n_i of n, assuming that this child exists.*
3. *Decide if n has exactly (at least) i siblings that precede n in the left-to-right ordering. If n has at least i preceding siblings, compute the number $Id_s(n_i)$ of the i-th preceding sibling n_i of n.*
4. *Compute the number $Id_s(n_i)$ of the i-th right sibling n_i of n, assuming that this sibling exists.*

Proof. 1. Obviously, n has an ancestor n' reached in exactly i parent steps iff $m := \Phi(n)$ has such an ancestor, m'. Using Ind_{w_s} we may decide this question, finding m' in the affirmative case. By Proposition 1, $Id_s(n')$ is a multiple of $w_s(m')$. By Proposition 2, $Id_s(n')$ is the greatest multiple of $w_s(m')$ smaller than $Id_s(n)$.

2. Using Ind_{w_s} we fetch the weight $w = w_s(m')$ of the children m' of m, which are assumed to exist. By definition, $Id_s(n_1)$ is the smallest multiple of w larger than $Id_s(n)$, and for $i > 1$ we have $Id_s(n_i) = Id_s(n_1) + w(i-1)$.

3. We may assume that n has a parent node n'. Let $Id_s(n')$ denote its number, calculated as described in 1. Let $w = w_s(m)$. By Proposition 2, n has at least i preceding siblings iff $Id_s(n') < Id_s(n) - i \cdot w$. From Definition 7 it follows that n has exactly i preceding siblings iff $Id_s(n) - (i+1) \cdot w \leq Id_s(n') < Id_s(n) - i \cdot w$. If the i-th preceding sibling exists, it has the number $Id_s(n) - i \cdot w$.

4. Similar. □

The totally balanced scheme (where $s = h$, the height of the database) has a number of special features that lead to stronger statements and refinements of Propositions 1 and 2 and Lemma 1. See [20] for details.

6 Deciding Generalized XPath Relations

Let R be any of the "generalized XPath axes" $Child$, $Child^*$, $Child^+$, $NextSibling$, $NextSibling^*$, $NextSibling^+$, and $Following$. For two nodes n and n' of the database we write $DB \models R(n, n')$ iff the relation R holds in DB between n and n' (e.g., $DB \models Child(n, n')$ iff n' is a child of n). As before we fix a structural summary Ind_{w_s} with index mapping Φ.

Lemma 2. *[Deciding generalized XPath axes] Let $s \geq 1$. Assume we are given*

- *the number $Id_s(n)$ of the database node n,*
- *the index node $m = \Phi(n)$ corresponding to n,*
- *the number $Id_s(n')$ of a second database node n'.*

Let R be any of the aforementioned relations. Then, using Ind_{w_s} we may decide if $DB \models R(n, n')$ (or if $DB \models R(n', n)$) without access to the database DB.

Proof. See Table 1. □

For the sake of completeness, we mention some other problems that may be decided with similar methods. Proofs are simple.

Lemma 3. *[Deciding proximity relations] Let $s, i \geq 1$. Assume we are given the number $Id_s(n)$ of the database node n, the index node $m = \Phi(n)$, and the number $Id_s(n')$ of a second node $n' \in N$. Then, using Ind_{w_s} we may decide the following questions without access to the database DB:*

1. $DB \overset{?}{\models} Parent^i(n, n')$
2. $DB \overset{?}{\models} PreviousSibling^i(n, n')$
3. $DB \overset{?}{\models} NextSibling^i(n, n')$

Table 1. Proof for Lemma 2. Relations decidable using any s-balanced BIRD scheme with $s \geq 1$. Given the BIRD numbers $Id_s(n)$ and $Id_s(n')$ of two database nodes $n, n' \in DB$ as well as the index node $m = \Phi(n)$ holding the weight corresponding to n, all relations are decidable without access to the database. Corresponding XPath axes are given with n as context node. For example, $Child(n', n)$ means n is a child of n', corresponding to the `parent` axis. For further notation details, see Lemma 2

$DB \models Child(n, n')$ `child`	We check if m has any child, say, m', using Ind_{w_s}. In the negative case, n' is not a child of n. In the positive case let $w = w_s(m')$. Then $DB \models Child(n, n')$ iff $Id_s(n')$ is a multiple of w and $Id_s(n) < Id_s(n') < Id_s(n) + w_s(m)$. The numbers $w_s(m')$ and $w_s(m)$ are obtained from Ind_{w_s}.
$DB \models Child^+(n, n')$ `descendant`	We retrieve $w_s(m)$ using Ind_{w_s}. Then $DB \models Child^+(n, n')$ iff $Id_s(n) < Id_s(n') < Id_s(n) + w_s(m)$.
$DB \models Child^*(n, n')$ `descendant-or-self`	The relation holds iff $DB \models Child^+(n, n')$ or $n = n'$.
$DB \models Child(n', n)$ `parent`	We proceed as in Lemma 1, 1. with $i = 1$ and compare the resulting BIRD number to n'.
$DB \models Child^+(n', n)$ `ancestor`	We iterate the procedure described in Lemma 1 for $i = 1$ until reaching either n' (positive result) or a node n'' where $Id_s(n'') < Id_s(n')$ (negative result).
$DB \models Child^*(n', n)$ `ancestor-or-self`	The relation holds iff $DB \models Child^+(n', n)$ or $n = n'$.
$DB \models NextSibling(n, n')$	We obtain $w_s(m)$ and m's parent m'' from Ind_{w_s} and compute the number $Id_s(n'')$ of the parent n'' of n in DB (cf. Lemma 1, 1.). $DB \models NextSibling(n, n')$ holds iff $Id_s(n') = Id_s(n) + w_s(m)$ and $Id_s(n') < Id_s(n'') + w_s(m'')$.
$DB \models NextSibling^+(n, n')$ `following-sibling`	We obtain $w_s(m)$, m'' and $Id_s(n'')$ as above (cf. $DB \models NextSibling(n, n')$). $DB \models NextSibling^+(n, n')$ holds iff $Id_s(n') - Id_s(n)$ is positive and a multiple of $w_s(m)$ and if $Id_s(n') < Id_s(n'') + w_s(m'')$.
$DB \models NextSibling^*(n, n')$	The relation holds iff $DB \models NextSibling^+(n, n')$ or $n = n'$.
$DB \models NextSibling(n', n)$	We proceed as in Lemma 1, 3. with $i = 1$ and compare the resulting BIRD number to n'.
$DB \models NextSibling^+(n', n)$ `preceding-sibling`	We obtain $w_s(m)$ and m's parent m'' from Ind_{w_s} and compute the number $Id_s(n'')$ of the parent n'' of n in DB (cf. Lemma 1). $DB \models NextSibling^+(n', n)$ holds iff $Id_s(n) - Id_s(n')$ is positive and a multiple of $w_s(m)$ and if $Id_s(n'') < Id_s(n')$.
$DB \models NextSibling^*(n', n)$	The relation holds iff $DB \models NextSibling^+(n', n)$ or $n = n'$.

Table 1. (Continued)

$DB \models \mathit{Following}(n, n')$ `following`	The relation holds iff $Id_s(n) + w_s(m) \leq Id_s(n')$, by Propositions 1 and 2. The weight $w_s(m)$ is obtained from Ind_{w_s}.
$DB \models \mathit{Following}(n', n)$ `preceding`	The relation holds iff $Id_s(n') < Id_s(n)$ and n' is not an ancestor of n. The latter problem is decided as described above (cf. $DB \models \mathit{Child}^+(n', n)$/`ancestor`).

7 Positioning BIRD and Other Node Identification Schemes

Node ID schemes can be classified along several dimensions. *Number-based* schemes use integers to identify nodes, whereas *path-based* schemes encode the sequence of horizontal positions of ancestors among their respective siblings in IDs of varying length. Many schemes use a *sparse ID set*: the ID space is not contiguous and contains unused IDs, which helps to cope with limited node insertions without reassigning node IDs. Number-based schemes follow either a *depth-first* or *breadth-first* traversal of the document tree. In a *multiplier-driven* scheme, decision and reconstruction steps are reduced to simple arithmetic computations by only assigning multiples of a certain basic weight as node IDs. Finally, some ID schemes are *DataGuide-based*: they extract path-related information and store it in a DataGuide [19] for use during reconstruction or decision. In this terminology, BIRD represents a number-based multiplier-driven DataGuide-based scheme with sparse ID set.

An example for path-based ID schemes is *Dewey Order* [23]. Since offsets in paths are independent of each other, Dewey Order supports updates where renumbering is restricted to the descendants and following siblings of the node being inserted [23]. The Dewey-based *ORDPATH* scheme [25] uses skew binary encodings privileging smaller offset numbers to reduce storage. For reconstruction, the encoded ORDPATH IDs are parsed and split into their offset components. Removing the last offset is equivalent to going up one level. All path relations are decided by bitwise comparison of the encoded IDs. However, IDs must be decoded into their offset components first in order to find the component boundaries in the bit string. [25] describes an update mechanism for ORDPATH which reserves unused IDs for future insertions at any position in the document tree. ORDPATH is the only known scheme to allow for arbitrary updates without changing any existing ID. However, when using this update mechanism, ORDPATH cannot decide the *NextSibling* relation, and reconstruction of sibling or child nodes is no longer possible.

Another path-based scheme was proposed in [15]: binary *Path Identifiers (PIDs)* encode complete root paths as sequences of offsets representing relative positions among children with the same label. To save space, offsets for children which do not have any sibling with the same label are not encoded. Information about which path steps are skipped this way is stored in a DataGuide. Compared to ORDPATHs, PIDs facilitate reconstruction by storing the number of bits used to encode a given offset in the corresponding DataGuide node. Updates in PID are not supported, but only a local renaming of node IDs is necessary if new nodes are inserted. The PID scheme is the least expressive scheme among those supporting both path reconstruction and decision, but as our experiments showed, its node IDs are the smallest in size.

Virtual Nodes [14] is the only number-based scheme identifying nodes by their breadth-first rank. It uses a sparse ID set: all inner nodes are treated as if they had exactly k children. This means that many IDs are reserved for so-called *virtual nodes* which do not exist physically in the document tree. This leads to a significantly higher space consumption compared to other schemes (see Section 9). On the other hand, the resulting multiplier-driven scheme needs only simple arithmetic operations to decide tree relations and to reconstruct ancestors or siblings of a node.

Several number-based node identification schemes have been proposed that support only path decision: node IDs are pairs $\langle pre, post\rangle$ consisting of the node's pre- and postorder ranks. Simple comparison operations on the interval $[pre, post]$ decide the $Child^+$ and $Child^*$ relations [26]. [22] shows how to decide *Following* and *After*. The paper also describes how *XPath Accelerator* combines the pre-/postorder encoding with appropriate index structures embedded into a relational system to decide the remaining XPath axes. A related scheme is the *extended preorder* [9, 16], whose IDs are pairs $\langle pre, size\rangle$ where *pre* is the node's preorder rank and *size* is an integer equal to or greater than the number of descendants of that node.

8 Expressivity Comparison

Table 2. Expressivity of ID schemes

scheme	path decision								path reconst.				
	$Child^i(m,n)$	$Child^+(m,n)$	$NextSibling^i(m,n)$	$NextSibling^+(m,n)$	$Following(m,n,i)$	$Following(m,n)$	$NextInDocOrder^i(m,n)$	$NextInDocOrder^+(m,n)$	$parent^i(n)$	$j\text{-th-child}(n)$	$prevSibling^i(n)$	$nextSibling^i(n)$	$i\text{-th-commonAnc}(m,n)$
BIRD	•	•	•	•		•		•	•	2	•	2	•
ORDPATH	•	•		•		•		•	•				•
(non-updatable)	•	•	•	•		•		•	•	2	•	2	•
Virtual Nodes	•	•	•	•					•	2	•	2	•
PID	•	•							•				•
pre-/postorder	1	•				•	•	•					
interval encod.	1	•				•		•					
preorder							•	•					

• supported 1 requires level
2 supported, but may not exist physically

The expressivity of the various ID schemes is described in Table 2: A bullet in a cell indicates that the scheme in that row supports the evaluation of the tree relation in that column *without* access to any database table. Table 2 is divided into node identification schemes supporting path reconstruction (the first five rows) and those supporting decision only. Numbers in the table cells describe the following restrictions: (1) Reconstruction along forward axes, i.e., of children or right siblings, is hypothetical in the sense that the IDs obtained may not correspond to actual nodes of the database tree. (2) For pre-/postorder encoding, the child decision problem can only be solved with additional information about the node level. In the reconstruction part of Table 2 (right-hand side), *j-th-child*(n) denotes the j-th child of n in left-to-right sibling order, and *i-th-commonAnc*(m, n) the i-th common ancestor of m and n (bottom-up). All decision results mentioned also hold for the corresponding reverse axes.

BIRD, ORDPATH and Virtual Nodes support most decision problems. ORDPATH cannot decide horizontal proximity (for instance, `following-sibling::*[1]` in XPath) in its original (updatable) version. Virtual Nodes decides order among siblings, but not the more general *NextInDocOrder* and *Following* relations. BIRD, ORDPATH and Virtual Nodes have roughly the same expressivity in the reconstruction part, although ORDPATH cannot reconstruct siblings in its original form.

9 Efficiency Comparison

Table 3. Document collections

name	XML size	# nodes	# label paths	depth
DBLP	157 MB	5,390,160	129	7
XMark	1,145 MB	20,532,979	549	13
IMDb	8,633 MB	83,404,825	276	5

We evaluated four different identification schemes, namely BIRD (child-balanced, i.e., $s = 1$), ORDPATH [25] (encoded with max. 9 bits for length and max. 20 bits for offset components), Virtual Nodes [14] and PID [15]. We applied each scheme to the first two document collections listed in Table 3, which differ considerably w.r.t. size and structural characteristics. All tests were carried out sequentially on an i686 computer with AMD Athlon XP 2600+ CPU running at 2.1 GHz with 256 kB cache. Further details of the experimental setting and a more detailed analysis can be found in [20].

Reconstruction. Figures 3 and 4 plot the computation time needed for various reconstruction problems on the *DBLP* and the *XMark* collection. Schemes were tested with the same set of synthetically generated problems. Since the speed of individual operations cannot be measured with sufficient confidence, the figures represent the accumulated time (in milliseconds) needed for 50,000 repetitions of each decision or reconstruction. The figures subsume all necessary operations including, e.g., DataGuide accesses for BIRD or PID and ID comparison during decision.

Figure 3 shows the time needed to reconstruct the parents of nodes at different levels. For *DBLP* (left-hand side) and *XMark* (right-hand side), PID is almost as fast as BIRD, whereas ORDPATH and Virtual Nodes are slower by at least a factor 4. On *XMark*, the difference between BIRD and ORDPATH is up to one order of magnitude. Obviously the performance of both BIRD and PID is independent of the level of the context node. For ORDPATH and Virtual Nodes, the computation time grows with the depth of the context node. Figure 4 illustrates the orthogonal situation: here the *parent*$^i(n)$ relation is reconstructed from context nodes at a fixed depth in the tree (level 7 for *DBLP*, 13 for *XMark*), with varying distance i. Again, BIRD and PID are much faster than ORDPATH and Virtual Nodes.

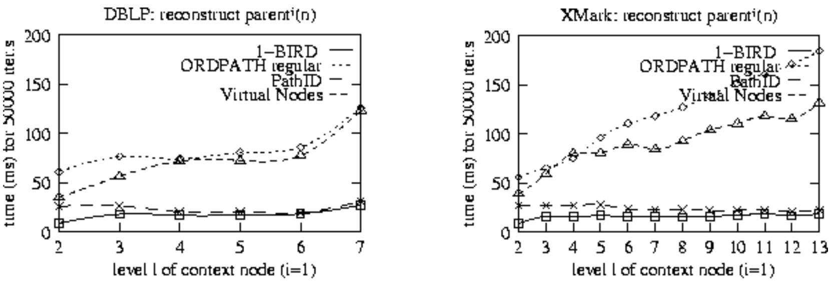


Fig. 3. Reconstructing ancestors from varying levels

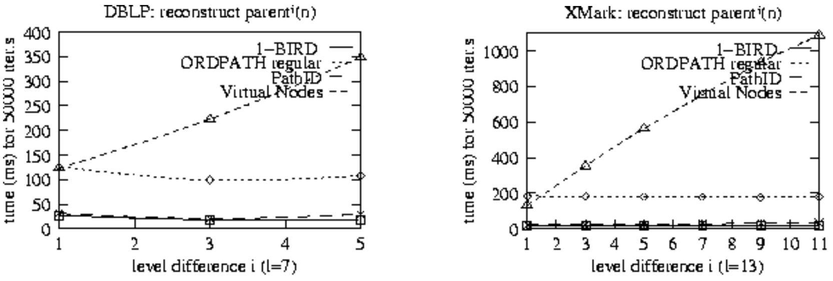


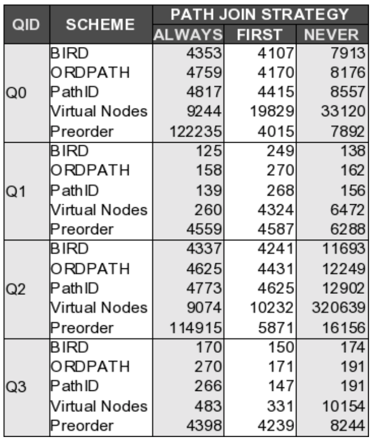
Fig. 4. Reconstructing ancestors in varying proximity

Decision. Results for decision problems [20] show the efficiency gain of BIRD even stronger (up to two orders of magnitude compared to ORDPATH), but are not included here due to space constraints.

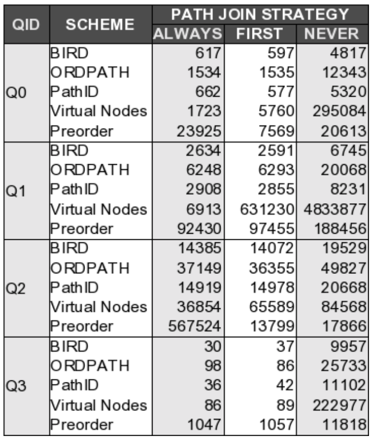
Runtime Performance for Tree Queries. To quantify how much the differences in decision and reconstruction speed just observed affect the overall performance for entire tree queries, we evaluated four sample queries against the *XMark* collection and another four againt *DBLP*, using the same schemes as in the previous section. To avoid artefacts due to file system cache effects, the best and the worst result of six consecutive iterations of each query were discarded. The remaining four iterations of the same query were then averaged in Table 4. Each tree query was executed using three different path join strategies determining the use of reconstruction (ALWAYS, NEVER, only FIRST child of a given branching node). A detailed description of these join algorithms and their impact on query evaluation time is found in [20].

The following key results sum up the outcome of these experiments: (1) The BIRD scheme performs best for virtually all queries and path join strategies on *XMark*. (2) Reconstruction is of paramount importance to efficient query evaluation because it saves ID fetching and comparison (ALWAYS versus NEVER in Table 4). Further experiments [20] show also that (3) inefficiencient ID comparisons (e.g. due to 128-bit IDs) can spoil the performance gained by ID reconstruction and decision; (4) ID schemes preserving document order benefit greatly from path join optimizations.

Table 4. Performance (avg, ms) for queries against *DBLP* (left) and *XMark* (right)



QID	SCHEME	PATH JOIN STRATEGY		
		ALWAYS	FIRST	NEVER
Q0	BIRD	4353	4107	7913
	ORDPATH	4759	4170	8176
	PathID	4817	4415	8557
	Virtual Nodes	9244	19829	33120
	Preorder	122235	4015	7892
Q1	BIRD	125	249	138
	ORDPATH	158	270	162
	PathID	139	268	156
	Virtual Nodes	260	4324	6472
	Preorder	4559	4587	6288
Q2	BIRD	4337	4241	11693
	ORDPATH	4625	4431	12249
	PathID	4773	4625	12902
	Virtual Nodes	9074	10232	320639
	Preorder	114915	5871	16156
Q3	BIRD	170	150	174
	ORDPATH	270	171	191
	PathID	266	147	191
	Virtual Nodes	483	331	10154
	Preorder	4398	4239	8244



QID	SCHEME	PATH JOIN STRATEGY		
		ALWAYS	FIRST	NEVER
Q0	BIRD	617	597	4817
	ORDPATH	1534	1535	12343
	PathID	662	577	5320
	Virtual Nodes	1723	5760	295084
	Preorder	23925	7569	20613
Q1	BIRD	2634	2591	6745
	ORDPATH	6248	6293	20068
	PathID	2908	2855	8231
	Virtual Nodes	6913	631230	4833877
	Preorder	92430	97455	188456
Q2	BIRD	14385	14072	19529
	ORDPATH	37149	36355	49827
	PathID	14919	14978	20668
	Virtual Nodes	36854	65589	84568
	Preorder	567524	13799	17866
Q3	BIRD	30	37	9957
	ORDPATH	98	86	25733
	PathID	36	42	11102
	Virtual Nodes	86	89	222977
	Preorder	1047	1057	11818

Table 5. Storage consumption for *DBLP*(left) and *XMark*(right)

scheme	ID size (bits)			total storage (MB)			
	min.	max.	avg.	var ID size		fixed ID size	
				abs	% pre	abs	% pre
BIRD	1	37	36	25	170	25	161
ORDPATH	2	53	37	26	186	36	240
(non-dyn)	2	52	36	25	179	35	233
Virtual N.	1	95	37	25	174	64	413
PID	1	28	21	14	99	19	122
preorder	1	23	21	14	100	15	100

scheme	ID size (bits)			total storage (MB)			
	min.	max.	avg.	var ID size		fixed ID size	
				abs	% pre	abs	% pre
BIRD	1	44	43	113	188	113	177
ORDPATH	2	86	48	124	207	221	345
(non-dyn)	2	77	43	111	185	198	309
Virtual N.	1	198	81	210	350	508	794
PID	1	29	20	54	90	74	116
preorder	1	25	23	60	100	64	100

10 Storage Consumption Comparison

The storage consumption of various identification schemes on *DBLP* and *XMark* are given in Table 5. The first three columns after the scheme name contain the minimum, maximum and average number of bits used for a single ID, respectively. The remaining columns list the storage needed for all IDs together, both as an absolute value in MB in columns 5, 7, and relative to the corresponding result obtained for the preorder scheme (columns 6, 8). In a first calculation we sum up the exact bit counts needed for all IDs, assuming that IDs can be stored with variable size (columns 5, 6). On the other hand, it may be more realistic to assume that when stored in the database, all IDs assigned to nodes in the same document collection take up the same space. The total storage taken up by such fixed-size IDs appears in columns 7, 8. The BIRD scheme almost always takes up considerably less space than ORDPATH and especially Virtual Nodes, the two schemes which are closest to BIRD in terms of expressivity (see Section 7). When assigning fixed-size IDs, BIRD reduces the space consumption by nearly a factor 2 for ORDPATH and between 2.2 and 4.5 for Virtual Nodes.

11 Robustness Comparison

Among all identification schemes discussed here, only ORDPATH supports unlimited updates. BIRD, PID and Virtual Nodes only allow for a limited number of node insertions, until an overflow occurs when a node has more children than its ID range allows for. If this happens, a global (for BIRD and Virtual Nodes) or local (for PID) reallocation of IDs is necessary.

In some scenarios, updates occur either rarely (like in static databases containing, e.g., medical, juridical, geographical or historical information), or new data are first collected and then added to the database in a bulk update once in a while (e.g., in digital archives, linguistic corpora, encyclopedias and dictionaries, product catalogues, or digital libraries). Yet for other scenarios the robustness of an ID scheme against node insertions is an important requirement. We empirically tested the robustness of BIRD IDs using a large real-world data set.

We indexed the 8.6 GB *IMDb* collection ([27], nearly 2,000,000 documents converted to XML) in chunks of 100,000 documents, reserving extra IDs for 100 potential child node insertions below any overflowing node during the weight calculation. Figure 5 (solid line, "BIRD + 100") illustrates how often at least one weight in the DataGuide was changed while adding 100,000 documents, thus causing a reindexing of the entire collection. The two peaks at the beginning show that BIRD weights were reasonably stable after indexing the first 400,000 documents, or 20% of the data. In the sequel, only one additional weight update is necessary before adding 1,300,000 documents without any overflow. Note that reserving extra IDs to increase the robustness of the scheme is not expensive in terms of storage: the greatest BIRD ID in the extra-sparse encoding ("BIRD + 100", at most 54 bits per ID compared to 45 bits for ordinary BIRD) still occupies far less than 64 bits, a critical boundary in our runtime experiments.

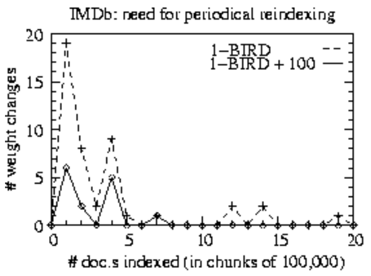


Fig. 5. Robustness of BIRD IDs for *IMDb*. Dashed line, BIRD; solid line, extra-sparse BIRD

We are currently looking for variants that make BIRD IDs even more robust against updates. [20] outlines the *layered BIRD scheme* that allows to construct hierarchical BIRD IDs similar to Dewey IDs [23] and ORDPATH [25]. The layers of an ID can be adapted to specific needs and do not have to follow the tree levels as in Dewey Order and ORDPATH. Layers can be introduced at critical positions in the underlying DataGuide and allow unlimited node insertions directly below a layer boundary. In the extreme case, there are as many layers as levels in the tree, allowing for unlimited updates in general. In this situation BIRD coincides with ORDPATH. All decision and reconstruction operations are easily adapted to the layered variant. First experiments with layered BIRD are promising and prove that robustness is not necessarily conflicting with space or time efficiency.

12 Conclusion

In this paper we introduced the BIRD family of tree numbering schemes based on structural summaries that allows to decide and reconstruct tree relations efficiently with simple arithmetic operations. We showed that decision and reconstruction of tree relations is a central building block of most query strategies. We analyzed and compared properties and expressivity of other node identification schemes and identified a trade-off between evaluation time, storage consumption and expressivity, where BIRD appears to be a favourable choice. We presented the results of extensive tests, proving that BIRD is almost always faster than ID schemes of comparable expressivity (up to two orders of magnitude), while being reasonably small in size.

As an application of this work, we are currently integrating BIRD with a purely relational retrieval system for conjunctive XML queries. Future work may include a generalization of the notion of structural summaries in order to reduce the storage consumption of BIRD IDs further. Besides, the update mechanism mentioned above (*layered BIRD scheme*) needs to be elaborated in detail.

Acknowledgements

We would like to thank Gerhard Weikum and Ralf Schenkel (MPI Saarbrücken) for their XML convertor for the *IMDb* collection, and Sebastian Hick (CIS, University of Munich) for helpful comments on the experiments.

References

1. Gottlob, G., Koch, C.: Monadic Datalog and the Expressive Power of Web Information Extraction Languages. Journal of the ACM **51** (2004) 74–113
2. University of Pennsylvania: The Penn Treebank Project. (Available at `www.cis.upenn.edu/~treebank/home.html`)
3. Boag, S., Chamberlin, D., , Fernández, M.F., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML Query Language. W3C Working Draft (2004)
4. Berglund, A., Boag, S., Chamberlin, D., Fernández, M.F., Kay, M., Robie, J., Siméon, J.: XML Path Language (XPath) 2.0. W3C Working Draft (2004)
5. Schlieder, T., Naumann, F.: Approximate tree embedding for querying xml data. In: Proc. ACM SIGIR Workshop On XML and Information Retrieval. (2002)
6. McHugh, J., Abiteboul, S., Goldman, R., Quass, D., Widom, J.: Lore: A Database Management System for Semistructured Data. SIGMOD Record **26** (1997) 54–66
7. Baeza-Yates, R.A., Navarro, G.: XQL and proximal nodes. Journal American Society for Information Science and Technology (JASIST) **53** (2002) 504–514
8. Kanne, C.C., Moerkotte, G.: Efficient Storage of XML Data. In: Proc. 16th Int. Conf. on Data Engineering (ICDE). (2000) 198
9. Li, Q., Moon, B.: Indexing and Querying XML Data for Regular Path Expressions. In: Proc. 27th Int. Conf. on Very Large Data Bases (VLDB). (2001) 361–370
10. Grust, T., Sakr, S., Teubner, J.: XQuery on SQL Hosts. In: Proc. 30th Int. Conf. on Very Large Data Bases (VLDB). (2004) 252–263
11. Pal, S., et al.: Indexing XML Data Stored in a Relational Database. In: Proc. 30th Int. Conf. on Very Large Data Bases (VLDB). (2004) 1134–1145
12. Meuss, H., Schulz, K.U., Weigel, F., et al.: Visual Exploration and Retrieval of XML Document Collections with the Generic System X^2. Journ. Dig. Lib. **5** (2005) 1–70

13. Gottlob, G., Koch, C., Schulz, K.U.: Conjunctive Queries over Trees. In: Proc. 23rd ACM Symposium on Principles of Database Systems (PODS). (2004) 189–200
14. Lee, Y.K., Yoo, S.J., Yoon, K., Berra, P.B.: Index structures for structured documents. In: Proc. 1st ACM Int. Conf. on Digital Libraries. (1996) 91–99
15. Bremer, J.M., Gertz, M.: An Efficient XML Node Identification and Indexing Scheme. Technical Report CSE-2003-04, University of California at Davis (2003)
16. Zhang, C., et al.: On Supporting Containment Queries in Relational Database Management Systems. In: Proc. 20th ACM SIGMOD Conference. (2001) 425–436
17. Al-Khalifa, S., et al.: Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In: Proc. 18th Int. Conf. on Data Engineering (ICDE). (2002) 141–152
18. Chien, S.Y., Vagena, Z., Zhang, D., Tsotras, V.J.: Efficient Structural Joins on Indexed XML Documents. In: Proc. 28th Int. Conf. on Very Large Data Bases. (2002) 263–274
19. Goldman, R., Widom, J.: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In: Proc. 23rd VLDB Conf. (1997) 436–445
20. Weigel, F., Schulz, K.U., Meuss, H.: The BIRD Numbering Scheme for XML and Tree Databases. Technical report, CIS, University of Munich (2005) `http://www.cis.uni-muenchen.de/~weigel/Literatur/weigel05birdtech.pdf`.
21. McHugh, J., Widom, J., Abiteboul, S., Luo, Q., Rajamaran, A.: Indexing Semistructured Data. Technical report, Stanford University, Computer Science Dept. (1998)
22. Grust, T.: Accelerating XPath location steps. In: Proc. 21st ACM SIGMOD Int. Conf. on Management of Data. (2002) 109–120
23. Tatarinov, I., et al.: Storing and Querying Ordered XML Using a Relational Database System. In: Proc. 21st SIGMOD Int. Conf. on Management of Data. (2002) 204–215
24. Milo, T., Suciu, D.: Index Structures for Path Expressions. In: Proc. 7th Int. Conf. on Database Theory (ICDT). (1999) 277–295
25. O'Neil, P., et al.: ORDPATHs: Insert-Friendly XML Node Labels. In: Proc. 23rd ACM SIGMOD Int. Conf. on Management of Data. (2004) 903–908
26. Dietz, P., Sleator, D.: Two Algorithms for Maintaining Order in a List. In: Proc. 19th ACM Symposium on Theory of Computing (STOC). (1987) 365–372
27. IMDb: Internet Movie Database. (Available at `www.imdb.com`)

Efficient Handling of Positional Predicates Within XML Query Processing

Zografoula Vagena[1], Nick Koudas[2], Divesh Srivastava[3], and Vassilis J. Tsotras[1]

[1] UC Riverside
{foula,tsotras}@cs.ucr.edu
[2] University of Toronto
koudas@cs.toronto.edu
[3] AT&T Labs-Research
divesh@research.att.com

Abstract. The inherent order within the XML *document-centric* data model is typically exposed through `positional predicates` defined over the XPath navigation axes. Although processing algorithms for each axis have already been proposed, the incorporation of positional predicates in them has received very little attention. In this paper, we present techniques that leverage the power of existing, state of the art methods, to efficiently support positional predicates as well. Our preliminary experimental comparisons with alternative approaches reveal the performance benefits of the proposed techniques.

1 Introduction

XML is gradually becoming the standard for data sharing and information exchange among B2B and other applications over the Internet. Thanks to standard specifications for web services (such as SOAP, WSDL, etc.), user programs can receive requests for data (specified in XML) and return their answers tagged in XML. In addition, via the use of specific query languages such as XPath [6] or XQuery [7], users and applications can compose declarative specifications of their interests, as well as filter and transform data items represented in XML. This widespread acceptance and employment of XML, calls for novel data processing techniques, pertaining to XML's effective storage and retrieval. One key issue in XML query processing is the effective support of the *ordered* model for data and document representation that the language employs. Order is particularly important in the *document-centric* view of the language as it enables exposing the logical structure of the document. Popular XML query languages (e.g., XQuery [7] and XPath [6]) express order sensitive queries through the ability to address parts of a document in a navigational fashion, based on the sequence of nodes in the document tree.

Consider, for example, an XML database of journal articles. For each article entry, the order in which its authors and its sections are listed is relevant. Information extraction tools can take into consideration the tree structure and the positional node order to extract the first two authors of each article, with the following query:

```
//article/child::author[position()<=2]
```

Similarly, the section of an article immediately following the *Related Work* section of the article can be extracted using the query:

```
//article/child::section[./child::title =
   'Related Work']/following-sibling::section[1]
```

Supporting such ordered predicates is thus an important requirement for XML query processing. In the current work we focus on the efficient processing of `positional predicates` within the various XPath navigation axes.

Initial efforts [11, 14] investigated the extent to which relational database technology can support order-based queries. These efforts have concluded that although a relational database performs well in general, in many cases it needs to be extended (i.e. new processing algorithms should be devised) to efficiently support general order-based queries. Although processing techniques that employ direct navigation of the XML tree structure can be trivially adapted to identify nodes that satisfy any positional predicate, recent works have shown that, in the absence of those predicates, such navigational techniques are outperformed by set-based approaches [4, 8, 16]. As a result, it would be beneficial if the latter techniques could be extended to support positional predicates as well.

Set-based solutions transform axes predicates to value conditions and thus reduce the navigation problem to a relational join computation (with the additional requirement of producing the final output in a specific order). Such techniques were originally proposed for queries where navigation is restricted to the `child` or `descendant` axes ([4]). Very recently, set-based methods that take into consideration other navigation axes have appeared ([12, 13, 15]). Moreover, *holistic* solutions (i.e., where multiple steps are grouped and computed as a single operator), have been shown to outperform step-by-step processing, as they avoid unnecessary computations of intermediate results that do not participate in the final answer of the query [8, 9, 15]. All these works introduce new, tree-aware operators to speed up tree traversal, however, none of them considers the existence of positional predicates.

Taking into consideration the abundance and efficiency of processing techniques for the ordered axes, the aim of this paper is to investigate whether such techniques can be modified or extended so as to handle positional predicates as well. In the process, we came to the conclusion that, contrary to common belief, efficient support of positional predicates is non-trivial. Furthermore, not all of the above techniques are applicable when positional predicates are present.

Consider, for example, the `Staircase Join` [12] family of algorithms. Their efficiency is attributed to the fact that while processing queries with navigation, not all context nodes are needed in order to identify the result nodes. Consider the document in Figure 1.a and assume that the `descendant` axis query: `a/descendant::*` is invoked over this document (which finds all descendants of *context* nodes `a`). Since the descendants of a particular document node `x` are also descendants of any ancestor of `x` in the same document, only the context nodes that do not have other context nodes as ancestors are necessary for answering

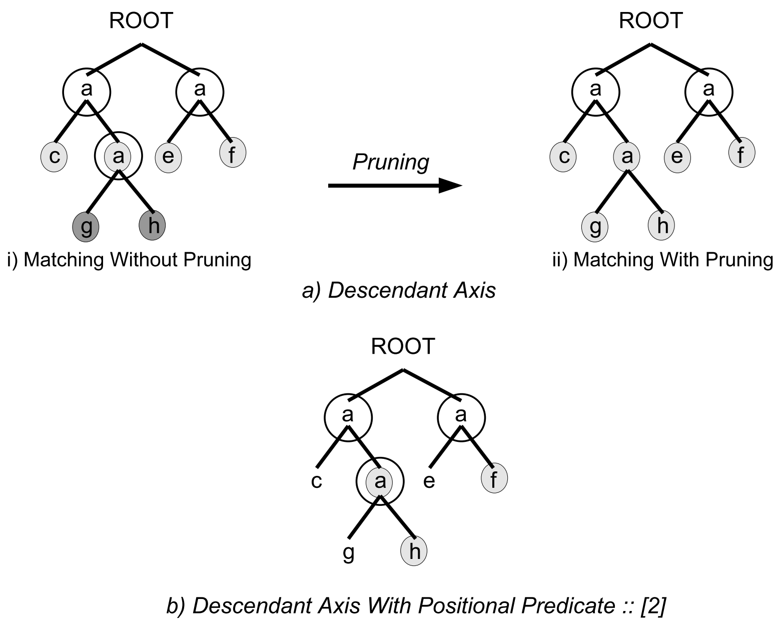

ROOT
a
c
e
f
g
h
Pruning
i) Matching Without Pruning
ii) Matching With Pruning
a) Descendant Axis
b) Descendant Axis With Positional Predicate :: [2]

Fig. 1. Inapplicability of Staircase Join in the Presence of Positional Predicates

this `descendant` axis query. In terms of the particular example, only the first and third (in document order) context nodes that are labeled `a` are necessary, while the second `a` can be safely discarded. Similar observations can be made for the other axes as well (e.g. `following`, `following-sibling` etc).

By discarding unnecessary context nodes, one can save matching time from the duplicate results that would otherwise be produced and subsequently be discarded as superfluous. In Figure 1.a(i), if the second `a` node is not discarded, the `g` and `h` nodes are produced twice. Instead, the `Staircase Join` involves a first phase, where it prunes unnecessary context nodes, before performing the actual matching with the appropriate input descendant streams. Figure 1.a(ii) illustrates the matching process after the pruning phase (the result nodes are depicted in gray). Note that for the pruning process, the algorithm only considers: (i) which is the axis involved in the query, and, (ii) the set of context nodes.

Assume now that the positional predicate [position() = 2] is added to the `descendant` axis of the previous example. That is, the query requests only the second descendants under the context nodes. In this case the pruning performed by the `Staircase Join` is not applicable as it would result in false negatives. By discarding the second `a` node in the first phase, one cannot identify the `h` node that belongs to the result, as the second descendant of this `a` node. Preserving only the first `a` node is not enough because `h` is not the second descendant of this `a`. The situation is illustrated in Figure 1.b. While unnecessary context nodes can still be pruned, to identify them, the actual matching over the input streams has to be performed anyways. As a result, the optimization introduced by the `Staircase Join` is inapplicable when positional predicates are present.

In fact, there is a straightforward technique so that positional predicates can be answered as well. In particular, we can: (a) identify all the matching combinations of context and result nodes that satisfy the axis predicate (using any applicable processing algorithm), (b) group on the context nodes and sort each such group on the result nodes, (c) filter from each group the pairs that satisfy the numeric predicate while preserving the result nodes, and, (d) take the union of the result nodes. However this straightforward solution, although acceptable, might lead to unnecessary computation, as it incurs the overhead of identifying context nodes that do not have any matching counterpart that also satisfies the positional predicates. For example, assume the query `a/child::b[4]`, where the fourth `b` child of `a` nodes is requested. If in the document no `a` node has more than three children, then this approach will incur a lot of unnecessary overhead. The question is whether the *"useful"* context nodes can be identified, *prior* to performing all possible matchings. In this paper we will present algorithms that can identify such useful context nodes by effectively maintaining appropriate states of the computation. The contributions of this paper are summarized as:

1. We investigate the problem of supporting queries with positional predicates over XML data. We extend existing state of the art processing algorithms for each of the proposed navigation axes so as to effectively handle positional predicates. To the best of our knowledge, this is the first work that provides a complete, scalable, XML model-aware solution.
2. We target branched queries that may contain *any* number of axes of the same type as well as their backward counterparts (e.g., `descendant` and `ancestor`, or `following-sibling` and `preceding-sibling` etc.)
3. We present a preliminary comparison with (i) the naive solution which checks positional predicates as a post-processing step, (ii) previous techniques that leverage the relational engines to efficiently support tree shaped XML data, and (iii) DOM based XPath query processors. The experimental evaluation reveals the performance advantages of our solutions.

We proceed with Section 2 that provides necessary background while our methods are described in Section 3. In Section 4 we experimentally investigate the performance of the proposed solutions. Finally, Section 5 concludes the paper.

2 Background

Ordered Model. In the data model employed by popular XML query languages [6, 7], a document is represented as an ordered tree, where each node has a unique ID. The *document order* is defined over all nodes and corresponds to the sequence in which nodes occur within the pre-order traversal of the ordered tree [10].

Order Based Predicates. Positional predicates appear in XML query languages in the form of `numeric predicates`. A predicate, filters a sequence of nodes with respect to an axis to produce a new sequence of the nodes for which the predicate evaluates to true. `Numeric predicates` specifically take order into account and are of the form: `position()` *op* n, where `position()` returns the position of the node within the axis node set, *op* is one of the operators $=, <, >, <=, >=, !=$ and n is an integer. If a predicate expression evaluates to a number, the result will be true if the number is equal to the context position, otherwise it is false. Thus, a location path `para[3]` is equivalent to `para[position()=3]`. For ease of presentation we describe our techniques in terms of the $=$ operator while pointing out the necessary changes to support the other operators.

Document Representation. The approaches we are going to discuss in this paper project the document into sequences of nodes (we call them `element lists` from now on). There is one sequence per document tag, which maintains all nodes with the same tag. Each node is augmented with information that identifies its position within the XML tree. The position of a node is represented with the triplet: `(left, right, pright)` where: (a) `left` and `right` are generated by counting tags from the beginning of the document until the start and the end tags of the node are visited, respectively, and, (c) `pright` is the `right` value of its parent node. Using this scheme, the relative positions between any two nodes can be identified in constant time. A node y is a child of a node x if the `right` value of x is equal to the `pright` value of y. Similarly, a node y is `following-sibling` of a node x if the `right` value of x is smaller than the `left` value of y and the `pright` value of x is equal to the `pright` value of y.

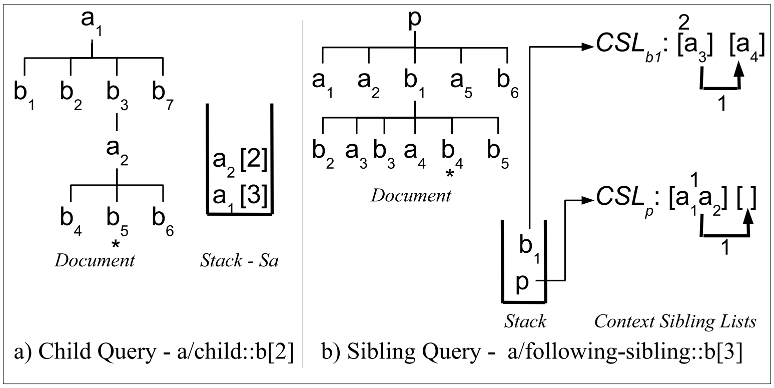
3 Algorithmic Approaches

We proceed with the description of our algorithms for supporting XML queries with positional predicates. For the discussion we focus on the `child` and the `following-sibling` axes, and their backward counterparts. The remaining axes can be supported with trivial modifications to the proposed methods. For simplicity we restrict the comparison operator to `equality`. Moreover, while our algorithms follow a set-based evaluation, the results can be easily returned in document order (or any other desired order) providing list-based semantics.

3.1 One-Step Queries

Such queries have the form: `a/child::b`$[n]$ or `a/following-sibling::b`$[n]$, where n is a numerical value. The former query identifies the n^{th} b child of each a document node, while the latter query identifies the n^{th} b sibling of each a document node, which resides after that a in document order.

Child Axis. The optimal processing technique for one-step queries with child axis ('structural joins') was presented in [4]. The input is provided in the form of `element lists` (i.e. the element list of the a nodes and the element list of the b nodes). Each list is sorted on the `left` value of the nodes it contains. The lists for `a, b` are joined in a merge-like fashion. A stack S_a is employed to maintain those `a` nodes that have been visited and whose descendants (and their children) are currently being accessed. At all times during the execution of the algorithm,



Document
Stack - Sa
a) Child Query - a/child::b[2]
Document
Stack
Context Sibling Lists
b) Sibling Query - a/following-sibling::b[3]

Fig. 2. Processing of positional predicates within one-step queries

each node in S_a is a descendant of the nodes below it in the stack. When a b node is visited, the parent-child relationship, if any, is determined between that node and the current top of S_a. If the accessed b node is not a descendant of the current top of the stack, then it is guaranteed that the top of the stack will not produce further matchings and can thus safely be removed from S_a. The process is then resumed with the new top of the stack.

From the previous discussion it is obvious that in the absence of positional predicates, the relationship between a specific pair of a and b nodes can be determined based solely on their node numberings; the stacks are useful to efficiently identify relationships between sets of a and b nodes. When dealing with positional predicates, however, node numberings do not suffice to determine whether a specific pair of a and b nodes satisfy `a/child::b`[3], for example. Additional state needs to be maintained while processing the sets of a and b nodes to make this inference.

For addressing positional predicates, we note that when a b node is accessed, its a parent (if any) resides in the top of the stack. Moreover, a node a is maintained in S_a until all its children are visited. Since such children are accessed from their element list in document order, the stack provides the appropriate means to keep track of the children nodes for each of the a nodes residing in S_a. In order to achieve that, for each a element already in the stack, we maintain a (child) *counter*. Each time a new b element is accessed and is determined to be child of the current top of the stack, the corresponding counter is incremented. The new value of the counter determines the participation of the b node in the result. In particular, if this value equals to the numerical value n of the positional predicate, the node participates in the result and is joined; otherwise it is discarded. Figure 2.a illustrates the stack state after having accessed node b_5. At this point in the execution, the third child of node a_1 and the second child of node a_2 have been accessed so far.

Following-Sibling Axis. An advantage of the child axis algorithm is that it can perform the join in a *single* pass over the two input element lists. Very recently, novel merge-like algorithms with the same one-pass property over the input element lists have been independently proposed for the following-sibling axis as well [13, 15]. Here we extend the ideas summarized in [15] so as to support positional predicates as well.

An important observation is that since the input is visited in document order, the following-siblings for some a node occur after all its descendants have been encountered. Equivalently, an a node has to be buffered until the following-siblings of its descendants have been processed first. Moreover, an a node may have other a nodes as following siblings. We call the a nodes that share the same parent as "context-siblings". Context-sibling nodes conceptually form a linked-list, called the `Context-Sibling List`, or CSL in short, which is associated to the parent node of the context-siblings. For example, Figure 2.b shows snapshots of two CSLs, associated with parent nodes `p,` b_1. New a nodes that are context-siblings are appended at the end of their parent's CSL. CSLs provide an effective way to capture the following-sibling predicate: If a b node becomes a following-sibling to the node at the end of a context-sibling list, it is also a following-sibling to all other nodes currently in that list. There can be many CSLs at a given time (up to the depth of the currently accessed context node in the document tree).

A stack keeps track of the existing CSLs. Each node in the stack is the common parent of the nodes within the corresponding CSL. Moreover it is a descendant of the nodes below it. When a node is removed from the top of the stack (i.e. when all its children have been accessed), its corresponding CSL (if any) is erased. One important point to note here is that although a stack is employed for the processing of both the `child` and the `following-sibling` axis, the contents of the stacks are very different for each algorithm. In particular in the case of the `child` axis, stack entries are from the list of the context nodes, while for the `following-sibling` axis stack entries represent parents of the context nodes. The latter is possible because of the `pright` value that is maintained in the positional representation of each document node.

When the following-sibling query contains a positional predicate, only those following-siblings with position that satisfies the predicate should be joined. Our solution is built on the properties: (i) a context element is accessed before any of its following-siblings, (ii) it is maintained within its CSL, and (iii) its following-siblings are accessed in document order. Therefore, a *counter* could be maintained for each context element in a CSL, that keeps track of the number of following-siblings that have already been identified with respect to this node. In this case whenever a new following-sibling is accessed, the counters of *all* context elements currently within the corresponding CSL would be incremented and only those pairs (if any) that satisfy the numerical predicate would be returned.

Although the previous way of updating the counters is correct, it is not optimal. For example, if there are N a's and b's that are all siblings of each other, then this approach would take time $O(N^2)$ in the worst case for answering `a/following-sibling::b`[1], even though the output is only $O(N)$. In order to achieve optimality we propose a more complex way of updating the counters, which divides the nodes that reside within a particular CSL into several partitions. We call those partitions `predicate groups` or PGs from now on. Each PG maintains all the nodes whose counters have the same value (i.e. a nodes for which the same number of following-sibling b nodes have been accessed). Moreover each PG is associated with a counter, which is the value of the counter of each node within this PG. The values of those counters from the beginning to the end of a particular CSL list follow a strictly descending order. Furthermore, there exists a pointer from a PG pg_1 to the PG whose counter has the largest value that is smaller than the counter associated with pg_1. Each such pointer is associated with a value, which represents the difference of the values of the counters associated with the two connected PGs. An example of this data structure is represented in Figure 2.b, which captures the state of the algorithm, after node b_4 has been accessed.

Using the above structure, only a constant number of actions have to be performed for counter updates. In particular, when an a node is visited it is appended within the PG that resides at the end of a's associated CSL. When a node b that is associated with a particular CSL is visited, the value of the counter associated with the PG at the beginning of this CSL (we call this PG pg_1) is incremented by one. If no empty PG exists at the end of the CSL, a new empty PG (pg_2) is created at the end of this CSL and the PG that was previously at the end of the CSL is set to point to pg_2, with an associated value of 1. If an empty PG pg_3 exists in the end of the CSL, the value associated with the pointer that points to this pg_3 is incremented by one. Subsequently, if the value of the counter associated with pg_1 does not match the value of the predicate the algorithm resumes with the next document node. Otherwise, the node b is matched with the nodes in pg_1 and the results are returned. At this point the nodes of pg_1 can be discarded as they are not going to create any future results. The counter of the PG referenced by pg_1 is set to the value of the counter of PG minus the associated difference. The following theorem holds for the correctness and performance of processing positional predicates for one step queries (the pseudocode is provided in the appendix):

Theorem 1. *Given a one step query T with an equality positional predicate,* `a/axis::b`$[n]$*, where* `axis` *can be* `following-sibling` *or* `child`*, which is invoked over a database D, the proposed processing of the intermediate state for each algorithm correctly identifies all nodes that participate in answers for T in D. Moreover, they have worst-case I/O and CPU time complexities linear to the sum of the sizes of the input lists. Furthermore, the maximum space used is $O(h * f)$, where h is the height of the XML data tree, and f is the maximum fanout at any data tree node.* □

For the one step queries of the theorem, output size is bounded by the sum of the sizes of the input lists. This is not the case for other operators like $position() > 2$. In such cases, our algorithms can be modified to yield correct results with worst-case I/O and CPU time complexities linear to the sum of the sizes of the input lists and the output.

Similar buffering (of context nodes) can be performed for each of the remaining forward axes and hence the functionality of the *counters* can be easily realized for all forward axes. We thus proceed with our discussion of multi-step queries by focusing on the `following-sibling` axis.

3.2 Multiple Step Queries with *Following-Sibling* Axes

Multiple-step queries involve many query nodes. Each node q_i, may be connected with zero or more other nodes through `following-sibling` axes. We call each q_i that is connected to one or more other nodes a *step-parent* from now on. Such queries can be represented with tree structures whose vertices correspond to query nodes and the edges correspond to the connecting `following-sibling` axes.

One straightforward way to compute a multi-step query with positional predicates, is to divide it into individual steps and use the techniques described in Section 3.1. Nevertheless, this may lead to unnecessary processing of intermediate results. When no positional predicates are present, a more efficient approach is to regard the whole query as a *single* operant and holistically compute the results with a single pass over the input. The proposed algorithm traverses the input in document order and attempts to identify the nodes that satisfy all the predicates of the query at once.

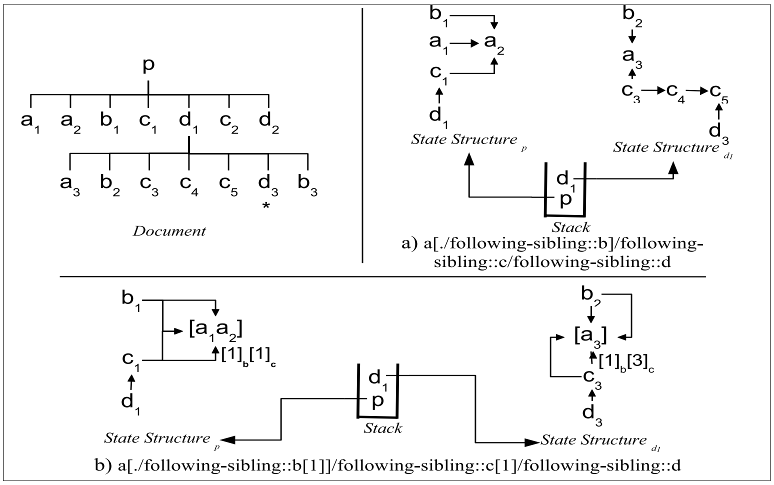


Fig. 3. Processing of positional predicates within multiple-step queries

An important observation that the algorithm utilizes is that any result instance involves only sibling nodes (i.e. nodes that have the same parent). For each such *sibling group* a *state structure* is maintained, which encodes all partial and total results that can be produced from this group with size proportional to the size of the sibling group. The *state structure* consists of one CSL per *step* *parent*. The role of a CSL is similar with that in Section 3.1, i.e. to hold context-siblings in document order. An element y within a CSL, maintains a reference (the `step-pointer`) to the latest (in document order) element within the CSL that corresponds to its query parent, with which y is matched. All elements from the beginning of this CSL until the node referenced by the `step-pointer` of y can be matched with y. At each point during the execution of the algorithm multiple *sibling groups* (up to the height of the document tree) can be active (i.e. some but not necessarily all of their nodes have been accessed). A stack is utilized to keep track of all active *sibling groups*. An example of the state structure that is maintained is illustrated in Figure 3.a, where the *state structure* after node d_3 has been accessed is presented.

When positional predicates exist within the multi-step query, counters associated with context nodes within CSLs are also maintained. These counters enable tracking of the following-siblings that have already been seen. PGs are utilized to enable efficient update of those counters in a similar way as described in Section 3.1. The only difference is that when the positional predicate is satisfied, the PG with the matching nodes cannot be discarded until the actual matching takes place. As a result, an additional pointer needs to be maintained pointing to the PG whose counter is updated at each point. To be able to identify the PGs with the matching nodes when total results are produced, the element y maintains two *step-pointers*, one at the first and one at the last PG. All the PGs in between those references contain nodes that can potentially be matched with y. Figure 3.b shows the *state structure* for the multi-step query of Figure 3.a where positional predicates have been added. The computation is depicted right after node d_3 has been accessed.

3.3 Backward Axes

When the multiple-step query contains both *following-sibling* and *preceding-sibling* axes, an adaptation of the method proposed in [5] can convert the query into one with only `following-sibling` axes [15]. The resulting query can be represented with a DAG structure whose vertexes correspond to the query nodes and edges to `following-sibling` axes. In the absence of positional predicates, the additional issue (when compared to the queries in Section 3.2) that needs to be addressed is that a query node may have multiple parents. We call such query nodes as `join` nodes from now on. Our multiple-step algorithm can be easily extended to check those additional predicates. More precisely, a document node y that becomes a `join` node must have at least one corresponding sibling within each of the CSLs of its query parents, before it is inserted within its corresponding CSL, or it triggers the production of results (if y corresponds to a leaf query node).

When considering positional predicates we differentiate whether they are specified on a forward or a backward axis in the original query. If one (or more) of the edges of the DAG that corresponds to a `following-sibling` axis in the original query is associated with a positional predicate, the counters described in

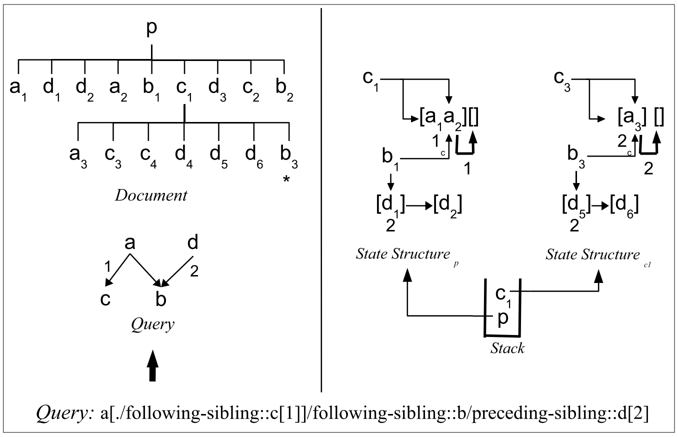

p
a1 d1 d2 a2 b1 c1 d3 c2 b2
a3 c3 c4 d4 d5 d6 b3
*
Document
a d c b 1 2
Query
c1 [a1a2][] 1 1 b1 [d1]→[d2] 2
State Structure p
c3 [a3] [] 2 2 b3 [d5]→[d6] 2
State Structure c1
c1 p
Stack
Query: a[./following-sibling::c[1]]/following-sibling::b/preceding-sibling::d[2]

Fig. 4. Integration of positional predicates with backward axes

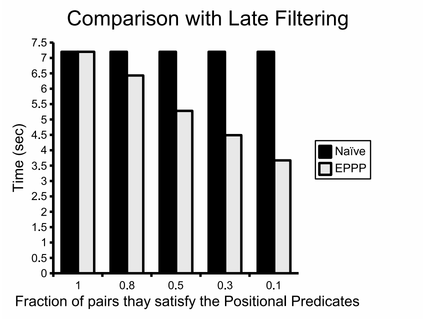
Section 3.2 provide the necessary means to identify the nodes that satisfy those numeric predicates.

If, on the other hand, the positional predicate is on an edge e that corresponds to a `preceding-sibling` axis in the original query, then the following necessary condition is needed so as a document node y that corresponds to the destination node of e, participates into a result. In particular, there must exist a document node x that corresponds to the source node of e that is the n^{th} preceding sibling of y in *reverse document order*. Nevertheless, such a predicate is easy to check, because our algorithm will have already buffered all the necessary preceding-siblings of y within their associated CSL. As a result, we only need to check whether there exist n elements within this CSL (starting however from the *end* of the list). If such an x element exists, it is the only matching for the y element under consideration. A reference from y to x is then maintained and will be used later in the result production phase. In order to identify those x elements efficiently we utilize the same method as in the case of the forward axes, and we partition the nodes within each CSL into PGs. In the case of the backward axis, however, each PG will contain at most one node and as a result, there is no need to maintain pointers between PGs. Furthermore, in this case it is also unnecessary to maintain the differences between the PG counter values, as those are always equal to 1.

As an example, consider the query in Figure 4, where two positional predicates exist, one on a following-sibling and one on a preceding-sibling axis. The figure also presents the intermediate state after node b_3 has been accessed.

4 Experimental Evaluation

In order to investigate the effectiveness of our techniques, we performed a number of experiments over synthetic, benchmark and real data.



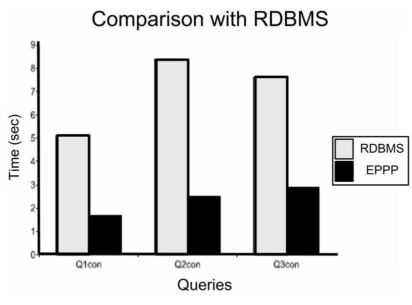


Fig. 5. (a) Comparison with late filtering, (b) Comparison with an RDBMS

4.1 Experimental Setup

We implemented all the algorithms in C++ on top of a native storage manager. All the experiments were conducted on a 2.6 GHz Pentium 4 with 512MB of main memory running RedHat Linux 9. The code was compiled with the GNU compiler version 3.2.2.

To evaluate join performance we measure total execution time for each algorithm. The times with hot cache are reported. We used 8K pages and a 100 block buffer cache.

For the purposes of this section, we refer to our proposed algorithms under the name `EPPP` (for Effective Positional Predicate Processing). We begin by investigating the importance of early identification of nodes that satisfy the positional predicates. We also compare the performance of `EPPP` with a relational DBMS. Finally, we investigate the behavior of `EPPP` with regard to existing main memory XPath processors (Xalan and Saxon, available in [3] and [1]).

4.2 Importance of Early Filtering

This group of experiments investigates whether it is advantageous to identify the context nodes that satisfy the positional predicates *before* performing the matching process. For this experiment we assumed the simple one-step query `a/following-sibling::b`$[position() <= n]$ and generated a large synthetic input dataset (around 100,000 a nodes and around 1 million b nodes). Without any positional predicate the participation of input nodes in the result is 100%.

We first considered the query where no positional predicates exist and compared EPPP with the "positional predicate oblivious" processing algorithm for the following-sibling axes (we call it the `naive` algorithm from now on). Subsequently, we begun changing the value of n, so that at each time only a proportion of the previous results, satisfies the query. For each case we measured the time to completion of each algorithm. For the `naive` algorithm we identify all the results and then filter out the ones that do not satisfy the predicates.

Figure 5.a presents the results. Clearly, when no positional predicates exist, the two algorithms perform very similar. Hence the overhead introduced in

EPPP is very small. This is because the examination of predicate satisfaction has been integrated within the original processing algorithm by introducing simple computations (like counter incrementing or simple boolean expression checking) only among objects that the `naive` algorithm would also access. However, when positional predicates are present, EPPP is becoming more efficient. In particular, the more selective the predicates, the larger the performance gap between the two methods. This is due to the fact that EPPP can discard unnecessary nodes early and before performing the actual matching, while the `naive` algorithm has to incur the overhead of creating all answers, and discard them in a later, post-processing step.

It should be noted that the time differences are expected to be much larger for multiple step queries. This is because the overhead of creating the results is higher (due to the recursive traversal of the CSL through the step pointers). As a result, the performance gain of EPPP will become even more prominent.

4.3 Comparison with an RDBMS

The next set of experiments compares the performance of the EPPP techniques with a commercial relational DBMS, so as to compare against a pure relational approach, that translates the query into SQL. In particular, we followed an approach that combines ideas from [14] and [16]. We first created one table for each of the input streams, where we stored the positional representation of each stream node indexed on `left` and `right`. The dataset used was the 1G (text) database generated by the XMark benchmark, with 825043 `incategory` nodes, 217500 `mailbox` nodes, and 217500 `location` nodes. We used the following-sibling axis queries shown in Table 1(a). Query Q1 is a single-step query, while Q2 and Q3 are multi-step queries.

Table 1. Queries for (a) XMark and (b) real data

Q1 :	incategory/following-sibling::mailbox[2]
Q2 :	location/following-sibling::incategory/following-sibling::mailbox[2]
Q3 :	location[./following-sibling::incategory]/following-sibling::mailbox[2]
Q4 :	LINE/following-sibling::STAGEDIR[2]
Q5 :	TITLE/following::STAGEDIR[2]
Q6 :	PERSONAE[/descendant::PGROUP[2]]/descendant::TITLE

To produce the SQL code, we followed the technique described in [14] and used the *Rank*() function to produce the results. The relational query was pre-optimized, and in order to decrease the overheads of the logging and recovery subsystems, the query (being the only user query running in the system at the time of the experiment) was evaluated in 'READ UNCOMMITED' access mode.

The results are presented in Figure 5.b; clearly, the relational DBMS approach performs worse than EPPP. As also mentioned in [12, 14], this is because the RDBMS is agnostic to the tree shape of the data and cannot take advantage of it. Moreover, when the number of steps increases, the generated SQL becomes increasingly complex, which is more prone to optimization errors. Our results confirm the observation that new operators must be added for a relational DBMS to efficiently support XML data retrieval (even more so when positional predicates are present).

4.4 Comparison with XPath Engines

We then compared EPPP with two commonly used XPath engines, namely Xalan [3] and Saxon [1]. Their use is limited to main memory data. For these experiments we utilized real data sets, namely Shakespeare's plays [2]. Each of the documents is small enough to be held in main memory. We evaluated queries Q4, Q5 and Q6 shown in table 1(b) and present the average execution time.

Table 2. Comparison with Xalan and Saxon

	Xalan		Saxon		EPPP	
	with	w/o	with	w/o	with	w/o
Q4	0.20	0.21	0.20	0.12	0.08	0.05
Q5	0.41	0.41	0.18	0.14	0.05	0.07
Q6	0.22	0.21	0.21	0.11	0.04	0.04

The results appear in Table 2. For each query, we report the time with the positional predicate and the time without it. These results show that the EPPP techniques are also efficient when the data is in main memory. These results are very encouraging as they reveal that the EPPP techniques are typically better than specialized XML processors. Moreover, they scale to large datasets and can be easily integrated into general purpose repositories.

4.5 Discussion

We introduced new techniques (EPPP) that provide a general and efficient solution to support positional predicates within navigation axes XPath queries. Our techniques can take advantage of the positional predicates and avoid computation that straightforward approaches, which check the positional predicates as a post-processing step, would entail. An important characteristic of the EPPP techniques is that they identify nodes that participate in the result early in the processing phase of the associated axis. Our preliminary experimental evaluation gives strong evidence that the proposed solutions show more robust performance when compared with pure relational approaches and main-memory specialized XPath engines.

5 Conclusion

We studied the problem of supporting the ordered, tree shaped model of XML data. We proposed efficient methods that extend state of the art processing techniques for any of the navigation axes, so as the latters can efficiently support positional predicates as well. Most importantly, we showed that the intermediate state (i.e. buffering of nodes) that is maintained by those algorithms provides the necessary means to identify the nodes that satisfy any positional predicate. To the best of our knowledge, this is the first approach that addresses positional predicates in a complete, scalable, XML model-aware fashion.

As future work we intend to investigate query processing techniques that will target heterogeneous queries (i.e. queries where interleaving of any axes is possible). Moreover, we plan to develop methods to support combined efficient evaluation of structural and value-based predicates.

References

1. Saxon xslt and xquery processor. In *Available at http://saxon.sourceforge.net/.*
2. Shakespeare's plays in xml. In *Available at http://www.oasis-open.org/cover/bosakShakespeare200.html.*
3. Xalan xslt processor. In *Available at http://xml.apache.org /xalan-c/index.html.*
4. S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, , and Y. Wu. Structural joins: A primitive for efficient xml query pattern matching. In *Proc. of IEEE ICDE*, 2002.
5. C. Barton, P. Charles, M. Fontoura, and V. Josifovski. Streaming xpath processing with forward and backward axes. In *Proc. of ICDE*, 2003.
6. A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Key, J. Robie, and J. Simeon. Xml path language (xpath) 2.0. In *W3C Recommendation. Available from http://www.w3.org/TR/xpath20*, 2005.
7. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. Xquery 1.0: An xml query language. In *W3C Working Draft. Available from http://www.w3.org/TR/xquery*, 2005.
8. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal xml pattern matching. In *Proc. of ACM SIGMOD*, 2002.
9. T. Chen, J. Lu, and T. W. Ling. On boosting holism in xml twig pattern matching using structural indexing techniques. In *Proc. of SIGMOD*, 2005 (to appear).
10. M. Fernandez, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. Xquery 1.0 and xpath 2.0 data model. In *W3C Working Draft. Available from http://www.w3.org/TR/xpath-datamodel/*, 2005.
11. T. Grust. Accelerating xpath location steps. In *Proc. of ACM SIGMOD*, 2002.
12. T. Grust, M. van Keulen, and J. Teubnem. Staircase join: Teach a relational dbms to watch its (axis) steps. In *Proc. of VLDB*, 2003.
13. G. V. Subramanyam and P. S. Kumar. Efficient handling of sibling axis in xpath. In *Proc. of COMAD*, 2005.
14. 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 *Proc. of ACM SIGMOD*, 2002.
15. Z. Vagena, N. Koudas, D. Srivastava, and V. J. Tsotras. Answering order-based queries over xml data. In *Proc. of WWW*, 2005 (poster presentation).
16. C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proc. of ACM SIGMOD*, 2001.

A Appendix

A.1 Pseudocode for One-Pair Queries

In what follows we provide the pseudocode for one-pair queries of the form: `a/following-sibling::b`[n]. The algorithm takes as parameters the element lists that correspond to the a and b query nodes, sorted on the $left$ position of each node.

Algorithm 1 FSwPosPredicate(EL a, EL b, PREDICATE n)

```
1: while !eof(a) AND !eof(b) do
2:    $l_a$ = nextLeft(a) {left position of next node in $a$ list}
3:    $l_b$ = nextLeft(b)
4:    if $l_a$ LESSTHAN $l_b$ then
5:       $n_a = next(a)$ {retrieve next node from $a$ list}
6:       clearStack($l_a$) {also discard CSLs corresponding to popped pright values}
7:       if emptyStack() OR (topStack() NOT EQUALS $pr_{n_a}$) then
8:          pushStack($pr_{n_a}$) {pr returns pright value of node}
9:       end if{append $a$ node to the PG at the end of corresponding CSL}
10:      appendToCSL($n_a$, CSL(topStack()))
11:   else
12:      $n_b = next(b)$
13:      clearStack($l_b$)
14:      if !emptyStack() AND (topStack() EQUALS $pr_{n_b}$) then
15:         ++Counter(FIRST-PG(CSL($pr_{n_b}$)))
16:         if NOT-EXISTS-EMPTY-PG-AT-THE-END() then
17:            pg = createEmptyPG()
18:            appendPG(CSL($pr_{n_b}$), pg)
19:            Difference(pg) = 1
20:         else
21:            ++Difference(LAST-PG(CSL($pr_{n_b}$)))
22:         end if
23:         if Counter(FIRST-PG(CSL($pr_{n_b}$))) == n then
24:            OutputMatches() {identify PG referenced by first PG in CSL}
25:            NextPG = NEXT-PG(FIRST-PG(CSL($pr_{n_b}$)))
26:            Counter(NextPG)=Counter(FIRST-PG(CSL($pr_{n_b}$)))-
               Difference(NextPG)
27:            DISCARD-PG(FIRST-PG(CSL($pr_{n_b}$)))
28:         end if
29:      end if
30:   end if
31: end while
```

Relational Index Support for XPath Axes*

Leo Yuen and Chung Keung Poon

Department of Computer Science
City University of Hong Kong
{leo,ckpoon}@cs.cityu.edu.hk

Abstract. In this paper, we designed efficient indexing structure for XML documents so that each basic XPath axis step is supported. The indexing structure is built on top of the B$^+$-tree which is available in practically all commercial relational database systems. For most of the basic axis steps, we are able to derive theoretical worst case execution time bounds. We also perform experimental evaluation to substantiate those bounds.

1 Introduction

The Extensible Markup Language (XML) [6] is becoming the *de facto* standard for information representation and exchange over the Internet. Owing to its hierarchical (recursive) and self-describing syntax, XML is flexible enough to express a large variety of information. XPath is a query language promoted by W3C for addressing parts of an XML document and is a core fragment of several other major XML query languages like XSLT, XPointer and XQuery. Thus, it is important to support XPath queries efficiently.

An XML document can be naturally modelled as a tree in which nodes represent XML elements while edges represent nesting between elements. A sample XML document is shown in Figure 1.

Starting from a given *context node*, an XPath expression specifies a set (or a sequence, in XPath 2.0 [2]) of nodes to be reported as follows. An XPath expression is comprised of a number of *location steps*, each consisting of an *axis*, a node test, and possibly a predicate as well. The axis specifies how the document tree is to be traversed from a context node. The node test then filters the set of reached nodes using a test on the node's tag name or type. The predicate, if present, specifies further conditions for filtering. The result set of a location step will then become the context nodes for the next location step. For example, the expression

```
/descendant::book/child::author
```

consists of two location steps. Starting from the root as the default initial context node, the first location step "`/descendant::book`" specifies all nodes in the tree with `book` as their tag names. Then, the second location step specifies all their children with `author` as their tag names.

* This research was fully supported by a grant from the Research Grants Council of the Hong Kong SAR, China [Project No. 9040906 (RGC Ref. No. CityU 1164/04E)].

```
<?xml version="1.0"?>
<a>
  <b>
    <d><g/><h/></d>
  </b>
  <c>
    <e/>
    <f><i/><j/></f>
  </c>
</a>
```

Fig. 1. An example XML document

Handling large XML documents in secondary storage turns out to be a big challenge. Earlier approaches [7, 9, 15, 16, 21] construct structural summaries of the XML documents to improve the query efficiency. They often require large index storage unless query performance is sacrificed.

Newer approaches apply various labelling schemes on the document tree so that structural information about the nodes (such as ancestor-descendant or parent-child) can be checked by just examining their labels. Then an XPath expression can be evaluated by preparing the lists of nodes satisfying the node tests for each location step in the expression, and pairwisely joining the lists according to the structural requirements on the nodes. Because of the labelling, this type of joins (called *containment* or *structural* joins) does not require traversing the actual document tree, thereby saving a lot of disk I/Os. A substantial body of research [1, 5, 14, 20, 27, 29] has been on the efficient implementation of such joins. All of them, except for [29], are for native XML databases. Also, most of these works mainly emphasized the "vertical" axes, i.e., the ancestor, descendant, parent and child axes. More recent work focus on evaluating certain patterns of XPath expressions including the holistic twig joins [3, 14, 15, 22, 26], sequences of consecutive child axis steps [4] and the next-of-kin pattern [30].

There are also investigations on evaluating the whole XPath expressions. On the main memory model, Gottlob et al. studied the query and data complexities of XPath expression evaluation. They designed generic algorithms [10, 12] and determined the complexity for evaluating the whole path expression [11]. For example, for a fragment of XPath which they called Core XPath, they designed an algorithm for its evaluation using $O(n \cdot |Q|)$ time where n and $|Q|$ represent the size of the XML document and query respectively. In the relational domain, Schmidt et al. [23] and Yoshikawa et al. [28] proposed path-based approaches which, again, favours the vertical axes. DeHaan et al. [8] proposed generic translations of an XPath expression into a single SQL statement. In practice, their method is unlikely to be efficient without modifying the relational engine.

In this paper, we focus on indexing the XML document on a relational database to support each basic axis step efficiently. This is important despite the many research on matching patterns of path expressions or even the whole path expressions holistically. First, by concentrating on a basic step, we often obtain more efficient indexing method. For example, Gottlob et al.'s algorithm

[10] requires $O(n)$ time when the XPath expression contains only one axis step, i.e., $|Q| = 1$. This is slow compared to our index structure. Second, our techniques may well be applicable to these other methods. Relatively few works have been done on this direction.

Kha et al. [17] suggested a recursive version of the UID [19] to support all XPath axes. It requires adding dummy nodes to the original tree to make it a complete b-ary tree before labelling in a breadth-first manner, where b is the maximum fanout of the original tree. Thus, the label size can be very large ($O(n \log n)$ bits per node in the worst case compared with $O(\log n)$ bits in common interval or prefix labelling).

In contrast, Grust et al. [13]'s XPath Accelerator maps each node in an XML document to a point on the so-called pre/post plane; and translates the four global axes, i.e., descendant, ancestor, preceding and following, into 2-d range queries over this 2-d plane. To support the 2-d range queries, they have implemented their index structure on top of an R-tree (a spatial index structure) as well as a B$^+$-tree which does not naturally suit for 2-d range queries. Thus, no worst case performance bounds are given.

Here, we refined their work by mapping the nodes to 1-dimensional intervals instead of points on a 2-d plane. Then the four global axes are translated to either 1-dimensional range queries or interval queries (also called 1-d *point enclosure* in some literature). Consequently, we reduce the number of dimensions in the index structure by one. Note that Grust et al. already observed that the descendant axis can actually be computed as 1-d range queries. However, they did not go further to consider the other three axes in the way we do.

One-dimensional range queries are well-supported by the ubiquitous B$^+$-tree index in relational database with good worst case performance bound. For the interval queries, we employ the RI-tree of Kriegel et al. [18]. Thus, instead of indexing the pre/post-plane by an RI-tree as suggested by Grust et al., we directly map the nodes to intervals which is naturally supported by RI-tree. Interestingly, Jiang et al. [14] also made use of certain variants of interval tree index structure to support interval queries. However, they require modifying the database kernel to incorporate their XR-tree. In contrast, the RI-tree is much simpler and directly implementable on top of a B$^+$-tree.

We observe that the set of intervals derived from the document tree possesses many nice properties. First, the interval boundaries are confined to a limited range, thereby allowing us to derive good theoretical bounds for many of the axes. Second, the set of intervals are nested, i.e., for any two intervals in the set, either they have no intersection or one is completely contained in another. This allows us to use RI-tree, as an alternative to B$^+$ tree, to support range queries, which may be of independent theoretical interest. This in turn permits us to support both XPath axes with and without name tests.

The rest of this paper is organized as follow. In the next section, we explain some basic concept of XPath axes and the main idea of Grust et al. Then we gradually build up to our final index structure by describing the design for handling the descending, preceding and following axes in Section 3, the ancestor axis in Section 4, the local axes in Section 5 and the name test in Section 6. We present our experimental results in Section 7. The paper is then concluded in Section 8.

2 Preliminaries

2.1 The XPath Axes

There are altogether 13 axes in XPath. Besides the *attribute* and *namespace* axes, the other 11 axes deal with traversals of the document tree. In the same spirit as Zhang et al. [30], we divide them into two types. The *local axes* include the *self*, *parent*, *child*, *preceding-sibling* and *following-sibling* axes. The *global axes* include the *descendant*, *descendant-or-self*, *ancestor*, *ancestor-or-self*, *preceding* and *following* axes.

2.2 Grust et al.'s Document Region

Given an XML document tree, Grust et al. label every node u by the pair $(pre(u), post(u))$ where $pre(u)$ and $post(u)$ represent the pre-order and post-order labelling of u in a depth-first traversal of the tree. For example, in Figure 2, each node is labelled with a pair of numbers, the left is its pre-order number and the right is its post-order number. As such, each node is naturally mapped to a point on the 2-dimensional pre/post-plane. Then the descendants of u are precisely those nodes v such that $pre(u) < pre(v)$ and $post(v) < post(u)$. In other words, v lies on the lower-right quadrant of u. Similarly, the ancestors of u are those nodes lying on the top-left quadrant of u while the preceding and following nodes of u are on the lower-left and upper-right quadrants of u respectively. See Figure 2.

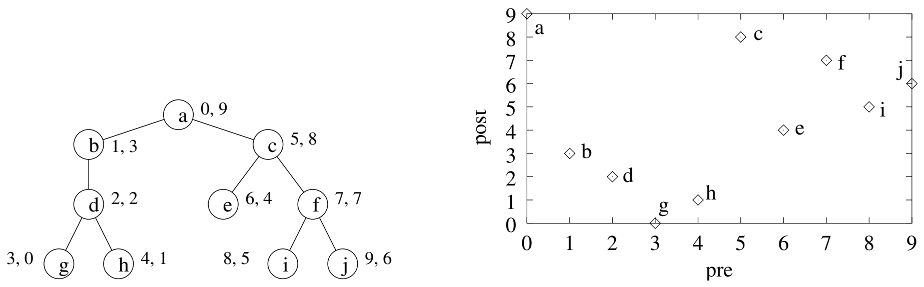


Fig. 2. Left: Pre/post-order of tree nodes. Right: The pre/post-plane

Thus, supporting these four XPath axes amounts to indexing the pre/post-plane to allow for 2-d range searching. To do this, Grust et al. have experimented with ordinary B^+-tree index as well as R-tree (a spatial index structure).

3 The Descendant, Preceding and Following Axes

3.1 Mapping Nodes to Intervals

We will label each node in the document tree T by an interval as follows. It is folklore that the structure of a tree can be represented by a set of nested parentheses. For convenience, we will call it a *bracket expression*. The bracket expression of T can be obtained by performing a depth-first traversal on T. When we start traversing the subtree rooted at u, we output an opening bracket for node u. When we have finished traversing the subtree of u, we output a closing bracket for u. Then the *first-order* number of a node u, denoted $f(u)$, is the position of its opening bracket in the bracket expression of T. Similarly, the *last-order* number of u, denoted $l(u)$, is the position of its closing bracket. See Figure 3 for an example.

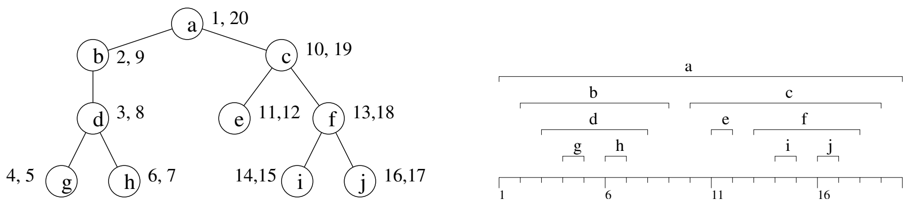


Fig. 3. Left: Interval labelling of tree nodes. Right: The corresponding intervals

Note that for a tree with n-nodes, there are $2n$ brackets. Thus, the range of the first- and last-order number is in $[1..2n]$. Moreover, $f(u) < l(u)$ for every node u. We can view that each node u is labelled with the interval $[f(u), l(u)]$. The following properties are obvious.

Proposition 1. *For any tree labelled with the above interval labelling scheme, node u is an ancestor of v (or equivalently, v is a descendant of u) iff the interval $[f(v), l(v)]$ is contained in $[f(u), l(u)]$, i.e., $f(u) < f(v) < l(v) < l(u)$.*

Proposition 2. *For any tree labelled with the above interval labelling scheme, the set of intervals is nested, i.e., any two intervals are either non-overlapping or one is completely contained in another.*

3.2 Finding Descendants, Preceding and Following Nodes

Based on Proposition 1, the descendants v of a context node c can be characterised by

$$f(c) < f(v) < l(c) \tag{1}$$

which is a 1-d range searching. An alternative characterisation is:

$$f(c) < l(v) < l(c). \tag{2}$$

This will turn out to be useful as one can choose either condition to check as convenient. For the preceding nodes of context node c, they are those nodes v whose opening tag comes before that of c except when they are the ancestors of c ([6]). That means, if we perform a depth-first traversal, we will finish the traversal of the subtree rooted at v before we start traversing the subtree rooted at c. Hence we have the condition

$$l(v) < f(c). \tag{3}$$

Symmetrically, the following nodes of c are those nodes v whose opening tag comes after that of c, except those of c's descendants. That is, we completed traversing the subtree of c before starting the traversal of v. Hence the nodes v can be characterised as:

$$l(c) < f(v). \tag{4}$$

Note that such characterization should be quite obvious. For example, it has been mentioned in [25, 28].

As all three axes involve range queries, we store the first and last-order number of the nodes as attributes in a relational table and build a B$^+$ tree index on it. More specifically, we will have the following two tables:

$\mathsf{XTfirst}(\underline{\mathsf{first}}, \mathsf{last}, \mathsf{data})$
$\mathsf{XTlast}(\underline{\mathsf{last}}, \mathsf{first}, \mathsf{data})$.

The attributes first and last store $f(u)$ and $l(u)$ of a node u respectively. The attribute data contains other information of u. An underlined attribute indicates that it is (part of) the primary key for indexing and the tuples are sorted on the primary key.

To compute the following axis of c, we select those tuples t from XTfirst where $l(c) < t.\mathsf{first}$. The preceding axis of c is given by selecting those tuples t from XTlast where $t.\mathsf{last} < f(c)$. Descendants of c can be found by selecting tuples t from XTlast where $f(c) < t.\mathsf{last} < l(c)$. Since the tuples are sorted according to the primary key, the search will take only $O(\log_B n + k/B)$ time where k is the output size and B is a parameter depending on the block size.

4 The Ancestor Axis

Using Proposition 1, the ancestors of a context node c are those nodes v such that $[f(v), l(v)]$ contains $f(c)$ (or equivalently $l(c)$). To find those intervals enclosing $f(c)$, we make use of an RI-tree index.

4.1 RI-Tree

The original RI-tree ([18]) of height h consists of an implicit complete binary tree U of height h (with the root at height $h-1$ and leaves at height 0). Each node will get an ID (i.e., an integer) and for convenience, we identify a node with its ID. The root is 2^{h-1} and its left and right children are $2^{h-1} - 2^{h-2} = 1 \cdot 2^{h-2}$ and $2^{h-1} + 2^{h-2} = 3 \cdot 2^{h-2}$ respectively. In general, for a node $x = x'2^{\ell}$ where 2^{ℓ} is the largest power of two that divides x, its left and right sons are $x'2^{\ell} - 2^{\ell-1} = (2x' - 1)2^{\ell-1}$ and $x'2^{\ell} + 2^{\ell-1} = (2x' + 1)2^{\ell-1}$, respectively.

Construction. Suppose we are to store a set of n intervals $[l_i, r_i]$, $1 \leq i \leq n$, whose boundaries are taken from a bounded universe $\{1, \ldots, 2^h - 1\}$, i.e., each boundary can be represented as an h-bit binary number. The main idea is to associate each interval $[l_i, r_i]$ with the highest node x such that $l_i \leq x \leq r_i$.

To compute such an x, we can search from the root. Alternatively, one can observe that x is nothing but the lowest common ancestor of l_i and r_i in U. Thus x can be computed easily by considering the binary representation of l_i and r_i and finding the most significant bit on which they differ. Let's say this is the $(l-1)$-st bit counting from bit 0. Then $x = \lfloor l_i/2^l \rfloor \cdot 2^l$ which is the same as $\lfloor r_i/2^l \rfloor \cdot 2^l$.

To facilitate the searching of intervals associated with a node x, the intervals are stored twice, once in sorted order of left boundaries and once in right boundaries. Thus, there will be two database tables, $\mathsf{L}(\underline{\mathsf{node}, \mathsf{left}}, \mathsf{right})$ and $\mathsf{R}(\underline{\mathsf{node}, \mathsf{right}}, \mathsf{left})$ where the node attribute stores the ID of a node in the binary tree U and the left, right attributes store the left and right boundaries of an interval assoicated with the node specified in the node attribute. Clearly, the two tables L and R require $O(n)$ space in total. They can be constructed in $O(n \log_B n)$ time by scanning through the n intervals once. For each interval $[l, r]$, we compute the associated node x as described above and then insert the record (x, l, r) in L and (x, r, l) in R. Note that U is never explicitly stored. In particular, if no intervals are associated with a node x in U, the tables L and R will not have an entry with the node attribute storing value x.

Querying. To search for the intervals enclosing a query point $q \in \{1, \ldots, 2^h - 1\}$, we start from the root 2^{h-1} and search down the path to q. Suppose we are at a node x along this path. If $q \leq x$, then we report those interval $[l, r]$ with $l \leq q$ since its right endpoint $r \geq x \geq q$. (Those interval $[l, r]$ such that $l > q$ need not be reported because q is outside the interval.) This is done by searching the table L. Similarly, if $q \geq x$, then we need only report those interval $[l, r]$ with $r \geq q$. This is done by searching table R. If $q < x$ or $q > x$, we search down x's left or right son respectively. Otherwise, we can stop.

In general, a query requires accessing $\leq h$ nodes in the RI-tree. For $0 \leq i \leq h - 1$, if the node at height i contains k_i intervals enclosing the query point, retrieving these intervals assuming a B$^+$-tree index requires $O(\log_B n + k_i/B)$ time. Summing up all the h levels, the complexity for a query is $O(h \log_B n + k/B)$ where $k = \sum_i k_i$ is the total number of intervals to be reported.

4.2 Finding Ancestors

By Proposition 2, our intervals are nested. Thus, for those intervals associated a node in U, if they are already sorted in order of the left endpoints, they must also be sorted in (reverse) order of their right endpoints. So, we will only keep one table, say, L.

To find the ancestors of a context node c, we search the RI-tree with the query point $f(c)$. We start from the root 2^{h-1}. If $f(c) \leq 2^{h-1}$, we examine the table L and report all records t such that $t.\mathsf{node} = 2^{h-1}$ and $t.\mathsf{left} < f(c)$. If $f(c) \geq 2^{h-1}$, we examine the table L and report all records t such that $t.\mathsf{node} = 2^{h-1}$ and $t.\mathsf{right} > f(c)$. After that, we move down the tree one level to examine either the node $1 \cdot 2^{h-2}$ or $3 \cdot 2^{h-2}$ depending on $f(c) < 2^{h-1}$ or $f(c) > 2^{h-1}$. The process is then repeated until we reached a node x where $f(c) = x$. To bound the query complexity, note that our interval boundaries lie within the range $[1, ..., 2n]$. Thus, we have $h = O(\log n)$ and the query time becomes $O(\log n \log_B n + k/B)$.

We remark that the RI-tree can also be used to find the parent of a node though it is not as efficient as the method we mention in the next section. Observe that the intervals enclosing the query point $f(c)$ are nested and the innermost one corresponds to the parent of node c. We can make use of the RI-tree to find the innermost interval enclosing the query point. This is done by computing the interval with the largest left endpoint among those enclosing $f(c)$. Thus the query time is the same as that of finding ancestors.

5 The Local Axes

In this section, we modify our previous table $\mathsf{XTfirst}$ to support the child, parent and preceding/following-sibling axes.

To find the children of a context node c, enumerating all descendants of c and picking the maximal ones, i.e., the ones not contained in any other descendant of c, would be slow. Instead, we change one of the tables, $\mathsf{XTfirst}$, to $\mathsf{XTfirst}(\underline{\mathsf{parent}, \mathsf{first}}, \mathsf{last}, \mathsf{data})$ with tuples sorted in the order of the key $(\mathsf{parent}, \mathsf{first})$. With this change, the children of context node c can be obtained by selecting the tuples t where $t.\mathsf{parent} = f(c)$. Querying on the child axis will take $O(\log_B n + k/B)$ time where k is the number of children in the result set.

In addition, finding parent is also easy: Suppose the parent attribute of (the record in $\mathsf{XTfirst}$ for) the context node c contains the value $f(p)$. Then we search for the record t where $t.\mathsf{first} = f(p)$. This takes $O(\log_B n)$ time assuming a B$^+$-tree index is built on attiribute first. Finding the preceding or following siblings of c can be done by finding the parent p of c and then the children v of p with $f(v) < f(c)$ (for preceding-siblings) or $f(v) > f(c)$ (for following-siblings). Thus, these axes take $O(\log_B n + k/B)$ time as well, where k is, again, the output size.

The drawback of introducing the parent attribute to table $\mathsf{XTfirst}$ is that the complexity for the following axis will be larger as the records are now clustered around the parent attribute instead of the first attribute. However, as we have a B$^+$-tree index on the first attribute, we can still bound the complexity by the depth d of the document tree as follows. Recall that the tuples are arranged in the order of their parents' first-order number. Consider an arbitrary context node c and let its ancestors be $u_1, u_2, \ldots, u_{d'}$, $d' \leq d$, as we walk up the path from c to the root of the document. For convenience, define $u_0 = c$. Then for $1 \leq i \leq d'$,

u_{i-1} is a child of u_i. Let $w_{i1}, w_{i2}, \ldots$ be the children of u_i that follows u_{i-1}; and let $W_{i1}, W_{i2}, \ldots$ be the subtrees rooted at these nodes respectively. The following axis of c is then the union of those nodes in $W_{i1}, W_{i2}, \ldots$, for $1 \leq i \leq d'$. Fix an i and observe that the first-order numbers of the nodes in $W_{i1}, W_{i2}, \ldots$ are larger than that of u_i but smaller than those of the nodes in $W_{i+1,1}, W_{i+1,2}, \ldots$. The first-order number of the parent of a node in $W_{i1}, W_{i2}, \ldots$ is either the first-order number of u_i or that of some node in $W_{i1}, W_{i2}, \ldots$. Hence, we can conclude that the following nodes of c are partitioned into at most $2d'$ segments in the physical storage. Hence retrieving all the segments require $O(d \log_B n + k/B)$ time. In practice, d is often a small value.

6 Handling Name Tests

We are now ready to present our final design of the index structure which supports name tests. This is important because XPath expressions often contain tests on the tag names. To support such expressions efficiently, we introduce a tag attribute in the database table. Thus the modified tables are: XTfirst($\underline{\text{tag, parent, first}}$, last, data) and XTlast($\underline{\text{tag, last}}$, first, data). The B$^+$-tree index for attribute first in XTfirst is now extended to (tag, first). Then with name test, the descendant, preceding, child, preceding- and following-sibling axes are now supported in $O(\log_B n + k/B)$ time. The following axis with name test is supported in $O(d \log_B n + k/B)$ time. The ancestor axis, with name test, takes $O(\log n \log_B n + d/B)$ (recall d is the document tree height) while taking $O(\log n \log_B n + k/B)$ time without name test. The parent axis remains to be $O(\log_B n)$ with or without name test.

For the descendant, preceding and following axes without name test, we make use of the RI-tree, exploiting the nesting properties of the intervals. To compute the preceding axis of context node c, we need to find all document nodes v such that $1 \leq l(v) < f(c)$. Consider an RI-tree node x. If $x \geq f(c)$, then it follows from the construction of RI-tree that any interval associated with x must have right endpoint at least $f(c)$ and hence not in the answer set. Hence we need only consider those node $x < f(c)$. For each such node, we examine its associated intervals and report only those with right endpoint less than $f(c)$. It can be shown that there are at most $\log n$ RI-tree nodes which contain intervals not in the answer set. (The main idea is that each level of the RI-tree U can have at most one node containing intervals not in the answer set. Formal proof will be given in the full paper.) The query complexity is then $O(\log n \log_B n + k/B)$ where k is the output size. The following axis is handled similarly.

To compute the descendant axis of c, we can do even better by using the fact that the query range $[f(c), l(c)]$ is itself one of the nested intervals. As before, we need not consider those RI-tree node x outside $(f(c), l(c))$. Let $[f(c), l(c)]$ be associated with node z in the RI-tree. We claim that for any RI-tree node x in $(f(c), l(c))$ except for z, all its associated intervals $[l, r]$ must satisfy $f(c) < l < r < l(c)$. Otherwise, if $l < f(c) < l(c) < r$, then $[l, r]$ would have been associated with a node y at least as high as z in the RI-tree. So, y is higher than

x, a contradiction. Hence the claim follows. Thus, the query complexity for the descendant axis is $O(\log_B n + k/B)$.

Now, the only axes for which our structure does not guarantee a worst case time bound are the child, preceding- and following-siblings without name test.

7 Experimental Evaluation

In this section, we present our experimental results on the query performance of each axis. Our XML documents are generated by XMLgen of the XMark benchmark project [24]. The tool produces XML documents of a specific structure but allows us to specify the document size.

We implemented our index structure as well as the B$^+$-tree and R-tree version of Grust et al.'s design for comparison. All programs are written in Java. All the experiments were performed on an Intel Pentium 4 2.6 GHz PC with 1GB RAM running Windows XP. Our purpose here is not to compare the different designs on absolute terms but to elicit their behaviours as the document sizes and/or output sizes grow.

We used the PostgreSQL database since it is equipped with GiST that supports both of the indexes. For the B$^+$-tree version, we implemented its stretched version (Section 4.2 [13]). For the R-tree version, we use the optimization described in Section 4.1 of [13] that minimizes the query window size using the subtree size as additional information. However, we only use the R-tree for indexing the two dimensional *pre/post* plane rather than making the whole tuple as a 5-dimensional descriptor. It is well-known that the performance of an R-tree deteriorates as the number of dimensions increases. The indexes and clustering they mentioned are also applied.

7.1 Range Searching on B$^+$-Tree

Our descendant, preceding, child, preceding- and following-sibling axes with name test requires range searching on B$^+$-tree, taking $O(\log_B n + k/B)$ time. The following axis requires $O(d \log_B n + k/B)$ time. To verify the formula, we design the experiments so that in each set, either only the input size or the output size varies.

Input Size. To show the effect on varying the input size n, we choose a path that returns only one node in whatever input size.

Figure 4 shows that the times for Grust et al's design grows linearly instead of logarithmically in the input size n. The reason is that they need to filter out nodes that do not match the node test (false hits). Our design (near the horizontal axis in the graph) takes negligible time even for different input sizes. This indicates that the $\log_B n$-term is very small (at least for input size ranging from 11 to 113 MB).

Output Size. To show the effect on varying output size k, we use a large enough input file (56MB) and provide different name tests that yield different output sizes.

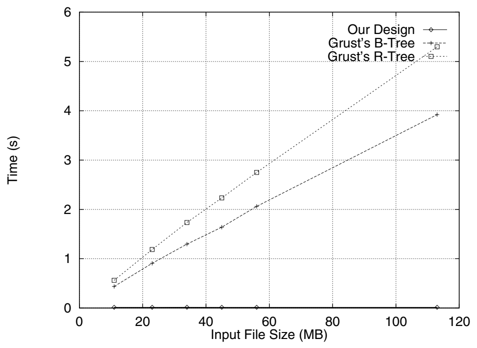


Fig. 4. Effect of Input Size on Descendant Axis with Name Test (Query: /descendant::open_auctions)

Figure 5 shows the result for descendant, child, preceding and following axes. In all cases, our design gives linear growth with respect to the output size k while Grust's design may sometimes give irregular growth, especially for their R-tree version. This is possibly due to the overhead of false hits, and the fact that R-tree's performance has no worst case guarantee.

7.2 Interval Queries of RI-Tree

Our ancestor axis (without name test) is done by an interval query on the RI-tree using $O(\log_2 n \log_B n + k/B)$ time. Due to the given DTD of the XMLgen, the generated XMLs have a depth of $d = 11$ which is too small for showing the effect of output size. Therefore we focus on the effect of the input size in this subsection. Here, we first use a descendant axis to select a single element node called 'africa' that has $k = 2$ ancestors for any input size. We can then vary the input size n. In measuring the time, we only count the second step (i.e., the ancestor axis).

The result in Figure 6 (left) shows that Grust et al.'s B$^+$-tree takes linear time in n while their R-tree and our design have negligible time. To magnify the processing time to observable magnitude, we perform the previous experiment again but for each point we repeat the operation 1,000 times. Then we plot the graph with x-axis in log-scale. Figure 6 (right) verified our claims that the time grows in $\log n \log_B n$ time where $\log_B n$ can be treated as constant for our range of input sizes.

7.3 Range Query on RI-Tree

Our descendant axis (without name test) is handled by range query on the RI-Tree and takes $O(\log_B n + k/B)$ time. Our preceding and following axes without name test are handled by range query on RI-Tree as well and take $O(\log n \log_B n + k/B)$ time.

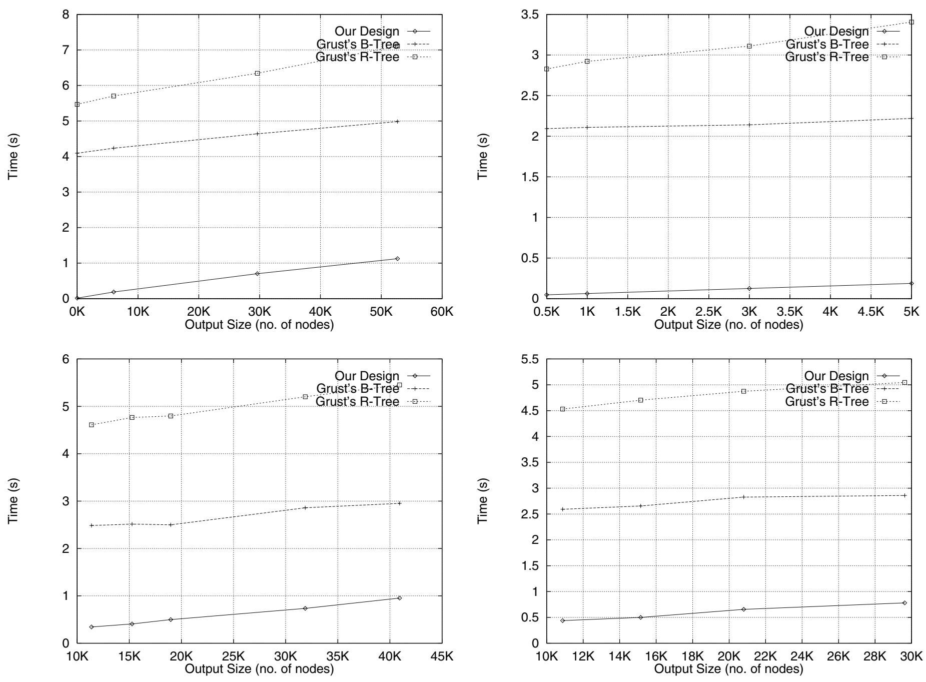


Fig. 5. Effect of Output Size. Top-left: descendant axis with name test. Top-right: child axis with name test. Bottom-left: preceding axis with name test. Bottom-right: following axis with name test

Output Size. For conciseness, we only show the graph for the descendant and following axes. The queries used for the descendant axis are:

```
/descendant::?/descendant::*
```

where ? is certain tag name. Those for the following axis are:

```
/descendant::?/following::*.
```

The graph for the preceding axis is similar to that of the following axes.

In these cases, Grust et al.'s B$^+$-tree should have the best performance since it favors range query. Figure 7 shows that our RI-tree (originally designed for interval queries) has comparable performance with B$^+$-tree on range queries. In comparison, the R-tree also grows linearly but at a somewhat larger slope.

Input Size. Without name test, it is difficult to control the output size when we vary the input size. So, we skip its evaluation in this paper.

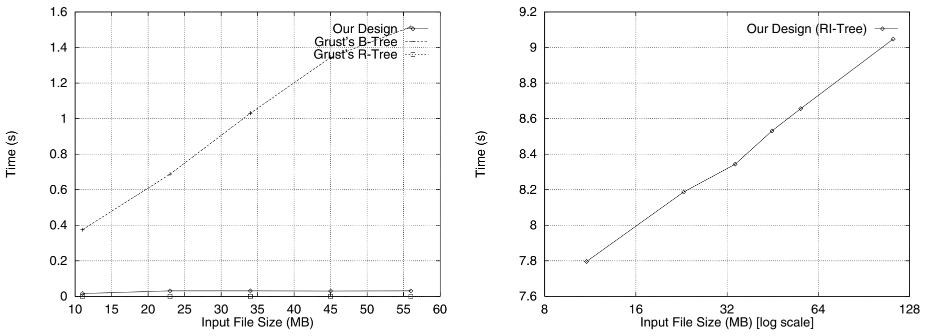


Fig. 6. Effect of Input Size on Ancestor Axis without Name Test. Left: linear scale. Right: x-axis in log-scale

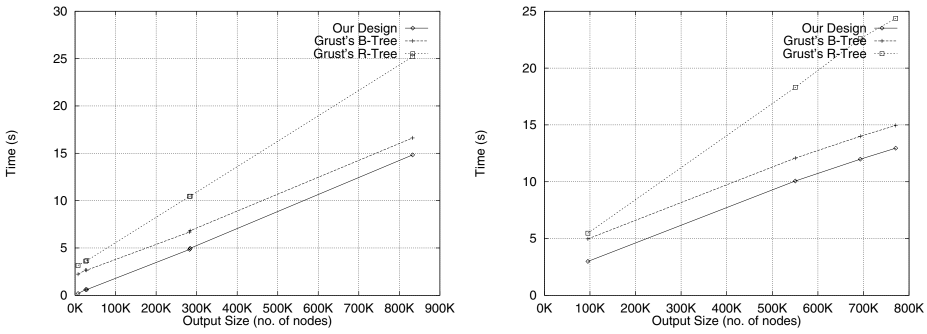


Fig. 7. Effect of Output Size on Descendant Axis (left) and Following Axis (right) both without Name Test

8 Conclusion

We studied the problem of indexing an XML document with traditional relational database to support all the XPath axes. We consider the physical ordering of the data and derived worst case upper bounds on the execution time for most of the XPath axes. The only axes for which our structure does not provide a worst case guarantee on the performance bound are the child, preceding- and following-siblings when name tests are not present.

We verifed experimentally our formulas by performing a series of experiments. The results show that in our design the input file size has very small effect compared to the output size. Thus, it is suitable for indexing large XML documents, even when they have varying structures. (Our design does not require a DTD of the documents.)

Our design requires only B$^+$-tree index which is available in practically any relational database. Without the need of R-tree index, our index structure will have more predictable performance compared with the XPath Accelerator of Grust et al. This may be a desirable feature when it is used as a basic building block for other methods of XPath evaluation.

Our observation that RI-Tree can handle range queries on nested intervals is also interesting. Perhaps more interesting properties of the RI-tree are waiting ahead for our discovery.

References

1. S. Al-Khalifa, H. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: a primitive for efficient XML query pattern matching. In *18th International Conference on Data Engineering*, pages 141–152, 2002.
2. A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, and J. Simeon. XML Path Language (XPath) 2.0. Technical Report W3C Working Draft, Version 2.0, World Wide Web Consortium, Aug. 2002.
3. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the 2002ACM SIGMOD Conference on the Management of Data*, pages 310–321, 2002.
4. Y. Chen, S. B. Davidson, and Y. Zheng. BLAS: an efficient xpath processing system. In *Proceedings of the 2004ACM SIGMOD Conference on the Management of Data*, pages 47–58, 2004.
5. S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 263–274, 2002.
6. W. W. W. Consortium. Extensible markup language (XML) 1.0 (second edition) – W3C recommendation. Available at http://www.w3.org/TR/2000/WD-xml-2e-20000814, 2000.
7. B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 341–350, 2001.
8. D. deHaan, D. Toman, M. P. Consens, and M. T. Ozsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proceedings of the 2003ACM SIGMOD Conference on the Management of Data*, pages 623–634, 2003.
9. R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23th International Conference on Very Large Data Bases*, pages 436–445, 1997.
10. G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 95–106, 2002.
11. G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Database Systems*, pages 179–190, 2003.
12. G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: improving time and space efficiency. In *19th International Conference on Data Engineering*, pages 379–390, 2003.
13. T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Transactions on Database Systems*, 29(1):91–131, 2004.
14. H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: indexing XML data for efficient structural joins. In *19th International Conference on Data Engineering*, pages 253–263, 2003.
15. R. Kaushik, P. Bohannon, J. Naughton, and H. Korth. Covering indexes for branching path queries. In *Proceedings of the 2002ACM SIGMOD Conference on the Management of Data*, pages 133–144, 2002.

16. R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *18th International Conference on Data Engineering*, pages 129–140, 2002.
17. D. D. Kha, M. Yoshikawa, and S. Uemura. A structural numbering scheme for XML data. In *EDBT Workshops*, pages 91–108, 2002.
18. H.-P. Kriegel, M. Potke, and T. Seidl. Managing intervals efficiently in object-relational databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 407–418, 2000.
19. Y. K. Lee, S. Yoo, K. Yoon, and P. B. Berra. Index structures for structured documents. In *Digital Libraries*, pages 91–99, 1996.
20. Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 361–370, 2001.
21. T. Milo and D. Suciu. Index structures for path expressions. In *7th International Conference on Database Theory*, pages 277–295, 1999.
22. P. Rao and B. Moon. PRIX: indexing and query XML using Prüfer sequences. In *20th International Conference on Data Engineering*, pages 288–300, 2004.
23. A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *Proceedings of the 3rd International Workshop on the Web and Databases*, pages 137–150, 2000.
24. A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: a benchmark for XML data management. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 974–985, 2002.
25. 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 *Proceedings of the 2002ACM SIGMOD Conference on the Management of Data*, pages 204–215, 2002.
26. H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A dynamic index method for queryinh XML data by tree structures. In *Proceedings of the 2003ACM SIGMOD Conference on the Management of Data*, pages 110–121, 2003.
27. W. Wang, H. Jiang, H. Lu, and J. X. Yu. PBiTree coding and efficient processing of containment joins. In *19th International Conference on Data Engineering*, pages 391–, 2003.
28. 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):110–141, 2001.
29. C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of the 2001ACM SIGMOD Conference on the Management of Data*, pages 425–436, 2001.
30. N. Zhang, V. Kacholia, and M. T. Ozsu. A succinct physical storage scheme for efficient evaluation of path queries in XML. In *20th International Conference on Data Engineering*, pages 56–65, 2004.

Supporting XPath Axes with Relational Databases Using a Proxy Index*

Olli Luoma

Department of Information Technology and Turku Centre for Computer Science
University of Turku, Finland
olli.luoma@it.utu.fi

Abstract. In recent years, a plethora of work has been done to develop methods for managing XML documents using relational databases. In order to support XPath or any other XML query language, the relational schema must allow fast retrieval of the parents, children, ancestors, or descendants of a given set of nodes. Most of the previous work has aimed at this goal using pre-/postorder encoding. Relying on this method, however, may lead to scalability problems, since the structural relationships have to be checked using nonequijoins, i.e., joins using $<$ or $>$ as their join condition. Thus, we discuss alternative methods, such as ancestor/descendant and ancestor/leaf indexes, and present a novel method, namely a so called proxy index. Our method allows us to replace nonequijoins with equijoins, i.e., joins using $=$ as their join condition. The results of our comprehensive performance experiments demonstrate the effectiveness of the proxy index.

1 Introduction

Because of its simplicity and flexibility, XML [1] has widely been adopted as the *lingua franca* of the Internet. Currently, XML is used not only as a platform-independent means to transfer data in computer networks, but also as a format to store large amounts of heterogeneous data in many modern application areas, such as bioinformatics [2]. It is thus widely agreed that without efficient means for managing XML documents, the potential of XML cannot be realized to its full extent, and the database community has been actively developing methods for storing and querying large amounts of XML data using XML query languages, most often XPath [3] and XQuery [4].

According to XPath [3], every well-formed XML document can be represented as an *XML tree*, a partially ordered tree with seven different node types. In addition to this tree representation, XPath also lists 12 *axes*[1], i.e., operators for tree traversal, such as `parent`, `child`, `ancestor`, `descendant`, `preceding`, and `following`. In order to provide adequate XPath support, all axes have to be implemented efficiently. The majority of the previous proposals, however,

* Supported by the Academy of Finland.

[1] In XPath 2.0, the `namespace` axis of XPath 1.0 was deprecated.

have considered only the `descendant` and `child` axes, and thus they lack the flexibility and generality intended by the original XPath recommendation.

In many previous proposals [5] [6] [7], the structural relationships between the nodes of an XML tree have been modeled using *pre-/postorder encoding*, i.e., by maintaining the pre- and postorder numbers of the nodes, or by using some similar method. This method forces us to check the nested relationships using expensive nonequijoins, which can easily lead to scalability problems [8] [9]. Thus, methods aiming at replacing nonequijoins with equijoins have been proposed; an *ancestor/descendant index* [10] maintains all ancestor/descendant pairs, an *ancestor/leaf index* [9] maintains the ancestor information only for the leaf nodes.

Maintaining the ancestor/descendant or ancestor/leaf information explicitly, however, consumes much more storage than pre-/postorder encoding. According to our previous experiments [9], a database based on ancestor/leaf approach consumes roughly twice and a database based on ancestor/descendant approach roughly four times the storage consumed by a database based on pre-/postorder encoding, which makes these methods somewhat impractical.

The main contributions of this paper can be listed as follows:

1. We present a novel method for modeling the structural relationships in XML trees, namely a *proxy index*. Our method supports all 12 XPath axes, provides good query performance, and encodes the structural information much more compactly than the ancestor/descendant and ancestor/leaf indexes.
2. We show that not only is it possible to support all axes of XPath using ancestor/descendant, ancestor/leaf, and proxy indexes, it is actually practical when large XML documents have to be queried efficiently. To support this claim, we present the results of our comprehensive performance experiments.

This paper proceeds as follows. In section 2, we review the related work and in section 3, we present the basics of the XPath query language; section 4 provides the reader with a more detailed discussion on the previous proposals. We present the proxy index in section 5 and the results of our performance evaluation in section 6. Section 7 concludes this article and discusses our future work.

2 Related Work

From a technical viewpoint, an XML management system (XMLMS) can be built in several ways. One option is to build a *native XML database*, i.e., to build an XML management system from scratch. This option allows one to design all features, such as query optimization and storage system, specifically for XML data. In the XML research literature, there are several examples of this approach, such as Lore [11] and NATIX [12].

Another approach followed, for instance, in [13] is to build an *XMLMS on top of an object-oriented database*. This approach allows us to take advantage of the rich data modeling capabilities of object-oriented databases. In this context, however, traversing large XML trees means traversing large object hierarchies, which can be a rigorous task, and thus the scalability of these systems might be somewhat questionable [14]. Relational databases, on the contrary, provide a technically sound platform for data management, and thus building an *XMLMS on top of a relational database* is a viable option. This method allows the relational data and the XML data to coexist the same database, which is advantageous, since it is unlikely that XML databases can completely replace the traditional database technology in the near future. Unfortunately, there is some undeniable mismatch between the hierarchical XML and flat relational data models, and thus, after half a decade of XMLMS research, it still is not clear which one or ones of these approaches will prevail.

The previous proposals to manage XML data using relational databases can roughly be divided into two categories [5]. In the *structure-mapping approach* (*schema-aware approach*), the relational models are designed in accordance with the DTDs or the logical structure of the documents. The standard approach is to create one relation for each element type [14], but more sophisticated methods have also been proposed [15]. All of these methods, however, are at their best in applications where a large number of documents corresponding to a limited number of DTDs have to be stored. If this is not the case, we are at risk to end up with a very large number of relations, which complicates the XPath-to-SQL query translation [7] [16]. The other approach is the *model-mapping approach* (*schema-oblivious approach*) in which the database schemas represent the generic constructs of the XPath data model, such as element, attribute, and text nodes. Unlike in the structure-mapping approach, the database schema is fixed, and thus we can store any well-formed XML document without changing the schema. Having a fixed schema also simplifies the XPath-to-SQL query translation.

Thus far, the model-mapping approach has been followed in many proposals, such as XRel [5] (based on a variant of pre-/postorder encoding), XParent [10] (based on an ancestor/descendant index), and SUCXENT [17] (based on a variant of ancestor/leaf index). All of these methods, however, have only considered the `descendant` and `child` axes of XPath, and thus they provide a very limited XPath support. The XPath accelerator proposed by Grust [7] took a leap forward by providing support for all XPath axes and relative addressing. However, since XPath accelerator is based on pre-/postorder encoding, the nested relationships have to be checked using nonequijoins, and thus the scalability of the method may be somewhat in doubt [9].

3 XPath Axes

As mentioned earlier, the tree traversals in XPath are based on 12 axes which are presented in Table 1. These axes are used in *location steps* which, starting from a *context node*, result in a set of nodes in the relation defined by the axis with the context node. A location step of the form `axis::nodetest[predicate]` also includes a *node test* which can be used to restrict the name or the type of the selected nodes. An additional predicate can be used to filter the resulting node set further. For example, the location step n`/descendant::record[child::*]` selects all descendants of the context node n which are named `record` and have one or more child nodes.

Table 1. The XPath axes and their semantics

Axis	Semantics of n/Axis
`child`	Children of n.
`parent`	Parent of n.
`ancestor`	Transitive closure of `parent`.
`descendant`	Transitive closure of `child`.
`ancestor-or-self`	Like ancestor, plus n.
`descendant-or-self`	Like descendant, plus n.
`preceding`	Nodes preceding n, no ancestors.
`following`	Nodes following n, no descendants.
`preceding-sibling`	Nodes preceding n and with the same parent as n.
`following-sibling`	Nodes following n and with the same parent as n.
`attribute`	Attribute nodes of n.
`self`	Node n.

XPath also provides a means for checking the string values of the nodes. The XPath query `/descendant-or-self::record="John Scofield Groove Elation Blue Note"`, for instance, selects all element nodes with label "record" for which the value of all text node descendants concatenated in document order matches "John Scofield Groove Elation Blue Note". In the scope of this paper, however, we will not concentrate on querying the string values; our focus is on the XPath axes.

4 Previous Proposals

In this section, we discuss the previous proposals for modeling nested relationships in XML documents using relational databases in detail. For brevity, we have omitted the document identifiers from the relations, but as in [9], these could easily be added to support storage and retrieval of multiple documents in a single database.

4.1 Pre-/Postorder Encoding

The pre-/postorder encoding [18] makes use of the following very simple property of pre- and postorder numbers. For any two nodes n_1 and n_2, n_1 is an ancestor of n_2, iff the preorder number of n_1 is smaller than the preorder number of n_2 and the postorder number of n_1 is greater than the postorder number of n_2. As in [7], we store all the nodes in a single relation `Node`:

```
Node(Pre, Post, Par, Type, Name, Value)
```

In this schema, the database attributes `Pre`, `Post`, and `Par` correspond to the preorder and postorder numbers of the node and the preorder number of the parent of the node, respectively. The database attribute `Type` corresponds to the type of the node and the database attribute `Name` to the name of the node.

`Value` corresponds to the string value of the node. Like many earlier proposals [5] [10] [17], we do not store the string values of element nodes.

As noticed by Grust [7], the pre- and postorder numbers, combined with the parent information, are sufficient to perform all XPath axes by using the query conditions presented in Table 2; node tests can be supported by using the additional query conditions presented in Table 3.

Table 2. Pre-/postorder encoding query conditions for supporting the axes

Axis	Query conditions
`parent`	n_{i+1}`.Pre=`n_i`.Par`
`child`	n_{i+1}`.Par=`n_i`.Pre`
`ancestor`	n_{i+1}`.Pre<`n_i`.Pre AND `n_{i+1}`.Post>`n_i`.Post`
`descendant`	n_{i+1}`.Pre>`n_i`.Pre AND `n_{i+1}`.Post<`n_i`.Post`
`ancestor-or-self`	n_{i+1}`.Pre<=`n_i`.Pre AND `n_{i+1}`.Post>=`n_i`.Post`
`descendant-or-self`	n_{i+1}`.Pre>=`n_i`.Pre AND `n_{i+1}`.Post<=`n_i`.Post`
`preceding`	n_{i+1}`.Pre<`n_i`.Pre AND `n_{i+1}`.Post<`n_i`.Post`
`following`	n_{i+1}`.Pre>`n_i`.Pre AND `n_{i+1}`.Post>`n_i`.Post`
`preceding-sibling`	`preceding AND `n_{i+1}`.Par=`n_i`.Par`
`following-sibling`	`following AND `n_{i+1}`.Par=`n_i`.Par`
`attribute`	n_{i+1}`.Par=`n_i`.Pre AND `n_{i+1}`.Type="attr"`
`self`	n_{i+1}`.Pre=`n_i`.Pre`

Table 3. Additional query conditions for supporting the node tests

Axis	Query conditions
`node()`	
`text()`	n_{i+1}`.Type="text"`
`comment()`	n_{i+1}`.Type="comm"`
`processing-instruction()`	n_{i+1}`.Type="proc"`
`name`	n_{i+1}`.Type="elem" AND `n_{i+1}`.Name="name"`
`*`	n_{i+1}`.Type="elem"`

For brevity, we will not discuss the XPath-to-SQL query translation in full detail. For our purposes, it is sufficient to say that the SQL queries can be generated simply by walking through all the location steps in an XPath query and by using the query conditions to perform each step. For example, the XPath query n_1`/descendant::record` can be translated into the following SQL query:

```
SELECT DISTINCT n2.*
FROM Node n1, Node n2
WHERE n2.Pre>n1.Pre AND n2.Post<n1.Post
AND n2.Type="elem" AND n2.Name="record"
ORDER BY n2.Pre;
```

The last row of the query is added to ensure that the nodes are returned in document order, as required by the XPath recommendation. Optional predicates in queries can be evaluated by changing the tuple variable in the SELECT part of the query. The XPath query n_1`/descendant::*/following-sibling::record [child::title]`, for example translates into the following SQL query:

```
SELECT DISTINCT n3.*
FROM Node n1, Node n2, Node n3, Node n4
-- First step /descendant::*
AND n2.Pre>n1.Pre AND n2.Post<n1.Post
AND n2.Type="elem"
-- Second step /following-sibling::record
AND n3.Pre>n2.Pre AND n3.Post>n2.Post AND n3.Par=n2.Par
AND n3.Type="elem" AND n3.Name="record"
-- Third step /child::title
AND n4.Par=n3.Pre
AND n4.Type="elem" AND n4.Name="title"
ORDER BY n3.Pre;
```

In [7], Grust also described how the evaluation of `descendant` and `descendant-or-self` axes can remarkably be accelerated by tightening the query conditions for the pre- and postorder numbers of the descendant nodes. Using this idea, the XPath query n_1`/descendant::record` can be translated into the following "accelerated" SQL query in which *height* denotes the height of the tree:

```
SELECT DISTINCT n2.*
FROM Node n1, Node n2
WHERE n2.Pre>n1.Pre AND n2.Pre<=n1.Post + height
AND n2.Post<n1.Post AND n2.Post>=n1.Pre - height
AND n2.Type="elem" AND n2.Name="record"
ORDER BY n2.Pre;
```

However, if n_1 is the root of the tree the query is not accelerated at all, since every node in the tree has a preorder (postorder) number smaller (larger) than the postorder (preorder) number of n_1. In general, the closer to the root the context node resides, the less will the query be accelerated.

The XML research literature knows several methods similar to pre-/postorder encoding, such as the *nested sets model* [2] and the *order/size scheme* [6]. XRel [5] makes use of *region coordinates* which tell the indexes of the first and last character of the piece of text corresponding to an element, attribute or text node. Nevertheless, all of these methods impose the nested relationships to be checked using expensive nonequijoins.

4.2 Ancestor/Descendant Index

An extreme approach for modeling the nested relationships is to build an ancestor/descendant index, i.e., to explicitly maintain all ancestor/descendant pairs.

In order to evaluate `ancestor-or-self` and `descendant-or-self` axes efficiently, all nodes actually have to be in relation `AncDesc` with themselves, which raises the storage requirements even higher. Nevertheless, this approach can provide good query performance, and thus it has been followed in [10], for instance. To maintain the ancestor/descendant information, we use an `AncDesc` table:

```
Node(Pre, Par, Type, Name, Value)
AncDesc(Anc, Desc)
```

It is easy to find the query conditions for ancestor/descendant index; these are presented in Table 4. Evaluating the `ancestor` and `descendant` axes involves an equijoin on preorder numbers to ensure that the context node is not included in the result of the location step. When evaluating `following` and `preceding` axes, we use subqueries to filter the descendants and ancestors from the result.

Table 4. Ancestor/descendant index query conditions for performing the axes using `AncDesc` tuple variable a_i

Axis	Query conditions
parent	n_{i+1}.Pre=n_i.Par
child	n_{i+1}.Par=n_i.Pre
ancestor	ancestor-or-self AND NOT n_{i+1}.Pre=n_i.Pre
descendant	descendant-or-self AND NOT n_{i+1}.Pre=n_i.Pre
ancestor-or-self	n_{i+1}.Pre=a_i.Anc AND a_i.Desc=n_i.Pre
descendant-or-self	n_{i+1}.Pre=a_i.Desc AND a_i.Anc=n_i.Pre
preceding	n_{i+1}.Pre<n_i.Pre AND n_{i+1} NOT IN ancestor
following	n_{i+1}.Pre>n_i.Pre AND n_{i+1} NOT IN descendant
preceding-sibling	preceding AND n_{i+1}.Par=n_i.Par
following-sibling	following AND n_{i+1}.Par=n_i.Par
attribute	n_{i+1}.Par=n_i.Pre AND n_{i+1}.Type="attr"
self	n_{i+1}.Pre=n_i.Pre

Using the query conditions presented in Table 4 and the additional query conditions presented in Table 3, we can now translate the XPath query n_1`/descendant::record` into the following SQL query:

```
SELECT DISTINCT n2.*
FROM Node n1, Node n2, AncDesc a1
WHERE n2.Pre=a1.Desc AND a1.Anc=n1.Pre AND NOT n2.Pre=n1.Pre
AND n2.Type="elem" AND n2.Name="record"
ORDER BY n2.Pre;
```

It is worth noticing that in this query, there are no expensive nonequijoins, and thus we can expect this query to run faster than the corresponding query that is based on pre-/postorder encoding.

4.3 Ancestor/Leaf Index

Simply put, an ancestor/leaf index is an ancestor/descendant index built only the leaf nodes as descendants. To maintain the ancestor/leaf information, we employ an `AncLeaf` table:

```
Node(Pre, Par, Type, Name, Value)
AncLeaf(Anc, Leaf)
```

Obviously, the database attribute `Leaf` corresponds to the preorder number of the leaf node and `Anc` corresponds to the preorder number of the ancestor of the leaf node. Similarly to the ancestor/descendant approach, all leaf nodes are in relation `AncLeaf` with themselves.

To evaluate all the axes using ancestor/leaf index, we first define a "special axis" `special` which, for a `Node` tuple variable n_i, can be evaluated using `AncLeaf` variables a_i and b_i with query conditions n_{i+1}`.Pre=`a_i`.Anc AND` a_i`.Leaf=`b_i`.Leaf AND` b_i`.Anc=`n_i`.Pre`. In simple terms, the result of the `special` axis contains all ancestors and descendants of the context node and the context node itself. To filter the ancestors or descendants from the result of this query, we can use the preorder numbers of the nodes, as presented in Table 5.

Table 5. Ancestor/leaf index query conditions for performing the axes

Axis	Query conditions
`parent`	n_{i+1}`.Pre=`n_i`.Par`
`child`	n_{i+1}`.Par=`n_i`.Pre`
`ancestor`	`special AND` n_{i+1}`.Pre<`n_i`.Pre`
`descendant`	`special AND` n_{i+1}`.Pre>`n_i`.Pre`
`ancestor-or-self`	`special AND` n_{i+1}`.Pre<=`n_i`.Pre`
`descendant-or-self`	`special AND` n_{i+1}`.Pre>=`n_i`.Pre`
`preceding`	n_{i+1}`.Pre<`n_i`.Pre AND` n_{i+1} `NOT IN ancestor`
`following`	n_{i+1}`.Pre>`n_i`.Pre AND` n_{i+1} `NOT IN descendant`
`preceding-sibling`	`preceding AND` n_{i+1}`.Par=`n_i`.Par`
`following-sibling`	`following AND` n_{i+1}`.Par=`n_i`.Par`
`attribute`	n_{i+1}`.Par=`n_i`.Pre AND` n_{i+1}`.Type="attr"`
`self`	n_{i+1}`.Pre=`n_i`.Pre`

Using these query conditions and the additional query conditions presented in Table 3, we can evaluate our example query n_1`/descendant::record` using the following SQL query:

```
SELECT DISTINCT n2.*
FROM Node n1, Node n2, AncLeaf a1, AncLeaf b1
WHERE n2.Pre=a1.Anc AND a1.Leaf=b1.Leaf AND b1.Anc=n1.Pre
AND n2.Pre>n1.Pre
AND n2.Type="elem" AND n2.Name="record"
ORDER BY n2.Pre;
```

Although this query involves one nonequijoin on preorder numbers, it can be evaluated quite efficiently, since most of the work can be carried out using inexpensive equijoins. Provided that the database management system optimizes the query correctly, the nonequijoin is very likely performed after the equijoins, and thus the nonequijoin is performed on only a small number of tuples. To avoid unnecessary disk I/O during queries, the `AncLeaf` table should be sorted according to the database attribute `Leaf`.

5 Proxy Index

Both ancestor/descendant index and ancestor/leaf index suffer from the same problem: they maintain a lot of redundant information. In this section, we thus describe a *proxy index* as a method for representing the structural relationships in a more compact manner. Our idea is to select a set of inner nodes to act as *proxy nodes* and to maintain the ancestor/descendant information only for these proxies. The concept of proxy node can formally be defined as follows:

Definition 1. *Node n is a proxy node of level i, if the length of the path from n to some leaf node is i or if n is the root node and the length of the path from n to some leaf node is smaller than i.*

To couple the proxy nodes together with their ancestors and descendants, we use the relation `ProxyReach`:

```
Node(Pre, Post, Par, Type, Name, Value)
ProxyReach(Proxy, Reach)
```

In this schema, the database attribute `Proxy` corresponds to the preorder number of the proxy node and `Reach` corresponds to the preorder number of a node that is reachable from the proxy, i.e., a node that is an ancestor or descendant of the proxy node or the proxy node itself.

Notice that each proxy node corresponds to an equivalence class induced by an equivalence relation $\equiv_{pi}$ on the leaf nodes defined as follows:

Definition 2. *For any two leaf nodes n_1 and n_2, $n_1 \equiv_{pi} n_2$, iff there exists a proxy node n_3 of level i such that n_3 is the nearest proxy node ancestor for both n_1 and n_2.*

Using this notion, we can now define the ancestor/leaf index as a proxy index corresponding to the relation $\equiv_{p0}$. Obviously, the structural information can be encoded more compactly by increasing the value of i. Building an index corresponding to the relation $\equiv_{p1}$, however, will probably not get us far, since in the case of typical XML trees, most leaf nodes are text nodes without any siblings. Thus, with value 1 we will end up with almost as many tuples in the `ProxyReach` table as with value 0. For this reason, our experimental evaluation was carried out using value 2 which proved to provide a good balance between query speed and storage consumption.

Similarly to the ancestor/leaf index, we use a "special axis" `special` which, for a `Node` tuple variable n_i, can be evaluated using `ProxyReach` variables a_i and b_i with query conditions n_{i+1}`.Pre=`a_i`.Reach AND` a_i`.Proxy=`b_i`.Proxy AND` b_i`.Reach=`n_i`.Pre`. Here, the result of the `special` axis contains all descendants and ancestors of the proxy nodes that are reachable from n_i. Thus, we can use query conditions similar to those used in the pre-/postorder encoding approach to filter the unwanted nodes from the result of the axis, as presented in Table 6.

Table 6. Proxy index query conditions for performing the axes

Axis	Query conditions
`parent`	n_{i+1}`.Pre=`n_i`.Par`
`child`	n_{i+1}`.Par=`n_i`.Pre`
`ancestor`	`special AND` n_{i+1}`.Pre<`n_i`.Pre AND` n_{i+1}`.Post>`n_i`.Post`
`descendant`	`special AND` n_{i+1}`.Pre>`n_i`.Pre AND` n_{i+1}`.Post<`n_i`.Post`
`ancestor-or-self`	`special AND` n_{i+1}`.Pre<=`n_i`.Pre AND` n_{i+1}`.Post>=`n_i`.Post`
`descendant-or-self`	`special AND` n_{i+1}`.Pre>=`n_i`.Pre AND` n_{i+1}`.Post<=`n_i`.Post`
`preceding`	n_{i+1}`.Pre<`n_i`.Pre AND` n_{i+1}`.Post<`n_i`.Post`
`following`	n_{i+1}`.Pre>`n_i`.Pre AND` n_{i+1}`.Post>`n_i`.Post`
`preceding-sibling`	`preceding AND` n_{i+1}`.Par=`n_i`.Par`
`following-sibling`	`following AND` n_{i+1}`.Par=`n_i`.Par`
`attribute`	n_{i+1}`.Par=`n_i`.Pre AND` n_{i+1}`.Type="attr"`
`self`	n_{i+1}`.Pre=`n_i`.Pre`

Using these query conditions and the additional query conditions presented in Table 3, we can evaluate our example query n_1`/descendant::record` using the following SQL query:

```
SELECT DISTINCT n2.*
FROM Node n1, Node n2, ProxyReach a1, ProxyReach b1
WHERE n2.Pre=a1.Reach AND a1.Proxy=b1.Proxy AND b1.Reach=n1.Pre
AND n2.Pre>n1.Pre AND n2.Post<n1.Pre
AND n2.Type="elem" AND n2.Name="record"
ORDER BY n2.Pre;
```

Although this query involves two nonequijoins, it can still be evaluated rather efficiently, as explained in the section discussing ancestor/leaf index.

6 Experimental Results

This section presents the results of our experimental evaluation carried out using MySQL 5.0 Alpha running on Windows XP and a 2.00 GHz Pentium PC with 512 MB of RAM and standard IDE disks. As in [9], the database schemas corresponding to pre-/postorder encoding (PP), ancestor/descendant index (AD), ancestor/leaf index (AL), and proxy index (PR) were all extended with document identifiers; the proxy index was built on relation $\equiv_{p2}$.

6.1 Storage Requirements

To start with, we compared the storage consumption of the methods by storing three different sets of XML documents into databases designed according to PP, AD, AL, and PR[2]. For this purpose, we used the 1998 baseball statistics and a collection of four religious texts, both available at [19]. We also studied the storage requirements using a deeply nested and structurally complex XML document generated with XMLgen [20] using factor 1. The database sizes in both tuples and megabytes, as well as the sizes of original XML documents, are presented in Table 7.

Table 7. Database sizes

		PP		AD		AL		PR	
	MB	Tuples	MB	Tuples	MB	Tuples	MB	Tuples	MB
Baseball	0.6	52707	3.6	393095	15.1	233142	9.9	110608	5.2
Religious	6.8	94616	11.7	599005	29.0	366178	20.6	195006	15.0
XMLgen	113	3221932	305	23134116	998	13952004	686	8339906	486

As can be seen in Table 7, PR performs quite well in terms of storage consumption. Although PR consumes more storage than PP, the difference between PP and PR is not nearly as significant as between PP and AD or AL. In the next section, we will see that PR allows us to evaluate queries much more efficiently than PP, which justifies the bigger database size.

6.2 Query Performance

We evaluated the query performance of the different approaches using synthetic documents generated with XMLgen using factors 0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64, and 1.28. The sizes of these documents ranged approximately from 1.1 MB to 146 MB. From each document, we randomly selected 20 context nodes, and performed the `parent`, `child`, `ancestor`, `descendant`, `following-sibling`, and `preceding-sibling` axes starting from these context nodes.

According to our experiments, the evaluation time of all axes grows linearly with respect to the document size. Regardless of the document size, however, axes `parent`, `child`, `preceding-sibling`, `following-sibling`, and `descendant` can be evaluated in almost no time in all approaches, provided that the `descendant` queries in PP are accelerated.

The results concerning the `ancestor` axis, on the contrary, reveal one of the weaknesses of PP. Since this axis cannot be accelerated, the query evaluation time in PP grows rapidly compared to other approaches. In approaches AD, AL, and PR, not only the `descendant` axis, but also the `ancestor` axis can be

[2] Auxiliary indexes were built on `Node(Doc, Post)` `Node(Doc, Par)`, `Node(Type)`, `AncDesc(Doc, Desc)`, `AncLeaf(Doc, Leaf)`, `Proxy(Doc, Reach)`, and the first five characters of `Node(Name)`.

evaluated very efficiently. Evaluating this axis using the largest test document, for example, took almost 16 seconds in PP, whereas in AD, AL, and PR, almost no time at all was needed. The results are presented in Fig. 1; all times are averages for the 20 context nodes.

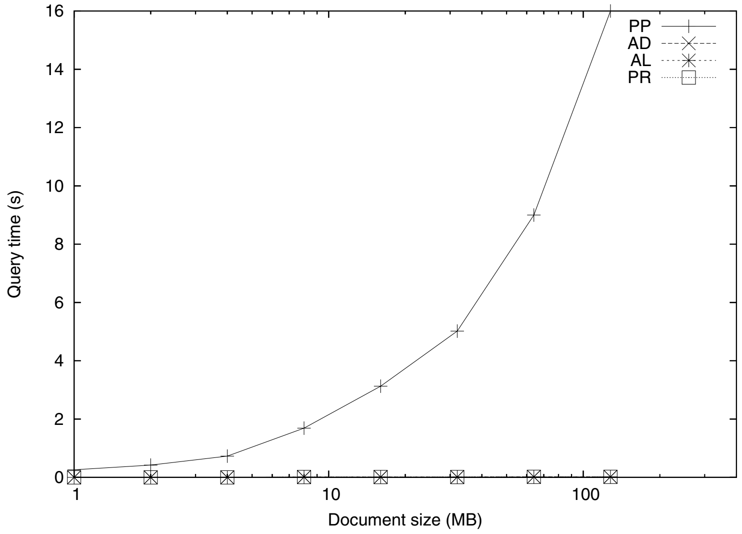


Fig. 1. Times for the `ancestor` axis (logarithmic scale for size)

As discussed section 4.1, the `descendant` and `descendant-or-self` axes are not accelerated in PP if the root is the context node. This is actually a rather serious drawback, since all queries based on absolute addressing traverse the tree starting from the root. To exemplify this problem, Fig. 2 presents the query times for the XPath query `/descendant-or-self::open_auction`. According to these results, the scalability of PP leaves a lot to be desired when the traversal starts from the root. Also in AL, the query times grow rather rapidly, but both PR and AD perform very well indeed even in the case of the largest test document. Although the SQL queries in PR look similar to the queries in AL, PR clearly outperforms AL, since it issues less disk accesses.

After these tests, we still wanted to see how the approaches perform when a large set of context nodes is used. To do this, we stored the result sets of our previous query `//open_auction` into separate relations and performed the location step `/descendant-or-self::description` starting from these node sets; the size of the context node set varied from 120 to 15360. In these tests, PP suffered from severe scalability problems with respect to the number of context nodes. For instance, even in the case of the smallest test document, PP took almost two minutes, whereas the other approaches needed less than a second, which clearly supports our presentiments on the lack of scalability in PP. Indeed, although the acceleration can make a substantial difference when the set of context nodes is small, PP does not scale well with respect to the number of

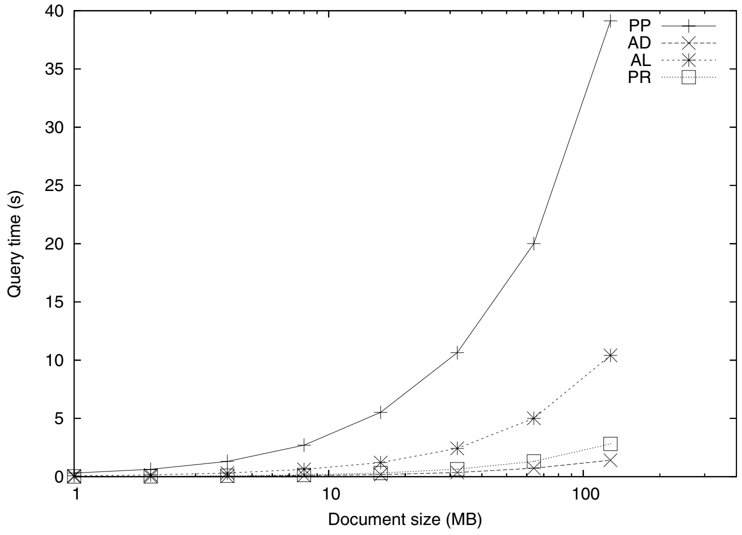


Fig. 2. Times for query `//open_auction` (logarithmic scale for size)

context nodes. PR performed better than AL and also in these tests, AD provided the best performance. These results are presented in Fig. 3.

All in all, our results were similar to those obtained in [10] and [17]. It must be pointed out that according to Grust [7], PP performs much better if the pre- and postorder numbers are indexed using R-trees. The purpose of this study, however, was to rely on standard relational database technology, and thus we did not consider this possibility.

7 Conclusion and Future Work

In this paper, we discussed methods for modeling the nested relationships in XML documents using relational databases. We also proposed a novel method, namely a proxy index, which maintains the ancestor/descendant information for a selected set of inner nodes. Our proposal makes it possible to check the nested relationships mainly using equijoins instead of nonequijoins, and thus our method can clearly outperform the widely used pre-/postorder encoding. Furthermore, our method encodes the structural relationships in a compact manner, and thus the storage consumption is low compared to other methods that model the structural relationships explicitly.

It would be interesting to study how the approaches discussed in this paper perform when relative addressing is used. Since PR performed quite well with large sets of context nodes, we believe that our approach performs well also when relative addressing of XPath is used. It would also be interesting to study how different clusterings, i.e., orderings of the tuples, affect the performances of the methods. Ordering the tuples according to their names instead of their preorder numbers, for example, might provide good results.

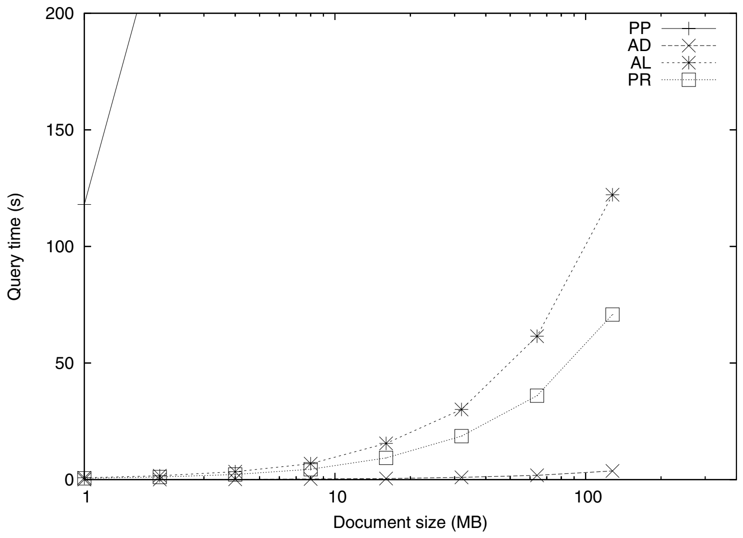


Fig. 3. Times for query `//open_auction//description` (logarithmic scale for size)

References

1. W3C. Extensible Markup Language (XML) 1.0. `http://www.w3c.org/TR/REC-xml`.
2. A.B. Chaudri, A. Rashid, and R. Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems.* Addison-Wesley, 2003.
3. W3C. XML path language (XPath) 2.0. `http://www.w3c.org/TR/xpath20`.
4. W3C. XQuery 1.0: An XML query language. `http://www.w3c.org/TR/xquery`.
5. 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 Technologies*, 1(1): 110-141, 2001.
6. Q. Li and B. Moon. Querying XML data for regular path expressions. In *Proc. of the 27th Intl Conf. on Very large Databases*, pages 361-370, 2001.
7. T. Grust. Accelerating XPath location steps. In *Proc. of the 2002 ACM SIGMOD Conf. on Management of Data*, pages 109-120, 2002.
8. C. Zhang, J. Naughton, D. DeWitt, Qiong Luo, G. Lohman. On supporting containment queries in relational database management systems. In *Proc. of the 2001 ACM SIGMOD Conf. on Management of Data*, pages 425-436, 2001.
9. O. Luoma. Modeling nested relationships in XML documents using relational databases. In *Proc. of the 31st Annual Conf. on Current Trends in Theory and Practice of Informatics*, pages 259-268, 2005.
10. H. Jiang, H. Lu, W. Wang, and J. Xu Yu. Path materialization revisited: An efficient storage model for XML data. In *Proc. of the 13th Australasian Database Conf.*, pages 85-94, 2002.
11. J. McHugh, S. Abiteboul, R. Goldman, R. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3): 54-66, 1997.
12. C.C. Kanne and G. Moerkotte. Efficient storage of XML data. Poster abstract in *Proc. of the 16th Intl Conf. on Data Engineering*, page 198, 2000.

13. V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proc. of the 1994 ACM SIGMOD Intl Conf. on Management of Data*, pages 313-324, 1994.
14. D. Florescu and D. Kossmann. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical Report, INRIA, 1999.
15. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D.J. DeWitt, and J.F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. of the 25th Intl Conf. on Very Large Databases*, pages 302-314, 1999.
16. R. Krishnamurthy, R. Kaushik, and J.F. Naughton. XML-to-SQL query translation literature: The state of the art and open problems. In *Proc. of the 1st Intl XML Database Symposium*, pages 1-18, 2003.
17. S. Prakash, S.S. Bhowmick, and S. Madria. SUCXENT: An efficient path-based approach to store and query XML documents. In *Proc. of the 15th Intl Conf. on Database and Expert Systems Applications*, pages 285-295, 2004.
18. P.F. Dietz. Maintaining order in a linked list. In *Proc. of the 14th ACM Symposium on Theory of Computing*, pages 122-127, 1982.
19. `http://www.ibiblio.org/xml/examples`.
20. `http://monetdb.cwi.nl/xml/index.html`.

An Extended Preorder Index for Optimising XPath Expressions

Martin F. O'Connor[1], Zohra Bellahsène[2], and Mark Roantree[1]

[1] Interoperable Systems Group, Dublin City University, Ireland
{moconnor,mark.roantree}@computing.dcu.ie
[2] LIRMM, UMR 5506 CNRS Université Montpellier II, France
bella@lirmm.fr

Abstract. Many of the problems with native XML databases relate to query performance and subsequently, it can be difficult to convince traditional database users of the benefits of using semi- or unstructured databases. Presently, there still lacks an index structure providing efficient support for structural queries and the traditional data-centric and content queries. This paper presents an extended index structure based on the preorder traversal rank and the level (or depth) rank of each node in a document tree. The extended index fully supports the navigation of all XPath axes while efficiently supporting data-centric queries. The ability to start path traversals from arbitrary nodes in a document tree also enables the extended index to support the evaluation of path traversals embedded in XQuery expressions. Furthermore, an encoding technique is presented where properties of the level ranking may be exploited to provide efficient and optimised level-based XPath evaluations.

1 Introduction

XML has been adopted as the new standard for data exchange on the World Wide Web and increasingly so in industry as the standard data interchange format. The key ingredient to its successful adoption is the expressive and extensible nature of XML. The basic structure underlying XML is the tree, which represents semi-structured data. Semi-structured data consists of an irregular and non-uniform organisation; it may have data with missing attributes and some attributes may be of different types within different data items. All of these variations are acceptable in XML documents. Thus, it may be seen that XML provides for an unlimited number for tree dialects, some of which have been formally described by Document Type Descriptors (DTDs) or XML Schemas, while others are employed in an ad-hoc schema-less manner. The database community is well advanced in adapting its technology to host large XML collections and to query these collections efficiently. It will be essential, though, that these new technologies support the XML query language specifications such as XPath [13] and XQuery [12]. These specifications are key enablers in maintaining the interoperability among XML repositories.

1.1 Motivation

The operations and path traversals required in the querying of tree structured data present difficult challenges. There has been much activity on the specification and provision of extensions to existing indexing mechanisms and processing models to enable the efficient exploitation of the structural properties of XML. The goal of this activity is to support, not only rapid navigational or structural queries but efficient content-based queries as well [1] [15]. There have also been several proposals [5] [7] for new index structures to deal with these problems. However, virtually all of the proposals focus on support for step evaluation along the child and descendant-or-self axes, to the detriment of the remaining XPath axes. Moreover, these proposals often rely on query processing algorithms which call for implementation techniques that lie outside their natural domain. An example is the relational domain where such proposals incur associated drawbacks such as additional software layers and transactional and performance issues. Indeed, as trees in their abstract form may be queried using path expressions, the XPath language was defined to model and query an XML document as a tree of nodes. The XQuery specification moreover facilitates embedded path traversals that may commence from any arbitrary node. Presently, there still lacks an index structure facilitating embedded XPath traversals from arbitrary nodes while providing at the same time, efficient and optimised XPath traversal evaluations incorporating both structural and navigational queries and the traditional content and data-centric queries. Our PreLevel Index structure fills this gap.

1.2 Contribution

In this paper, we present a new tree encoding mechanism based solely on the preorder traversal rank and the level (or depth) rank of each node in the document tree. We define new conjunctive range predicates based on our tree encoding to support the evaluation of location steps along the principle XPath axes and provide proofs to validate them. We then present an Extended Index structure (hereafter, referred to as the PreLevel Structure) based on our tree encoding that fully supports all XPath axes. Both the preorder traversal rank and level rank values may be determined during the initial parsing of the XML document and thus, the PreLevel Structure has minimal computational overhead associated with its construction. The ability to start traversals from arbitrary context nodes in an XML tree also enables the PreLevel Structure to support the evaluation of path traversals embedded in XQuery expressions. Furthermore, using our PreLevel Structure, the properties of the level rank of a node may be exploited to provide efficient and optimised level-based XPath evaluations.

The paper is organised as follows: Section 2 reviews the *partition* property of the XPath language and presents our PreLevel encoding and the newly derived conjunctive range predicates that facilitate XPath axis navigation, together with formal proofs of their derivation. Section 3 presents the tabular representation of the PreLevel Structure, explaining its construction and illustrating an evaluation of a step location along the descendant axis. Section 4 highlights various features of the PreLevel Structure and outlines some of the optimised XPath queries possible. Section 5 reviews related work and we conclude in Section 6.

2 Presenting the PreLevel Structure Encoding

In this section, the XPath partition property is reviewed and the PreLevel encoding mechanism is introduced. For each of the *primary* XPath axes, the new conjunctive range predicates for performing a location step along the axis are presented and the corresponding proofs provided. The conjunctive range predicates have been derived from the intrinsic properties of the *preorder* traversal ranks and *level* ranks alone.

2.1 XPath Partition Property

The basic data type underlying XML is the tree. Thus, the XPath language was defined to model and query an XML document as a tree of nodes. The XPath 2.0 working draft [13] also specifies the following partitioning property: the **ancestor**, **descendant**, **preceding**, **following** and **self** axes partition an XML document (ignoring attribute and namespace nodes), partitions are disjoint and together they contain all nodes in the XML document. Thus, as a given context node resides in the **self** axis, all other nodes in the XML document fall into one of four partitions, as identified by the axes specified above (hereafter referred to as the *primary* axes).

2.2 The PreLevel Encoding

The PreLevel structural index is an extension to the XPath Accelerator presented in [2]. The PreLevel encoding is based solely on the *preorder traversal* rank encoding and a *level* rank encoding. The *size* information is not recorded as in the encoding mechanism in [3]. The *level* (or depth) function takes one parameter, a node, and returns the *level* rank value of the node. Figure 1(a) depicts a sample XML document and Figure 1(b) depicts the corresponding XML tree with a *preorder* and *level* rank encoding.

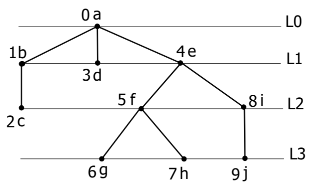
Thus, *level(v)* = m if the path from the root of the tree to the node v has length m; for example, *level(a)* = 0 and *level(f)* = 2. The *XPath Partition* property introduced in Section 2.1 is preserved by the combined *preorder traversal* and *level* rank encoding. The remaining XPath axes (**parent**, **child**, **descendant-or-self**, **ancestor-or-self**, **following-sibling** and **preceding-sibling**) determine either supersets or subsets of one of the *primary* axes and may be evaluated from them.

2.3 Navigating the Descendant Axis

The **descendant** axis selects all children of the given context node, and their children recursively, with the resulting nodes in document order [13]. The new conjunctive range predicate defining a location step along the **descendant** axis, based on the PreLevel encoding, is as follows:

```
<a>
  <b><c></c></b>
  <d></d>
  <e>
    <f><g></g><h></h></f>
    <i><j></j></i>
  </e>
</a>
```

(a) Sample XML document



(b) PreLevel encoded XML tree.

Fig. 1. Sample XML document and associated PreLevel encoded tree.

Lemma 1.

$$\begin{aligned} v \in c/\textbf{descendant} \Leftrightarrow\ & pre(v)\ >\ pre(c) && (i)\\ \wedge\ & level(v)\ >\ level(c) && (ii)\\ \wedge\ & \forall x : pre(x) \in (pre(c)\,,\, pre(v)) && (iii)\\ & \Rightarrow\ level(x) \neq level(c) && \end{aligned}$$

Lemma 1 states that an arbitrary node v is a **descendant** of a given context node c if and only if:

(i) the *preorder* rank of v is greater than the *preorder* rank of c, and
(ii) the *level* rank of v is greater than the *level* rank of c, and
(iii) for all nodes (let us label them x) having a *preorder* rank greater than *pre(c)* and less than *pre(v)*, that none of those nodes have a *level* rank the same as *level(c)*.

Proof: Condition *(i)* ensures that the *preorder* rank of node v is greater than the *preorder* rank of the context node c. In essence, the first condition exploits the properties of *preorder traversal* to ensure that the arbitrary node v appears, in document order after the given context node c. Condition *(ii)* ensures the *level* rank of node v is greater than the *level* rank of node c. Conditions *(i)* and *(ii)* are intuitive if node v is to be a **descendant** of node c. The third condition ensures that node v does not have another **ancestor** at the same *level* as the given context node c. If there is another **ancestor** at the same *level* as the context node c, then the context node could not be the **ancestor** of node v. This can be stated with certainty due to the properties of *preorder traversal* - namely that a node is visited immediately before its children, and the children are visited from left to right. So, if there is another node at the same *level* as node c, then that node must have a higher *preorder* rank than node c but also a *preorder* rank less than node v (the range requirement of condition *(iii)* ensures this). Thus, although the identity of the **ancestor** at *level(c)* has not been definitely established, it has been definitively determined that the **ancestor** of node v cannot be node c – by finding any other node at the same *level* and within the range specified. Only if there is no node at the same *level* as the context node c and within the range specified, can it be stated with certainty that the context node c is an **ancestor** of node v, and conversely that node v is a **descendant** of the context node c.

An illustration of Lemma 1 now follows. While referring to the conjunctive range predicate in Lemma 1 and to the illustration in Figure 2; let v = node h; let c = node e. To determine if node h is a **descendant** of the context node e, one must examine the conditions:

(i) Is *pre(h)* > *pre(e)*...(7 > 4)...condition holds true.
(ii) Is *level(h)* > *level(e)*...(3 > 1)...condition holds true
(iii) For all nodes whose *preorder* rank is greater than *pre(e)* and less than *pre(h)*, these nodes are located within the shaded area in Figure 2, do any of them have a *level* rank the same as *level(e)*, in this case 1? No, they do not and therefore, the condition holds true.

All three conditions are true, thus node h is a **descendant** of the context node c.

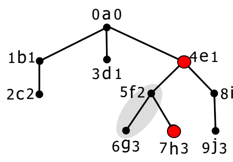


Fig. 2. Example of navigating the descendant axis of a PreLevel encoded XML tree.

Now, let us take an example whereby the conjunctive range predicate will return false. By following the above example, but assigning node d to be the context node c, conditions *(i)* and *(ii)* hold true, but condition *(iii)* fails because node e has the same *level* rank as node d.

2.4 Navigating the Ancestor Axis

The **ancestor** axis selects all nodes in the document that are ancestors of a given context node [13]. Thus, the new conjunctive range predicate defining a location step along the **ancestor** axis, based on the PreLevel encoding, is:

Lemma 2.

$$\begin{array}{rll} v \in c/\textbf{ancestor} \Leftrightarrow & pre(v) < pre(c) & (i) \\ & \wedge\ level(v) < level(c) & (ii) \\ & \wedge\ \forall x : pre(x) \in (pre(v)\,,\, pre(c)) & (iii) \\ & \quad \Rightarrow level(x) \neq level(v) & \end{array}$$

Lemma 2 states that an arbitrary node v is an **ancestor** of a given context node c if and only if:

(i) the *preorder* rank of v is less than the *preorder* rank of c, and
(ii) the *level* rank of v is less than the *level* rank of c, and
(iii) for all nodes (let us label them x) having a *preorder* rank greater than *pre(v)* and less than *pre(c)*, that none of those nodes have a *level* rank the same as *level(v)*.

Proof: Condition *(i)* exploits the properties of *preorder traversal* to ensure the arbitrary node v appears in document order before the given context node c. Condition *(ii)* exploits the *level* rank properties to ensure node v appears higher in the document tree than node c. Condition *(iii)* ensures that the given context node c does not have another **ancestor** at the same *level* as node v. If there is any other node at the same *level* as node v, then node v could not be the **ancestor** of the context node c. This can be stated with certainty due to the properties of *preorder traversal* – namely that a node is visited immediately before its children, and the children are visited from left to right. So, if there is another node at the same *level* as node v, then that node must have a higher *preorder* rank than node v but also a *preorder* rank less than the context node c (the range requirement of condition *(iii)* ensures this). Only if there is no node at the same *level* as node v and within the range specified, can it be stated with certainty that node v is an **ancestor** of the context node c.

2.5 Navigating the Preceding Axis

The **preceding** axis selects all nodes in document order that appear before the given context node, excluding all **ancestors** of the context node [13]. The new conjunctive range predicate, based on the PreLevel encoding, defines a location step along the **preceding** axis as follows:

Lemma 3.

$$\begin{aligned} v \in c/\textbf{preceding} \Leftrightarrow\ & pre(v) < pre(c) && (i) \\ \wedge\ & \exists x : pre(x) \in (pre(v)\,,\, pre(c)] && (ii) \\ & \Rightarrow level(x) \in (0\,,\, level(v)] && \end{aligned}$$

Lemma 3 states that an arbitrary node v is member of the **preceding** axis of a given context node c if and only if:

(i) The *preorder* rank of v is less than the *preorder* rank of c, and
(ii) There exists a node (let us label it x) whose *preorder* rank is greater than *pre(v)* and less than or equal to *pre(c)*, and that the *level* rank of x is less than or equal to *level(v)*.

Proof: Condition *(i)* exploits the properties of *preorder traversal* to ensure the arbitrary node v appears, in document order, before the given context node c. Condition *(ii)* ensures that node v is not an **ancestor** of the context node c. Due to the properties of *preorder traversal,* the existence of any other node which has a *preorder* rank greater than *pre(v)* and less than or equal to *pre(c)*, and which has a *level* rank less than or equal to node v, rules out any possibility that node v is the **ancestor** of node c. Thus, conditions *(i)* and *(ii)* together ensure that an arbitrary node v is a member of the **preceding** axis of given context node c.

2.6 Navigating the Following Axis

The **following** axis selects all nodes that appear after the given context node in document order, excluding the **descendants** of the context node [13]. The new conjunctive range predicate defining a location step along the **following** axis based on the PreLevel encoding is:

Lemma 4.

$$\begin{aligned} v \in c/\textbf{\textit{following}} \Leftrightarrow\ & pre(v) > pre(c) & (i)\\ \wedge\ & \exists x : pre(x) \in (pre(c)\,,\, pre(v)] & (ii)\\ & \Rightarrow level(x) \in (0\,,\, level(c)] & \end{aligned}$$

Lemma 4 states that an arbitrary node v is member of the **following** axis of a given context node c if and only if:

(i) The *preorder* rank of v is greater than the *preorder* rank of c, and
(ii) There exists a node (let us label it x) whose *preorder* rank is greater than *pre(c)* and less than or equal to *pre(v)*, and that the *level* rank of x is less than or equal to *level(c)*.

Proof: Condition *(i)* exploits the properties of *preorder traversal* to ensure the arbitrary node v appears in document order after the given context node c. Condition *(ii)* ensures that node v is not a **descendant** of the context node c. The second condition is validated by verifying that there is another node, with a *preorder* rank greater than *pre(c)* and less than or equal to *pre(v)*, and which has a *level* rank less than or equal to the *level* rank of the context node c. If any such node exists, then due to the properties of *preorder traversal* - namely that a node is visited immediately before its children and the children are visited from left to right - the context node c cannot be the **ancestor** of node v, and conversely node v cannot be the **descendant** of the context node c. Thus, conditions *(i)* and *(ii)* together ensure that an arbitrary node v is a member of the **following** axis of given context node c.

3 Extended Index Structure

In this section we present a tabular representation for the PreLevel encoding that facilitates optimised algorithms for the efficient evaluation of XPath expressions. We adapt the tabular encoding of the *XPath Accelerator* originally proposed in [2] and extend it to incorporate our Extended Preorder Index and Level Index.

3.1 Tabular Encoding

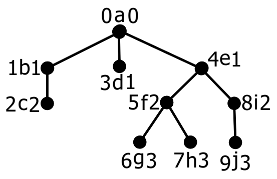
The PreLevel encoding facilitates a tabular representation of XML documents, namely the *Extended Preorder Index*. The primary column of the Extended Preorder Index consists of the *preorder* ranks sorted in ascending order. The second column contains the *level* ranks that correspond to the associated *preorder* ranks of the primary column. Extra columns may be added to the Extended Preorder Index to hold further node properties as defined by the XPath/XQuery data model, such as name, node type (node, element, attribute, comment) and more. In particular, to support the **parent** axis in our tabular encoding, we add a column containing the parent's *preorder* rank of each node to the Extended Preorder Index. However, in order to efficiently evaluate an XPath location step along all of the XPath axes, a second index is required. This second index is introduced (hereafter referred to as the *Level Index*) and consists of two columns only, the *level* rank column and the *preorder* rank column. The first column in the Level Index is the *level* rank column, sorted in ascending order, the second column being the *preorder* rank column, again sorted in ascending order. The Extended Preorder Index and Level Index combined may also be referred to as the *PreLevel Structure*. Several observations should be made at this point.

- Both the *preorder* ranks and the *level* ranks may be determined during the initial parsing of the XML document tree, and thus have minimal computational overheads associated with them.
- Each node in the XML tree has a single *preorder* rank and a single *level* rank associated with it. Thus, the Extended Preorder Index contains a one-to-one mapping. However, as many nodes may reside at the same *level*, the Level Index contains a one-to-many mapping - it is an inverted index.
- Both the Extended Preorder Index and the Level Index can be constructed in parallel during the initial parsing of the XML document tree. The act of parsing of an XML document (reading from top to bottom and left to right) corresponds to a *preorder traversal*. Thus, the Extended Preorder Index is constructed in a sorted list, sorted on the *preorder* rank in ascending order. It may not be obvious that the Level Index is also constructed in a sorted list. When the *preorder traversal* begins, the *level* information is recorded also (*level* 0 for the root node). As the *preorder traversal* progresses, all new *levels* and the associated *preorder* ranks are recorded. As the *preorder traversal* encounters nodes on a *level* already recorded, the *preorder* ranks are simply appended to the list of existing *preorder* ranks at that *level*. Thus, depending on the structure used at implementation time, for example a linked list, when the *preorder traversal* has been completed, we are left with a column of unique *level* ranks, sorted in ascending order with each *level* rank pointing to a linked list of *preorder* ranks and each linked list also sorted in ascending order.
- Lastly, in order to facilitate a lookup of the Level Index in constant time, a *position* column is included in the Extended Preorder Index. During the construction of the Level Index, before any *preorder* ranks have been inserted, each *level* is assigned a counter initialised to zero. As a *preorder* rank is added (or appended) to the Level Index, the counter at that *level* is incremented by one and its value is written in the *position* column of the Extended Preorder Index, in the row of the related *preorder* rank. Thus, the *position* value, when obtained using a lookup of the Extended Preorder Index, facilitates a direct jump to a given *preorder* rank within the Level Index in constant time. The *position* column is the key to enabling the evaluation of location steps on the *primary* XPath axes in constant time and to the optimised evaluations of *level-based* queries (to be introduced in §4.2).

The main issue is to compute the conjunctive range predicates for each of the XPath *primary* axes in *constant time*. This is demonstrated in Section 3.2.

3.2 Example of an Evaluation Along the Descendant Axis

The sample PreLevel encoded tree and the corresponding PreLevel Structure, are illustrated in Figure 3. A high level algorithm detailing the steps to evaluate a location step along the **descendant** axis in constant time is provided in Algorithm 1.



(a) A PreLevel encoded XML tree

Pre	Level	Pos	Parent	Kind	Name	...
0	0	1		Elem	A	
1	1	1	0	Elem	B	
2	2	1	1	Elem	C	
3	1	2	0	Elem	D	
4	1	3	0	Elem	E	
5	2	2	4	Elem	F	
6	3	1	5	Elem	G	
7	3	2	5	Elem	H	
8	2	3	4	Elem	I	
9	3	3	8	Elem	J	

(b) Extended Preorder Index

Level	Pre
0	0
1	1, 3, 4
2	2, 5, 8
3	6, 7, 9

(c) Level Index

Fig. 3. Sample XML tree and the corresponding PreLevel Structure.

Let us now illustrate Algorithm 1. Let v = node h; let c = node e; nodes are represented by their *preorder* rank. It can be verified that *pre(h)* is greater than *pre(e)* (i.e. 7>4), and that *level(h)* is greater than *level(e)* (i.e. 3>1). The Level Index is used to identify the next *preorder* rank greater than *pre(e)* at *level(e)* (i.e. *null*). This information is obtained in constant time as the *position* column of the Extended Preorder Index facilitates a direct jump to *pre(e)* within the *level(e)* index. Note, the next *preorder* rank greater than *pre(e)* at *level(e)*, should it exist, must appear immediately after *pre(e)* because the index is sorted in ascending order. If the next *preorder* rank after *pre(e)* at *level(e)* is greater than *pre(h)*, the node being tested, then node h must be a **descendant** of node e. This can be stated with certainty as the properties of *preorder traversal* require a node's children to be visited immediately after its parent. Also, as in this case, if there are no *preorder* ranks greater than *pre(e)* at *level(e)*, indicated with *null*, node h must be a **descendant** of node e. The fact that there may be no *preorder* ranks greater than *pre(e)* at *level(e)* simply means that node e is the root node of the rightmost subtree rooted at *level(e)*.

This subsection has illustrated an evaluation of a location step along the **descendant** axis in constant time, however an evaluation along the **ancestor** axis in constant time may be illustrated in a similar fashion by adapting the algorithm appropriately. An evaluation along the **following** and **preceding**

Algorithm 1 To determine if an arbitrary node v is a **descendant** of a given context node c

```
Name:    IsNodeDescendant
Given:   An arbitrary node $v$, a context node $c$.
Returns: Boolean (TRUE or FALSE)
begin
    //Using the Extended Preorder Index
    if ($pre(v) <= pre(c)$) or ($level(v) <= level(c)$) then
        return FALSE;
    endif
    //Using the Level Index
    next_pre := next preorder rank after $pre(c)$ at $level(c)$;
    if (next_pre > $pre(v)$) or (next_pre == null) then
        return TRUE;
    else
        return FALSE;
    endif
end
```

axes may also be evaluated in constant time however lack of space prevents this demonstration here but may be referenced in [8].

4 Optimised XPath Queries

The PreLevel Structure enables an efficient encoding mechanism that supports highly optimised structural and navigational queries as well as content and data-centric queries.

4.1 Evaluating the Size of a Subtree

Using our PreLevel Structure, the size of a subtree tree rooted at an arbitrary node v can be determined very efficiently. The evaluation of the subtree size is independent of the actual size of the subtree (and indeed the size of the entire document tree) but rather dependent on the number of levels between the given node v and the root node of the entire document tree. In [6], a comprehensive study of over 190,000 XML trees was performed revealing that 99% of all the documents had less than 8 levels. The vast majority of the remaining 1% of documents had less than 30 levels, with only a tiny fraction having more than 30 levels. Thus, it may be seen that the number of levels (or depth) in an XML tree is sufficiently small so as to be deemed to have a minimal computational impact on our evaluation. The size of the subtree evaluated with our algorithm is accurate and no extra information beyond the *preorder* and *level* ranks are necessary to determine the size of the subtree. A more detailed explanation of this algorithm may be found in [8]. An algorithm demonstrating the steps required to evaluate the size of a subtree rooted at an arbitrary node v using our PreLevel Structure is provided in Algorithm 2.

Algorithm 2 To determine the size of subtree rooted at an arbitrary node v

```
Name:     SizeOfSubtree
Given:    An arbitrary node v,
          The maximum preorder rank in document tree max_pre.
Returns:  subtree_size
begin
  //Using the Extended Preorder Index, determine if node v is a leaf node
  if (level(pre(v) + 1) <= level (v)) then subtree_size := 1;
    return subtree_size;
  endif
  //Using the Level Index
  next_pre := next preorder rank after pre(v) at level(v);
  //limit will contain the maximum upper preorder rank of the preorder interval (non-inclusive)
  //specifying the subtree nodes.
  limit := next_pre;
  init_level := level(v) - 1;
  //par(v) returns the preorder rank of the parent node of v
  par_pre := par(v);
  //For each level between level(v) and root node, find first node with preorder rank > pre(v)
  for (count = init_level; count > 0; count --)
    next_pre := next preorder rank after par_pre at level(par_pre);
    if (limit != null) then
      if (next_pre != null) and (next_pre < limit) then limit := next_pre;
      endif
    endif
    par_pre := par(par_pre);
  endfor
  if (limit != null) then subtree_size := limit - pre(v);
  else subtree_size := (max_pre - pre(v)) + 1;
  endif
  return subtree_size;
end
```

The *SizeOfSubtree* function facilitates the efficient evaluation of *all members* of the **descendant** and **following** axes of a given node v. By exploiting the *parent* column in the Extended Preorder Index we can also very efficiently evaluate *all members* of the **ancestor** and **preceding** axes for any given arbitrary node v. The remaining XPath axes (**parent**, **child**, **descendant-or-self**, **ancestor-or-self**, **following-sibling**, and **preceding-sibling**) determine either supersets or subsets of one of the *primary* axes and may be evaluated from them.

4.2 Optimised Level-Based Queries

The PreLevel Structure makes a notable contribution to the efficient processing of XPath expressions by facilitating optimised evaluations of *level-based* queries. A *level-based* query is such that the results of the query reside at a particular level in the XML tree.

Taking the **descendant** axis as an example, all nodes that are a **descendant** of an arbitrary node v will reside in a *preorder-defined* interval, delimited by lower and upper *preorder* ranks. Thus, using our Level Index, it is easy to identify a sequence of nodes residing at a particular *level* that belong to a *preorder-defined* interval. For example, given a query to select all grandchildren of an arbitrary node v; the result of such a query will be represented using the Level Index as an interval or array with lower and upper *preorder* bounds residing at a specific *level*. The *position* column of the Extended Preorder Index facilitates a direct jump to the lower and upper *preorder* bounds within the Level Index.

The Level Index is sorted in ascending order and can be searched very efficiently using a binary search algorithm with a time complexity of $O(lg\ n)$. The lower bound of the *preorder* interval containing node v's **descendants** at a given level l, is obtained by performing a binary search at level l for the first *preorder* rank greater than *pre(v)*. In a similar fashion, the upper bound of the *preorder* interval containing node v's **descendants** at a given level l, is obtained by performing a binary search at level l for the last *preorder* rank preceding a *container preorder* rank of node v's **descendants**. A *container preorder* rank is a *preorder* rank greater than the largest *preorder* rank in node v's **descendants**. Due to the properties of *preorder traversal*, a valid *container preorder* rank for node v's **descendants** is the next *preorder* rank greater than *pre(v)* at *level(v)*. The *container preorder* rank can be obtained in constant time using a lookup of the Level Index and provides an upper bound for node v's **descendants** at an arbitrary level l.

Thus, given the *preorder* rank of a context node, the upper and lower bounds of the interval containing the context node's **descendants** at an arbitrary level l can be obtained using the Level index, requiring only two lookup operations of time complexity $O(lg\ n)$ each, at level l. The processing of nodes at intermediary levels is unnecessary for all levels between the context node and the level to be queried.

The optimal time complexity for reading n values from an array of size n is linear, i.e. $O(n)$. Thus, given that the results of a *level-based* query is an array subset of the Level Index, which is always sorted in document order; and given that the *position* column of the Extended Preorder Index facilitates a direct jump to the lower and upper *preorder* bounds within the Level Index; when both lower and upper bounds of the interval have been obtained, the actual results of the *level-based* query may be retrieved in *optimal* time. Indeed, once the interval is know, the solution is optimal for retrieving all **descendants** of a given node v that reside at an arbitrary level l. A sample algorithm to evaluate all **descendants** of a given node v residing at an arbitrary level l is provided in Algorithm 3.

In a similar fashion, the solution for identifying all members of the **following-sibling** and **preceding-sibling** axes are also optimal. It should be noted that queries along the **descendant**, **descendant-or-self** and **child** axes of an arbitrary node constitute the core of XPath subexpressions embedded in XQuery statements and provide the most challenging and highly computational tasks for XPath/XQuery processors.

Algorithm 3 To determine all the **descendants** of an arbitrary node v at a given level m

```
Name:     AllDescendantsAtLevelM
Given:    An arbitrary node $v$,
          The maximum preorder rank in document tree $max\_pre$,
          A level $m$, where $m$ is the path length from root node to node $v$.
Returns: A sequence of document nodes labelled descendants or the empty_sequence
begin
  //Using Extended Preorder Index, determine if v is a leaf node
  if ($level(pre(v) + 1) <= level(v)$) then
    return empty_sequence;
  endif
  //Using the Level Index
  next_pre := next preorder rank after $pre(v)$ at $level(v)$;
  //Convert relative level rank to absolute level rank of document tree.
  queryLevel := $level(v) + m$;
  if (next_pre != null) then
    start_pre := next preorder value > $pre(v)$ at queryLevel;
    descendants = all nodes in interval [start_pre , next_pre) at queryLevel;
  else
    descendants = all nodes in interval ($pre(v)$ , $max\_pre$] at queryLevel;
  endif
  return descendants;
end
```

5 Related Work

In [11], the experience of building Jungle, a secondary storage manager for Galax, an open source implementation of the family of XQuery 1.0 specifications is presented. They chose to implement the Jungle XML indexes using the *XPath Accelerator*. However, one significant limitation they encountered was the evaluation of the **child** axis, which they found to be as expensive as evaluating the **descendant** axis. They deemed this limitation to be unacceptable and designed their own alternative indexes to support the **child** axis. Although the XPath Accelerator *pre/post* encoding scheme has since been updated in [3] to use *pre/level/size,* which Jungle has yet to incorporate, our PreLevel Structure as demonstrated in Section 4.2 supports highly efficient evaluations of not just children, but grandchildren and indeed all nodes at any particular level of an arbitrary node. The ability to efficiently evaluate level-based queries by considering only the nodes at the level concerned and eliminating the need for large scale processing at the intermediary levels, is the principle contribution of the PreLevel Structure. The Jungle implementation experience also highlighted the significant overhead imposed at document loading time by a *postorder traversal*, a necessary component in the construction of the indexing system proposed in [14].

There has been much research into the development and specification of new indexing structures to efficiently exploit the properties of XML. There have been several initiatives to extend the relational data model to facilitate the XML data model and once again the XPath Accelerator has been at the forefront [4] [1] [15]. In [10], the key issue of whether the ordered XML data model can be efficiently represented by the relational tabular data model is examined and the authors propose three new encoding methods to support their belief that it can. In [5], a new system called XISS is proposed for indexing and storing XML data, specifying three new structures to support content queries, and a new numbering scheme, based on the notion of extended preorder to facilitate the evaluation of **ancestor-descendant** relationships between elements and attributes in constant time. In [9], a hierarchical labelling scheme called ORDPATH, implemented in the upcoming version of Microsoft SQL Server, is proposed. Each node on an XML tree is labelled with an ordinal value, a compressed binary representation of which, provides efficient document ordering evaluation as well as structural evaluation. In addition, the ORDPATH scheme supports insertion of new nodes in arbitrary positions in the XML tree, without requiring the re-labelling of any nodes.

6 Conclusion

There is an urgent need for an indexing structure capable of supporting very efficient structural, navigational and content-based queries over both document-centric and data-centric XML. Our PreLevel Structure makes a significant contribution toward this goal. In this paper we have presented a new tree encoding mechanism based solely on the *preorder traversal* rank and the *level* rank of a node. We constructed new conjunction range predicates based on the PreLevel encoding to support the evaluation of location steps along the *primary* XPath axes and provided proofs of their derivation. We then presented a tabular encoding for our PreLevel Structure – the Extended Preorder Index and Level Index – to enable the navigation of all XPath axes and demonstrated how these indexes have a minimal computational overhead associated with their construction. The tabular representation of the PreLevel Structure allows for flexible implementation strategies. Finally, accompanied by several algorithms, we detailed how our tabular encoding facilitates efficient XPath queries and expression evaluations. In particular, the properties of the Level index may be exploited to provide highly optimised level-based query evaluations as well as the optimal retrieval of their results.

As part of our future work, we are investigating the possibility of supporting efficient XML updates. In tandem with our research, we have short listed several open-source native XML databases and are examining them with a view to providing an implementation of our work to date.

References

1. Daniela Florescu and Donald Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

2. Torsten Grust. Accelerating XPath Location Steps. In *Proceedings of the 2002 ACM SIGMOD International Conference on the Management of Data*, volume 31, pages 109–120. SIGMOD Record, ACM Press, 2002.
3. Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*, pages 252–263. Morgan Kaufmann, 2004.
4. Torsten Grust and Jens Teubner. Relational Algebra: Mother Tongue–XQuery: Fluent. In *1st Twente Data Management Workshop on XML Databases and Information Retrieval*. Enschede, The Netherlands, 2004.
5. Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the 27th International Conference on Very Large Databases (VLDB)*, pages 361–370. Morgan Kaufmann, 2001.
6. Laurent Mignet, Denilson Barbosa, and Pierangelo Veltri. The XML Web: A First Study. In *Proceedings of the 12th International World Wide Web Conference (WWW2003)*, pages 500–510. ACM Press, 2003.
7. Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *Proceedings of the 7th International Conference on Database Theory*, pages 277–295. LNCS 1540, Springer, 1999.
8. Martin O'Connor, Zohra Bellashène, and Mark Roantree. Level-based Indexing for Optimising XPath Expressions. Technical report, Interoperable Systems Group, Dublin City University, 2005. Available from: `www.computing.dcu.ie/~isg/technicalReport.html`.
9. Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proceedings of the 2004 ACM SIGMOD International Conference on the Management of Data*, volume 33, pages 903–908. SIGMOD Record, ACM Press, 2004.
10. Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and Querying Ordered XML using a Relational Database System. In *Proceedings of the 2002 ACM SIGMOD International Conference on the Management of Data*, volume 31, pages 204–215. SIGMOD Record, ACM Press, 2002.
11. Avinash Vyas, Mary F. Fernández, and Jérôme Siméon. The Simplest XML Storage Manager Ever. In *Proceedings of the 1st International Workshop on XQuery Implementation, Experience and Perspectives <XIME-P/> in cooperation with ACM SIGMOD*, pages 37–42, 2004.
12. World Wide Web Consortium. *XQuery 1.0: An XML Query Language*, W3C Working Draft edition, April 2005.
13. World Wide Web Consortium. *XML Path Language (XPath) 2.0*, W3C Working Draft edition, February 2005.
14. Pavel Zezula, Giuseppe Amato, Franca Debole, and Fausto Rabitti. Tree Signatures for XML Querying and Navigation. In *Proceedings of the 1st International XML Database Symposium 2003*, pages 149–163. Springer, September 2003.
15. Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on the Management of Data*, volume 30, pages 425–436. SIGMOD Record, ACM Press, 2001.

XPathMark: An XPath Benchmark for the XMark Generated Data

Massimo Franceschet[1,2]

[1] Informatics Institute, University of Amsterdam
Kruislaan 403 – 1098 SJ Amsterdam, The Netherlands
[2] Dipartimento di Scienze, Università "Gabriele D'Annunzio"
Viale Pindaro, 42 – 65127 Pescara, Italy

Abstract. We propose XPathMark, an XPath benchmark on top of the XMark generated data. It consists of a set of queries which covers the main aspects of the language XPath 1.0. These queries have been designed for XML documents generated under XMark, a popular benchmark for XML data management. We suggest a methodology to evaluate the XPathMark on a given XML engine and, by way of example, we evaluate two popular XML engines using the proposed benchmark.

1 Introduction

XMark [1] is a well-known benchmark for XML data management. It consists of a scalable document database modelling an Internet auction website and a concise and comprehensive set of XQuery queries which covers the major aspects of XML query processing.

XQuery [2] is much larger than XPath [3], and the list of queries provided in the XMark benchmark mostly focuses on XQuery features (joins, construction of complex results, grouping) and provides little insight about XPath characteristics. In particular, only `child` and `descendant` XPath axes are exploited. In this paper, we propose XPathMark [4], an XPath 1.0 benchmark for the XMark document database. We have developed a set of XPath queries which covers the major aspects of the XPath language including different axes, node tests, Boolean operators, references, and functions. The queries are concise, easy to read and to understand. They have a natural interpretation with respect to the semantics of XMark generated XML documents. Moreover, we have thought most of the queries in such a way that the sizes of the intermediate and final results they compute, and hence the response times as well, increase as the size of the document grows. XMark comes with an XML generator that produces XML documents according to a numeric scaling factor proportional to the document size.

The targets of XPathMark are:

– *functional completeness*, that is, the ability to support the features offered by XPath;
– *correctness*, that is, the ability to correctly implement the features offered by XPath;

– *efficiency*, that is, the ability to efficiently process XPath queries;
– *data scalability*, that is, the ability to efficiently process XPath queries on documents of increasing sizes.

Since XPath is the core retrieval language for XSLT [5], XPointer [6] and XQuery [2], we think that the proposed benchmark can help vendors, developers, and users to evaluate these targets on XML engines implementing these technologies.

Our contribution is as follows. In Section 2 we describe the proposed XPath benchmark. In Section 3, we suggest how to evaluate the XPath benchmark on a given XML engine and, by way of example, we evaluate, using XPathMark, two popular XML engines, namely Saxon [7] and Galax [8]. Finally, in Section 4, we outline future work.

2 XPathMark: An XPath Benchmark for the XMark Generated Data

XPathMark has been designed in XML and is available at the XPathMark website [4]. In this section we describe a selection of the benchmark queries.

We first motivate our choice of developing the benchmark as XML data. This solution has all the advantages of XML [9]. In particular:

– the benchmark can be easily read, shipped, and modified;
– the benchmark can be queried with any XML query language;
– it is easier to write a *benchmark checker*, that is an application that automatically checks the benchmark against a given XML engine, that computes performance indexes, and that shapes the performance outcomes in different formats (plain text, XML, HTML, Gnuplot).

Figure 1 contains the Document Type Definition (DTD) for the XML document containing the benchmark. The root element is named `benchmark` and has the attributes `targets` (the targets of the benchmark, for instance, functional completeness), `language` (the language for which the benchmark has been written, for instance XPath 1.0), and `authors` (the authors of the benchmark). The benchmark element is composed of a sequence of `document` elements followed by a sequence of `query` elements. Each `document` element is identified by an attribute called `id` of type ID and contains, enclosed into a Character Data (CDATA) section, a possible target XML document for the benchmark queries. Each `query` element is identified by an attribute called `id` of type ID and has an attribute called `against` of type IDREF that refers to the document against which the query must be evaluated. Moreover, each `query` element contains the following child elements:

– `type`, containing the category of the query;
– `description`, containing a description of the query in English;
– `syntax`, containing the query formula in the benchmark language syntax;

```
<!ELEMENT benchmark     (document*,query*)>
<!ELEMENT document      (#PCDATA)>
<!ELEMENT query         (type,description,syntax,answer)>
<!ELEMENT type          (#PCDATA)>
<!ELEMENT description   (#PCDATA)>
<!ELEMENT syntax        (#PCDATA)>
<!ELEMENT answer        (#PCDATA)>

<!ATTLIST benchmark targets   CDATA #REQUIRED
                    language  CDATA #REQUIRED
                    authors   CDATA #REQUIRED>
<!ATTLIST document  id        ID #REQUIRED>
<!ATTLIST query     id        ID #REQUIRED
                    against   IDREF #REQUIRED>
```

Fig. 1. The benchmark DTD

– `answer`, containing the result of the evaluation of the query against the pointed document, enclosed within a CDATA section. The result is always a sequence of XML elements with no separator between two consecutive elements (not even a whitespace).

We have included in the benchmark two target documents. The first document corresponds to the XMark document generated with a scaling factor of 0.0005. A document type definition is included in this document. The set of queries that have been evaluated on this document are divided into the following 5 categories: axes, node tests, Boolean operators, references, and functions. In the following, for each category, we give a selection of the corresponding benchmark queries (see [4] for the whole query set). See [1] for the XMark DTD.

Axes. These queries focus on the navigational features of XPath, that is on the different kinds of axes that may be exploited to browse the XML document tree. In particular, we have the following sub-categories.

Child Axis. One short query (Q1) with a possibly large answer set, and a deeper one (Q2) with a smaller result. Only the child axis is exploited in both the queries.

Q1 *All the items*

```
/site/regions/*/item
```

Q2 *The keywords in annotations of closed auctions*

```
/site/closed_auctions/closed_auction/annotation/
description/parlist/listitem/text/keyword
```

Descendant Axes. The tag `keyword` may be arbitrarily nested in the document tree and hence the following queries can not be rewritten in terms of `child` axis. Notice that `listitem` elements may be nested in the document. During the processing of query Q4 an XPath processor should avoid to search the same subtree twice.

Q3 *All the keywords*

```
//keyword
```

Q4 *The keywords in a paragraph item*

```
/descendant-or-self::listitem/descendant-or-self::keyword
```

Parent Axis. Elements named `item` are children of the world region they belong to. Since XPath does not allow disjunction at axis step level, one way to retrieve all the items belonging to either North or South America is to combine the parent axis with disjunction at filter level (another solution is query Q22 that uses disjunction at the query level).

Q5 *The (either North or South) American items*

```
/site/regions/*/item[parent::namerica or parent::samerica]
```

Ancestor Axes. Elements named `keyword` may be arbitrarily deep in the document tree hence the ancestor operator in the following queries may have to ascend the tree of an arbitrarily number of levels.

Q6 *The paragraph items containing a keyword*

```
//keyword/ancestor::listitem
```

Q7 *The mails containing a keyword*

```
//keyword/ancestor-or-self::mail
```

Sibling Axes. Children named `bidder` of a given open auction are siblings, and the XPath sibling axes may be exploited to explore them. As for query Q4 above, during the processing of query Q9, the XPath processor should take care to visit each bidder only once.

Q8 *The open auctions in which a certain person issued a bid before another person*

```
/site/open_auctions/open_auction[bidder[personref/@person=
'person0']/following-sibling::bidder[personref/@person='person1']]
```

Q9 *The past bidders of a given open auction*

```
/site/open_auctions/open_auction[@id='open_auction0']
/bidder/preceding-sibling::bidder
```

Following and Preceding Axes. `following` and `preceding` are powerful axes since they may potentially traverse all the document in document or reverse document order. In particular, `following` and `preceding` generally explore more than `following-sibling` and `preceding-sibling`. Compare query Q8 with query Q11: while in Q8 only sibling bidders are searched, in Q11 also bidders of different auctions are accessed.

Q10 *The items that follow, in document order, a given item*

```
/site/regions/*/item[@id='item0']/following::item
```

Q11 *The bids issued by a certain person that precedes, in document order, the last bid in document order of another person*

```
/site/open_auctions/open_auction/bidder[personref/
@person='person1']/preceding::bidder[personref/@person='person0']
```

Node Tests. The queries in this category focus on node tests, which are ways to filter the result of a query according to the node type of the resulting nodes.

Q18 *The children nodes of the root that are comments*

```
/comment()
```

Q21 *The text nodes that are contained in the keywords of the description element of a given item*

```
/site/regions/*/item[@id='item0']/description//keyword/text()
```

Boolean operators. Queries may be disjuncted with the | operator, while filters may be arbitrarily combined with conjunction, disjunction, and negation. This calls for the implementation of intersection, union, and set difference on context sets. These operations might be expensive if the XPath engine does not maintain the context sets (document) sorted.

Q22 *The (either North or South) American items*

```
/site/regions/namerica/item | /site/regions/samerica/item
```

Q23 *People having an address and either a phone or a homepage*

```
/site/people/person[address and (phone or homepage)]
```

Q24 *People having no homepage*

```
/site/people/person[not(homepage)]
```

References. References turns the data model of XML documents from trees into graphs. A reference may potentially point to any node in the document having an attribute of type ID. Chasing references implies the ability of coping with arbitrary jumps in the document tree. References are crucial to avoid redundancy in the XML database and to implement joins in the query language. In summary, references provide data and query flexibility and they pose new challenges to the query processors.

Reference chasing is implemented in XPath with the function `id()` and may be static, like in query Q25, or dynamic, like in queries Q26-Q29. The `id()` function may be nested (like in query Q27) and its result may be filtered (like in query Q28). The `id()` function may also be used inside filters (like in query Q29).

Q25 *The name of a given person*

```
id('person0')/name
```

Q26 *The open auctions that a given person is watching*

```
id(/site/people/person[@id='person1']/watches/watch/@open_auction)
```

Q27 *The sellers of the open auctions that a given person is watching*

```
id(id(/site/people/person[@id='person1']
/watches/watch/@open_auction)/seller/@person)
```

Q28 *The American items bought by a given person*

```
id(/site/closed_auctions/closed_auction[buyer/@person='person4']
/itemref/@item)[parent::namerica or parent::samerica]
```

Q29 *The items sold by Alassane Hogan*

```
id(/site/closed_auctions/closed_auction
[id(seller/@person)/name='Alassane Hogan']/itemref/@item)
```

Functions. XPath defines many built-in functions for use in XPath expressions. The following queries focus on some of those.

Q30 *The initial and last bidder of all open auctions*

```
/site/open_auctions/open_auction
/bidder[position()=1 and position()=last()]
```

Q31 *The open auctions having more than 5 bidders*

```
/site/open_auctions/open_auction[count(bidder)>5]
```

Q36 *The items whose description contains the word 'gold'*

```
/site/regions/*/item[contains(description,'gold')]
```

Q39 *Mails sent in September*

```
/site/regions/*/item/mailbox/mail
[substring-before(substring-after(date,'/'),'/')='09']
```

Q44 *Open auctions with a total increase greater or equal to 70*

```
/site/open_auctions/open_auction[floor(sum(bidder/increase))>=70]
```

XMark documents do not contain any comment or processing instruction. Moreover, they do no declare namespaces and language attributes. Although we have used these features in the first part of the benchmark, the corresponding queries do not give interesting insights when evaluated on XMark documents, since their answer sets are trivial. Therefore, we included in the benchmark a second document and a different set of queries in order to test these features only. For space reasons, we do not describe this part of the benchmark here and we invite the interested reader to consult the XPathMark website [4].

3 Evaluation of XML Engines

In this section we suggest how to evaluate XPathMark on a given XML engine. Moreover, by way of example, we evaluate, using XPathMark, two popular XML engines, namely Saxon [7] and Galax [8].

3.1 Evaluation Methodology

XPathMark can be checked on a set of XML processors and conclusions about the performances of the processors can be drawn. In this section, we suggest a method to do this evaluation.

We describe a set of *performance indexes* that might help the evaluation and the comparison of different XML processors that have been checked with XPathMark. We say that a query is *supported* by an engine if the engine processes the query without giving an error. A supported query is *correct* with respect to an engine if it returns the correct answer for the query. We define the *completeness index* as the number of supported queries divided by the number of benchmark queries, and the *correctness index* as the number of supported and correct queries divided by the number of supported queries. The completeness index gives an indication of how much of the benchmark language (XPath in our case) is supported by the engine, while the correctness index reveals the portion of the benchmark language that is correctly implemented by the engine.

XMark offers a document generator that generates XML documents of different sizes according to a numeric scaling factor. The document size grows linearly with respect to the scaling factor. For instance, factor 0.01 corresponds to a document of (about) 1,16 MB and factor 0.1 corresponds to a document of (about) 11,6 MB. Given an XMark document and a benchmark query, we can measure the time that the XML engine takes to evaluate the queries on the document. The *query response time* is the time taken by the engine to give the answer for the query on the document, including parsing of the document, parsing, optimization, and processing of the query, and serialization of the results. It might be interesting to evaluate the *query processing time* as well, which is the fraction of the query response time that the engine takes to process the query only, excluding the parsing of the document and the serialization of the results. We define the *query response speed* as the size of the document divided by the query response time. The measure unit is, for instance, MB/sec.

We may run the query against a documents series of documents of increasing sizes. In this case, we have a *speed sequence* for the query. The *average query response speed* is the average of the query response speeds over the document series. Moving from one document (size) to another, the engine may show either a positive or a negative *acceleration* in its response speed, or the speed may remain constant.

The concept of speed acceleration is intimately connected to that of *data scalability*. Consider two documents d_1 of size s_1 and d_2 of size s_2 in the document series with $s_1 < s_2$, and a query q. Let t_1 and t_2 be the response times for query q on documents d_1 and d_2, respectively. Let $v_1 = s_1/t_1$ be the speed of q over d_1 and $v_2 = s_2/t_2$ be the speed of q over d_2. The *data scalability factor* for query q is defined as:

$$\frac{v_1}{v_2} = \frac{t_2 \cdot s_1}{t_1 \cdot s_2}$$

If the scalability factor is lower than 1, that is $v_1 < v_2$, then we have a *positive speed acceleration* when moving from document d_1 to document d_2. In this case, we say that the scalability is *sub-linear*. If the scalability factor is higher than 1, that is $v_1 > v_2$, then we have a *negative speed acceleration* when moving from document d_1 to document d_2. In this case, we say that the scalability is *super-linear*. Finally, if the scalability factor is equal to 1, that is $v_1 = v_2$, then the speed is constant when moving from document d_1 to document d_2. In this case, we say that the scalability is *linear*. A sub-linear scalability means that the response time grows less than linearly, while a super-linear scalability means that the response time grows more than linearly. A linear scalability indicates that the response time grows linearly. For instance, if $s_2 = 2 \cdot s_1$ and $t_2 = 4 \cdot t_1$, then the scalability factor is 2 and the time grows quadratically on the considered segment.

Once again, we may run query q against series of documents of increasing sizes and generate a *data scalability sequence* for query q. The *average data scalability factor* for query q is the average of the data scalability factors for query q over the document series.

All these indexes can be computed for a single query or for an arbitrary subset of the benchmark. Of particular interest is the case when the whole benchmark is considered. Given a document d, we define the *average benchmark response time* for d as the average of the response times of all the benchmark queries on document d. Moreover, the *benchmark response speed* for d is defined as the size of d divided by the average benchmark response time. Notice that the benchmark response speed is different from the average of the response speeds for all the benchmark queries. Finally, the *data scalability factor for the benchmark* is defined as above in terms of the benchmark response speed. If we take the average of the benchmark response speed (respectively, data scalability factor for the benchmark) over a document series we get the *average benchmark response speed* (respectively, *average data scalability factor for the benchmark*). The former indicates how fast the engine processes XPath, while the latter reveals how well the engine scales-up with respect to XPath when the document size increases.

The outcomes of the evaluation for a specific XML engine should be formatted in XML. In Figure 2 we suggest a DTD for this purpose. The document root is named `benchmark`. The engine under evaluation and its version are specified as attributes of the element `benchmark`. The benchmark element has an `index` child and zero or more `query` children.

The `index` element contains the performance indexes of the engine and has attributes describing the testing environment. The testing environment contains

```
<!ELEMENT benchmark        (indexes,query*)>

<!ELEMENT indexes          (completeness,correctness,times?,speeds?,
                            scalas?,avgspeed?,avgscala?)>
<!ELEMENT completeness     (#PCDATA)>
<!ELEMENT correctness      (#PCDATA)>
<!ELEMENT times            (time+)>
<!ELEMENT speeds           (speed+)>
<!ELEMENT scalas           (scala+)>
<!ELEMENT time             (#PCDATA)>
<!ELEMENT speed            (#PCDATA)>
<!ELEMENT scala            (#PCDATA)>
<!ELEMENT avgspeed         (#PCDATA)>
<!ELEMENT avgscala         (#PCDATA)>

<!ELEMENT query            (type,description,syntax,supported,error?,
                            correct,given_answer?,expected_answer?,
                            times?,speeds?,scalas?,avgspeed?,avgscala?)>
<!ELEMENT type             (#PCDATA)>
<!ELEMENT description      (#PCDATA)>
<!ELEMENT syntax           (#PCDATA)>
<!ELEMENT supported        EMPTY>
<!ELEMENT error            (#PCDATA)>
<!ELEMENT correct          EMPTY>
<!ELEMENT given_answer     (#PCDATA)>
<!ELEMENT expected_answer  (#PCDATA)>

<!ATTLIST benchmark engine    CDATA #REQUIRED
                    version   CDATA #REQUIRED>
<!ATTLIST query     id        ID #REQUIRED>
<!ATTLIST indexes   cpu       CDATA #IMPLIED
                    memory    CDATA #IMPLIED
                    os        CDATA #IMPLIED
                    time_unit (msec | csec | dsec | sec)  #IMPLIED
                    time_type (response | processing) #IMPLIED>
<!ATTLIST supported value     (yes | no) #REQUIRED>
<!ATTLIST correct   value (yes | no | undef) #REQUIRED>
<!ATTLIST time      factor    CDATA #REQUIRED>
<!ATTLIST speed     factor    CDATA #REQUIRED>
<!ATTLIST scala     factor1   CDATA #REQUIRED
                    factor2   CDATA #REQUIRED>
```

Fig. 2. The DTD for a benchmark outcome

information about the processor (`cpu`), the main memory (`memory`), the operating system (`os`), the time unit (`time_unit`), and the time type, that is, either response or processing time (`time_type`). The performance indexes are: the completeness index (`completeness`), the correctness index (`correctness`), a sequence of average benchmark response times for a document series (`times`, a sequence of `time` elements), a sequence of benchmark response speeds for a document series (`speeds`, a sequence of `speed` elements), a sequence of data scalability factors for the benchmark for a document series (`scalas`, a sequence of `scala` elements), the average benchmark response speed (`avgspeed`), and the average data scalability factor for the benchmark (`avgscala`). Each element of type `time` and `speed` has an attribute called `factor` indicating the factor of the XMark document on which it has been computed. Moreover, each element of type `scala` has two attributes called `factor1` and `factor2` indicating the factors of the two XMark documents on which it has been computed.

Each `query` element contains information about the single query and is identified by an attribute called `id` of type ID. In particular, it includes the category of the query (`type`), a description in English (`description`), the XPath syntax (`syntax`), whether or not the query is supported by the benchmarked engine (`supported`, it must be either `yes` or `no`), the possible error message (`error`, only if the query is not supported), whether or not the query is correctly implemented by the benchmarked engine (`correct`, it must be either `yes`, `no`, or `undef`. The latter is used whenever the query is not supported), the given and expected query answers (`given_answer` and `expected_answer`. They are used for comparison only if the query is not correct), a sequence of query response times for a document series (`times`, as above), a sequence of query response speeds for a document series (`speeds`, as above), a sequence of data scalability factors for the query for a document series (`scalas`, as above), the average query response speed (`avgspeed`), and the average data scalability factor for the query (`avgscala`).

The solution of composing the results in XML format has a number of advantages. First, the outcomes are easier to extend with different evaluation parameters. More importantly, the outcomes can be queried to extract relevant information and to compute performance indexes. For instance, the following XPath query retrieves the benchmark queries that are supported but not correctly implemented:

```
/benchmark/query[supported="yes" and correct="no"]/syntax
```

Moreover, the following XQuery computes the completeness and correctness indexes:

```
let $x := doc("outcome_engine.xml")/benchmark/query
let $y := $x[supported="yes"]
let $z := $x[correct="yes"]
return <indexes>
          <completeness> {count($y) div count($x)} </completeness>
          <correctness> {count($z) div count($y)} </correctness>
       </indexes>
```

Finally, the following XQuery computes the average query response time of queries over the axes category when evaluated on the XMark document with scaling factor 0.1:

```
let $x := doc("outcome_engine.xml")/
          benchmark/query[type="axes" and correct="yes"]
let $y := sum($x/times/time[@factor="0.1"])
let $z := count($x)
return <average_time> {$y div $z} </average_time>
```

More generally, one can easily program a benchmark checker that automatically tests and evaluates different XML engines with respect to XPathMark.

3.2 Evaluating Saxon and Galax

We ran the XPathMark benchmark on two state-of-the-art XML engines, namely Saxon [7] and Galax [8]. Saxon technology is available in two versions: the basic edition Saxon-B, available as an open-source product, and the schema-aware edition Saxon-SA available on a commercial license. We tested Saxon-B 8.4, with Java 2 Platform, Standard Edition 5.0. Galax is the most popular native XQuery engine available in open-source and it is considered a reference system in the database community for its completeness and adherence to the standards. We tested version 0.5. We ran all the tests on a 3.20 GHz Intel Pentium 4 with 2GB of main memory under Linux version 2.6.9-1.667 (Red Hat 3.4.2-6.fc3). All the times are response CPU times in seconds. For each engine, we ran all the supported queries on XMark documents of increasing sizes. The document series is the following (XMark factors):

(0.001, 0.002, 0.004, 0.008, 0.016, 0.032, 0.064, 0.128, 0.256, 0.512, 1)

corresponding to the following sizes (in MB):

(0.116, 0.212, 0.468, 0.909, 1.891, 3.751, 7.303, 15.044, 29.887, 59.489, 116.517)

It is worth noticing that in the computation of the completeness index we did not consider queries using the `namespace` axis, since this axis is no more supported in XQuery [2].

The whole evaluation outcomes can be accessed from the XPathMark website [4]. This includes the outcomes in XML for both the engines and some plots illustrating the behaviour of the performance indexes we have defined in this paper. In order to compare efficiency and scalability of the two engines, we also evaluated the subset of the benchmark corresponding to the intersection of the query sets supported by the two engines (which are different). This common base is the query set {Q1-Q9,Q12,Q13,Q15-Q24,Q30-Q47} of cardinality 39. In the following we report about our findings.

1. **Completeness and Correctness.** The completeness and the correctness indexes for Saxon are both 1, meaning that Saxon supports all the queries in the benchmark (excluding queries using the `namespace` axis, which are not counted) and all supported queries give the correct answer. The completeness index for Galax is 0.85. In particular, the axes `following` and `preceding` (which are in fact optional in XQuery) and the `id()` function

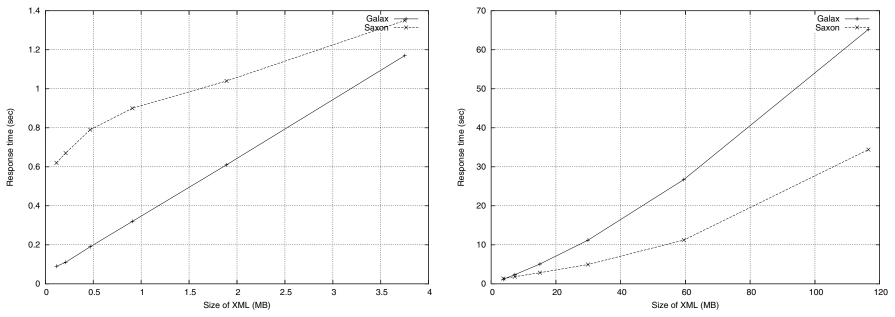


Fig. 3. Average benchmark response times

are not supported by Galax. However, all the supported queries give correct answers, hence the correctness index for Galax is 1.

2. **Efficiency.** On the common query set, the average benchmark response speed for Saxon is 2.80 MB/sec and that for Galax is 2.50 MB/sec. This indicates that Saxon is faster than Galax to process the (checked subset of the) benchmark. The average response time for a query in the benchmark, varying the document size, is depicted in Figure 3 (left side is from factor 0.001 to factor 0.032 and right side is from factor 0.032 to factor 1). Interestingly enough, Galax outperforms Saxon in the first track, corresponding to small documents (up to 3.7 MB), but Saxon catches up in the second track, corresponding to bigger documents. This trend is confirmed by the behaviour of the benchmark response speeds (see Figure 4 corresponding to the same segments of the document series).
3. **Scalability.** On the common query set, the average data scalability factor for the checked benchmark is 0.80 in the case of Saxon and it is 0.98 in the case of Galax. This indicates that Saxon scalas-up better than Galax as the size of the XML document increases. Figure 5 compares the data scalability factors for the two engines. Notice that Saxon's scalability is sub-linear up to XMark factor 0.256 (29.9 MB), and it is super-linear for bigger files. Galax's scalability is sub-linear up to XMark factor 0.032 (3.7 MB), and it is super-linear for bigger documents. This trend is confirmed by the behaviour of the benchmark response speeds (Figure 4). In particular, notice that Saxon's response speed increases (with a decreasing derivative) up to XMark factor 0.256, and then it decreases, while Galax has a positive acceleration up to XMark factor 0.032, and then the acceleration becomes negative. From this analysis, we conclude that, under our testing environment, Saxon is well performing up to a *break point* corresponding to an XML documents of size 29.9 MB, while the break point for Galax corresponds to a smaller file of 3.7 MB.

Finally, Figures 6 and 7 depict, for each query, the average response speeds and the average data scalability factors over the document series. Interestingly, the *qualitative* behaviour of the response speeds is the same for both the engines,

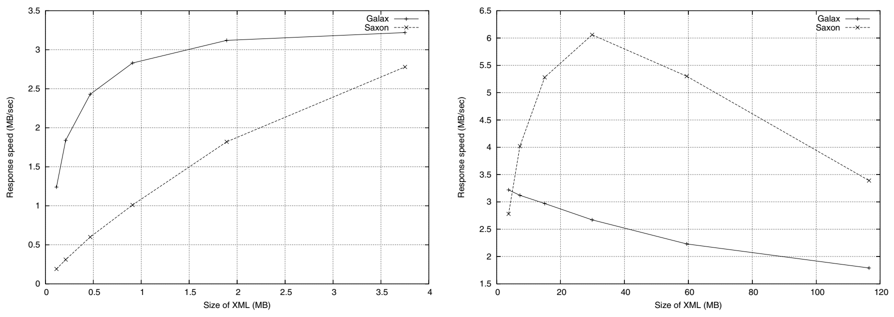


Fig. 4. Benchmark response speeds

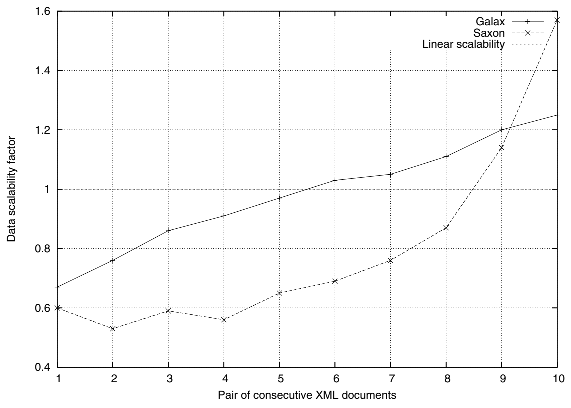


Fig. 5. Data scalability factors for the benchmark

with Saxon outperforming Galax in all the queries but Q35 (The elements written in Italian language: `//*[lang('it')]`). This might indicate that the two engines implement a similar algorithm to evaluate XPath queries. The data scalability factor for Galax is almost constant for all the queries, and it is less but close to linear scalability. The data scalability factor for Saxon is less stable. It is far below linear scalability for all the queries but the problematic Q35. In particular, the scalability factor for Q35 is higher than 3 in the last segment of the document series, indicating that the response time for Q35 grows more then quadratically (probably Saxon doesn't understand Italian very well!). Notice that Q35 is not problematic in Galax.

4 Future Work

We intend to improve XPathMark in different directions by: (i) enlarging the benchmark query set. In particular, we are developing a benchmark to test *query scalability*, that is the ability of an XML engine to process queries of increas-

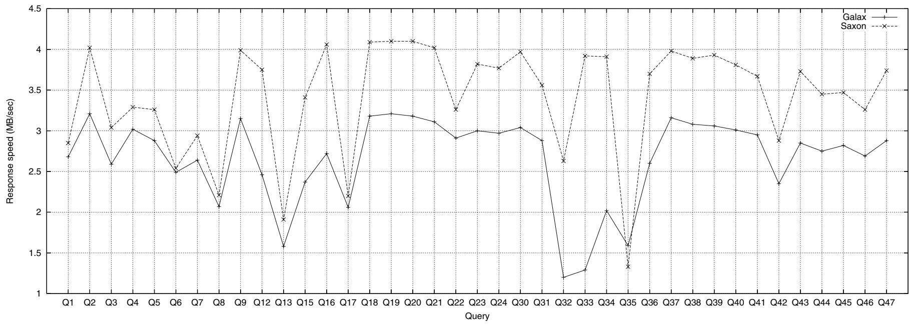


Fig. 6. Average query response speeds

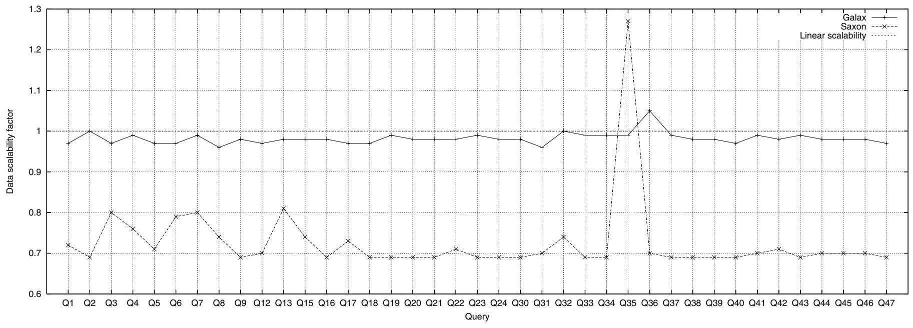


Fig. 7. Average data scalability factors for queries

ing lengths; (ii) studying different performance indexes to better evaluate and compare XML engines; (iii) implementing a *benchmark checker* in order to automatically compare the performance of different query processors with respect to XPathMark.

XPathMark can also be regarded as a benchmark for testing the navigational fragment of the XQuery language in isolation. Indeed, XQuery crucially uses XPath to navigate XML trees, saving the retrieved node sequences into variables that may be further elaborated by, e.g., joining, sorting, and filtering. In this respect, XPathMark can be considered as a fragment of a new version of the XMark benchmark or as a part of a bigger benchmark evaluation project for XQuery (e.g., the micro-benchmark repository for XQuery proposed in [10]).

References

1. Schmidt, A.R., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: XMark: A Benchmark for XML Data Management. In: Proceedings of the International Conference on Very Large Data Bases (VLDB). (2002) 974–985 `http://monetdb.cwi.nl/xml/`.

2. World Wide Web Consortium: XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery` (2005)
3. World Wide Web Consortium: XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath` (1999)
4. M. Franceschet: XPathMark: An XPath benchmark for XMark. `http://www.science.uva.nl/~francesc/xpathmark` (2005)
5. World Wide Web Consortium: XSL Transformations (XSLT). `http://www.w3.org/TR/xslt` (1999)
6. World Wide Web Consortium: XML Pointer Language (XPointer). `http://www.w3.org/TR/xptr` (2002)
7. Kay, M.H.: Saxon. An XSLT and XQuery processor. `http://saxon.sourceforge.net` (2005)
8. Fernández, M., Siméon, J., Chen, C., Choi, B., Gapeyev, V., Marian, A., Michiels, P., Onose, N., Petkanics, D., Ré, C., Stark, M., Sur, G., Vyas, A., Wadler, P.: Galax. The XQuery implementation for discriminating hackers. `http://www.galaxquery.org` (2005)
9. Harold, E.R., Means, W.S.: XML in a Nutshell. 3rd edn. O'Reilly (2004)
10. Afanasiev, L., Manolescu, I., Michiels, P.: MemBeR: a micro-benchmark repository for XQuery project description. In: Proceedings of the International XML Database Symposium (XSym). (2005)

MemBeR: A Micro-benchmark Repository for XQuery

Loredana Afanasiev[1,⋆], Ioana Manolescu[2,⋆⋆], and Philippe Michiels[3,⋆⋆⋆]

[1] University of Amsterdam, The Netherlands
lafanasi@science.uva.nl
[2] INRIA Futurs & LRI, France
ioana.manolescu@inria.fr
[3] University of Antwerp, Belgium
philippe.michiels@uia.ua.ac.be

Abstract. XQuery is a feature-rich language with complex semantics. This makes it hard to come up with a benchmark suite which covers all performance-critical features of the language, and at the same time allows one to individually validate XQuery evaluation techniques. This paper presents MemBeR, a *micro-benchmark repository*, allowing the evaluation of an XQuery implementation with respect to precise evaluation techniques. We take the view that a fixed set of queries is probably insufficient to allow testing for various performance aspects, thus, the users of the repository must be able to add new data sets and/or queries for specific performance assessment tasks. We present our methodology for constructing the micro-benchmark repository, and illustrate with some sample micro-benchmarks.

1 Introduction

The development of XML query engines is currently being held back by a lack of systematic tools and methodology for evaluating algorithms and optimization techniques. The essential role of benchmark tools in the development of XML query engines, or any type of data management systems for that matter, is well established. Benchmarks allow one to assess a system's capabilities and help determine its strengths or potential bottlenecks.

Since the introduction of XML query languages like XPath 1.0 [6], XPath 2.0 [9] and XQuery [3], many benchmark suites have been developed. Most of them, including XMark [24], XMach-1 [4], X007 [5] and XBench [25], fall into the category of *application benchmarks*. Application benchmarks are used to evaluate the overall performance of a database system by testing as many query

⋆ Loredana Afanasiev is supported by the Netherlands Organization for Science and Research (NWO), grant number 017.001.190.

⋆⋆ Ioana Manolescu is partly supported by the ACIMD Tralala (*Tra*nsformations, *l*ogics *a*nd *la*nguages for XML).

⋆⋆⋆ Philippe Michiels is supported by IWT – Institute for the Encouragement of Innovation by Science and Technology Flanders, grant nr. 31016.

language features as possible, using only a limited set of queries. The XML data sets and the queries are typically chosen to reflect a particular user scenario. The influence of different system components on their overall performance is, however, difficult to analyze from performance figures collected for complex queries. The evaluation of such queries routinely combines a few dozen execution and optimization techniques, which in turn depend on numerous parameters. Nevertheless, due to a lack of better tools, application benchmarks have been used for tasks they are not suited for, such as the assessment of a particular XML join or optimization technique.

Micro-benchmarks, as opposed to application benchmarks, test individual performance-critical features of the language, allowing database researchers to evaluate their query evaluation technique (e.g. query optimization, storage and indexing schemes etc.) in isolation. Micro-benchmarks provide better insight in how XQuery implementations address specific performance problems. They allow developers to compare performance with and without the technique being tested, while reducing to a minimum the interaction with other techniques or implementation issues.

To the authors' knowledge, the Michigan benchmark [23] is the only existing micro-benchmark suite for XML. It proposes a large set of queries, allowing one to assess an engine's performance for a variety of elementary operations. These queries are used on a parametrically generated XML data set. However, this micro-benchmark suffers from some restrictions. For instance, the maximum document depth is fixed in advance to 16, and there are only two distinct element names. Furthermore, the query classes identified in [23] are closely connected to a particular evaluation strategy[1]. The queries of [23] are very comprehensive on some aspects, such as downward XPath navigation and ignore others, such as other XPath axes, or complex element creation.

The Need for Micro-benchmarks and Associated Methodology. The first problem we are facing is the lack of micro-benchmarks allowing system designers and researchers to get *precise and comprehensive* evaluations of XML query processing systems and prototypes. An evaluation is *precise* if it allows one to study language features *in isolation*, to understand which parameters impact a system's behavior on that feature, without "noise" in the experimental results due to processing other features. The need for precision, which is a general aspect of scientific experiments, leads to the choice of micro-benchmarks, one for each feature to study. For an evaluation to be *comprehensive*, several conditions must be met. For every interesting feature, there must be a micro-benchmark allowing to test that feature. When studying a given feature, *all* parameters which may impact the system's behavior in the presence of that feature must be described, and *it must be possible to vary their values in a controlled way.* Finally, *all interesting aspects of the system's behavior during a given measure must be documented.* For instance, a performance micro-benchmark should be

[1] Evaluating path expressions by an n-way structural join, where n is the total number of path steps. The query classification criteria are no longer appropriate e.g., if a structural index is used.

accompanied by information regarding the memory or disk consumption of that measure.

Second, a *micro-benchmark user methodology* is needed, explaining how to choose appropriate micro-benchmarks for a given evaluation, why the micro-benchmark parameters are likely to be important, and how to choose value combinations for these parameters. Such a methodology will bring many benefits. It will ease the task of assessing a new implementation technique. It will facilitate comprehension of system behavior, and reduce the time to write down and disseminate research results. And, it will ease the task of reviewers assessing "Experiments" sections of XML querying papers, and help unify the authors' standards with the reviewers' expectations.

Our Approach: Micro-benchmark Repository and Methodology. We are currently building a repository of micro-benchmarks for XQuery and its fragments, which will provide for precise and comprehensive evaluation. We endow micro-benchmarks with precise guidelines, easing their usage and reducing the risks of mis-use.

Given the wide range of interesting XQuery features, and the presence of interesting ongoing developments (such as extensions for text search [7], and for XML updates [8]), a fixed set of micro-benchmarks devised today is unlikely to be sufficient forever. Thus, we intend to develop our repository as a *continuing, open-ended community effort*:

- users can contribute to the repository by *creating*, or *enhancing* an existing micro-benchmark;
- additions to the repository will be subject to *peer review* from the other contributors, checking if the feature targeted by the addition was not already covered and if the addition adheres to the micro-benchmark principles, and ensuring the quality of the benchmark's presentation.

Such a repository will allow consolidating the experience of many individual researchers having spent time and effort in "carving out" micro-queries from existing application benchmarks unfit for the task. It will be open to the addition of new performance challenges, coming from applications and architectures perhaps not yet available today. This way, the micro-benchmarks will be continuously improved, and, we hope, widely used in the XML data management community. The repository is hosted on the Web and freely accessible. This should further simplify the task of setting up experimental studies, given that individual micro-benchmarks will be available at specific URLs.

This paper describes our approach. Section 2 formalizes the concepts behind the micro-benchmarks repository, and Section 4 illustrates them with several examples. Section 3 outlines a preliminary micro-benchmark classification and delves into more details of performance-oriented micro-benchmarks. Section 5 describes the test documents in the repository. Section 6 briefly describes the repository's Web interface, and concludes with some perspectives.

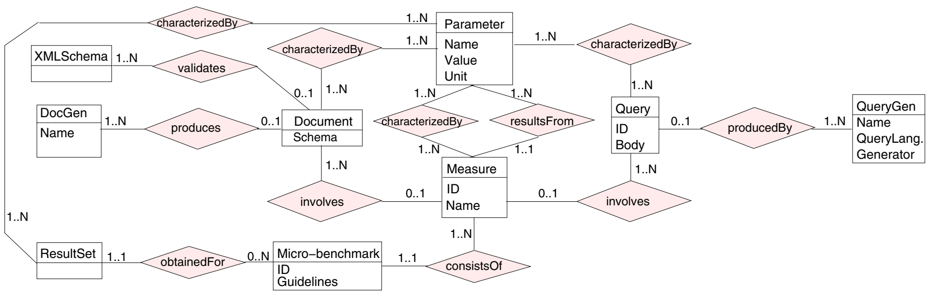


Fig. 1. Entity-Relationship diagram of the micro-benchmark repository contents.

2 Our Approach: A Micro-benchmark Repository

Our goal is *to build a repository of micro-benchmarks for studying the performance, resource consumption, correctness and completeness of XQuery implementations techniques.*

Performance: how well does the system perform, e.g., in terms of completion time, or query throughput? The primary advantage of a data management system, when compared with an ad-hoc solution, should be its efficiency.

Resource consumption: performance should be naturally evaluated against the system's resource needs, such as the size of a disk-resident XML store, with or without associated indexes, or the maximum memory needs of a streaming system.

Completeness: are all relevant language features supported by the system? Some aspects of XQuery, such as its type system, or its functional character, have been perceived as complex. Correspondingly, many sub-dialects have been carved out [16, 20, 22]. Implementations aiming at completeness could use a yardstick to compare against.

Correctness: does the output of the system comply with the query language specifications? For a complex query language such as XQuery, and even its fragments, correctness is also a valid target of benchmarking.

In this paper we will mainly focus on *performance and resource consumption micro-benchmarks*. Nevertheless, we stress the importance of correctness for interpreting performance results. In devising correctness and completeness benchmarks, we expect to draw inspiration from the use cases and examples used in the W3C XQuery specifications and from existing benchmarks, like [12].

We intend our benchmark repository mainly for system designers, to help them analyze and optimize their system.

Micro-benchmarking Design Principles. We adopt the following design principles:

There should be a minimal number of micro-benchmarks for every language feature, and we will usually strive to keep this number to 1. However, if a new example provides a very different perspective, and the current ones cannot be extended to simulate it, then it will be included in the repository.

A micro-benchmark should explicitly list, and provide value ranges for all data, query and resource parameters which may impact the results. This is a crucial condition for benchmark results to be reproducible, interpretable, and trustworthy. In this way a micro-benchmark will contain well-documented and thorough measures.

The above implies that for any micro-benchmark measure, and any data parameter likely to impact the measure's result, *at least one data set can be constructed by controlling the value of that parameter in its interesting range.* This will have an impact on our choice of data sets (see Section 5).

A micro-benchmark should reduce to a minimum the influence of all but the tested language feature. Thus, if the purpose is to test path expressions navigating downward, the queries should not use sibling navigation, and vice versa. An important consequence is the following. The presence of an XML Schema for the input document enables a large number of optimizations, at the level of an XML store, XML index, XQuery rewriting and optimization, automata-based execution etc. *In any micro-benchmark measure where the focus is not on schema-driven optimizations, one should use documents without a schema.* Otherwise, schema-driven optimizations might effect the system's performance in a non-transparent manner and make results uninterpretable.

Measure (also) individual query processing steps. To get a precise evaluation, often is needed to measure individual processing steps, such as: query normalization, query rewriting, query optimization, data access, (structural) join processing, output construction etc. For instance, XPath micro-benchmarks may measure the time to *locate* the elements which must be returned (this often means finding their IDs). Measuring such processing steps requires hooks into the execution engine. We consider it worth the trouble, as even if queries are chosen with care, query execution times may still reflect the impact of too many factors.

A micro-benchmark should be extensible. A micro-benchmark should aim to remain useful even when systems will achieve much higher performance. The parameters should therefore allow for a wide enough range. The micro-benchmarks should also be regularly updated to reflect new performance standards.

From these principles, we derive the following micro-benchmark repository structure (Fig. 1):

- *XML documents.* A document may have an XML Schema or not. It is characterized by a number of *parameters*, which we model as name-value pairs. Benchmarks typically benefit from using collections of documents similar in some aspects, but characterized by different parameters. For synthetic data sets, a *document generator* is also provided.
- *Queries.* Each query aims at testing exactly one feature. Queries can also be characterized by *parameters*, such as: number of steps in a path expression query, numbers of query nesting levels, selectivity of a value selection predicate etc. Similar queries make up a *query set*, for which a *query generator* is provided.

- *Measures*: a measure is one individual experiment, from which experimental results is gathered. We model an experimental result also as a parameter, having a name and a value. A measure may involve a document, or none; XML fragments are legal XQuery expressions, and thus an XML query may carry "its own data" (see micro-benchmark μB_4 in Section 4). A measure may involve zero, one, or more queries; the latter case is reserved to multi-query scenarios, such as, for instance, XML publish/subscribe. Intuitively, one measure yields "one point on a curve in a graph". We will provide examples shortly.
- *Micro-benchmarks.* A micro-benchmark is a collection of measures, studying a given (performance, consumption, correctness or completeness) aspect of a XML data management and querying. Intuitively, a micro-benchmark yields a set of points, which can be presented as "one or several curves or graphs", depending on how many parameters vary, in the documents and queries considered. A micro-benchmark includes *guidelines*, explaining which data and/or query parameters may impact the results and why, and suggesting ranges for these parameters. Measure methodologies may also specify the scenario(s) for which the measure is proposed, including (but not limited to): persistent database scenario, streaming query evaluation, publish/subscribe etc.
- *Micro-benchmark result sets*, contributed by users. A result set consists of a set of points corresponding to each of the micro-benchmark's prescribed measures, and of a set of parameters characterizing the measure enactment. Commonly used parameters describe hardware and software configurations. Other important parameters are the technique(s) and optimization(s) employed. For instance, a result set may specify the particular XML index used, or the fact that the query automaton was lazily constructed etc.

Micro-benchmark Usage Methodology. Even a carefully designed (micro-) benchmark can be misused. As an attempt to limit this, we require micro-benchmark results to adhere to the following usage principles:

Always declare the language and/or dialect supported by the system, even for features not used by the micro-benchmark. Many efficient evaluation techniques are conditioned by some underlying language simplifications, such as: unordered semantics, simplified atomic types set, unsupported navigation axes, unsupported typing mechanism etc. Such simplifications, if any, should be clearly stated next to performance figures. In relational query processing research, the precise SQL or Datalog dialect considered is always clearly stated. XQuery not being simpler than SQL, at least the same level of precision is needed.

Micro-benchmark results which vary less parameters than specified by the micro-benchmark are only meaningful if they are accompanied by a short explanation as to why the simplifications are justified. Omitting this explanation ruins the effort spent in identifying useful parameters, and compromises the comprehensive aspect of the evaluation.

Extra values for a parameter may always be used in results. Range changes or restrictions should be justified. In some special situations, new parameter values may be needed. In such cases, a revision of the micro-benchmark should be considered.

When presenting micro-benchmark results, parameters should vary one at a time, while keeping the other parameters constant. This will typically yield a family of curves where the varying parameter values are on the x axis, and the measure result on the y axis. For space reasons, some curves may be omitted from the presentation. In this case, however, *the measure points for all end-of-range parameter values should be provided*. Trying the measure with these values may give the system designer early feedback, by exposing possible system shortcomings. And, when performance is robust on such values, a convincing case has been made for the system's performance.

3 Preliminary Taxonomy of Measures and Micro-benchmarks

In this section, we outline a general classification of measures (and thus, of micro-benchmarks). This classification guides a user looking for a specific micro-benchmark, and serves as a road map for our ongoing micro-benchmark design work.

A first micro-benchmark classification criterion, introduced in Section 2, distinguishes between *performance*, *consumption*, *correctness* and *completeness* benchmarks.

We furthermore classify micro-benchmarks according to the following other criteria:

- The result metric: it may be *execution time*, *query normalization or optimization time*, *query throughput*, *memory occupancy*, *disk occupancy* etc. It may also be a simple boolean value, in the case of correctness measures.
- Benchmarks may test *data scalability* (fixed query on increasingly larger documents) or *query scalability* (increasing-size queries on fixed documents).
- Whether or not a micro-benchmark uses an *XMLSchema*, and the particular schema used.
- The *query processing scenarios* to which a micro-benchmark applies, such as: persistent database (store the document once, query it many times), streaming (process the query in a single pass over the document), or programming language-based (the document is manipulated as an object in some programming language).
- The *query language* and perhaps dialect which must be supported in order to run the micro-benchmark.
- The *language feature being tested* in a micro-benchmark is a precise classification criteria. We strive to provide exactly one micro-benchmark for each interesting feature.

The next section contains several micro-benchmark examples together with their classification and methodology.

4 Examples of Micro-benchmarks for XPath and XQuery

We start by a very simple micro-benchmark, involving a basic XPath operation.

Example 1 (Micro-benchmark μB_1: simple node location). In a persistent XML database scenario, we are interested in the time needed to locate elements of a given tag in a stored document. We study this time on increasingly large and complex documents (data scalability measure). We are not concerned with schema optimizations, thus, we will use schema-less documents. We measure the time to *locate* the elements, not to *return* their subtrees.

Let Q_1 be the query `//a`$_1$. Let n, t, d and f be some positive integers, and p be a real number between 0 and 1. Let $D_1(n, t, d, f, p)$ be a document whose n elements are labeled `a1`, `a2`, ..., `a`t, having the depth d, the fan-out (maximum number of children of an element) f, and such that exactly $p * n$ elements are labeled `a1`. Elements may nest freely, that is, the parent of an element labeled `a`i can have any `a`j label, $1 \leq i, j \leq t$.

The measure $M_1(h, b)$ involves Q_1 and a document $D_1(n, t, d, f, p)$, for some n, t, d, $f \in \mathbb{N}$ and $p \in [0, 1]$. The parameter h may take values in $\{true, false\}$ and specifies whether the measure is taken with a hot cache ($h = true$) or with a cold cache ($h = false$). The parameter b is the size of the memory buffer in KB. For any pair of (h, b) values, $M_1(h, b)$ results in an execution time for locating the persistent identifiers of the elements in Q_1's result, in document $D_1(n, t, d, f, p)$. A M_1 measure is characterized by a (h, b, n, t, d, f, p) tuple.

The micro-benchmark μB_1 consists of applying M_1 on a subset of $\{true, false\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times [0, 1]$, such as, for instance:

$h \in \{true, false\}$	$b \in \{50, 200, 1000\}$	$n \in \{10^6, 10^9, 10^{12}\}$	$d \in \{10, 20, 50\}$
$f \in \{2, 10, 20\}$	$t \in \{1, 5, 20\}$	$p \in \{0.001, 0.01, 0.1, 0.5, 0.8, 1.0\}$	

Parameters n and p impact the number of retrieved elements. Parameter t determines the document's structural complexity, which may impact performance if the evaluation relies on a path or structural index. Together, d and f determine whether the document is deep or shallow; this may impact performance for strategies based on navigation or structural pattern matching. Such strategies are also affected by n, even when $n * p$ is constant. Varying h and b allows to capture the impact of the system's memory cache and memory buffer size on the evaluation times.

μB_1 results in several thousand points, all of which are not needed in all circumstances. If the system implements M_1 by a lookup in a tag index as in [17], d, f and t may make little difference, thus μB_1 may be reduced to only 60 individual measures. As another example, if M_1 is implemented by CAML-based pattern matching as in [2], then all evaluation takes place in memory and b is irrelevant.

Example 2 (Micro-benchmark μB_2: simple node location in the presence of a schema). Important performance gains can be realized on a query like Q_1 if schema information is available. μB_2 aims at quantifying such gains. Let $D_{2,l}(n, t, f, d, p, \Sigma_l)$ be a document satisfying the same conditions as $D_1(n, t, f, d, p)$ from μB_1, but furthermore valid with respect to an XML Schema Σ_l, specifying that `a1` elements can only occur as children of elements labeled `a1, a2, ... a`l. Σ_l prescribes a maximum number of f children for any element, and does not constrain the structure of $D_{2,l}$ in any other way.

Measure $M_2(n, t, f, d, p, l)$ records the running time of query Q_1 on $D_{2,l}(n, t, f, d, p, \Sigma_l)$. We choose not to include h and b here, since μB_2 is only concerned with the impact of the schema, orthogonal to cache and buffer concerns.

The micro-benchmark μB_2 consists of running M_2 for some fixed n, t, f, d and p, and for all $l = 1, 2, \ldots, t$. A set of suggested values is: $n = 10^6$, $t = 15$, $f = 10$, $d = 10$, $p = 1/t$.

An efficient system would use Σ_l to narrow the search scope for `a1` elements. This can take many forms. A fragmented storage, or a structural index, may make a distinction between elements `a1`, ..., `a`l and the others, making it easier to locate `a1`s. A streaming processor may skip over subtrees rooted in `a`$(l+1)$, ..., `a`t elements etc.

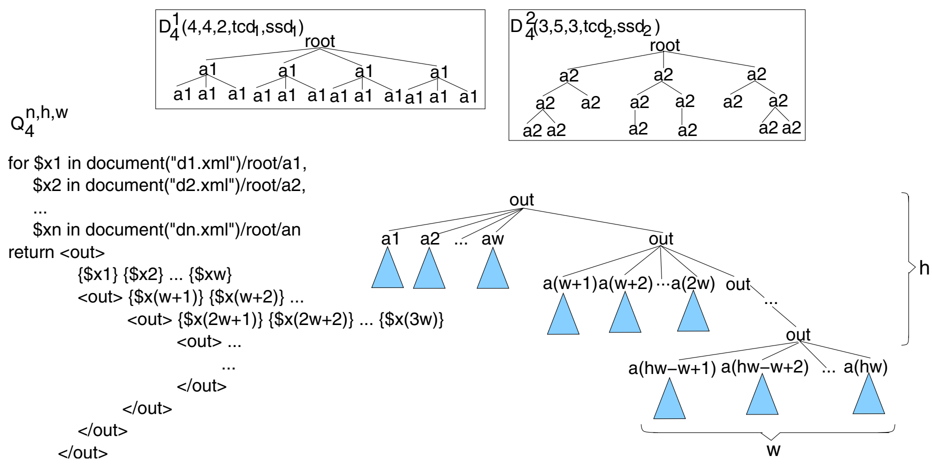
Example 3 (Micro-benchmark μB_3: returning subtrees, no schema). This micro-benchmark is meant for the persistent database scenario. It captures the performance of sending to the output sub-trees from the original document. This operation, also known as serialization, or reconstruction, is challenging in many respects: systems whose storage is fragmented (e.g. over several relational tables) will have to make an effort to reconstruct the input; systems using a persistent tree will have to follow disk-based pointers, thus they depend on the quality of node clustering etc.

We aim at measuring the cost of reconstruction alone, not the cumulated cost of locating some nodes and then reconstructing them. Thus, we choose the query Q_2: `/*`, and will apply it on the root of some input document.

Document depth, fan-out, and size, all impact reconstruction performance. Furthermore, document leaves (in a schema-less context, interpreted as strings) also may have an impact. Some storage models separate all strings from their parents, others inline them, others inline only short strings and separate longer ones etc.

We vary string size according to a normal distribution $ssd(sa, sv)$, of average sa and variance sv. We vary the number of text children of a node according to another (normal) distribution $tcd(ca, cv)$ of average ca and variance cv. Let $D_3(n, t, d, f, tcd, ssd)$ be an XML document having n elements labeled `a1`, `a2`, ... `a`t, of depth d and fanout f, such that the number of text children of given element is obtained from tcd, and the number of characters in each individual text child is given by ssd. As in D_1, elements nest arbitrarily. The actual string content does not matter, and is made of randomly chosen characters.

Measure $M_3(n, t, d, f, sa, sv, ca, cv)$ consists of measuring the execution time of Q_2 on $D_3(n, t, d, f, tcd(ca, cv), ssd(sa, sv))$. Micro-benchmark μB_3 consists of M_3 measures for:


$D_4^1(4,4,2,tcd_1,ssd_1)$
root
a1 a1 a1 a1
a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1 a1
$D_4^2(3,5,3,tcd_2,ssd_2)$
root
a2 a2 a2
a2 a2 a2 a2 a2 a2
a2 a2 a2 a2 a2 a2
$Q_4^{n,h,w}$
out
a1 a2 ... aw
out
a(w+1) a(w+2) ... a(2w) out
...
out
a(hw−w+1) a(hw−w+2) ... a(hw)
h
w

```
for $x1 in document("d1.xml")/root/a1,
    $x2 in document("d2.xml")/root/a2,
    ...
    $xn in document("dn.xml")/root/an
return <out>
          {$x1} {$x2} ... {$xw}
          <out> {$x(w+1)} {$x(w+2)} ...
                <out> {$x(2w+1)} {$x(2w+2)} ... {$x(3w)}
                      <out> ...
                            ...
                      </out>
                </out>
          </out>
       </out>
```

Fig. 2. Documents and queries involved in the micro-benchmark μB_5.

$n \in \{10^3, 10^6, 10^9\}$	$t \in \{2, 20\}$	$d \in \{5, 20, 50\}$	$f \in \{5, 10\}$
$(ca, cv) \in \{(2,1), (5,2), (10,5)\}$		$(sa, sv) \in \{(5,2), (100,50), (1000,500)\}$	

Example 4 (Micro-benchmark μB_4: XPath duplicate elimination). This micro-benchmark is taken from [15]. The purpose is to assess the processor's performance in the presence of potential duplicates. Let Q_3^i be the query:

`<a><b/><b/></a>/b/parent::a/b/parent::a ... /b/parent::a`

where the sequence of steps `/b/parent::a` is repeated i times. Any Q_3^i returns exactly one `a` element. An evaluation following directly the XPath [9] specification requires eliminating duplicates after every path step, which may hurt performance. Alternatively, duplicates may be eliminated only once, after evaluating all path expression steps. However, in the case of Q^i, this entails building intermediary results of increasing size: 2 after evaluating `/b/parent::a`, 2^2 after evaluating `/b/parent::a/b/parent::a` etc., up to 2^i before the final duplicate elimination. Large intermediary results may eat up available memory and hurt performance.

$M_4(i)$ measures the running time of Q_3^i; it does not need a document. The micro-benchmark μB_4 consists of the measures $M_4(i)$, for $i \in \{1, 5, 10, 20\}$.

Example 5 (Micro-benchmark μB_5: element creation). This micro-benchmark targets the performance of new element construction, an important feature in XQuery.

The size, depth, and fanout of the copied input subtrees, as well as the text nodes therein, clearly impact performance. Another important factor is the complexity of the constructed XML output. Thus, we consider a set of documents $D_4^i(fr_i, n_i, d_i, tcd_i, ssd_i)$, as follows:

- D_4^i's root element is labeled `root`. All other elements in D_4^i are labeled `a`i.
- The root of any D_4^i document has exactly fr_i children elements.

- Any sub-tree rooted in an `a`i child of the root is a tree of depth d_i consisting of n_i elements.
- Root elements do not have text children. The number, and size, of text children of `a`i elements is dictated by the text child distribution tcd_i and the text length distribution ssd_i. These are normal distributions, as in the micro-benchmark μB_3 previously described.

For instance, $D_4^1(4, 4, 2, tcd_1, ssd_1)$ and $D_4^2(3, 5, 3, tcd_2, ssd_2)$ are outlined in the upper part of Fig. 2, for some unspecified distributions. Text nodes are omitted for readability.

To vary the complexity of result construction, we consider a family of queries $Q_4^{n,h,w}$, where: n is the number of input documents from which subtrees are extracted, h is the "stacking height" of such sub-trees in the output, and w is the "stitching width" of the sub-trees in the output, where $h * w$ must be at most n. This is better understood by looking at $Q_4^{n,h,w}$, pictured in the lower part of Fig. 2, together with the outline of an element it produces. The shaded triangular shapes represent (deep copies of) the subtrees rooted in the `a`i input elements. This query uses very simple navigation paths in the `for` clause, in order to minimize the noise introduced in the measure by the effort spent in locating these subtrees. Two sample instances of $Q_4^{n,h,w}$ are shown in Fig. 3.

Measure M_5 records the execution time of query $Q_4^{n,h,w}$ for some n, h and w values, on n input documents D_4^1, D_4^2, …, D_4^n. M_5 is characterized by numerous input parameters, including those describing each D_4^i.

Choosing the recommended parameters ranges for μB_5 without blowing up the number of measures is quite delicate. Without loss of generality, we restrict ourselves to the distributions:

- tcd_{low} with average 2 and variance 1, ssd_{low} with average 10 and variance 5; these correspond to a mostly data-centric document.
- tcd_{high} with average 10 and variance 2, ssd_{high} with average 1000 and variance 200; these correspond to a text-rich document.

To measure scale up with w, the following set of parameter ranges are recommended:

$n \in \{1, 2, 5, 10, 20\}$ $\quad w = n \quad h = 1$
$fr_1 = 10^5$, choose a combination of fr_2, …, fr_n values such that $\Pi_{i=1}^{n} fr_i = 10^8$
For any n and $i \in \{1, \ldots, n\}$, $(tcd_i, ssd_i) = (tcd_{high}, ssd_{high})$, $d_i = 5$, and $n_i = 100$

In the above, the values of related parameters, such as fr_i, are part of the micro-benchmark specification. To measure scale up with h, set $w = 1$, $h = n$, and proceed as above.

To measure scale up with the output size, set $n = 12$, $w = 4$, $h = 3$, set all distributions, first, to (tcd_{high}, ssd_{high}) and second, to (tcd_{low}, ssd_{low}), choose some fr_i such that $\Pi_{i=1}^{n} fr_i = 10^6$, and let each n_i take values in $\{1, 10, 25\}$. An output tree will thus contain between $12+3 = 15$ and $12*25+3 = 303$ elements.

All these five micro-benchmarks are performance-oriented. μB_1 and μB_2 measure node location time, while μB_3, μB_4 and μB_5 measure query execution

$Q_4^{3,1,3}$

```
for $x1 in document(''d1.xml'')/root/a1
    $x2 in document(''d2.xml'')/root/a2
    $x3 in document(''d3.xml'')/root/a3
return <out>
        {$x1} {$x2} {$x3}
       </out>
```

$Q_4^{3,3,1}$

```
for $x1 in document(''d1.xml'')/root/a1
    $x2 in document(''d2.xml'')/root/a2
    $x3 in document(''d3.xml'')/root/a3
return <out>
        {$x1} <out>
              {$x2} <out> {$x3} </out>
             </out>
       </out>
```

Fig. 3. Sample queries from the micro-benchmark μB_5.

Table 1. XPath performance micro-benchmarks (not an exhaustive list).

μB_5	*(dq)*	Simple linear path expressions of the form `/a1/a2/.../a`k.
μB_6	*(dq)*	Path expressions of the form `//a1//a2//.../a`k.
μB_7	*(dq)*	Path expressions of the form `/a1/*/*/.../a2`, where the number of `*` varies.
μB_8	*(d)*	For each XPath axis [9], one path expression including a step on that axis. Interestingly, some node labeling schemes such as ORDPATH [21] allow "navigating" along several axes, such as parent, but also child, preceding-sibling etc. directly on the element label.
μB_9	*(dq)*	Path expressions with a selection predicate, of the form `/a1/a2/.../a`k`[text()=`c`]`, where c is some constant.
μB_{10}	*(dq)*	Path expressions with inequality comparisons.
μB_{11}	*(d)*	Path expressions with positional predicates of the form `/a1[`n`]`, where $n \in \mathbb{N}$.
μB_{12}	*(d)*	Path expressions with positional predicates, such as `/a1[position()=last()]`.
μB_{13}	*(dq)*	Path expressions with increasingly many branches, of the form `//a1[`$p1$`]//...//ak[`pk`]`, where each pi is a simple path expression.
μB_{14}	*(dq)*	Path expressions involving several positional predicates, of the form `//a1[`$n1$`]//...//ak[`nk`]`, where each $ni \in \mathbb{N}$.
μB_{15}	*(d)*	Aggregates such as `count`, `sum` etc. over the result of path expressions.

time. μB_1, μB_2, μB_3 and μB_4 require downward XPath, while μB_4 requires XQuery node creation. Only μB_3 uses a schema. μB_1, μB_2 and μB_3 test data scalability; μB_4 tests query scalability.

In Table 1 and Table 2 we outline some other interesting XPath and XQuery performance-oriented micro-benchmarks; the list is clearly not exhaustive. We mark by *(d)* and *(q)* micro-benchmarks where data scalability (respectively query scalability) should be tested.

Other XQuery micro-benchmarks should target feature such as: explicit sorting; recursive functions; atomic value type conversions implied by comparison predicates; explicit type conversion to and from complex types; repeated subexpressions etc. Two ongoing XQuery extension directions will require specific micro-benchmarks: *full-text search* [7], and *updates* [8].

5 Data Sets for the Micro-benchmark Repository

Precise performance evaluation requires carefully characterizing the test documents. Some document characteristics were already considered in [23, 24]:

Q_5^k

```
for $x1 in document(''doc.xml'')//a1
    $x2 in $x1//a2
    ...
    $xk in $x1//ak
return {$x1, $x2, ... $xk}
```

Q_6^k

```
for $x in document(''doc.xml'')//a1
return <res>
         {$x//a1} {$x//a2} ... {$x//ak}
       </res>
```

Q_7^k

```
for $x1 in document(''doc.xml'')//a1
    $x2 in document(''doc.xml'')//a2
where $x1/text() θ $x2/text()
return {$x1, $x2}
```

Fig. 4. Queries involved in the micro-benchmark μB_{16}, μB_{17} and μB_{18}.

Table 2. XQuery performance micro-benchmarks (not an exhaustive list).

μB_{16} *(dq)* The time needed to locate the elements to which are bound the `for` variables of a query. These variables can be seen as organized in a tree pattern of varying width and depth. For instance, width k and depth 2 yield the simple query Q_5^k (Fig. 4). This task is different from similar XPath queries, such as `//a1[//a2]...[//a`k`]`, or `//a1[//a2]...[//a`$(i-1)$`][//a`$(i+1)$`]...[//a`k`]//a`i, since unlike these XPath queries, all tuples of bindings for Q_5^k variables must be retained in the result.

μB_{17} *(dq)* measures the time to locate the roots of the sub-trees to be copied in the output of query Q_6^k, shown in Fig. 4. Q_6^k will return some `a1` elements lacking some `a`i, while Q_5^k discards them. Thus, an evaluation technique using frequency estimations of `a`i elements to reduce intermediary results will improve performance for Q_5^k, and it would not affect Q_6^k.

μB_{18} *(d)* measures the time needed to: locate the elements corresponding to `$x1` and `$x2` in the query Q_7^k, shown in Fig. 4, as well as its total evaluation time. In the query, θ stands for a comparison operator such as $=$, $<$, $\leq$ etc. When θ is $=$, separate micro-benchmarks should address: (i) the schema-less case, with the exact query above; then, replacing the condition with `$x/@y1=$x2/@y2`, (ii) the schema-less case, and (iii) the case when an XML schema specifies that the `y1` and `y2` attributes are in a key-foreign key relationship.
The time to locate the elements will show whether an efficient technique such as an index-based join is used to retrieve e.g. only those `$x2` elements with matching `$x1` elements. Measuring the total query evaluation time, especially for non-equality joins, exposes the join's performance, and how it interacts with serialization: If a `$x1` sub-tree must appear n times in the result, is it fully copied n times, or are some operations factorized ? The results of the micro-benchmarks based on Q_7^k should also be interpreted in conjunction with the results of μB_5 described in the previous section.

- **Document size** is the most obvious parameter for data scalability measures. It is also one of the most mis-used. For instance, some studies use queries addressing the `category` hierarchy of a "500 Mb XMark document". If the fact that the category hierarchy makes up at most 3% of XMark documents is omitted, document size is probably misleading.
- **Document tree depth** has an impact on document size, and determines the maximum length of downward path queries with non-empty results.
- **Fan-out** is the maximum number of children of a node. It has an impact on document size; it may also impact some storage strategies, and thus query performance.

Size, depth and fan-out, however, are insufficient to account for all interesting characteristics of a document. We identify the following important data set characteristics:

Presence of Schemas and Constraints. A DTD or XML Schema may be exploited for optimization purposes. More generally, one could envision XML query processors taking advantage of other classes of constraints, such as different type systems [2], or a-posteriori schemas extracted from schema-less data sets [13, 14]. A performance-oriented benchmark should state which constraints are used, if any. Proper type handling is one aspect of correctness.

Text-Centric vs. Data-Centric. Different XML data sets exhibit different ratios between the complexity of the XML structure tree, and the weight of the leaves (text). Both extremes are useful in different applications, and real-life data sets are in-between; the tree-to-text ratio may impact query performance. **Mixed contents** elements, frequent in text-centric documents, may raise correctness issues, as some systems do not support them.

Atomic Value Types. The XQuery data model provides a rich set of atomic value types. However, most existing value-based XML indexing techniques proposed so far ignore these types and XQuery's many value coercions [10, 11]; meshing a value index with coercion-based semantics is quite complex [18].

Frequency of Recursion. Data recursion is an interesting feature, encountered in real-life XML documents [19], and affecting many evaluation techniques. Thus, precise evaluations must specify whether recursive elements were absent, rare, or frequent in the input.

Tag Distribution. In some documents, each tag may appear on only one path; in some others, numerous paths may lead to elements having the same name, maybe due to recursion, maybe not. For instance, in some (but not all) systems, the XMark-inspired query `//item//keyword` would be evaluated by a structural join of all `item` and all `keyword` element IDs, even though many keywords do not appear under items.

Values. The actual values found in the document's text nodes must also be described for measures using queries with conditions on values. Value distribution controlled in synthetic data sets [23]. Value domain, size distribution, and contents also impact query evaluation.

Partitioning of Data in Documents. The XML and XQuery specifications give an important place to the notion of document, which can be seen as a physical segment of an XML data set. XQueries may combine information from several documents. Thus, it is important to understand how processors cope with input being fragmented over several documents.

Coming up with one unified data set, even a parametric one, on which all the above aspects can be varied at will, is hardly feasible. Notice that some parameters are inter-related and thus cannot be independently controlled, such as size, depth, and fan-out. We thus include in the benchmark repository two broad classes of synthetic documents. Documents in the first class are on purpose schema-less, and allow full control over the above mentioned parameters.

Documents in the second class are schema-driven; we rely on ToXGene [1] to generate those documents. We briefly describe each class of documents next.

Schema-Less Parametric Data Set. This data set is produced by a data generator we implemented. It allows controlling: the maximum node fanout, maximum depth, total tree size (number of elements), document size (disk occupancy), the number of distinct element names in the document, and the distribution of tags inside the document. Required parameters are: either tree size or document size; and, either depth or fan-out. The number of distinct element names is 1 by default; elements are named `a1`, `a2` etc.

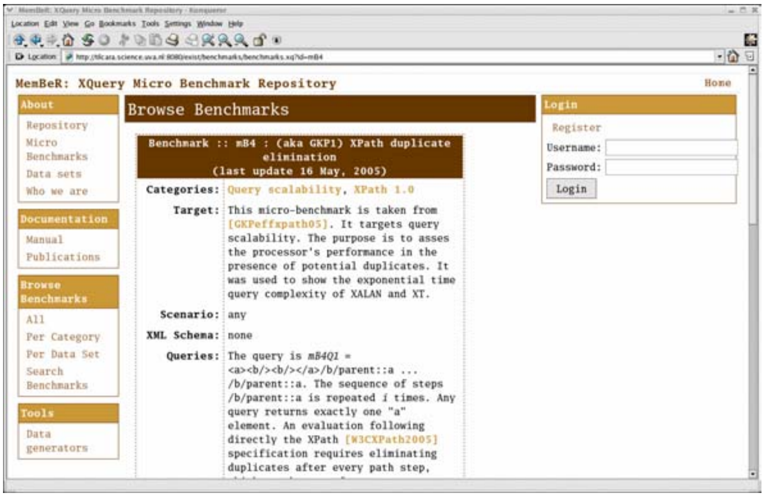
The distribution of tags within elements can be controlled in two ways. *Global* control allows tuning the overall frequency of element named `a1`, `a2`, ..., an. Labels may nest arbitrarily. Uniform and normal distributions are available. *Per-tag* control allows specifying, for every element name `ai`, the minimum and maximum level at which `ai` can appear may be set; furthermore, the relative frequency of `ai` elements at that level can be specified as a number between 0.0 and 1.0[2]. Global distributions allow generating trees where any `ai` may appear at any level. Close to this situation, for instance, is the Treebank data set[3], corresponding to annotated natural language; tags represent parts of speech and can nest quite freely. Per-tag distributions produce more strictly structured documents, whereas e.g., some names only appear at level 3, such as `article` and `inproceedings` in the DBLP data set[4], other elements appear only below level 7, such as `keywords` in XMark etc.

Fan-out, depth and tag distribution impact: the disk occupancy of many XML storage and structural indexing schemes; the complexity and precision of XML statistical synopses; the size of in-memory structures needed by an XML stream processor; and, the performance of path expression evaluation for many evaluation strategies. Thus, we will rely on this data set, and devise measures varying *all* these parameters, for assessing such aspects.

The number and size of text values follow uniform or normal distributions, as illustrated in μB_3 in Section 4. Values can be either filled with random characters, or taken from the Wikipedia text corpus (72 Mb of natural language text, in several languages). The latter is essential in order to run full-text queries; neither XMark nor MBench consider this issue.

Schema-Derived Data Sets. The ToXGene [1] XML generator produces XML documents conforming to a type description expressed in a subset of XML Schema. Furthermore, ToXGene provides hooks for controlling: the frequency of a given element type, the simple values found in leaf nodes, the sharing of values in several nodes (thus, the size of various join results), the specific values to fill in specific places in the document etc. Thus, we adopt ToXGene as a useful tool for controlled generation of schema-endowed documents.

[2] The generator checks the frequencies of several `ai`s at a given level for consistency.
[3] Available at `http://www.cs.washington.edu/research/xml/datasets`.
[4] Available at `http://dblp.uni-trier.de/xml`.



Browse Benchmarks

Benchmark :: mB4 : (aka GKP1) XPath duplicate elimination
(last update 16 May, 2005)

Categories:	Query scalability, XPath 1.0
Target:	This micro-benchmark is taken from [GKPeffxpath05]. It targets query scalability. The purpose is to asses the processor's performance in the presence of potential duplicates. It was used to show the exponential time query complexity of XALAN and XT.
Scenario:	any
XML Schema:	none
Queries:	The query is *mB4Q1* = <a><b/><b/></a>/b/parent::a ... /b/parent::a. The sequence of steps /b/parent::a is repeated *i* times. Any query returns exactly one "a" element. An evaluation following directly the XPath [W3CXPath2005] specification requires eliminating duplicates after every path step,

Fig. 5. Sample snapshot of Member's Web interface.

6 Conclusion and Perspectives

We have described a micro-benchmark repository approach for evaluating XQuery processing techniques. Micro-benchmarks provide the proper tools for systematically evaluating approaches to XML query evaluation. We have described the design principles underlying our repository and we have given several examples of micro-benchmarks and their applications.

We have started implementing our repository over a Web interface, available at `http://ilps.science.uva.nl/Resources/MemBeR/`. For now, it contains some micro-benchmarks, with their categorization criteria, targets, scenarios etc. A sample snapshot corresponding to the page of micro-benchmark μB_4 is depicted in Figure 5.

We are currently adding to the repository several other micro-benchmarks, including those mentioned in Section 3. We plan to extract candidate features from the XQuery specification itself, and also from XML query processing articles recently published in major database conferences. These may also yield inspiration for micro-benchmark queries, although our methodology is more rigorous. In parallel, we are inviting researchers and system designers to send us suggestions of features, measures, or micro-benchmarks they deem interesting, or even better, contribute some benchmarks and become a micro-benchmark reviewers.

The need for precise and meaningful assessment of XML query processors is becoming stringent for research to advance, and for communicating results. Micro-benchmarks as a step forward; we started MemBeR to develop comprehensive, well-documented ones.

References

1. D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: a template-based data generator for XML. In *WebDB*, 2002.
2. V. Benzaken, G. Castagna, and C. Miachon. A full pattern-based paradigm for XML query processing. In *PADL*, pages 235–252, 2005.
3. S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0 An XML Query Language, W3C Working Draft, April 2005. `http://www.w3.org/TR/xquery`.
4. Timo Böhme and Erhard Rahm. Xmach-1: A benchmark for XML data management. In *Proceedings of BTW2001, Oldenburg, 7.-9. März, Springer, Berlin*, March 2001.
5. S. Bressan, G. Dobbie, Z. Lacroix, M. Lee, Y. Li, U. Nambiar, and B. Wadhwa. X007: Applying 007 benchmark to XML query processing tool. In *CIKM*, pages 167–174. ACM, 2001.
6. World Wide Web Consortium. XML path language (XPath) version 1.0 – W3C Recommendation, 2000. `http://www.w3.org/TR/xpath.html`.
7. World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Full-Text – W3C Working Draft, July 2004. `http://www.w3.org/TR/xquery-full-text/`.
8. World Wide Web Consortium. W3C XQuery Update Requirements – W3C Working Draft, 2005. `http://www.w3.org/TR/xquery-update-requirements/`.
9. World Wide Web Consortium. XML path language (XPath) version 2.0 – W3C Working Draft, 2005. `http://www.w3.org/TR/xpath20/`.
10. World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Formal Semantics – W3C Working Drafts, 2005. `http://www.w3.org/TR/xquery-semantics/`.
11. World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Functions and Operators, 2005. `http://www.w3.org/TR/xpath-functions/`.
12. M. Francescet. XPathMark: an XPath benchmark for the XMark Generated Data. In *XSym*, 2005.
13. M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: Learning document type descriptors from XML document collections. *Data Min. Knowl. Discov.*, 1(7):23–56, 2003.
14. R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.
15. G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *PODS*, pages 179–190, 2003.
16. J. Hidders, J. Paredaens, R. Vercammen, and S. Demeyer. A light but formal introduction to XQuery. In *XSym*, pages 5–20, 2004.
17. H .V. Jagadish, S. Al-Khalifa, A. Chapman, L. V.S. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, , and C. Yu. Timber: a native XML database. *VLDB Journal*, 11(4), 2002.
18. J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Query optimization for semistructured data, 1998. Tech. report.
19. L. Mignet, D. Barbosa, and P. Veltri. The XML web: A first study. In *WWW Conference*, 2003.
20. G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. In *PODS*, 2002.
21. P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *SIGMOD*, pages 903–908, 2004.

22. S. Paparizos, Y. Wu, L. Lakshmanan, and H. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *SIGMOD*, 2004.
23. K. Runapongsa, J. Patel, H.V. Jagadish, Y. Chen, and S. Al-Khalifa. The Michigan benchmark: Towards XML query performance, 2001. `http://www.eecs.umich.edu/db/mbench`.
24. A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, M. J. Carey, I. Manolescu, and R. Busse. Why and How to Benchmark XML Databases. *SIGMOD Record*, 3(30):27–32, 2001.
25. B. Yao, T. Özsu, and N. Khandelwal. XBench benchmark and performance testing of XML DBMSs. In *ICDE*, pages 621–633. IEEE Computer Society, 2004.

Main Memory Implementations for Binary Grouping

Norman May and Guido Moerkotte

University of Mannheim
B6, 29
68131 Mannheim, Germany
{norman,moer}@pi3.informatik.uni-mannheim.de

Abstract. An increasing number of applications depend on efficient storage and analysis features for XML data. Hence, query optimization and efficient evaluation techniques for the emerging XQuery standard become more and more important. Many XQuery queries require nested expressions. Unnesting them often introduces binary grouping.
We introduce several algorithms implementing binary grouping and analyze their time and space complexity. Experiments demonstrate their performance.

1 Motivation

Optimization and efficient evaluation of queries over XML data becomes more and more important because an increasing number of applications work with XML data. In XQuery – the emerging standard query language for XML – queries including restructuring or aggregation often require nested queries. For example, the following query returns for each of the fifty richest persons of the world the number of countries with smaller gross domestic product (GDP) than the person's total capital.

```
for $p in document("richest-fifty.xml")//person
return
  <result>
    <person> { $p/name } </person>
    <count-richer> {
        count(for $c in document("countries.xml")//country
              where  $p/capital gt $c/gdp
              return $c) }
    </count-richer>
  </result>
```

This query combines data of two different documents and performs grouping and aggregation over the XML data. Note that each country can contribute to the count of multiple persons, and that a non-equality predicate is used to relate items from both documents.

Direct nested evaluation of this query is highly inefficient because for each person the nested FLWR expression is evaluated, demanding a scan of the countries document. Fortunately, the query can be unnested introducing binary grouping [17]. Moreover, optimizers can then apply algebraic equivalences to further improve performance. However, efficient implementations for binary grouping are not available yet. If they were, the optimizer could choose among them, ensuring an efficient query evaluation. We fill this gap and present several main-memory algorithms for implementing binary grouping. Further, we analyze their time and space complexity. The different algorithms will require different conditions to hold. Enumerating them then enables the query optimizer to select the most efficient implementation of binary grouping for a given situation. Experiments demonstrate that performance can be improved by orders of magnitude. Due to space constraints, we restrict ourselves to the formulation of algorithms working on sets of tuples. However, an extension to bags or sequences is not difficult (see [18]). Let us stress that binary grouping is useful not only in the context of XQuery. It has also been successfully applied to unnest nested OQL-queries [4, 20] and to evaluate complex OLAP queries [2].

The paper is structured as follows. Section 2 presents the definition of binary grouping and surveys properties of predicates and aggregate functions. They form the basis for the selection of an efficient implementation for the binary grouping operator. The main contribution of this paper – Section 3 – introduces several algorithms for binary grouping and analyzes their time and space complexity. Exemplary performance results are given in Section 4. More detailed experimental data is presented in [18]. Before concluding this paper, Section 5 reviews related work.

2 Preliminaries

2.1 The Algebra

We will only present the operators needed for our exposition. For an extensive treatment of our algebra we refer to [4]. Our framework is extendible to sequences as required in XQuery (cf. [17] for this algebra and related work).

The algebra works on sets of unordered tuples. Each tuple contains a set of variable bindings representing the attributes of the tuple. Single tuples are constructed by 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 $\mathcal{F}(e)$.

For an expression e_1 possibly containing free variables and a tuple t, $e_1(t)$ denotes the result of evaluating e_1 where bindings of free variables are taken from variable bindings provided by t – this requires $\mathcal{F}(e_1) \subseteq \mathcal{A}(t)$. Note that this can also be used for function application. We denote NULL values by $\_$.

The semantics of the binary grouping operator is defined by the map operator (χ) and the selection (σ). If their input is the empty set ($\emptyset$), their output is also empty.

Let us briefly recall **selection** with predicate p defined as $\sigma_p(e) := \{x | x \in e, p(x)\}$ and **map** defined as $\chi_{a:e_2}(e_1) := \{y \circ [a : e_2(y)] | y \in e_1\}$. The latter extends a given input tuple $y \in e_1$ by a new attribute a whose value is computed by evaluating $e_2(y)$.

Definition 1. *We define the* **binary grouping** *operator as:*

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

In this definition we call e_1 **grouping input** *and e_2* **aggregation input***.*

Note that the result of the binary grouping operator is empty if and only if the grouping input evaluates to an empty set. When the aggregation input is empty we assume that $f(\emptyset)$ is well-defined, and $f(\emptyset)$ is returned as the result. In many cases f will be an aggregation function such as `sum`. We refer to [18] for examples of applying these operators.

2.2 Properties of Predicates

To find the most efficient implementation for binary grouping, we take a closer look at the properties of predicates. Therefore, we distinguish, for example, *symmetric, irreflexive* predicates ($\neq$) from *antisymmetric, transitive* predicates ($<, \leq, >, \geq$).

2.3 Properties of Aggregate Functions

Aggregate functions can be *decomposable* and *reversible* [3]. These properties help us to find the most efficient implementation for binary grouping. To make the paper self-contained, we recall the definitions of these properties.

Let $\mathcal{N}$ be the codomain of a *scalar aggregate function* $f : X \rightarrow \mathcal{N}$ over some set X of tuples. In the definitions below, we will make use of (sub-) sets X, Y, and Z, with $X = Y \dot{\cup} Z$ and $Y \cap Z = \emptyset$.

Definition 2. *We say $f : X \rightarrow \mathcal{N}$ is* **decomposable** *if there exist functions*

$$\begin{aligned} \alpha &: X \rightarrow \mathcal{N}' \\ \beta &: \mathcal{N}', \mathcal{N}' \rightarrow \mathcal{N}' \\ \gamma &: \mathcal{N}' \rightarrow \mathcal{N} \end{aligned}$$

with $f(X) = \gamma(\beta(\alpha(Y), \alpha(Z)))$

Decomposable aggregate functions allow us to aggregate on subsets of the whole data and combine the results of these computations to the aggregate over the whole data. Obviously, the common aggregate functions are decomposable.

Definition 3. *A decomposable scalar function $f : X \rightarrow \mathcal{N}$ is called* **reversible** *if for β there exists a function $\beta^{-1} : \mathcal{N}', \mathcal{N}' \rightarrow \mathcal{N}'$ with*

$$f(Z) = \gamma(\beta^{-1}(\alpha(X), \alpha(Y)))$$

for all X, Y, and Z with $X = Y \dot{\cup} Z$ and $Y \cap Z = \emptyset$.

α

A_1	s	c
1	5	2
2	9	2
3	0	0
-	14	4

(a) after matching

β^{-1}

A_2	s	c
1	9	2
2	5	2
3	14	4

γ

A_2	a
1	4.5
2	2.5
3	3.5

(b) $\neq$-table

β

A_2	a	
1	14	4
2	9	2
3	0	0

γ

A_2	a
1	3.5
2	4.5
3	-

(c) $\leq$-table

Fig. 1. Example of the reversible aggregate function `avg`

Reversible scalar aggregates allow us to compute the value of an aggregate function over some subset by computing the aggregate function over some superset. Using this result, we can use the inverse function β^{-1} to compute the desired value for the subset. As examples `sum`, `count`, and `avg` are reversible, `min` and `max` are not.

For function `avg`, we define $\alpha(X) = [s : sum(X), c : |X|]$ computing the sum and cardinality of each group, $\beta([s : s_1, c : c_1], [s : s_2, c : c_2]) = [s : s_1 + s_2, c : c_1 + c_2]$, $\beta^{-1}([s : s_1, c : c_1], [s : s_2, c : c_2]) = [s : s_1 - s_2, c : c_1 - c_2]$ combining the sums and counts of two groups, and $\gamma([s : s_1, c : c_1]) = [a : s_1/c_1]$ yielding the average for each group.

The θ-table proposed in [3] exploits the properties of decomposable and reversible aggregate functions. Conceptually, the θ-table is an array with an entry for each group that stores data collected during aggregation. First, partial aggregation for some subset of the matching data is done. Then the results of the first step are combined to the final result for each group. The first step avoids duplicate work and is the source of improved efficency, while the second step benefits from the properties of the predicate and the aggregation function.

To make this more concrete, let us assume that after matching the grouping input and aggregation input the θ-table contains the data shown in Figure 1(a). In case of the $\neq$-table, aggregation is done with data matched with $=$ instead of $\neq$. In addition, the values for sum and count over the whole data set are collected in an auxiliary entry shown in the last row of the table (c.f. Fig. 1(b)). This auxiliary entry is used to obtain the sum and count values of each group using function β^{-1}. The final result is computed using function γ. For the first row in Figure 1(b) we have $\beta^{-1}([14, 4], [5, 2]) = [14 - 5, 4 - 2] = [9, 2]$ and $\gamma([9, 2]) = 4.5$.

With a $\leq$-table aggregation is only done on the closest matching group. The final result of each group is computed in a walk backwards through the table, incrementally combining the aggregated values of each group using function β. Applying function γ to each group yields the final result for each group. For the second row in Figure 1(c), we have $\beta([0, 0], [9, 2]) = [9, 2]$ and $\gamma([9, 2]) = 4.5$.

3 Algorithms

3.1 Notation

The following notation will be used in the complexity formulas to describe the time and space complexity of the various algorithms:

$$\begin{aligned} f &:= \textit{duplication factor} \\ g &:= \text{storage space per group} \\ \alpha &:= \textit{load factor} \text{ of the hash table} \\ l &:= \Theta(1+\alpha) \\ n &:= max(|e_1|, |e_2|) \end{aligned}$$

The *duplication factor* as defined in [1] is the ratio of the number of tuples before duplicate elimination to the number of tuples after duplicate elimination. Note that α, the load factor of the hash table, changes while values are inserted into the hash table. We will ignore this fact and use the load factor as an upper bound after all values have been inserted into the hash table. Therefore, all complexity formulas will represent upper bounds. For brevity reasons, we denote $l = \Theta(1+\alpha)$ as the time for a lookup in the hash table [5], and n as the maximum cardinality of both inputs.

In the exposition of each alternative algorithm we will follow the same basic structure: First, we state the assumptions on the predicate and the aggregate function as introduced in Section 2. Then, we present the algorithm in pseudo code and deduce the time and space complexity from the code. Finally, we explain implementation details. All operators are implemented as iterators [9] consisting of an **open** function for initialization, a **next** function which returns one result tuple of the operator for each call, and a **close** function that does some deinitialization. The implementations in our experiments are set-based. The pseudo code uses the following notations:

$p(x, y)$	– returns the result of evaluating the predicate $A_1\theta A_2$, where $A_1 \in \mathcal{A}(e_1)$, $A_2 \in \mathcal{A}(e_2)$, and θ a comparison as described in Section 2
T	– a tuple of either input
G	– a tuple representing a group
GT	– an auxiliary grouping tuple
$\zeta_\alpha(G)$	– initializes a tuple G appropriately for α,
$\alpha(G, T)$	– returns the result of evaluating function α on a group G with tuple T from the aggregation input
$\beta(G_1, G_2)$, $\beta^{-1}(G_1, G_2)$	– return the result of evaluating β and β^{-1} on groups G_1 and G_2
$\gamma(G)$	– returns the result of γ on a group G

Figure 2 summarizes the algorithms we present in this paper. The left part of the table contains the algorithms with their time and space complexity derived from their code. The right part of the table surveys the assumptions for each algorithm. Thus, this table can be used as a guide to the most efficient implementation. The assumptions are related to the inputs e_1 and e_2, the predicate $A_1\theta A_2$, and the function f as used in Definition 1.

The last column indicates the ratio of improvement in execution time over the direct nested evaluation of the nested query. For simplicity, we restrict ourselves only to sorted input for both the grouping and aggregation input for an input size that all algorithms were capable to evaluate. We use the algorithm NestedSort

Algorithm			Assumptions				
Name	Time	Space	e_1	e_2	$A_1\theta A_2$	f	Δ
NESTED	$\frac{l}{f}\|e_1\|\|e_2\|$	$\frac{g}{f}\|e_1\|$	-	-	-	-	0.95-1.2
NLBINGROUP	$\frac{l}{f}\|e_1\|\|e_2\| + (l + \frac{1}{f})\|e_1\|$	$\frac{g}{f}\|e_1\|$	-	-	-	-	0.65-0.75
HASHBINGROUP	$(l + \frac{1}{f})\|e_1\| + O((\frac{\|e_1\|}{f} + \|e_2\|)\ lg \frac{\|e_1\|}{f})$	$\frac{(1+g)\|e_1\|}{f}$	-	-	¬SY, T	D	1300
TREEBINGROUP	$\frac{\|e_1\|}{f} + O((\|e_1\| + \|e_2\|) \lg \frac{\|e_1\|}{f})$	$\frac{g}{f}\|e_1\|$	-	-	¬SY, T	D	1300
EQBINGROUP	$l(\|e_1\| + \|e_2\|) + \frac{\|e_1\|}{f}$	$\frac{g}{f}\|e_1\|$	-	-	¬R, SY, ¬T	RE	1850
NESTEDSORT	$\frac{1}{f}\|e_1\|\|e_2\|$	$O(1)$	S	-	-	-	1.0
SORTBINGROUP	$\frac{1}{f}\|e_1\|\|e_2\|$	$O(1)$	S	-	-	-	1.1-1.2
LTSORTBINGROUP	$\|e_1\| + \|e_2\|$	$O(1)$	S	S	¬SY, T	-	2100

S sorted	**R** reflexive	**D** decomposable
	SY symmetric	**RE** reversible
	T transitive	

Fig. 2. Assumptions and complexity for the implementations of the binary grouping operator

as the basis defining it as $\Delta = 1.0$. For some algorithms ranges for Δ are given because they are applicable for different types of predicates. Values of $\Delta > 1.0$ indicate an improvement by a factor Δ. Obviously, algorithms with more assumptions evaluate up to three orders of magnitude faster than the nested-loops-based algorithms with fewer assumptions.

3.2 Direct Evaluation of Nested Query

Nested evaluation is most generally applicable and the basis of comparison for implementations of the binary grouping operator.

In general, nested queries are implemented by calling the nested query for each tuple given to the map operator. However, more efficient techniques were proposed to evaluate nested queries [11]. The general idea is to memoize the result of the nested query for each binding of the nested query's free variables. When the same combination of free variables is encountered, the result of the previous computation is returned. In general, a hash table would be employed for memoizing which demands linear space in the size of the grouping input. For sorted grouping input, only the last result needs to be stored resulting in constant space.

We have implemented both strategies, and we will refer to these strategies by NESTED and NESTEDSORT. Because of its simplicity we omit the pseudo code for the nested strategies and restrict ourselves to the analysis of the complexity (cf. Fig. 2). Both strategies expose quadratic time complexity because the nested query must be executed for each value combination of free variables generated by the outer query. In absence of duplicates, this is also true when memoization is used.

3.3 Nested-Loop-Implementation of Binary Grouping

NLBinGroup. There are no assumptions on the predicate, the aggregate function, or the sortedness of any input.

```
OPEN
1  open e_1
   ▷ detect groups
2  while $T \leftarrow$ next e_1
3       do $G \leftarrow$ HT.LOOKUP($T$)
4          if $G$ does not exist
5             then $G \leftarrow$ HT.INSERT($T$)
                   ▷ initialize group
6                  $\zeta_\alpha(G)$
7  close e_1
   ▷ match aggregation input to groups
8  open e_2
9  while $T \leftarrow$ next e_2
10      do for each group $G$
                    in the HT
11            do if $p(G, T)$
12                  then $G \leftarrow \alpha(G,T)$
13 close e_2
14 htIter $\leftarrow$ HT.ITERATOR

NEXT
   ▷ next group in the hash table
1  if $G \leftarrow$ htIter.NEXT
2     then return $\gamma(G)$
3     else  return _

CLOSE
1  HT.CLEANUP
```

(a) NLBinGroup

```
OPEN
1  open e_1
2  $\zeta_\alpha(GT)$            ▷ initialize group tuple
3  while $T \leftarrow$ next e_1
4       do $G \leftarrow$ HT.LOOKUP($T$)
5          if $G$ does not exist
6             then $G \leftarrow$ HT.INSERT($T$)
                   ▷ initialize group
7                  $\zeta_\alpha(G)$
8  close e_1

9  open e_2
10 while $T \leftarrow$ next e_2
11      do $G \leftarrow$ HT.LOOKUP($T$)
12         if $G$ exists
13            then $G \leftarrow \alpha(G, T)$
14         if predicate is $\neq$
15            then $GT \leftarrow \alpha(GT, T)$
16 close e_2
17 htIter $\leftarrow$ HT.ITERATOR

NEXT
1  if $G \leftarrow$ htIter.NEXT
2     then if predicate is $\neq$
3              then $G \leftarrow \beta^{-1}(G, GT)$
4           return $\gamma(G)$
5     else  return _

CLOSE
1  HT.CLEANUP
```

(b) EQBinGroup

Fig. 3. Pseudo code of NLBinGroup and EQBinGroup

We call the naive nested-loops-based implementation proposed in [2, 9] NLBinGroup. The pseudo code for this algorithm is shown in Figure 3(a). Most work is done in function OPEN. First, the grouping input is scanned, and all groups are detected and stored in a hash table ($l|e_1|$ time). Most of the following algorithms will follow this pattern. Next, the aggregation input is scanned once for each group in the hash table. The tuples from the aggregation input are matched with the tuple of the current group using the predicate. This matching phase is similar to a nested-loop join and requires $O(\frac{l}{f}|e_1||e_2|)$ time. When a match is found, function α is used for aggregation. After this matching phase a traversal through all groups in the hash table is done to execute function γ to finalize the groups ($\frac{|e_1|}{f}$ time). The complete complexity formulas can be found in Figure 2.

From the complexity equations we see that this algorithm introduces some overhead compared to the hash-based case of NESTED because several passes through the hash table are needed. Hence, the time complexity is slightly higher then the direct nested evaluation. The following sections discuss more efficient algorithms for restricted cases.

3.4 Implementation of Binary Grouping with $=$ or $\neq$-Predicate

EQBinGroup. If the predicate is not an equivalence relation, the aggregate function must be decomposable and reversible.

We generalize the $\neq$-Table defined in [3] for predicate $\neq$. Instead of an array, we use a hash table to store an arbitrary number of groups. When collision lists do not degrade, the asymptotic runtime will not change, however. The algorithm in Figure 3(b) extends NLBINGROUP.

In function OPEN detecting all groups requires $l|e_1|$ time. In line 11 we do matching with equality for both kinds of predicates. But in line 15 all tuples are aggregated in a separate tuple GT using function α if the predicate is $\neq$. Alltogether matching requires $l|e_2|$ time.

When we return the result in a final sweep through the hash table ($\frac{|e_1|}{f}$ time) we have to apply the reverse function β^{-1} when the predicate is $\neq$ (cf. line 3 in NEXT). For that, we use the auxiliary grouping tuple GT and the group G matched with $=$ and compute the aggregation result for $\neq$. For scalar aggregate functions, this computation can be done in constant time and space. For both types of predicates, groups are finalized using function γ.

Compared to the directly nested evaluation and hash-based grouping, the time complexity can be improved to linear time and linear space complexity (cf. Fig. 2).

Figure 4 shows how EQBINGROUP implements the idea of the $\neq$-table introduced in Section 2. Figure 4(b) shows the content of the hash table after function OPEN. For each detected group, the tuple for attribute a stores the value of attribute A_1, the `sum`, and the `count` of all matching tuples of the group. The additional tuple GT is added at the bottom of the table. Note that the group with value 3 did not find any match, but a properly initialized tuple for it exists in the hash table. Applying function β^{-1} to each group and GT and then function γ as described in Section 2 produces the final result (cf. Fig. 4(c)).

3.5 Implementation of Binary Grouping with $\leq$-Predicate

These algorithms are applicable if the predicate is antisymmetric, transitive, and the aggregate function is decomposable; no assumptions are made on the sortedness of any inputs.

In this paper we investigate a hash table and a balanced binary search tree to implement the $\leq$-table proposed in [3]. The advantage of this approach compared

R_1
A_1
1
2
3

R_2	
A_2	B
1	2
1	3
2	4
2	5

(a) Input data

$(R_1)\Gamma_{a;A_1 \neq A_2;avg(B)}(R_2)$	
A_1	a
1	$\langle[1, 5, 2]\rangle$
2	$\langle[2, 9, 2]\rangle$
3	$\langle[3, 0, 0]\rangle$
GT	$\langle[\_, 14, 4]\rangle$

(b) After **open**

$(R_1)\Gamma_{a;A_1 \neq A_2;avg(B)}(R_2)$	
A_1	a
1	$\langle[1, 4.5]\rangle$
2	$\langle[2, 2.5]\rangle$
3	$\langle[3, 3.5]\rangle$
GT	$\langle[\_, 14, 4]\rangle$

(c) Final result

Fig. 4. Example of the evaluation of EQBinGroup

```
Open
open e1
for T ← next e1
      do G ← HTl.Lookup(T)
          if G does not exist
             then G ← HT.Insert(T)
                    ▷ initialize group
                    $\zeta_\alpha(G)$
close e1
sort groups by matching
predicate of e1
open e2
for T ← next e2
      do G ← minimal group in
                    ≤-Table ≥ T
          G ← $\alpha(G, T)$
close e2
htIter ← ≤-Table.Iterator

Next
   ▷ next group in the ≤-table
if G ← htIter.Next
   then G ← $\beta(G, successor(G))$
          return $\gamma(G)$
   else  return _

Close
HT.Cleanup
≤-Table.CleanUp
```

(a) HashBinGroup

```
Open
open e1
for T ← next e1
      do if _ == RB-Tree.Lookup(T)
            then G ← RB-Tree.Insert(T)
                    ▷ initialize group G
                    $\zeta_\alpha(G)$
close e1

open e2
while T ← next e2
      do G ← minimal group in
                    RB-Tree ≥ T
         G ← $\alpha(G, T)$
close e2
G ← RB-Tree.Maximum

Next
if G ≠ RB-Tree.Minimum
   then G ← $\beta(G,$ RB-Tree.Succ(G))
          G' ← $\gamma(G)$
          G ← RB-Tree.Pred(G)
          return G'
   else  return _

Close
RB-Tree.CleanUp
```

(b) TreeBinGroup

Fig. 5. Pseudo code of HashBinGroup and TreeBinGroup

to using an array is that no upper bound for the number of groups needs to be known. Since the assumptions are the same for both alternatives, we will only discuss implementation details.

R_1

A_1
1
2
3

R_2

A_2	B
1	2
1	3
2	4
2	5

(a) Input data

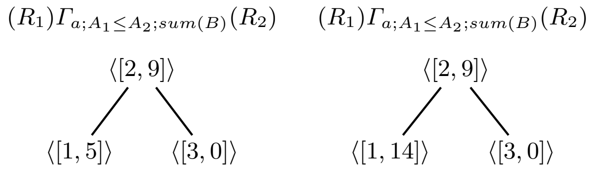
$(R_1)\Gamma_{a;A_1 \leq A_2;sum(B)}(R_2)$

A_1	a
1	$\langle[1,5]\rangle$
2	$\langle[2,9]\rangle$
3	$\langle[3,0]\rangle$

$(R_1)\Gamma_{a;A_1 \leq A_2;sum(B)}(R_2)$

A_1	a
1	$\langle[1,14]\rangle$
2	$\langle[2,9]\rangle$
3	$\langle[3,0]\rangle$

(b) HashBinGroup



(c) TreeBinGroup

Fig. 6. Example for the evaluation of HashBinGroup and TreeBinGroup

HashBinGroup. This algorithm, outlined in Figure 5(a), extends the NL-BinGroup operator. It is formulated in terms of predicate $<$.

First, all groups are identified using a hash table ($l|e_1|$ time). Before matching the tuples from the aggregation input, these groups are sorted according to the predicate ($O(\frac{|e_1|}{f} \lg \frac{|e_1|}{f})$ time). This can be done in a separate array in which the items in the hash table are referenced. In the matching phase binary search is employed to find the closest group that still matches with the predicate ($O(|e_2| \lg \frac{|e_1|}{f})$ time). Aggregation is done using function α. To compute the final result, one walk backwards through the array visits each group ($\frac{|e_1|}{f}$ time). First, the aggregated values of distinct groups are combined using function β. Then, function γ computes the final result of the group. One must be careful not to destroy the aggregated result of the previous group when applying function γ. The overall complexity can be found in Fig. 2.

TreeBinGroup. In an alternative implementation shown in Figure 5(b), we use a balanced search tree (e.g. a Red-Black-Tree) to identify all groups ($O(|e_1| \lg \frac{|e_1|}{f})$ time). The search tree structure implies the inclusion of groups. Thus, no sorting is needed after this step. Matching of tuples is done by a lookup in the search tree ($O(|e_2| \lg \frac{|e_1|}{f})$ time). When a group cannot be found, matching and aggregation is done on the last node in the tree that was visited. As with the previous algorithm, a backward traversal through the tree is done to aggregate the final result for each group using function γ ($\frac{|e_1|}{f}$ time). The resulting complexity is summarized in Fig. 2.

Comparison of the Implementations. Figure 6 resumes with the example in Section 2 to trace the evaluation of HashBinGroup and TreeBinGroup showing the state after **open**. Note that the groups must be sorted to find the closest matching group for aggregation with function α. This is achieved either by sorting the groups in the hash table or implicitly during insertion into the binary search tree. Each tuple stores the value of the grouping attribute and the aggregated result for the group. The result of the final walk backwards through the $\leq$-table computes the final result using function β and γ.

When we compare the complexity formulas we observe that sorting is dominant in HashBinGroup, and insertion is dominant in TreeBinGroup. Note that in both cases, we can remove duplicates during insertion. The hash-based implementation removes duplicates before sorting. In contrast, lookup of all items in e_1 in the balanced search tree demands $O(|e_1| \lg \frac{|e_1|}{f})$ time. This gives the hash-based implementation a potential advantage. On the other hand, the hash-based implementation does not degrade nicely when the collision lists on the hash table are not bounded by a constant any more. This can lead to linear search time in the collision lists ($l \in O(|e_1|)$, where l is the size of the collision list). Thus, the hash-based implementation depends on a good hash function.

3.6 Implementations of Binary Grouping on Sorted Input

When the grouping input or the aggregation input is sorted, we can improve the algorithm NLBinGroup.

SortBinGroup. First, we assume that only the grouping input is sorted.

Figure 7(a) presents the pseudo code for this algorithm. With sorted grouping input, groups can be detected efficiently because only subsequent tuples need to be compared (line 1 in Next). This can be done in constant space.

Matching the tuples of the aggregation input can be done with an algorithm similar to a 1:N sort-merge join, i.e. a sort-merge join algorithm that assumes that no duplicates occur on the left input. In the general case of an arbitrary predicate, the aggregation input needs to be scanned once for each group. This is done in $O(\frac{1}{f}|e_1||e_2|)$ time. It is also the reason for having no assumptions on the sortedness of the aggregation input.

Since the algorithm iterates through each group and matches all tuples from the aggregation input, groups does not have to be combined. Thus, the aggregation function need not be decomposable.

LTSortBinGroup. In addition to the assumptions of the previous algorithm, we now assume a antisymmetric and transitive predicate (e.g. $<$, or $\geq$). Both inputs need to be sorted. The direction of sorting depends on the predicate used. For example, for predicates $<$ and $\leq$ both inputs need to be sorted in descending order, for $>$ and $\geq$ in ascending order. No restrictions apply to the aggregation function.

These assumptions allow us to scan both inputs only once resulting in a time complexity of $|e_1| + |e_2|$. Each group resumes aggregation on the aggregated result of the previous group. For aggregation, we always use function α. The

```
OPEN
1  open $e_1$
2  open $e_2$

NEXT
1  if $G \leftarrow$ next group in $e_1$
2     then while $T \leftarrow$ next $e_2$
3                  do if $p(G, T)$
4                        then $G \leftarrow \alpha(G, T)$
5           close $e_2$
6           open $e_2$
7           return $\gamma(G)$
8     else  return _

CLOSE
1  close $e_1$
2  close $e_2$
```

(a) SortBinGroup

```
OPEN
1  open $e_1$
2  open $e_2$
3  $\zeta_\alpha(GT)$      ▷ initialize group tuple

NEXT
1  if $G \leftarrow$ next group in $e_1$
2     then copy group attributes of
                  $G$ into $GT$
3          while ($T \leftarrow$ next $e_2$) $\wedge$
                  $p(GT, T)$
4             do $GT \leftarrow \alpha(GT, T)$
          ▷ keep aggregated result in $GT$
5          $G \leftarrow \gamma(GT)$
6          return $G$
7     else return _

CLOSE
1  close $e_1$
2  close $e_2$
```

(b) LTSortBinGroup

Fig. 7. Pseudo code of SortBinGroup and LTSortBinGroup

result of finalizing a group using function γ is stored in a separate tuple, so that the current value of aggregation is not destroyed (cf. line 5 in Next). The algorithm stated in Figure 7(b) is formulated in terms of $<$ or $\leq$ as predicates.

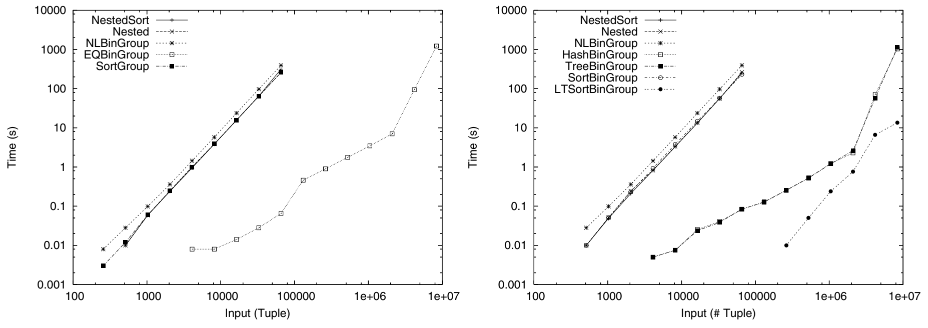
4 Experiments

We have implemented all algorithms in a prototype run-time system using GCC C++ version 3.3.4. All queries were executed on an Intel Pentium M with 1.4 GHz and 512MB RAM running Linux with 2.6.8 Kernel. In several experiments the performance of each algorithm was evaluated for different distributions. For space reasons we can only present a tiny fraction of the experimental data and refer to [18] for the details of the benchmark and the complete set of experimental results.

The cardinality of the input sequences e_1 and e_2 ranged between 128 and 8388608. The grouping input e_1 and the aggregation input e_2 were of equal size. The largest data set contained 63MB of data. The input for a query was loaded into main memory before executing the queries.

Figure 8 summarizes the most interesting results of our experiments. It shows the elapsed time for sorted input for both the grouping input and aggregation input for the predicate $\neq$ and $>$.

Figure 8(a) clearly shows that EQBinGroup is the most efficient algorithm for predicate $\neq$. It performs orders of magnitude faster than the nested-loops-based algorithms Nested, NestedSort, NLBinGroup and SortBinGroup



(a) Predicate $\neq$ (b) Predicate $>$

Fig. 8. Group input and aggregation input sorted

which are hard to distinguish in their query performance. However, since EQBinGroup loads all detected groups into a main memory data structure, its performance suffers when memory gets scarce. In our experiments, this happens for more than 2 million groups.

Figure 8(b) presents the experimental results for predicate $>$. The most efficient algorithm is LTSortBinGroup which is suited best for sorted input. When the input is not sorted, both HashBinGroup and TreeBinGroup are efficient algorithms with similar performance. When they run out of memory, both reveal the same weakness as EQBinGroup.

Among the nested-loop-based algorithms NLBinGroup is slowest. The inefficiency was caused by the iterator used for traversing the hash table. Only SortBinGroup exposes slightly improved efficiency compared to direct nested evaluation using memoization.

Summarizing, our experiments confirm the theoretical results from Section 3. We refer to [18] for more experimental results and a more detailed analysis.

5 Related Work

To the best of our knowledge, this paper is the first to investigate *efficient* implementations for binary grouping. Only one implementation corresponding to the NLBinGroup was presented so far [2].

However, previous work justifies the importance of binary grouping. Slightly different definitions of it can be found in [2, 3, 20]. Only [3] describes possible implementations. These papers enumerate use cases for binary grouping. In this paper we propose efficient implementations of binary grouping and evaluate their efficiency.

In addition, implementation techniques known for other operators apply for the binary grouping operator as well. The idea of merging the functionality of different algebraic operators to gain efficiency is well known. In [21] query patterns for OLAP queries are identified. One of these patterns – a sequence of grouping and equi-join – is similar to the implementation of the binary grouping operator. Sharing hash tables among algebraic operators was proposed in [12].

Our work also relates to work comparing sort-based and hash-based implementations of algebraic operators [7, 9, 10, 13, 14, 19]. However, they concentrate on implementations of equijoins. Non-Equality joins have been studied first in [8].

We presented main-memory implementations of the binary grouping operator. Implementation techniques that materialize data that does not fit into main memory can be applied to the binary grouping operator. We refer to [1, 6, 9, 15, 16] for such proposals.

6 Conclusion and Future Work

Binary grouping is a powerful operator to evaluate analytic queries [2] or to unnest nested queries [4, 17]. We have introduced, analyzed, and experimentally evaluated main memory implementations for binary grouping. Further, we have identified the conditions under which each algorithm is applicable.

The results show that query processing time can be improved by orders of magnitude, compared to nested evaluation of the query. Hence, binary grouping is a valuable building block for database systems that support grouping and aggregation efficiently.

For space reasons we refer to [18] for extensions of our algorithms to data models working on bags or sequences.

Acknowledgements

We would like to thank Carl-Christian Kanne and Simone Seeger for their comments on the manuscript.

References

1. D. Bitton and D. J. DeWitt. Duplicate record elimination in large data files. *ACM TODS*, 8(2):255–265, June 1983.
2. D. Chatziantoniou, M. Akinde, T. Johnson, and S. Kim. The MD-Join: An Operator for Complex OLAP. In *Proc. ICDE*, pages 524–533, 2001.
3. S. Cluet and G. Moerkotte. Efficient evaluation of aggregates on bulk types. *Proc. of 5-th DBPL*, 1995.
4. S. Cluet and G. Moerkotte. Nested queries in object bases. Technical Report RWTH-95-06, GemoReport64, RWTH Aachen/INRIA, 1995.
5. T. Corman, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2nd edition, 2001.
6. Jochen Van den Bercken, Martin Schneider, and Bernhard Seeger. Plug&join: An easy-to-use generic algorithm for efficiently processing equi and non-equi joins. In *EDBT '00*, pages 495–509, 2000.

7. D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. of the ACM SIGMOD*, pages 1–8, June 1984.
8. D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *Proc. VLDB*, pages 443–452, 1991.
9. G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.
10. G. Graefe. Sort-merge-join: An idea whose time has(h) passed? In *Proc. ICDE*, pages 406–417, 1994.
11. G. Graefe. Executing nested queries. In *BTW*, pages 58–77, 2003.
12. G. Graefe, R. Bunker, and S. Cooper. Hash joins and hash teams in Microsoft SQL server. In *Proc. VLDB*, pages 86–97, 1998.
13. G. Graefe, A. Linville, and L. D. Shapiro. Sort vs. hash revisited. *IEEE TKDE*, 6(6):934–944, December 1994.
14. L. M. Haas, M. J. Carey, M. Livny, and A. Shukla. Seeking the truth about *ad hoc* join costs. *VLDB Journal*, 6(3):241–256, May 1997.
15. S. Helmer, T. Neumann, and G. Moerkotte. Early grouping gets the skew. Technical Report TR-02-009, University of Mannheim, 2002.
16. S. Helmer, T. Neumann, and G. Moerkotte. A robust scheme for multilevel extendible hashing. *Proc. 18th ISCIS*, pages 218–225, 2003.
17. N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. *Proc. ICDE*, pages 239–250, 2004.
18. N. May, S. Helmer, and G. Moerkotte. Main memory implementations for binary grouping. Technical report, University of Mannheim, 2005. available at: `http://pi3.informatik.uni-mannheim.de/publikationen.html`.
19. D. E. Simmen, E. J. Shekita, and T. Malkemus. Fundamental techniques for order optimization. *SIGMOD Record*, 25(2):57–67, 1996.
20. H. J. Steenhagen, P. M. G. Apers, H. M. Blanken, and R. A. de By. From nested-loop to join queries in OODB. *Proc. VLDB*, pages 618–629, 1994.
21. T. Westmann and G. Moerkotte. Variations on grouping and aggregation. Technical report, University of Mannheim, 1999.

Logic Wrappers and XSLT Transformations for Tuples Extraction from HTML

Costin Bădică[1] and Amelia Bădică[2]

[1] University of Craiova, Software Engineering Department
Bvd.Decebal 107, Craiova, RO-200440, Romania
badica_costin@software.ucv.ro

[2] University of Craiova, Business Information Systems Department
A.I.Cuza 13, Craiova, RO-200585, Romania
ameliabd@yahoo.com

Abstract. Recently it was shown that existing general-purpose inductive logic programming systems are useful for learning wrappers (known as L-wrappers) to extract data from HTML documents. Here we propose a formalization of L-wrappers and their patterns, including their syntax and semantics and related properties and operations. A mapping of the patterns to a subset of XSLT that has a formal semantics is outlined and demonstrated by an example. The mapping actually shows how the theory can be applied to obtain efficient wrappers for information extraction from HTML.

1 Introduction

Many Web resources can be abstracted as providing relational information as sets of tuples, including: search engines result pages, product catalogues, news sites, product information sheets, a.o. Recently, we have experimentally shown that general-purpose inductive logic programming (ILP hereafter) systems might be useful for learning logic wrappers (i.e. L-wrappers) to extract tuples from Web pages written in HTML ([4–6]). Our wrappers use patterns defined as logic rules. Technically, these rules are acyclic conjunctive queries over trees ([12]). Their patterns are matched against XHTML information sources to extract the relevant information. Here we complement this work by giving a formalization of L-wrappers and their patterns using directed graphs. Then we show how L-wrappers can be efficiently implemented using XSLT ([9]) and corresponding transformation engines.

Web pages and XML can be regarded as semi-structured data modeled as labeled ordered trees ([1]). In this paper we study the syntax, semantics, and properties of patterns used in L-wrappers, in the more general context of information extraction from semi-structured data. This study serves at least three purposes: i) as a concise specification of L-wrappers; this enables a theoretical investigation of their properties and operations and allows comparisons with related works; ii) as a convenient way for mapping L-wrappers to XSLT for efficient processing using available XSLT processing engines; the mathematically sound approach enables also the study of the correctness of the implementation of L-wrappers using XSLT (however this issue is not addressed in this paper; only an informal argument is given); iii) furthermore, the experimental work reported in [6] revealed some scalability problems of applying ILP to learn L-wrappers of arity greater than three (i.e. tuples containing at least three attributes). There, this was explained by the large number of negative examples required, that grows exponentially with the tuple arity. Here we show that this problem can be tackled as follows: in order to learn a wrapper to extract tuples of arity $k \geq 3$, we suggest learning $k - 1$ wrappers to extract tuples of arity 2 and then use the pattern merging operator to merge them, rather than learning the wrapper in one shot. This approach is demonstrated using an example.

The work described in this paper is part of an ongoing research project that investigates the application of general-purpose ILP systems (like FOIL [18] or Aleph [2]), logic representations of wrappers and XML technologies (including the XSLT transformation language [9]) to information extraction from the Web.

The paper is structured as follows. First, we present the syntax and semantics of extraction patterns. Syntax is defined using directed graphs, while semantics and sets of extracted tuples are defined using a model-theoretic approach. Then, we discuss pattern properties – subsumption, equivalence, and operations – simplification and merging. We follow with a case study inspired by our previous work on using ILP to learn L-wrappers for HTML. In the case study we analyze the example L-wrappers from paper [6]. We discuss their combination using pattern merging and describe their mapping to XSLT. An explanation of why this mapping works is also given. Note that the application of ILP, logic representations and XML technologies to information extraction from the Web is not an entirely new field; several approaches and tool descriptions have already been proposed and published ([3, 8, 11, 13, 14, 16, 19, 20]; see also the survey in [15]). Therefore, before concluding, we follow with a summary of these relevant works, briefly comparing them with our own work. The last section of the paper contains some concluding remarks and points to future research directions.

2 Patterns. Syntax and Semantics

We model semi-structured data as labeled ordered trees. A wrapper takes a labeled ordered tree and returns a subset of tuples of extracted nodes. An extracted node can be viewed as a subtree rooted at that node. Within this framework, a pattern can be interpreted as a conjunctive query over labeled ordered trees, yielding a set of tree node tuples as answer. The node labels of a labeled ordered tree correspond to attributes in semi-structured databases or tags in tagged texts. Let Σ be the set of all node labels of a labeled ordered tree.

For our purposes, it is convenient to abstract labeled ordered trees as sets of nodes on which certain relations and functions are defined. Note that in this paper we are using some basic graph terminology as introduced in [10].

Definition 1. *(Labeled ordered tree) A* labeled ordered tree *is a tuple* $t = \langle T, E, r, l, c, n \rangle$ *such that:*

i) (T, E, r) *is a rooted tree with root* $r \in T$*. Here,* T *is the set of tree nodes and* E *is the set of tree edges ([10]).*

ii) $l : T \rightarrow \Sigma$ *is a node labeling function.*

iii) $c \subseteq T \times T$ *is the parent-child relation between tree nodes.* $c = \{(v, u) |$ *node* u *is the parent of node* $v\}$*.*

iv) $n \subseteq T \times T$ *is the next-sibling linear ordering relation defined on the set of children of a node. For each node* $v \in T$*, its* k *children are ordered from left to right, i.e.* $(v_i, v_{i+1}) \in n$ *for all* $1 \leq i < k$*.*

A pattern is a labeled directed graph. Arc labels denote conditions that specify the tree delimiters of the extracted information, according to the parent-child and next-sibling relationships (eg. is there a parent node ?, is there a left sibling ?, a.o). Vertex labels specify conditions on nodes (eg. is the tag label *td* ?, is it the first child ?, a.o).

Conditions specified by arc and node labels must be satisfied by the extracted nodes and/or their delimiters. A subset of graph vertices is used for selecting the items for extraction.

We adopt a standard relational model. Associated to each information source is a set of distinct attributes. Let $\mathcal{A}$ be the set of attribute names.

Definition 2. *(Syntax) An* (extraction) pattern *is a tuple* $p = \langle V, A, U, D, \mu, \lambda_a, \lambda_c \rangle$ *such that:*

i) (V, A) *is a directed graph.* V *is the finite set of vertices and* $A \subseteq V \times V$ *is the set of directed edges or arcs.*

ii) $\lambda_a : A \rightarrow \{'c', 'n'\}$ *is the labeling function for arcs. The meanings of* $'c'$ *and* $'n'$ *are:* $'c'$ *label denotes the parent-child relation and* $'n'$ *label denotes the next-sibling relation.*

iii) $\lambda_c : V \rightarrow \mathcal{C}$ *is the labeling function for vertices. It labels each vertex with a condition from the set* $\mathcal{C} = \{\emptyset, \{'f'\}, \{'l'\}, \{\sigma\}, \{'f', 'l'\}, \{'f', \sigma\}, \{'l', \sigma\}, \{'f', 'l', \sigma\}\}$ *of conditions, where* σ *is a label in the set* Σ *of symbols. In this context, the meanings of* $'f'$*,* $'l'$ *and* σ *are:* $'f'$ *label requires the corresponding vertex to indicate a first child;* $'l'$ *label requires the corresponding vertex to indicate a last child;* σ *label requires the corresponding vertex to indicate a node labeled with* σ*.*

iv) $U = \{u_1, u_2, \ldots, u_k\} \subseteq V$ *is the set of* pattern extraction vertices *such that for all* $1 \leq i \leq k$*, the number of incoming arcs to* u_i *that are labeled with* $'c'$ *is 0.* k *is called the* pattern arity*.*

v) $D \subseteq \mathcal{A}$ *is the set of attribute names defining the relation scheme of the information source.* $\mu : D \rightarrow U$ *is a one-to-one function that assigns a pattern extraction vertex to each attribute name.*

Note that according to point iv) of definition 2, an extraction pattern does not state any condition about the descendants of an extracted node; i.e. it looks only at its siblings and its ancestors. This is not restrictive in the context of patterns for information extraction from HTML; see for example the rules in Elog$^-$, as described in [13].

In what follows, we provide a model-theoretic semantics for our patterns. In this setting, a labeled ordered tree is an interpretation domain for the patterns. The semantics is defined by an interpretation function assigning tree nodes to pattern vertices.

Definition 3. *(Interpretation) Let $p = \langle V, A, U, D, \mu, \lambda_a, \lambda_c \rangle$ be a pattern and let $t = \langle T, E, r, l, c, n \rangle$ be a labeled ordered tree. A function $I : V \to T$ assigning tree nodes to pattern vertices is called* interpretation.

Intuitively, patterns are matched against parts of a target labeled ordered tree. A successful matching asks for the labels of pattern vertices and arcs to be consistent with the corresponding relations and functions over tree nodes.

Definition 4. *(Semantics) Let $p = \langle V, A, U, D, \mu, \lambda_a, \lambda_c \rangle$ be a pattern, let $t = \langle T, E, r, l, c, n \rangle$ be a labeled ordered tree and let $I : V \to T$ be an interpretation function. Then I and t are* consistent *with p, written as $I, t \models p$, if and only if:*

i) If $(v, w) \in A$ and $\lambda_a((v, w)) = 'n'$ then $(I(v), I(w)) \in n$.
ii) If $(v, w) \in A$ and $\lambda_a((v, w)) = 'c'$ then $(I(v), I(w)) \in c$.
iii) If $v \in V$ and $'f' \in \lambda_c(v)$ then for all $w \in V$, $(I(w), I(v)) \notin n$.
iv) If $v \in V$ and $'l' \in \lambda_c(v)$ then for all $w \in V$, $(I(v), I(w)) \notin n$.
v) If $v \in V$ and $\sigma \in \Sigma$ and $\sigma \in \lambda_c(v)$ then $l(I(v)) = \sigma$.

A labeled ordered tree for which an interpretation function exists is called a model *of p.*

Our definition for patterns is quite general by allowing to build patterns for which no consistent labeled ordered tree and interpretation exist. Such patterns are called *inconsistent*. A pattern that is not inconsistent is called *consistent*. The following proposition states necessary conditions for pattern consistency.

Proposition 1. *(Necessary conditions for consistent patterns) Let $p = \langle V, A, U, D, \mu, \lambda_a, \lambda_c \rangle$ be a consistent pattern. Then:*

i) The graph of p is a DAG.
ii) For any two disjoint paths between two distinct vertices of p, one has length 1 and its single arc is labeled with $'c'$ and the other has length at least 2 and its arcs are all labeled with $'n'$, except the last arc that is labeled with $'c'$.
iii) For all $v \in V$ if $'f' \in \lambda_c(v)$ then for all $w \in V$, $(w, v) \notin A$ or $c((w, v)) =' c'$ and if $'l' \in \lambda_c(v)$ then for all $w \in V$, $(v, w) \notin A$ or $c((v, w)) =' c'$.

The proof of this proposition is quite straightforward. The idea is that a consistent pattern has at least one model and this model is a labeled ordered tree. Then, the claims of the proposition follow from the properties of ordered trees seen as directed graphs ([10]). Note that in what follows we are considering only consistent patterns.

The result of applying a pattern to a semi-structured information source is a set of extracted tuples. An extracted tuple is modeled as a function from attribute names to tree nodes, as in standard relational data modeling.

Definition 5. *(Extracted tuple) Let* $p = \langle V, A, U, D, \mu, \lambda_a, \lambda_c \rangle$ *be a pattern and let* $t = \langle T, E, r, l, c, n \rangle$ *be a labeled ordered tree that models a semi-structured information source. A tuple extracted by* p *from* t *is a function* $I \circ \mu : D \to T$ [1], *where* I *is an interpretation function such that* $I, t \models p$.

Note that if p is a pattern and t is a tree then p is able to extract more than one tuple from t. Let $Ans(p, t)$ be the set of all tuples extracted by p from t.

3 Pattern Properties and Operations

In this section we study pattern properties – subsumption, equivalence and operations – simplification and merging.

Subsumption and equivalence enable the study of pattern simplification, i.e. the process of removing arcs in the pattern directed graph without changing the pattern semantics. Merging is useful in practice for constructing patterns of a higher arity from two or more patterns of smaller arities (see the example in section 4).

3.1 Pattern Subsumption and Equivalence

Pattern subsumption refers to checking when the set of tuples extracted by a pattern is subsumed by the set of tuples extracted by a second (possibly simpler) pattern. Two patterns are equivalent when they subsume each other.

Definition 6. *(Pattern subsumption and equivalence) Let* p_1 *and* p_2 *be two patterns of arity* k. p_1 subsumes p_2, *written as* $p_1 \preceq p_2$, *if and only if for all trees* t, $Ans(p_1, t) \subseteq Ans(p_2, t)$. *If the two patterns mutually subsume each other, i.e.* $p_1 \preceq p_2$ *and* $p_2 \preceq p_1$, *then they are called* equivalent, *written as* $p_1 \simeq p_2$.

In practice, given a pattern p, we are interested in simplifying p to yield a new pattern p' equivalent to p.

Proposition 2. *(Pattern simplification) Let* $p = \langle V, A, U, D, \mu, \lambda_a, \lambda_c \rangle$ *be a pattern and let* $u, v, w \in V$ *be three distinct vertices of* p *such that* $(u, w) \in A$, $\lambda_a((u, w)) = 'c'$, $(u, v) \in A$, *and* $\lambda_a((u, v)) =' n'$. *Let* $p' = \langle V, A', U, D, \mu, \lambda'_a, \lambda_c \rangle$ *be a pattern defined as:*

i) $A' = (A \setminus \{(u, w)\}) \cup \{(v, w)\}$.
ii) If $x \in A \setminus \{(u, w)\}$ *then* $\lambda'_a(x) = \lambda_a(x)$, *and* $\lambda'_a((v, w)) = 'c'$.

Then $p' \simeq p$.

Basically, this proposition says that shifting one position right an arc labeled with $'c'$ in a pattern produces an equivalent pattern. The result follows from the property that for all nodes u, v, w of an ordered tree such that v is the next sibling of u then w is the parent of u if and only if w is the parent of v. Note

[1] The $\circ$ operator denotes function composition.

that if $(v, w) \in A$ then the consistency of p enforces $\lambda_a((v, w)) = 'c'$ and this results in no new arc being added to p'. In this case p gets simplified to p' by deleting arc (u, w).

A pattern p can be simplified to an equivalent pattern p' called normal form.

Definition 7. *(Pattern normal form) A pattern p is said to be in* normal form *if the out-degree of every pattern vertex is at most 1.*

A pattern can be brought to normal form by repeatedly applying the operation described in proposition 2. The existence of a normal form is captured by the following proposition.

Proposition 3. *(Existence of normal form) For every pattern p there exists a pattern p' in normal form such that $p' \simeq p$.*

Note that the application of pattern simplification operation from proposition 2 has the result of decrementing by 1 the number of pattern vertices with out-degree equal to 2. Because the number of pattern vertices is finite and the out-degree of each vertex is at most 2, it follows that after a finite number of steps the resulted pattern will be brought to normal form.

3.2 Pattern Merging

Merging looks at building more complex patterns by combining simpler patterns. In practice we found convenient to learn a set of simpler patterns that share attributes and then merge them into more complex patterns, that are capable to extract tuples of higher arity.

Merging two patterns first assumes performing a pairing of their pattern vertices. Two vertices are paired is they are meant to match identical nodes of the target document. Paired vertices will be fusioned in the resulting pattern.

Definition 8. *(Pattern vertex pairings) Let $p_i = \langle V_i, A_i, U_i, D_i, \mu_i, \lambda_{a_i}, \lambda_{c_i} \rangle$, $i = 1, 2$, be two patterns such that $V_1 \cap V_2 = \emptyset$. The set of vertex pairings of p_1 and p_2 is the maximal set $P \subseteq V_1 \times V_2$ such that:*

i) *For all $d \in D_1 \cap D_2$, $(\mu_1(d), \mu_2(d)) \in P$.*
ii) *If $(u_1, u_2) \in P$, $(u_1, v_1) \in A_1$, $(u_2, v_2) \in A_2$, and $\lambda_{a_1}((u_1, v_1)) = \lambda_{a_2}((u_2, v_2)) =' n'$ then $(v_1, v_2) \in P$.*
iii) *If $(u_1, u_2) \in P$, $w_0 = u_1, w_1, \ldots, w_n = v_1$ is a path in (V_1, A_1) such that $\lambda_{a_1}((w_i, w_{i+1})) = 'n'$ for all $1 \leq i < n - 1$, $\lambda_{a_1}((w_{n-1}, w_n)) = 'c'$, and $w'_0 = u_2, w'_1, \ldots, w'_m = v_2$ is a path in (V_2, A_2) such that $\lambda_{a_2}((w'_i, w'_{i+1})) = 'n'$ for all $1 \leq i < m - 1$, $\lambda_{a_2}((w'_{m-1}, w'_m)) = 'c'$ then $(v_1, v_2) \in P$.*
iv) *If $(u_1, u_2) \in P$, $(v_1, u_1) \in A_1$, $(v_2, u_2) \in A_2$, and $\lambda_{a_1}((u_1, v_1)) = \lambda_{a_2}((u_2, v_2)) =' n'$ then $(v_1, v_2) \in P$.*
v) *If $(u_1, u_2) \in P$, $(v_1, u_1) \in A_1$, $(v_2, u_2) \in A_2$, $\lambda_{a_1}((u_1, v_1)) = \lambda_{a_2}((u_2, v_2)) = 'c'$, and $('f' \in \lambda_{c_1}(v_1) \cap \lambda_{c_2}(v_2)$ or $'l' \in \lambda_{c_1}(v_1) \cap \lambda_{c_2}(v_2))$, then $(v_1, v_2) \in P$.*

Defining vertex pairings according to definition 8 deserves some explanations. Point i) states that if two extraction vertices denote identical attributes then they must be paired. Points ii), iii), iv) and v) identify additional pairings based on properties of ordered trees. Points ii) and iii) state that next-siblings or parents of paired vertices must be paired as well. Point iv) states that previous siblings of paired vertices must be paired as well. Point v) state that first children and respectively last children of paired vertices must be paired as well.

For all pairings (u, v), the paired vertices u and v are fusioned into a single vertex that is labeled with the union of the conditions of the original vertices, assuming that these conditions are not mutually inconsistent.

First, we must define the fusioning of two vertices of a directed graph.

Definition 9. *(Vertex fusioning) Let $G = (V, A)$ be a directed graph and let $u, v \in V$ be two vertices such that $u \neq v$, $(u, v) \notin A$, and $(v, u) \notin A$. The graph $G' = (V', A')$ obtained by fusioning vertex u with vertex v is defined as:*

i) $V' = V \setminus \{v\}$;
ii) *A' is obtained by replacing each arc $(x, v) \in A$ with (x, u) and each arc $(v, x) \in A$ with (u, x).*

Pattern merging involves the repeated fusioning of vertices of the pattern vertex pairings. For each paired vertices, their conditions must be checked for mutual consistency.

Definition 10. *(Pattern merging) Let $p_i = \langle V_i, A_i, U_i, D_i, \mu_i, \lambda_{a_i}, \lambda_{c_i}\rangle$, $i = 1, 2$, be two patterns such that $V_1 \cap V_2 = \emptyset$ and let P be the set of vertex pairings of p_1 and p_2. If for all $(u, v) \in P$ and for all $\sigma_1, \sigma_2 \in \Sigma$, if $\sigma_1 \in \lambda_{c_1}(u)$ and $\sigma_2 \in \lambda_{c_2}(v)$ then $\sigma_1 = \sigma_2$, then the pattern $p = \langle V, A, U, D, \mu, \lambda_a, \lambda_c\rangle$ resulted from* merging *patterns p_1 and p_2 is defined as follows:*

i) *(V, A) is obtained by fusioning u with v for all $(u, v) \in P$ in graph $(V_1 \cup V_2, A_1 \cup A_2)$.*
ii) *$U = U_1 \cup U_2$. $D = D_1 \cup D_2$. If $d \in D_1$ then $\mu(d) = \mu_1(d)$, else $\mu(d) = \mu_2(d)$.*
iii) *For all $(u, v) \in V$ if $u, v \in V_1$ then $\lambda_a((u, v)) = \lambda_{a_1}((u, v))$ else if $u, v \in V_2$ then $\lambda_a((u, v)) = \lambda_{a_2}((u, v))$ else if $u \in V_1$, $v \in V_2$ and $(u, u') \in P$ then $\lambda_a((u, v)) = \lambda_{a_2}((u', v))$ else if $u \in V_2$, $v \in V_1$ and $(v, v') \in P$ then $\lambda_a((u, v)) = \lambda_{a_2}((u, v'))$.*
iv) *If a vertex in $x \in V$ resulted from fusioning u with v then $\lambda_c(x) = \lambda_{c_1}(u) \cup \lambda_{c_2}(v)$, else if $x \in V_1$ then $\lambda_c(x) = \lambda_{c_1}(x)$, else $\lambda_c(x) = \lambda_{c_2}(x)$.*

Essentially this definition says that pattern merging involves performing a pattern vertex pairing (point i)), then defining of the attributes attached to pattern extraction vertices (point ii)) and of the labels attached to vertices (point iv)) and arcs (point iii)) in the directed graph of the resulting pattern.

An example of pattern merging is given in the next section of the paper. Despite these somehow cumbersome but rigorous definitions, pattern merging is a quite simple operation that can be grasped more easily using a graphical representation of patterns (see figure 2).

The next proposition states that the set of tuples extracted by a pattern resulted from merging two or more patterns is equal to the relational natural join of the sets of tuples extracted by the original patterns.

Proposition 4. *(Tuples extracted by a pattern resulted from merging) Let p_1 and p_2 be two patterns and let p be their merging. For all labeled ordered trees t, $Ans(p,t) = Ans(p_1,t) \bowtie Ans(p_2,t)$. $\bowtie$ is the relational natural join operator.*

This result follows by observing that a pattern can be mapped to a conjunctive query over the signature $(child, next, first, last, (tag_\sigma)_{\sigma \in \Sigma})$. Relations $child$, $next$, $first$, $last$ and tag_σ are defined as follows (here $\mathcal{N}$ is the set of tree nodes):

i) $child \subseteq \mathcal{N} \times \mathcal{N}$, $(child(P,C) = true) \Leftrightarrow$ (P is the parent of C).
ii) $next \subseteq \mathcal{N} \times \mathcal{N}$, $(next(L,N) = true) \Leftrightarrow$ (L is the left sibling of N).
iii) $first \subseteq \mathcal{N}$, $(first(X) = true) \Leftrightarrow$ (X is the first child of its parent node).
iv) $last \subseteq \mathcal{N}$, $(last(X) = true) \Leftrightarrow$ (X is the last child of its parent node).
v) $tag_\sigma \subseteq \mathcal{N}$, $(tag(N) = true) \Leftrightarrow$ (σ is the tag of node N).

A pattern vertex is mapped to a logic variable. The query defines a predicate with variables derived from the pattern extraction vertices, one variable per pattern vertex. Merging involves renaming with identical names the variables corresponding to paired pattern vertices and then taking the conjunction of queries corresponding to merged patterns. Now, by simple relational manipulation, it is easy to see that the result stated by proposition 4 holds.

4 An Example

Here we show how the theory developed in the previous sections can be applied to obtain practical wrappers for HTML, implemented as XSLT stylesheets.

First, using the observation stated at the end of the previous section, we redefine L-wrappers ([5, 6]) as sets of patterns. Second, we consider the two single-clause wrappers from [6] and describe them as single-pattern wrappers. Third, we consider the pattern resulted from their merging. Finally, we map the resulting wrapper to XSLT and give arguments for its correctness. Note that in this mapping we are using a subset of XSLT that has a formal semantics ([7]).

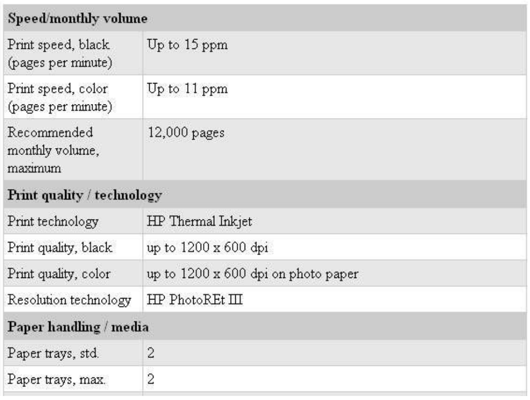
4.1 L-Wrappers as Sets of Patterns

L-wrappers (introduced in [5, 6]) can be redefined using patterns as follows.

Definition 11. *(L-wrapper) An* L-wrapper *of arity k is a set of $n \geq 1$ patterns $W = \{p_i | p_i = \langle V_i, A_i, U_i, D, \mu_i, \lambda_{a_i}, \lambda_{c_i} \rangle$, p_i has arity k, for all $1 \leq i \leq n\}$. The set of tuples extracted by W from a labeled ordered tree t is the union of the sets of tuples extracted by each pattern p_i, $1 \leq i \leq n$, i.e. $Ans(W,t) = \cup_{i=1}^{n} Ans(p_i,t)$.*

In [6] we considered learning L-wrappers for extracting printer information from Hewlett Packard's Web site, using general-purpose ILP. There, we also proposed a generic process for information extraction from the Web that consists of the following stages: page collection, pre-processing, manual information extraction, conversion to the input format of the learning program, learning, wrapper compilation, wrapper execution. In this section we are focusing on the last two stages of this process: wrapping compilation, i.e. the mapping of L-wrappers to XSLT and wrapper execution.

The printer information is represented in multi-section two column HTML tables (see figure 1). Each row contains a pair (feature name, feature value). Consecutive rows represent related features that are grouped into feature classes. For example, there is a row with the feature name ‘Print technology’ and the feature value ‘HP Thermal Inkjet’. This row has the feature class ‘Print quality/technology’. So actually this table contains triples (feature class, feature name, feature value). Some triples may have identical feature classes.



Speed/monthly volume	
Print speed, black (pages per minute)	Up to 15 ppm
Print speed, color (pages per minute)	Up to 11 ppm
Recommended monthly volume, maximum	12,000 pages
Print quality / technology	
Print technology	HP Thermal Inkjet
Print quality, black	up to 1200 x 600 dpi
Print quality, color	up to 1200 x 600 dpi on photo paper
Resolution technology	HP PhotoREt III
Paper handling / media	
Paper trays, std.	2
Paper trays, max.	2

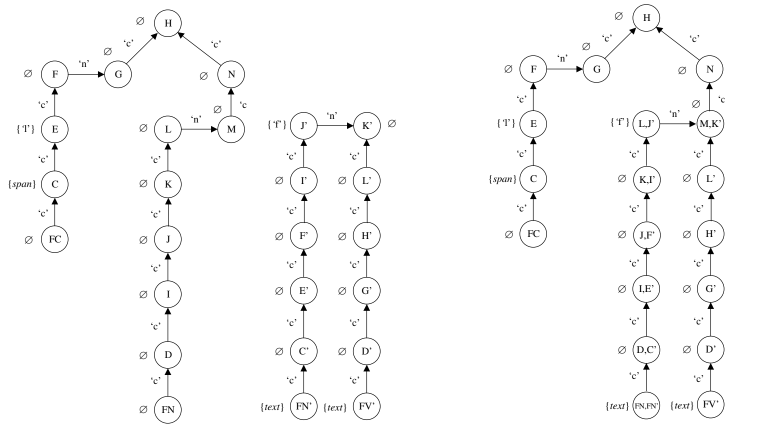
Fig. 1. An XHTML document fragment and its graphic view

In [6] we presented two single-clause L-wrappers for this example that were learnt using FOIL program ([18]): i) for pairs (feature class, feature name); ii) for pairs (feature name, feature value), together with figures of the precision and recall performance measures. The wrappers are (FC = feature class, FN = feature name, FV = feature value):

$$\begin{aligned}&extract(FC, FN) \leftarrow child(C, FC) \wedge child(D, FN) \wedge tag(C, span) \wedge child(E, C) \wedge\\&\quad child(F, E) \wedge next(F, G) \wedge child(H, G) \wedge last(E) \wedge child(I, D) \wedge child(J, I) \wedge\\&\quad child(K, J) \wedge child(L, K) \wedge next(L, M) \wedge child(N, M) \wedge child(H, N).\\&extract(FN, FV) \leftarrow tag(FN, text) \wedge tag(FV, text) \wedge child(C, FN) \wedge child(D, FV) \wedge\\&\quad child(E, C) \wedge child(F, E) \wedge child(G, D) \wedge child(H, G) \wedge child(I, F) \wedge\\&\quad child(J, I) \wedge next(J, K) \wedge first(J) \wedge child(K, L) \wedge child(L, H).\end{aligned}$$

Figure 2 illustrates the two patterns corresponding to the clauses shown above and the pattern resulted from their merging. One can easily notice that these patterns are already in normal form.

The experimental analysis performed in [6] also revealed some difficulties of learning triples (feature class, feature name, feature value) in a straightforward way. The problems were caused by the exponential growth of the number of negative examples, with the tuples arity. The idea of pattern merging presented here



$D_1 = \{Class, Name\}$	$D_2 = \{Name, Value\}$	$D = \{Class, Name, Value\}$
$U_1 = \{FC, FN\}$	$U_2 = \{FN', FV'\}$	$U = \{FC, (FN, FN'), FV'\}$
$\mu_1(Class) = FC$	$\mu_2(Name) = FN'$	$\mu(Class) = FC$
$\mu_1(Name) = FN$	$\mu_2(Value) = FV'$	$\mu(Name) = (FN.FN')$
		$\mu(Value) = FV'$
Pattern p_1	Pattern p_2	p_1 merged with p_2

Fig. 2. Patterns and pattern merging

is an approach of learning extraction patterns of a higher arity that overcomes these difficulties, and thus supporting the scalability of our approach.

4.2 Mapping Wrappers to XSLT

Paper [7] describes a subset of XSLT, called XSLT$_0$, that has a Plotkin-style formal semantics. The reader is invited to consult reference [7], for details on XSLT$_0$, its pseudocode notation and the formal semantics.

The XSLT$_0$ description of the single-pattern wrapper resulted from merging patterns p_1 and p_2 from figure 2 is shown in table 1[2]. The XSLT wrapper is shown in the appendix. XPath expressions xp_1, xp_2 and xp_3 are defined as follows:

xp_1 = `//*/*/preceding-sibling::*[1]/*[last()]/span/node()`
xp_2 = `parent::*/parent::*/parent::*/following-sibling::*[1]/parent::*/*/*/preceding-sibling::*[1][last()]/*/*/*/*/text()`
xp_3 = `parent::*/parent::*/parent::*/parent::*/parent::*/following-sibling::*[1]/*/*/*/*/text()`

The idea is simple: referring to figure 2, we start from the document root, labeled with *html*, then match node H and move downwards to FC, then move back upwards from FC to H and downwards to (FN, FN'), and finally move back upwards from (FN, FN') to (M, K') and downwards from (M, K') to FV'. The wrapper actually extracts the node contents rather than the nodes themselves, using the *content*(.) expression. The extracted information is passed between templates in template variables and parameters *varClass* and *varName*.

[2] Note that our version of XSLT$_0$ is slightly different from the one presented in [7].

Table 1. Description of the sample wrapper in XSLT$_0$ pseudocode

```
template start(html)
    return
        result(selclass($xp_1$))
end

template selclass(*)
    vardef
        varClass := content(.)
    return
        selname($xp_2$,varClass)
end

template selname(*,varClass)
    vardef
        varName := content(.)
    return
        display($xp_3$,varClass,varName)
end

template display(*,varClass,varName)
    vardef
        varValue := content(.)
    return
        triple[class$\rightarrow$ varClass;
               name$\rightarrow$ varName;
               value$\rightarrow$ varValue]
end
```

Note that this technique works for the general case of mapping a pattern to XSLT. As the pattern is a tree, it is always possible to move from a pattern extraction vertex to another via their common descendant in the pattern graph.

Below we give an informal argument of why this transformation works.

The XSLT$_0$ program contains four templates: i) the constructing templates $t_1 = start(html)$, $t_4 = display(*, varClass, varName)$, and ii) the selecting templates $t_2 = selclass(*)$, $t_3 = selname(*,\ varClass)$.

Initially, t_1 is applied. xp_1 selects the nodes that match the feature class. Then, for each feature class, t_2 is applied. xp_2 selects the feature names that are members of a feature class. Then, for each pair (feature class, feature name), t_3 is applied. xp_3 selects the feature values that are related to a pair (feature class, feature name). Then, for each triple (feature class, feature name, feature value), t_4 is applied. It constructs a triple and adds it to the output XML document.

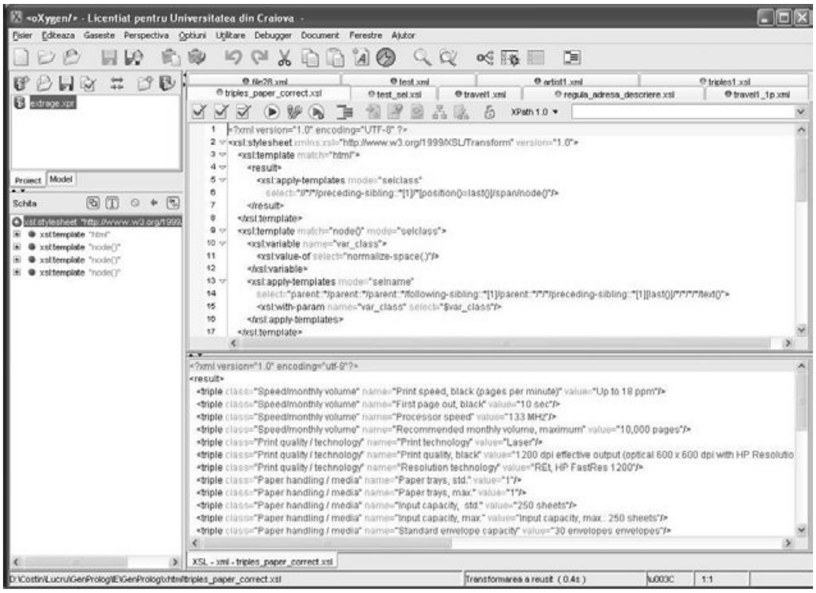
For wrapper execution we can use any of the available XSLT transformation engines. In our experiments we have used Oxygen XML editor ([17], a tool that incorporates some of these engines. Figure 3 illustrates our wrappers in action.

5 Related Work

With the rapid expansion of the Internet and the Web, the field of information extraction from HTML attracted a lot of researchers during the last decade. Clearly, it is impossible to mention all of their work here. However, at least we can try to classify these works along several axes and select some representatives for discussion.

First, we are interested in research on information extraction from HTML using logic representations of tree (rather than string) wrappers that are generated automatically using techniques inspired by ILP. Second, both theoretical and experimental works are considered.

[11] is one of the first papers describing a "relational learning program" called SRV. It uses a FOIL-like algorithm for learning first order information extraction rules from a text document represented as a sequence of lexical tokens. Rule



```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="html">
    <result>
      <xsl:apply-templates mode="selclass"
        select="//*/*/preceding-sibling::*[1]/*[position()=last()]/span/node()"/>
    </result>
  </xsl:template>
  <xsl:template match="node()" mode="selclass">
    <xsl:variable name="var_class">
      <xsl:value-of select="normalize-space(.)"/>
    </xsl:variable>
    <xsl:apply-templates mode="selname"
      select="parent::*/parent::*/parent::*/following-sibling::*[1]/parent::*/*/*/preceding-sibling::*[1][last()]/*/*/*/*/text()">
      <xsl:with-param name="var_class" select="$var_class"/>
    </xsl:apply-templates>
  </xsl:template>
```

```
<?xml version="1.0" encoding="utf-8"?>
<result>
  <triple class="Speed/monthly volume" name="Print speed, black (pages per minute)" value="Up to 18 ppm"/>
  <triple class="Speed/monthly volume" name="First page out, black" value="10 sec"/>
  <triple class="Speed/monthly volume" name="Processor speed" value="133 MHz"/>
  <triple class="Speed/monthly volume" name="Recommended monthly volume, maximum" value="10,000 pages"/>
  <triple class="Print quality / technology" name="Print technology" value="Laser"/>
  <triple class="Print quality / technology" name="Print quality, black" value="1200 dpi effective output (optical 600 x 600 dpi with HP Resolutio
  <triple class="Print quality / technology" name="Resolution technology" value="REt, HP FastRes 1200"/>
  <triple class="Paper handling / media" name="Paper trays, std." value="1"/>
  <triple class="Paper handling / media" name="Paper trays, max." value="1"/>
  <triple class="Paper handling / media" name="Input capacity,  std." value="250 sheets"/>
  <triple class="Paper handling / media" name="Input capacity, max." value="Input capacity, max.: 250 sheets"/>
  <triple class="Paper handling / media" name="Standard envelope capacity" value="30 envelopes envelopes"/>
```

<oXygen/> - Licentiat pentru Universitatea din Craiova
Fisier Editeaza Gaseste Perspectiva Optiuni Utilitare Debugger Document Ferestre Ajutor
file28.xml test.xml artist1.xml triples1.xsl
triples_paper_correct.xsl test_sel.xsl travel1.xml regula_adresa_descriere.xsl travel1_1p.xml
XPath 1.0
extrage.xpr
Project Model
Schita
xsl:stylesheet "http://www.w3.org/1999
xsl:template "html"
xsl:template "node()"
xsl:template "node()"
xsl:template "node()"
XSL - xml - triples_paper_correct.xsl
D:\Costin\Lucru\GenProlog\E\GenProlog\xhtml\triples_paper_correct.xsl
Transformarea a reusit (0.4s)
\u003C
1:1

Fig. 3. Wrapper execution inside Oxygen XML editor

bodies check various token features like: length, position in the text fragment, if they are numeric or capitalized, a.o. SRV has been adapted to learn information extraction rules from HTML. For this purpose new token features have been added to check the HTML context in which a token occurs. The most important similarity between SRV and our approach is the use of relational learning and a FOIL-like algorithm. The difference is that our approach has been explicitly devised to cope with tree structured documents, rather than string documents.

In [8] is described a generalization of the notion of string delimiters developed for information extraction from string documents ([14]) to subtree delimiters for information extraction from tree documents. The paper describes a special purpose learner that constructs a structure called candidate index based on trie data structures, which is very different from FOIL's approach. Note however, that the tree leaf delimiters described in that paper are very similar to our information extraction rules. Moreover, the representation of reverse paths using the symbols $Up(\uparrow)$, $Left(\leftarrow)$ and $Right(\rightarrow)$ can be easily simulated by our rules using the relations *child* and *next*.

In [20] is proposed a technique for generating XSLT-patterns from positive examples via a GUI tool and using an ILP-like algorithm. The result is a NE-agent (i.e. name extraction agent) that is capable of extracting individual items. A TE-agent (i.e. term extraction agent) then uses the items extracted by NE-agents and global constraints to fill-in template slots (tuple elements according to our terminology). The differences in our work are: XSLT wrappers are learnt indirectly via L-wrappers; our wrappers are capable of extracting tuples in a straightforward way, therefore TE-agents are not needed.

In [3] is described Elog, a logic Web extraction language. Elog is employed by a visual wrapper generator tool called Lixto. Elog uses a tree representation of HTML documents (similar to our work) and defines Datalog-like rules with patterns for information extraction. Elog is very versatile by allowing the refinement of the extracted information with the help of regular expressions and the integration between wrapping and crawling via links in Web pages. Elog uses a dedicated extraction engine that is incorporated into Lixto tool.

In [19] is introduced a special wrapper language for Web pages called token-templates. Token-templates are constructed from tokens and token-patterns. A Web document is represented as a list of tokens. A token is a feature structure with exactly one feature having name *type*. Feature values maybe either constants or variables. Token-patterns use operators from the language of regular expressions. The operators are applied to tokens to extract relevant information. The only similarity between our approach and this approach is the use of logic programming to represent wrappers.

In [16] is described the DEByE (i.e. Data Extraction By Example) environment for Web data management. DEByE contains a tool that is capable to extract information from Web pages based on a set of examples provided by the user via a GUI. The novelty of DEByE is the possibility to structure the extracted data based on the user perception of the structure present in the Web pages. This structure is described at example collection stage by means of a GUI metaphor called nested tables. DEByE addresses also other issues needed in Web data management like automatic examples generation and wrapper management. Our L-wrappers are also capable of handling hierarchical information. However, in our approach, the hierarchical structure of information is lost by flattening during extraction (see the printer example where tuples representing features of the same class share the feature class attribute).

As concerning theoretical work, [13] is one of the first papers that analyzes seriously the expressivity required by tree languages for Web information extraction and its practical implications. Combined complexity and expressivity results of conjunctive queries over trees, that also apply to information extraction, are reported in [12].

Finally, in [15] is contained a survey of Web data extraction tools. That paper contains a section on wrapper languages including HTML-aware tools (tree wrappers) and a section on wrapper induction tools.

6 Concluding Remarks

In this paper we studied a class of patterns for information extraction from semi-structured data inspired by logic. This complements our experimental work reported in [4–6]. There, we described how to apply ILP to learn L-wrappers for information extraction from the Web. Here the main focus were properties and operations of patterns used by L-wrappers and L-wrapper implementation using XSLT for information extraction from HTML. The results of this work provide a theoretical basis of L-wrappers and their patterns linking our work with related works in this field. They also show how the combination of general-purpose ILP, L-wrappers and XSLT transformations can be successfully applied to extract information from the Web. We plan to implement this in an information extraction tool. As future theoretical work, we would like to give a formal proof of the correctness of the mapping of L-wrappers to XSLT. As future experimental work we plan to investigate the generality of our approach by applying it on Web sites in other application areas.

References

1. Abiteoul, S., Buneman, P., Suciu, D.: *Data on the Web: From Relations to Semistructured data and XML*, Morgan Kauffman Publishers, (2000).
2. Aleph. `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/aleph.html`.
3. Baumgartner, R., Flesca, S., Gottlob, G.: The Elog Web Extraction Language. In: Nieuwenhuis, R., Voronkov, A. (eds.): *Proceedings of LPAR'2001*, LNAI 2250, Springer Verlag, (2001) 548–560.
4. Bădică, C., Bădică, A.: Rule Learning for Feature Values Extraction from HTML Product Information Sheets. In: Boley, H., Antoniou, G. (eds): *Proc. RuleML'04*, Hiroshima, Japan. LNCS 3323 Springer-Verlag (2004) 37–48.
5. Bădică, C., Popescu, E., Bădică, A.: Learning Logic Wrappers for Information Extraction from the Web. In: Papazoglou M., Yamazaki, K. (eds.) *Proc. SAINT'2005 Workshops. Computer Intelligence for Exabyte Scale Data Explosion*, Trento, Italy. IEEE Computer Society Press (2005) 336–339.
6. Bădică, C., Bădică, A., Popescu, E.: Tuples Extraction from HTML Using Logic Wrappers and Inductive Logic Programming. In: Szczepaniak, P.S., Kacprzyk, J., Niewiadomski, A. (eds.): *Proc.AWIC'05*, Lodz, Poland. LNAI 3528 Springer-Verlag (2005) 44–50.
7. Bex, G.J., Maneth, S., Neven, F.: A formal model for an expressive fragment of XSLT. *Information Systems*, No.27, Elsevier Science (2002) 21–39.
8. Chidlovskii, B.: Information Extraction from Tree Documents by Learning Subtree Delimiters. In: *Proc. IJCAI-03 Workshop on Information Integration on the Web* (IIWeb-03), Acapulco, Mexico (2003) 3–8.
9. Clark, J.: XSLT Transformation (XSLT) Version 1.0, W3C Recommendation, 16 November 1999, `http://www.w3.org/TR/xslt` (1999).
10. Cormen, T.H., Leiserson, C.E., Rivest, R.R.: *Introduction to Algorithms*. MIT Press (1990).
11. Freitag, D.: Information extraction from HTML: application of a general machine learning approach. In: *Proceedings of AAAI'98*, (1998) 517–523.
12. Gottlob, G., Koch, C., Schulz, K.U.: Conjunctive Queries over Trees. In: *Proc.PODS'2004*, Paris, France. ACM Press, (2004) 189–200.
13. Gottlob, G., Koch, C.: Monadic Datalog and the Expressive Power of Languages for Web Information Extraction. In: *Journal of the ACM*, Vol.51, No.1 (2004) 74–113
14. Kushmerick, N., Thomas, B.: Adaptive Information Extraction: Core Technologies for Information Agents, In: *Intelligent Information Agents R&D in Europe: An AgentLink perspective* (Klusch, Bergamaschi, Edwards & Petta, eds.). LNCS 2586, Springer-Verlag (2003).
15. Laender, A.H.F., Ribeiro-Neto, B., Silva, A.S., Teixeira., J.S.: A Brief Survey of Web Data Extraction Tools. In: *SIGMOD Record*, Vol.31, No.2, ACM Press (2002) 84–93.

16. Laender, A.H.F., Ribeiro-Neto, B., Silva, A.S.: DEByE - Data Extraction By Example. In: *Data & Knowledge Engineering* Vol.40, No.2, (2002) 121–154.
17. Oxygen XML Editor. http://www.oxygenxml.com/.
18. Quinlan, J. R., Cameron-Jones, R. M.: Induction of Logic Programs: FOIL and Related Systems, *New Generation Computing*, 13, (1995) 287–312.
19. Thomas, B.: Token-Templates and Logic Programs for Intelligent Web Search. *Intelligent Information Systems*. Special Issue: Methodologies for Intelligent Information Systems, 14(2/3) (2000) 241–261.
20. Xiao, L., Wissmann, D., Brown, M., Jablonski, S.: Information Extraction from HTML: Combining XML and Standard Techniques fro IE from the Web. In: Monostori, L., Vancza, J., Ali, M. (eds.): *Proc. IEA/AIE 2001*. LNAI 2070, Springer-Verlag (2001) 165–174.

A XSLT Code of the Sample Wrapper

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="html">
    <result>
      <xsl:apply-templates mode="selclass" select="//*/*/preceding-sibling::*[1]/
      *[last()]/span/node()"/>
    </result>
  </xsl:template>
  <xsl:template match="node()" mode="selclass">
    <xsl:variable name="var_class"> <xsl:value-of select="normalize-space(.)"/>
    </xsl:variable>
    <xsl:apply-templates mode="selname" select="parent::*/parent::*/parent::*/
      following-sibling::*[1]/parent::*/*/*/preceding-sibling::*[1][last()]/
      */*/*/*/text()">
      <xsl:with-param name="var_class" select="$var_class"/>
    </xsl:apply-templates>
  </xsl:template>
  <xsl:template match="node()" mode="selname">
    <xsl:param name="var_class"/>
    <xsl:variable name="var_name"> <xsl:value-of select="normalize-space(.)"/>
    </xsl:variable>
    <xsl:apply-templates mode="display" select="parent::*/parent::*/parent::*/parent::*/
      parent::*/following-sibling::*[1]/*/*/*/*/text()">
      <xsl:with-param name="var_class" select="$var_class"/>
      <xsl:with-param name="var_name" select="$var_name"/>
    </xsl:apply-templates>
  </xsl:template>
  <xsl:template match="node()" mode="display">
    <xsl:param name="var_class"/>
    <xsl:param name="var_name"/>
    <xsl:variable name="var_value"> <xsl:value-of select="normalize-space(.)"/>
    </xsl:variable>
    <triple>
      <xsl:attribute name="class"> <xsl:value-of select="$var_class"/>
      </xsl:attribute>
      <xsl:attribute name="name"> <xsl:value-of select="$var_name"/>
      </xsl:attribute>
      <xsl:attribute name="value"> <xsl:value-of select="$var_value"/>
      </xsl:attribute>
    </triple>
  </xsl:template>
</xsl:stylesheet>
```

Approximate Subtree Identification in Heterogeneous XML Documents Collections

Ismael Sanz[1], Marco Mesiti[2], Giovanna Guerrini[3], and Rafael Berlanga Llavori[1]

[1] Universitat Jaume I, Castellón, Spain
{berlanga,Ismael.Sanz}@uji.es
[2] Università di Milano, Italy
mesiti@dico.unimi.it
[3] Università di Pisa, Italy
guerrini@di.unipi.it

Abstract. Due to the heterogeneous nature of XML data for internet applications exact matching of queries is often inadequate. The need arises to quickly identify subtrees of XML documents in a collection that are similar to a given pattern. In this paper we discuss different similarity measures between a pattern and subtrees of documents in the collection. An efficient algorithm for the identification of document subtrees, approximately conforming to the pattern, by indexing structures is then introduced.

1 Introduction

The importance of tree-modeled data has spectacularly grown with the emergence of semistructured and XML-based databases. Nowadays, interoperability between systems is achieved through the interchange of XML files, which can represent a great variety of information resources: semi-structured objects, database schemas, concept taxonomies, ontologies, etc. As their underlying data structures use to be tree based, there is a great interest in designing mechanisms for retrieving subtrees according to user requests. In the case of heterogeneous XML collections, these mechanisms must also be approximate, that is, they must retrieve document subtrees that best fit the user requests.

Document heterogeneity poses several challenges to these retrieval systems. Firstly, they must face the problem of dealing with vocabulary discrepancies in the element names (e.g., synonymy, polysemy, etc.). Two elements should match also when their tags are similar relying on a given Thesaurus. Secondly, they must deal with the structural heterogeneity produced by the different DTDs or Schemas behind the stored XML documents. The common assumption of simply weakening the father-children relationships (e.g., the `address` element can be a child or a descendant of the `person` element) is not enough. They should consider the possibility that the father-children relationship is inverted (e.g., the `person` element is a child/descendant of the `address` element) or that the two elements appear as siblings. Moreover, as schemas can also include optional and complex components, structural heterogeneity can appear even for a single schema collection. For these reasons, a user query could be answered by a set of subtrees presenting different structures as well as slight variations in their labels.

In this paper, we stress the structure and tag heterogeneity of XML document collections that can lead to search a very large amount of documents exhibiting weak similarity to a given pattern and we propose an approach for identifying the portions of documents that are similar to the pattern. A pattern is an abstract representation of a user request whose structural constraints are not taken into account in identifying the portions of the target documents in which the nodes of the pattern appear. The structural similarity between the pattern and the identified portions of the target is evaluated as a second step, allowing to rank the identified portions and producing the result.

The proposed approach is thus highly flexible and allows one to choose the desired structural and tag constraints to be taken into account. The problem is however how to perform the first step efficiently, that is, how to efficiently identify *fragments*, i.e. portions of the target containing labels similar to those of the pattern, without relying on strict structural constraints. Our approach employs an ad-hoc data structure, an *inverted index* of the target and a *pattern index* extracted on the fly from the inverted index relying on the pattern labels. By the inverted index, nodes in the target with labels similar to those of the pattern are identified and organized in the levels in which they appear in the target. Fragments are generated by considering the ancestor-descendant relationship among such vertices. Moreover, identified fragments are combined in *regions*, allowing for the occurrence of nodes with labels not appearing in the pattern, if the region shows a higher structural similarity with the pattern than the fragments it is originated from. However, some heuristics are needed to avoid considering all the possible ways of merging fragments into regions and for the efficient computation of similarity.

In the paper, we formally define the notions of fragments and regions and propose the algorithms allowing their identification, relying on our pattern index structure. The use in the approach of different structural similarity functions taking into account different structural constraints (e.g. ancestor-descendant and sibling order) is discussed. The practical applicability of the approach is finally demonstrated by providing experimental results. The contribution of the paper can thus be summarized as follows: (*i*) characterization of different similarity measures between a pattern and regions in a collection of heterogeneous tree structured data; (*ii*) specification of an approach for the efficient identification of regions by specifically tailored indexing structures; (*iii*) realization of a prototype and experimental validation of the approach.

The remainder of the paper is organized as follows. Section 2 formally introduces the notions of pattern, target, fragments, and regions the approach relies on. Section 3 is devoted to discussing different approaches to measure the similarity between the pattern and a region identified in the target. Section 4 discusses how to efficiently identify fragments and regions. Section 5 presents experimental results while Section 6 compares our approach with related work. Finally, Section 7 concludes the paper.

Table 1. Notations

Symbol	**Meaning**
$root(T)$	The root of T
$\mathcal{V}(T)$	The set of vertices of T (i.e., V)
$\|V\|$,$\|T\|$	The cardinality of $\mathcal{V}(T)$
$label(T)$	The label associated with the root of T
$label(v)$,$label(V)$	The label associated with a node v and the nodes in V
$\mathcal{P}(v)$	The parent of vertex v
$desc(v)$	The set of descendants of v ($desc(v) = \{u \| (v,u) \in E^*\}$)
$level(T)$	The depth of T
$level(v)$	The level in which v appears in T
$pre(v)$,$post(v)$	The pre/post-order traversal rank of v
$nca(v,u)$	The nearest common ancestor of v and u
$Dist(v,u)$	The number of vertices from v to u in a pre-order traversal
$d(v)$,d^{max}	$d(v) = Dist(root(T), v)$, $d^{max} = max_{v \in \mathcal{V}(T)} d(v)$

2 Pattern, Target, Fragment and Region Trees

Trees. Following standard notations, a *tree* T is a pair (V,E), where V is a finite set of *vertices* ($root(T) \in V$ is the tree root) and E is a binary relation on V that satisfies the following conditions: (i) the root has no parent; (ii) every node of the tree except the root has exactly one parent; (iii) all nodes are reachable via edges from the root, i.e. $(root(T), v) \in E^*$ for all nodes in V (E^* is the Klein closure of E). u is the *parent* of v if an *edge* (u,v) belongs to E. A node labelling function $label$ assigns to each node in V a label in a set $\mathcal{L}$. Given a tree $T = (V,E)$, Table 1 reports functions/symbols used throughout the paper. In using the notations, the tree T is not explicitly reported whenever it is clear from the context. Otherwise, it is marked as a subscript of the operator.

Document order is determined by a pre-order traversal of the document tree [1]. In a pre-order traversal, a tree node v is visited and assigned its pre-order rank $pre(v)$ before its children are recursively traversed from left to right. A post-order traversal is the dual of the pre-order traversal: a node v is assigned its post-order rank $post(v)$ after all its children have been traversed from left to right. Each node v is thus coupled with a triple $(pre(v), post(v), level(v))$ as shown in Figure 1(a). For the sake of clarity, node triples are reported in the figure only when they are relevant for the discussion. The distance $Dist(v,u)$ between two vertices u, v in the tree is specified as the number of vertices traversed moving from v to u in the pre-order traversal.

Pre- and post- ranking can be used to efficiently characterize the descendants u of v. A node v is a descendant of u, $v \in desc(u)$, iff $pre(v) < pre(u) \wedge post(u) < post(v)$. Given a tree $T = (V,E)$ and two nodes $u, v \in V$, the *nearest common ancestor* of u and v, $nca(u,v)$, is the common ancestor of u and v whose distance to u (and to v) is smaller than the distance to any other common ancestor. Note that $nca(u,v) = nca(v,u)$ and $nca(u,v) = v$ if u is a descendant of v.

Two labels in a tree are *similar* if they are identical, or synonyms relying on a given Thesaurus, or *syntactically similar* relying on a string edit function [2].

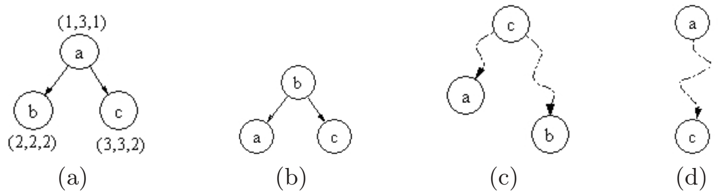


Fig. 1. (a) Pre/Post order rank, a matching fragment with a different order (b), missing levels (c), and missing elements (d)

Let l_1, l_2 be two labels, $l_1 \simeq l_2$ iff: (1) $l_1 = l_2$ or (2) l_1 is a synonym of l_2, or (3) l_1 and l_2 are syntactically similar. Given a label l and a set of labels L, we introduce the operator *similarly belongs*, $\propto$: $l \propto L$ iff $\exists n \in L$ s.t. $l \simeq n$.

Pattern and Target Trees. A pattern is a tree representing a collection of navigational expressions on the target tree (e.g., Xpath expressions in XML documents) or simply a set of labels for which a "preference" is specified on the hierarchical or sibling order in which such labels should occur in the target.

Example 1. Consider the pattern in Figure 1(a). Possible matches are reported in Figure 1(b,c,d). The matching tree in Figure 1(b) contains similar labels but at different positions, whereas the one in Figure 1(c) contains similar labels but at different levels. Finally, the matching tree in Figure 1(d) presents a missing element and both the elements appear at different levels. ○

The target is a set of heterogeneous documents in a source. The target is conveniently represented as a tree whose root is labelled `db` and whose subelements are the documents of the source. This representation relies on the common model adopted by native XML databases (e.g., eXist, Xindice) and simplifies the adopted notations. A target is shown in Figure 2(a).

Definition 1. *(*Target*). Let* $\{T_1, \ldots, T_n\}$ *be a collection of trees, where* $T_i = (V_i, E_i)$, $1 \leq i \leq n$. *A* target *is a tree* $T = (V, E)$ *such that:*

- $V = \cup_{i=1}^{n} V_i \cup \{r\}$, *and* $r \notin \cup_{i=1}^{n} V_i$,
- $E = \cup_{i=1}^{n} E_i \cup \{(r, root(T_i)), 1 \leq i \leq n\}$,
- $label(r) = \texttt{db}$. □

Fragment and Region Trees. The basic building blocks of matchings in our approach are *fragments*. Given a pattern P and a target T, a fragment F is a subtree of T, in which only nodes with labels similar to those in P are considered. Two vertices u, v belong to the same fragment F for a pattern P, iff their labels as well as the label of their common ancestor *similarly belong* to the labels of the pattern. A fragment should belong to a single document of the target.

Definition 2. *(*Fragment Node Set*). A* fragment node set *of a target* T *with respect to a pattern* P *is one of the maximal subsets* V *of* $\mathcal{V}(T)$ *for which* $root(T) \notin V$ *and* $\forall u, v \in V, label(u), label(v), label(nca(u, v)) \propto label(\mathcal{V}(P))$. □

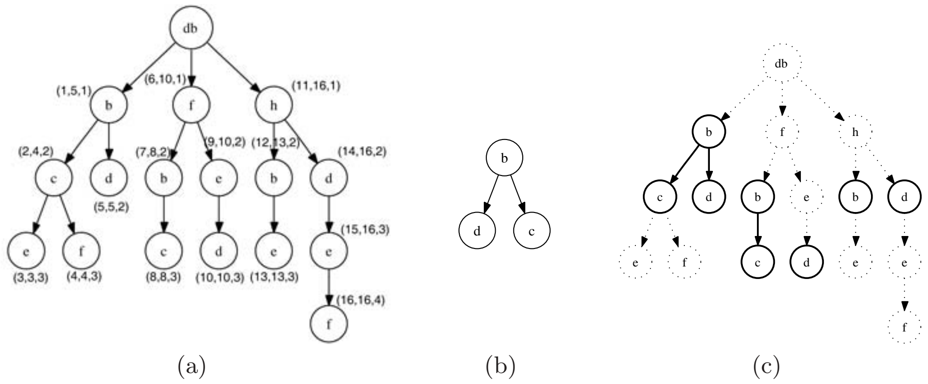


Fig. 2. (a) A target, (b) a pattern, and in bold (c) the corresponding fragments

The tree structure of each fragment is derived from the ancestor-descendant relationship existing among vertices in the target.

Definition 3. *(*Fragment Set*). Let $FN_P(T)$ be the collection of fragment node sets of target T with respect to a pattern P. Every set of vertices $V_F \in FN_P(T)$ defines a* fragment *as the tree $F = (V_F, E_F)$ such that:*

1. *For each $v \in V_F$, $nca(root(F), v) = root(F)$;*
2. *$(u, v) \in E_F$ if u is an ancestor of v in T, and there is no vertex $w \in V_F$, $w \neq u, v$ such that $w \in desc(u)$ and $v \in desc(w)$.* □

Example 2. Consider the pattern in Figure 2(b) and the target in Figure 2(a). The corresponding five fragments are shown in bold in Figure 2(c). ○

Starting from fragments, *regions* are introduced as combinations of fragments rooted at the nearest common ancestor in the target. Two fragments can be merged in a region only if they belong to the same document. In other words, the common root of the two fragments is not the `db` node of the source.

Example 3. Consider as tree T the tree rooted at node $(6, 10, 1)$ in Figure 2(a). Its left subtree contains elements `b` and `c`, whereas its right subtree contains element `d`. T could have a higher similarity with the pattern tree in Figure 2(b) than its left or right subtrees. Therefore, the need arises of combining fragments in regions to return subtrees with higher similarities. ○

Definition 4. *(*Regions*). Let $F_P(T)$ be the set of fragments btw a pattern P and a target T. The corresponding set of regions $R_P(T)$ is defined as follows.*

- *$F_P(T) \subseteq R_P(T)$;*
- *For each $F = (V_F, E_F) \in F_P(T)$ and for each $R = (V_R, E_R) \in R_P(T)$ s.t. $label(nca(root(F), root(R))) \neq$ `db`, $S = (V_S, E_S) \in R_P(T)$, where:*
  - *$root(S) = nca(root(F), root(R))$,*
  - *$V_S = V_F \cup V_R \cup \{root(S)\}$,*
  - *$E_S = E_F \cup E_R \cup \{(root(S), root(F)), (root(S), root(R))\}$.* □

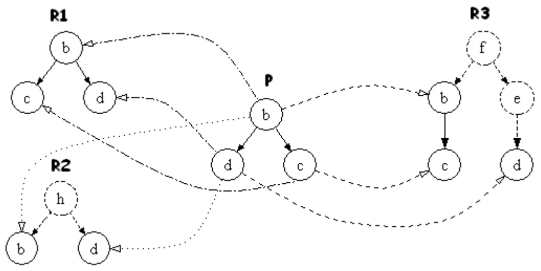


Fig. 3. Identification of different mappings

Figure 3 contains the three regions R_1, R_2, R_3 obtained from the fragments in Figure 2(c). This definition of regions allows one to identify as regions all possible combinations of fragments in a document of the target. This number can be exponential in the number of fragments. In Section 4.2 we will discuss the locality principle to reduce the number of regions to consider.

3 Similarity of a Region w.r.t. a Pattern

In this section we present an approach for measuring the similarity between a pattern and a region. We first identify the possible matches between the vertices in the pattern and the vertices in the region having similar labels. Then, the hierarchical structure of nodes is taken into account to choose, among the possible matches, those that are structurally more similar.

3.1 Mapping Between a Pattern and a Region

A mapping between a pattern and a region is a relationship among their elements that takes the tags used in the documents into account. Our definition differs from the mapping definition proposed by other authors ([3, 4]). Since our focus is on heterogeneous structured data, we do not consider the hierarchical organization of the pattern and the region in the definition of the mapping. We only require that the element labels are similar.

Definition 5. *(*Mapping M*). Let P be a pattern, and R a region subtree of a target T. A mapping M is a partial injective function between the vertices of P and those of R such that $\forall x_p \in \mathcal{V}(P), M(x_p) \neq \perp \Rightarrow label(x_p) \simeq label(M(x_p))$.*$\square$

Example 4. Figure 3 reports the pattern P in the center and the three regions, R_1, R_2, R_3, obtained from the pattern in Figure 2(a). Dashed lines represent the mappings among the vertices of the pattern and of each region. $\bigcirc$

Several mappings can be determined between a pattern and a region thus measures are required to evaluate the "goodness" of a mapping. The similarity between a pattern and a region depends on the similarity between the vertices having similar labels in the two structures. Possible similarity measures $Sim(x_p, x_r)$ between pairs of vertices will be introduced in next section.

Definition 6. *(*Evaluation of a Mapping *M). Let M be a mapping between a pattern P and a region R. The evaluation of M is:*

$$\mathcal{E}val(M) = \frac{\sum_{x_p \in \mathcal{V}(P) s.t. M(x_p) \neq \perp} \mathcal{S}im(x_p, M(x_p))}{max(|\mathcal{V}(P)|, |\mathcal{V}(R)|)} \quad \square$$

Once the evaluation of mappings is computed, we define the similarity between a pattern and a region as the maximal evaluation so obtained.

Definition 7. *(*Similarity between a Pattern and a Region*). Let $\mathcal{M}$ be the set of mappings between a pattern P and a region R. Their similarity is defined as:*

$$\mathcal{S}im(P, R) = max_{M \in \mathcal{M}} \mathcal{E}val(M) \quad \square$$

3.2 Similarity Between Matching Vertices

In this section we present three approaches for computing the similarity between a pair of matching vertices. In the first approach their similarity is one just because we have identified a match in the region. This definition of similarity does not take the structures of the pattern and of the region into account, but just the occurrence of a match. In the second approach, we consider the level at which x_p and $M(x_p)$ appear in the pattern and region structure, respectively. Whenever they appear in the same level, their similarity is equal to the similarity computed by the first approach. Otherwise, their similarity linearly decreases as the number of levels of difference increases. Since two vertices can be in the same level, but not in the same position, a third approach is introduced. Relying on the pre-order traversal of the pattern and the region, the similarity is computed by taking the distance of vertices x_p and $M(x_p)$ with respect to their roots into account. Thus, in this case, the similarity is the highest only when the two vertices are in the same position in the pattern/region.

Definition 8. *(*Similarity between Matching Vertices*). Let P be a pattern, R be a region in a target T, $x_p \in \mathcal{V}(P)$, and $x_r = M(x_p)$. The similarity $\mathcal{S}im(x_p, x_r)$ can be computed as follows:*

1. Match-based similarity*: $\mathcal{S}im_M(x_p, x_r) = 1$;*
2. Level-based similarity*: $\mathcal{S}im_L(x_p, x_r) = 1 - \frac{|level_P(x_p) - level_R(x_r)|}{max(level(P), level(R))}$;*
3. Distance-based similarity*: $\mathcal{S}im_D(x_p, x_r) = 1 - \frac{|d_P(x_p) - d_R(x_r)|}{max(d_P^{max}, d_R^{max})}$.* $\square$

Example 5. Let x_p be the vertex tagged `d` in the pattern P in Figure 3 and x_r^1, x_r^2, x_r^3 the corresponding vertices in the regions R_1, R_2, R_3. Table 2(a) reports the similarity of x_p to the corresponding vertices in the three regions. Since in each region a vertex tagged `d` appears, the match-based similarity is always 1. The level-based similarity is 1 both for region R_1 and R_2 because the vertex tagged `d` in that regions appears at the same level it appears in the pattern. By

Table 2. (a) Similarity of matching vertices (b) Similarity of a pattern with regions

(a)

	$\mathcal{S}im_M$	$\mathcal{S}im_L$	$\mathcal{S}im_D$
x_r^1	1	1	$\frac{2}{3}$
x_r^2	1	1	$\frac{2}{3}$
x_r^3	1	$\frac{2}{3}$	$\frac{1}{5}$

(b)

	$\mathcal{S}im_M$	$\mathcal{S}im_L$	$\mathcal{S}im_D$
R_1	1	1	$\frac{1+\frac{2}{3}+\frac{2}{3}}{3} = \frac{7}{9}$
R_2	$\frac{2}{3}$	$\frac{1+\frac{1}{2}}{3} = \frac{1}{2}$	$\frac{\frac{2}{3}+\frac{2}{3}}{3} = \frac{4}{9}$
R_3	$\frac{3}{5}$	$\frac{3\cdot\frac{2}{3}}{5} = \frac{2}{5}$	$\frac{\frac{4}{5}+\frac{1}{5}+1}{5} = \frac{2}{5}$

contrast, the level-based similarity between x_p and x_r^3 is $\frac{2}{3}$ because x_r^3 is at the third level while x_p is at the second level (thus one level of difference) and the maximal number of levels in P and R_3 is 3 $(1-\frac{1}{3})$. The distance-based similarity between x_p and x_r^1 and x_r^2 is the same because in regions R_1 and R_2 vertex `d` has distance 3 while in the pattern it has distance 2. Moreover, the maximal distance in the pattern/region is the same (3). Things are different for region R_3. Indeed, vertex `d` has distance 5 (which is the maximal) whereas the distance in the pattern is 1. Their distance-based similarity is thus $1 - \frac{5-1}{5}$.
Table 2(b) reports the similarity of P with each region. ○

Proposition 1. *$\mathcal{S}im_M$,$\mathcal{S}im_L$ are order-irrelevant. $\mathcal{S}im_D$ is order-relevant.* □

In the previous definition the context of the two vertices is not taken into account. The "context" is formed by vertices in the neighborhood of x_p and x_r that also match. Such vertices can be sibling, ancestor, or descendant vertices of x_p and x_r. Correction factors can be used to tune the similarity obtained by the previous similarity functions. For space constraints we do not present such factors even if they have already been included in our implementation.

4 Fragment and Region Construction

In this section we present an approach for the construction of regions in a target. The first step is to identify fragments F in the target T that satisfy Definition 3. A peculiarity of fragments is that the labels in the path from $root(F)$ and $root(T)$ do not appear in the pattern P. Thus fragments are disjoint subtrees of T. Then, we merge fragments in regions only when the similarity between P and the generated region is greater than the similarity of P with each single fragment. Thus, regions in the target are single fragments or combination of regions with fragments. The identification of the fragments in the target, their merging in regions, and the evaluation of similarity are efficiency performed by exploiting indexing structures.

4.1 Construction of Fragments

Inverted Index for a Target. Starting from the labels $label(T)$ of a target T, the set is normalized with respect to $\simeq$ obtaining $NL(T) = label(T)_{/\simeq}$.

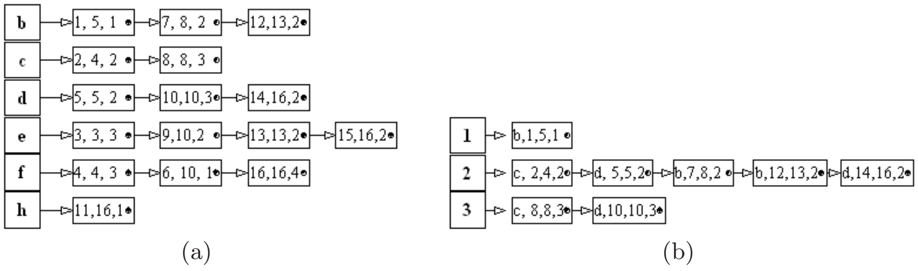


Fig. 4. (a) Inverted index, (b) pattern index

Each label $l \in NL(T)$ is associated with the list of vertices labeled by l or a similar label, ordered relying on the pre-order rank. For each $v \in \mathcal{V}(T)$, the list contains the 4-tuple $(pre(v), post(v), level(v), \mathcal{P}(v))$. Figure 4(a) depicts the inverted index for the target in Figure 2(a). For the sake of graphical readability, the parent of each vertex is not represented (only a $\cdot$ is reported).

Pattern Index. Given a pattern P, for every node v in P, all occurrences of nodes u in the target tree such that $label(v) \simeq label(u)$ are retrieved, and organized level by level in a *pattern index*. The number of levels of the index depends on the the levels in T in which vertices occur with labels similar to those in the patter. For each level, vertices are ordered according to the pre-order rank. Figure 4(b) contains the pattern index for the pattern in Figure 2(b) evaluated on the target in Figure 2(a).

Identification of Fragments from the Pattern Index. Once the pattern index is generated, the fragments are generated through a visit of the structure and the application of the *desc* function. Each node v in the first level of the pattern index is the root of a fragment because, considering the way we construct the pattern index, no other vertices can be the ancestor of v. Possible descendants of v can be identified in the underlying levels. Given a generic level l of the pattern index, a vertex v can be a root of a fragment iff for none of the vertices u in previous levels, v is a descendant of u. If v is a descendant of a set of vertices U, v can be considered the child of the vertex $u \in U$ s.t. $Dist(v, u)$ is minimal.

The developed algorithm visits each vertex in the pattern index only once by marking in each level the vertices already included in a fragment. Its complexity is thus linearly proportional to the number of vertices in the pattern index. Figure 5 illustrates fragments $F_1, \ldots, F_5$ obtained from the pattern index of Figure 4.

Proposition 2. *Let P be a pattern and T be a target. Let K be the maximal size of a level in the pattern index for P. The complexity of fragment construction is $\mathcal{O}(K \cdot |label(P)| \cdot |NL(T)|)$.* □

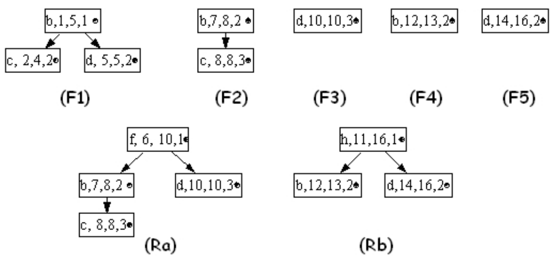


Fig. 5. Construction of fragments and regions

4.2 Construction of Regions

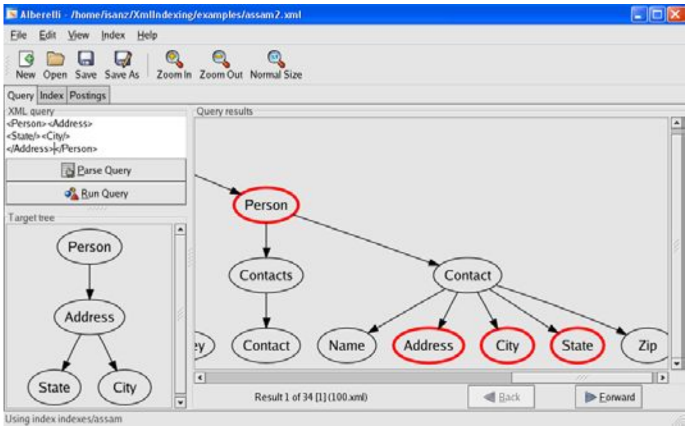
Two fragments should be merged in a single region when, relying on the adopted similarity function, the similarity of the pattern with the region is higher than the similarity with the individual fragments.

Whenever a document in the target is quite big and the number of fragments is high, the regions that should be checked can grow exponentially. To avoid such a situation we exploit the following *locality principle*: merging fragments together or merging fragments to regions makes sense only when the fragments/regions are close. Indeed, as the size of a region tends to be equal to the size of the document, the similarity decreases.

In order to meet such locality principle we evaluate regions obtained by merging adjacent fragments. Operatively, we start from a pair of fragments F, G obtained for a pattern P whose roots have the lowest pre-order rank and identify their common ancestor v. If v is the root of the target, the two fragments cannot be merged. If v is a vertex of the document, the similarity $\mathcal{S}im(P,R)$ is compared with $\mathcal{S}im(P,F)$ and $\mathcal{S}im(P,G)$. If $\mathcal{S}im(P,R)$ is greater than $\mathcal{S}im(P,F)$ and $\mathcal{S}im(P,G)$, F and G are removed and only R is kept. Otherwise, F is kept separate and we try to merge G with the right adjacent fragment.

Example 6. Considering the running example we try to generate regions starting from the fragments in Figure 5. Since the common ancestor between F_1 and F_2 is the root of the target, the two fragments cannot be merged. Since the common ancestor between F_2 and F_3 is a node of the same document, region R_a in Figure 5 is generated. Since the similarity of P with R_a is higher than its similarity with F_2 and F_3, R_a is kept and F_2, F_3 removed. Then, we try to merge region R_a with F_4, but their common ancestor is the root of the target, thus R_a is kept the merging of F_4 and F_5 is evaluated. The region R_b obtained from F_4 and F_5 has an higher similarity than the fragments, thus R_b is kept and F_4 and F_5 are removed. At the end of the process the identified regions are $\{F_1, R_a, R_b\}$. ○

We wish to remark that the construction of regions is quite fast because the target should not be explicitly accessed. All the required information are contained in the inverted indexes. Moreover, thanks to our *locality principle* the number of regions to check is proportional to the number of fragments. Finally, the regions obtained through our process do not present all the vertices occurring



```
<Person> <Address>
<State/> <City/>
</Address></Person>
```

Alberelli - /home/isanz/XmlIndexing/examples/assam2.xml
File Edit View Index Help
New Open Save Save As Zoom In Zoom Out Normal Size
Query Index Postings
XML query
Parse Query
Run Query
Target tree
Person
Address
State
City
Query results
Person
Contacts
Contact
Contact
Name
Address
City
State
Zip
Result 1 of 34 [1] (100.xml)
Back
Forward
Using index indexes/assam

Fig. 6. A screenshot of the GUI prototype on the ASSAM dataset

in the target but only those necessary for the computation of similarity. The evaluation of vertices appearing in the region but not in the pattern is computed through the pre/post order rank of each node.

Example 7. Consider the region R_3 in Figure 3 and the corresponding representation R_b in Figure 5. Vertex e is not explicitly present in R_b. However, its lack can be computed by considering the levels of vertex f and vertex d. ○

5 Prototype and Experimental Results

We have developed a prototype of the system, including a indexer module and a query tool written in Python using the Berkeley DB library. A GTK-based graphical user interface is also available; Figure 6 shows a screenshot. Several aspects of the system have been studied: its performance with respect to the dimension of the dataset, its behavior with respect to structural variations, and its effectiveness in a real dataset.

Performance. Two synthetic datasets and test patterns have been developed with different characteristics: Dataset 1 is designed to contain just a few matching results, embedded in a large number of don't-care nodes (around 7500 relevant elements out of 10^7 elements). Dataset 2 has a high proportion of relevant elements (3×10^5 out of 5×10^5). Moreover, to check the performance of pattern identification for a range of dataset sizes, smaller collections have been extracted from our datasets. The characteristics of each dataset are summarized in Figure 7. Results in Figure 8 show that performance is linearly dependent on the size of the result, and don't-care nodes are effectively discarded.

Effect of Structural Distortions. The second aspect we have evaluated is the effect of structural variations in fragments. In order to test this, we have generated another synthetic dataset, in which we have embedded potential matches of

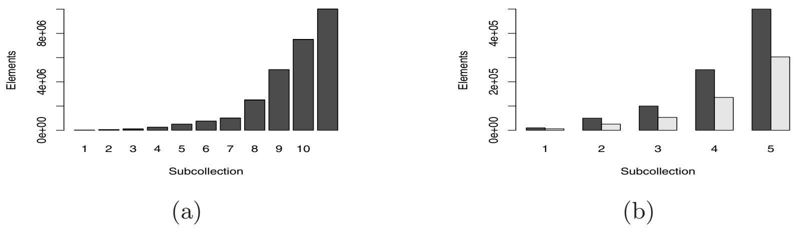


Fig. 7. (a) Total number of elements in each subcollection extracted from Dataset 1 (b) Total number of elements (dark) and number of relevant elements (light) in each subcollection extracted from synthetic Dataset 2

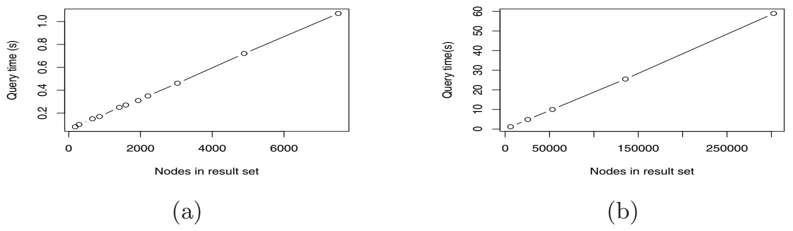


Fig. 8. Execution time in Dataset 1 (left) and Dataset 2 (right)

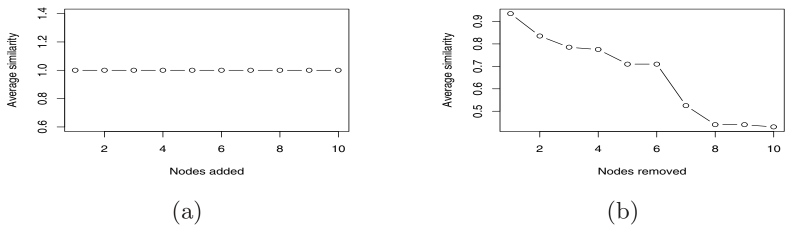


Fig. 9. Change in similarity with the addition and removal of nodes in regions

a 15-element test pattern with controlled distortions of the following kinds: *(1)* addition of n random nodes; *(2)* deletion of n random nodes; *(3)* switching the order of nodes in the same level, with a given probability; *(4)* switching parent and child nodes, with a given probability.

Results in Figure 9 show that added don't-care nodes are ignored by the system, while, predictably, removing relevant nodes in the target does have an effect in the average relevance of results. The results for switching nodes in the same level and for interchanging parent and child nodes are similar to those for nodes addition: fragments with these distortions are successfully retrieved.

Evaluation on the ASSAM Dataset. Preliminar experiments have been conducted on the ASSAM dataset `http://moguntia.ucd.ie/repository/datasets/`, a collection of heterogeneous schemas derived from Web Service descriptions. The original OWL-S schema specifications have been transformed in XML.

Figure 10 shows some sample patterns. The quality of the answers has been evaluated using the traditional information retrieval measures, namely: precision,

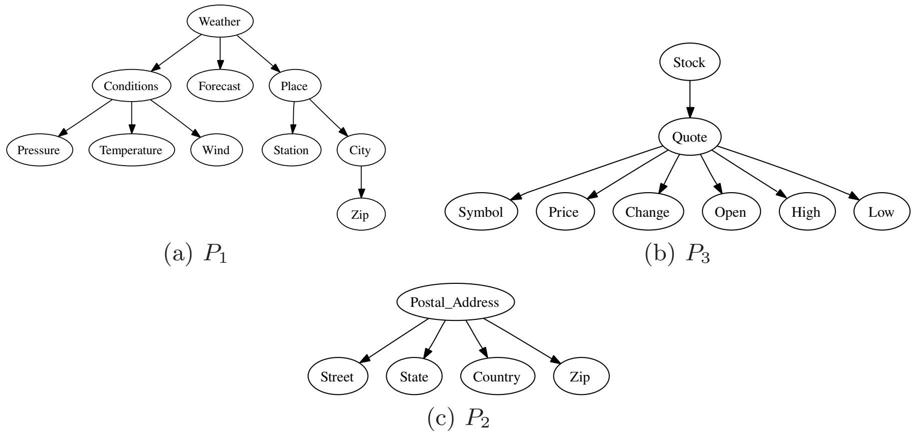


Fig. 10. Sample patterns searched for in the ASSAM dataset

Table 3. Results for the ASSAM datasets

Query	*MinSim*	*Prec.*	*Recall*	F_1
P_1	0.2	1	0.75	0.86
P_2	0.15	0.29	0.7	0.41
P_3	0.2	1	0.8	0.89

recall and the F_1 measure. A threshold $MinSim$ has been considered to discard outliers. $A_{MinSim}(P)$ denotes the results whose Sim_L similarity is greater than $MinSim$. Table 3 shows the results. The first column indicates the pattern, and the second column indicates the similarity threshold used for the answer set. While keeping in mind that these are preliminar tests using a simple, generic distance measure, the results seem encouraging. The low F_1 value for pattern 2 is due to the presence of some of the pattern tags in rather different contexts in the dataset; in a more realistic application, this should be solved by using a more suitable similarity measure.

6 Related Work

In the last decade, there has been a great interest in the *tree embedding* (or *tree inclusion*) problem and its application to semi-structured and XML-databases. Classes of tree inclusion problems are analyzed in [5], providing solutions for the ordered and unordered cases of all of them. The computational cost of the diverse tree-matching primitives varies widely. Simple ordered XPath-like matching has linear complexity, while the unordered version of tree embedding is NP-complete. Flexible and semiflexible matchings have also been proposed in [6] allowing a path in the pattern to match with a path in the target where nodesw appear in a different order. *Ranked tree-matching* approaches have been proposed as well in the same spirit of Information Retrieval (IR) approaches, where approximate answers are ranked on their matching scores. In these approaches, instead of generating all candidate subtrees, the algorithms return a ranked list of "good enough" matches. In [7] a dynamic programming algorithm for ranking query results according to a cost function is proposed. In [8] a data pruning algorithm is proposed where intermediate query results are filtered dynamically during evaluation process. `ATreeGrep` [9] uses as basis an exact matching algorithm, but allowing a fixed number of "differences" in the result.

As these approaches, ours also returns a ranked list of "good enough" subtree matches. However, our approach is highly flexible because it allows choosing the most appropriate structural similarity measures according to the application semantics. All the considered ranked approaches, indeed, enforce at least the ancestor-descendant relationship in pattern retrieval. Moreover, our approach also includes approximate label matching, which allows dealing with heterogeneous tag vocabularies. The proposed tool can thus be used in a wide range of tree-data based applications.

The retrieval of XML documents has also been investigated in the IR area [10]. These approaches mainly focus on the textual part of XML documents, so that the XML structure is used to guide user queries, and to improve the retrieval effectiveness of content indexes. However, current INEX evaluation collections present little heterogeneity in both the tags and structures `http://inex.is.informatik.uni-duisburg.de/2005/`. In the future, the combination of content-based IR techniques with the method proposed in this paper will allow us to extend the range of applications to large text-rich collections with many variations in tag names, structures and content.

Finally, some specific structural similarity measures have been proposed in the areas of Schema Matching [11] and more recently Ontology Alignment. In the former tree/graph-matching techniques allow determining which target schema portions can be mapped to the source ones. In this context, node matching depends on datatypes and domain constraints. However, schemas use to present simple structures that seldom exceed three depth levels (relation-attribute-type). In the latter, the problem consists in finding out which concepts of a target ontology can be associated to concepts of the source ontology, relying on neighborhood and cardinality constraints. However, there are few proposals to state useful tree-based similarity measures that can help this process.

7 Conclusions and Future Work

In this paper we have developed an approach for the identification of subtrees similar to a given pattern in a collection of highly heterogeneous tree structured documents. In this context, the hierarchical structure of the pattern cannot be employed for the identification of the target subtrees but only for their ranking. Peculiarities of our approach are the support for tag similarity relying on a Thesaurus, the use of indexing structures to improve the performance of the retrieval, and a prototype of the system.

As future work we plan to compare different similarity measures in order to identify those more adequate depending on the application context and the heterogeneity of the considered data. Such measures can be collected in a framework of functions that a user can select, compose, and apply depending on her needs. Moreover we plan to consider more sophisticated patterns in which further constraints on vertices and edges can be stated. For instance, an element or a portion of the pattern could be requested to mandatorily appear in the target regions, or the difference between the levels in which two elements appear could be constrained by fixing a threshold. The constraints should then be considered in the similarity evaluation. Finally, we wish to consider subtree identification in a collection of heterogeneous XML Schemas. In this context, the proposed approach should be tailored to the typical schema constraints (e.g., optionality and repeatability of elements, groups, types).

References

1. Grust, T.: Accelerating XPath Location Steps. In: ACM SIGMOD International Conference on Management of Data. (2002) 109–120
2. Wagner, R.A., Fischer, M.J.: The String-to-string Correction Problem. Journal of the ACM **21** (1974) 168–173
3. Nierman, A., Jagadish, H.V.: Evaluating Structural Similarity in XML Documents. In: 5th International Workshop on the Web and Databases. (2002) 61–66
4. Buneman, P., Davidson, S.B., Fernandez, M.F., Suciu, D.: Adding Structure to Unstructured Data. In: 6th International Conference on Database Theory. Volume 1186 of LNCS. (1997) 336–350
5. Kilpeläinen, P.: Tree Matching Problems with Applications to Structured Text Databases. PhD thesis, Dept. of Computer Science, University of Helsinki (1992)
6. Kanza, Y., Sagiv, Y.: Flexible Queries Over Semistructured Data. In: 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. (2001)
7. Schlieder, T., Naumann, F.: Approximate Tree Embedding for Querying XML Data. In: ACM SIGIR Workshop On XML and Information Retrieval. (2000)
8. Amer-Yahia, S., Cho, S., Srivastava, D.: Tree Pattern Relaxation. In: 8th International Conference on Extending Database Technology. Volume 2287 of LNCS. (2002) 496–513
9. Shasha, D., Wang, J.T.L., Shan, H., Zhang, K.: ATreeGrep: Approximate Searching in Unordered Trees. In: 14th International Conference on Scientific and Statistical Database Management. (2002) 89–98
10. Luk, R.W., Leong, H., Dillon, T.S., Chan, A.T., Croft, W.B., Allen, J.: A Survey in Indexing and Searching XML Documents. Journal of the American Society for Information Science and Technology **53** (2002) 415–438
11. Rahm, E., Bernstein, P.A.: A Survey of Approaches to Automatic Schema Matching. The VLDB Journal **10** (2001) 334–350

A Framework for XML-Based Integration of Data, Visualization and Analysis in a Biomedical Domain

Nathan Bales, James Brinkley, E. Sally Lee, Shobhit Mathur, Christopher Re, and Dan Suciu

University of Washington

Abstract. Biomedical data are becoming increasingly complex and heterogeneous in nature. The data are stored in distributed information systems, using a variety of data models, and are processed by increasingly more complex tools that analyze and visualize them. We present in this paper our framework for integrating biomedical research data and tools into a unique Web front end. Our framework is applied to the University of Washington's Human Brain Project. Specifically, we present solutions to four integration tasks: definition of complex mappings from relational sources to XML, distributed XQuery processing, generation of heterogeneous output formats, and the integration of heterogeneous data visualization and analysis tools.

1 Introduction

Modern biomedical data have an increasingly complex and heterogeneous nature, and are generated by collaborative yet distributed environments. For both technical and sociological reasons these complex data will often be stored, not in centralized repositories, but in distributed information systems implemented under a variety of data models. Similarly, as the data becomes more complex, the tools to analyze and visualize them also become more complex, making it difficult for individual users to install and maintain them.

The problem we are addressing is how to build a uniform Web interface that (a) gives users integrated access to distributed data sources, (b) allows users to formulate complex queries over the data without necessarily being competent in a query language, (c) allows access to existing visualization tools which do not need to be installed on the local workstation, and (d) allows control of existing data analysis tools, both for data generation, and processing of query results.

Our specific application is the integration of data sources containing multi-modality and heterogenous data describing language organization in the brain, known as the University of Washington's Human Brain Project [7]. The Web front end is targeted towards sophisticated and demanding users (neuroscience researchers). We examine in this paper the components that are needed to perform such a data integration task, and give a critical assessment of the available XML tools for doing that.

We have identified a few data management problems that need to be addressed in order to achieve integration: complex mappings from relational sources to XML, distributed XQuery processing, graphical XQuery interfaces, generation of heterogeneous output formats, and integration of data visualization and analysis tools. The main contribution in this paper is to describe our framework for achieving the integration of data, queries, visualization, and analysis tools. Specifically, we make the following contributions:

Complex Relational-to-XML Mapping In our experience, writing complex mappings from relational data to XML data was one of the most labor intensive tasks. We propose a simple extension to XQuery that greatly simplifies the task of writing complex mappings.

Distributed XQuery We identify several weaknesses of the *mediator* model for data integration, and propose an alternative, based on a *distributed query language*. It consists of a simple extension to XQuery, called XQueryD [28], which allows users to distribute computations across sites.

Heterogeneous Output Formats Users want to map the data into a variety of formats, either for direct visualization, or in order to upload to other data processing tools (spreadsheets, statistical analysis tools, etc). We describe a simple interface to achieve that.

Heterogeneous Visualization Tools We propose an approach for integrating multiple data visualization tools, allowing their outputs to be incorporated into query answers.

2 Application Description

The driving application for this work is the University of Washington Integrated Brain Project, the goal of which is to develop methods for managing, sharing, integrating and visualizing complex, heterogeneous and multi-modality data about the human brain, in the hope of gaining a greater understanding of brain function than could be achieved with a single modality alone [7]. This project is part of the national Human Brain Project (HBP) [21], whose long-term goal is to develop interlinked information systems to manage the exploding amount of data that is being accumulated in neuroscience research.

Within the UW HBP the primary data are acquired in order to understand language organization in the brain. Because each type of data is complex, we have developed and are continuously developing independent tools for managing each type, with the belief that each such tool will be useful to other researchers with similar types of data, and with the aim of integrating the separate tools, along with external tools, in a web-based data integration system that relies on XML as the medium of data exchange.

The web-based integration system we are developing is called XBrain [35, 36]. The components that this system seeks to integrate include data sources, visualization tools and analysis tools.

2.1 Data Sources

We describe here three of the data sources in XBrain, which illustrate data stored in three different data models: relational, ontology, and XML.

A Relational Database: CSM (Cortical Stimulation Mapping)**.** This is a patient-oriented relational database stored in MySQL, which records data obtained at the time of neurosurgery for epilepsy. The data primarily represent the cortical locations of language processing in the brain, detected by noting errors made by the patient during electrical stimulation of those areas. The database also contains the file locations of image volumes, 3-D brain models, and other data needed in order to reconstruct a model of the brain from MRI images, and to use that model as a basis for calculating the locations of the language sites. Data are entered by means of a web-based application [20], but only minimal browse-like queries were supported by the legacy application. The database has 36 tables containing 103 patients, and is 8MB in size.

An Ontology: FMA (Foundational Model of Anatomy)**.** This ontology is the product of a separate, major research project conducted over more than ten years at the University of Washington [30]. The FMA is a large semantic network containing over 70,000 concepts representing most of the structures in the body, and 1.2 million relationships, such as `part-of`, `is-a`, etc. The FMA relates to the CSM database through the names of anatomical brain regions where the stimulation sites are located: e.g. FMA could be used to find neighboring, contained, or containing regions of specific stimulation sites in CSM. The FMA ontology is stored and managed in Protege[1], which is a general purpose ontology managing system, and does not support a query language. A separate project [27] built a query interface to FMA, called OQAFMA, which supports queries written in StruQL [15], a query language specifically designed for graphs.

An XML File: IM (Image Manager)**.** As part of an anatomy teaching project we have developed a tool for organizing teaching images [5]. Each image has associated with it one or more annotation sets, consisting of one or more annotations. An annotation consists of a closed polygon specified by a sequence of image coordinates on the image and an anatomical name describing that region. As in the CSM database, the names are taken from the FMA. In the original project the data is stored in a relational database. For the purpose of integrating it in XBrain we converted it into a single XML document, because it is infrequently updated because it has a natural recursive structure. To query it, we use the Galax [13] XQuery interpretor.

2.2 Visualization Tools

Many tools for Web-based visualization and interaction with both 2-D and 3-D images have been developed, in our lab and elsewhere. For example, we have developed interactive tools for 2-D images [6], and tools for 3-D visualization of CSM and other functional language data mapped onto a 3-D model of a patient

[1] `http://protege.stanford.edu/`

or population brain [26]. These tools are being integrated as part of the XBrain project. While each tool is designed to display a single image at a time, in XBrain we allow users to integrate images generated by several visualization tools with the data returned by queries.

2.3 Analysis Tools

Finally, users are sophisticated, and they generally develop or use various analysis tools to generate the data that are entered into the various data sources, or to further process the results of a query. An example tool for the UW HBP is the Visualization Brain Mapper (VBM) [19], which accepts a specification file generated from the CSM database, then creates a mapping of stimulation sites onto a generated 3-D model. A second example is our X-Batch program [18] which provides a plugin to a popular functional image analysis program while transparently writing to a backend database. These tools are being integrated into XBrain, by having queries generate appropriate data formats for them.

2.4 The Problem

The UW HBP data sources and tools illustrate the increasingly complex and heterogenous nature of modern biomedical data, as well as the increasingly collaborative yet distributed environment in which they are generated. These complex data are stored, not in a centralized repository, but in distributed information systems implemented under a variety of data models. The tools to analyze and visualize them are also quite complex, making it difficult for individual users to install and maintain them. Thus, the problem we are addressing is how to build a uniform Web interface that (a) gives users integrated access to distributed data sources, (b) allows users to formulate complex queries over the data without necessarily being competent in a query language, (c) allows access to existing visualization tools which do not necessarily need to be installed on the local workstation, and (d) allows control of existing data analysis tools, both for data generation, and processing of query results. XBrain is our proposed framework for addressing these problems.

3 The XBrain Integration Architecture

Our architecture is shown in Fig. 1. All sources store data in their native format and have to map the data to XML when exported to the query processor. Most mapped sources accept XQuery over their data, with one exception: OQAFMA accepts StruQL queries, because the rich structure of the ontology describing all of human anatomy requires a richer language than XQuery for recursive path traversals in the ontology graph. The data from all sources are integrated by a module supporting a distributed extension of the XQuery language (XQueryD). This module sends queries to the local sources and integrates the resulting XML data fragments. The resulting XML query answer can be presented to the user

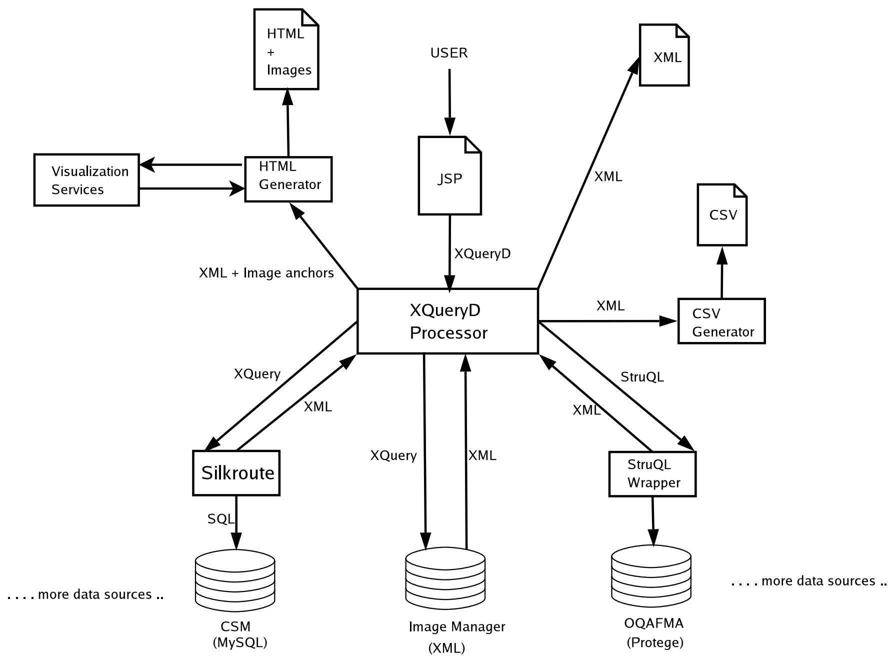


Fig. 1. The XBrain Integration Architecture.

in one of multiple formats: as a plain XML file, as a CSV (Comma Separated Values) file, or a nested HTML file. In the latter case, the image anchors embedded in the XML file are interpreted by calling the appropriate image generation Webservices, and the resulting HTML pages together with complex images is presented to the user. The user inputs queries expressed in XQueryD through a JSP page. We describe the specific data management tasks next.

4 Specific Data Management Tasks

4.1 Mappings to XML

We mapped the relational CSM database to XML using SilkRoute [11, 35]. To define the map, one needs to write an XQuery program that maps the entire relational database to a virtual XML document, called the *public view*. Users query this view using XQuery, which SilkRoute translates to SQL, then converts the answers back to XML. For example, the user query below finds the names of all structures over all patients, in which a CSM error of type 2 (semantic paraphasia) occurred at least once in one patient:

```
<results>
 {for $trial in PublicView("Scrubbed.pv")/patient/surgery/csmstudy/trial
  where $trial/trialcode/term/abbrev/text()="2"
  return $trial/stimsite/name()
 }
</results>
```

```
ExprSingle    ::= TblExpr | ... (* all expressions in XQuery remain here *)

TblExpr       ::= NameClause WhereClause? OmitClause? RenameClause? ReturnClause?

NameClause    ::= "table " TblName (" as " <NCName>)?
OmitClause    ::= "omit " ColName (", " ColName)*
RenameClause  ::= "rename " ColName as <NCName> (", " ColName as <NCName>)*
ReturnClause  ::= "return " EnclosedExpr

TblName       ::= <NCName>
ColName       ::= <NCName>
FunctionCall  ::= <QName "("> (ExprSingle ("," ExprSingle)*)? ")" ("limit" IntegerLiteral)?
```

Fig. 2. The Grammar for RXQuery.

Here `PublicView` indicates the file containing the public view definition (a large XQuery). SilkRoute translates the query automatically into a complex SQL statement:

```
  SELECT stimsite.name
  FROM  trial, csm, . . ., stimsite
  WHERE term.type = 'CSM error code' AND abbrev = '2' AND  . . .
```

Writing the public view was a major task. For XBrain, it had 583 lines of XQuery code, which was repetitive, boring to write, and error prone. We needed a more efficient tool to write such mappings, in order to easily extend XBrain to other sources. For that, we developed a simple extension to XQuery that allows the easy specification of complex mappings from relational data to XML.

RXQuery: A Language for Mapping Relations to XML. Our new language allows users to concisely specify complex mappings from relational databases to XML such that (1) default mappings are done automatically, by using the relational database schema, and (2) the user can override the defaults and has the full power of XQuery. The grammar is shown in Fig. 2. It extends the XQuery syntax with seven new productions, by adding "table expressions", `TblExpr` to the types of expressions in the language. The RXQuery preprocessor takes as input a relational database schema and an RXQuery expression and generates an XQuery expression that represents a public view.

Example 1. We illustrate RXQuery with three examples, shown in Fig. 3. In all of them we use a simple relational database schema consisting for the two relations below, which are a tiny, highly simplified fragment of CSM:

```
  patient(pid, name, dob, address)
  surgery(sid, pid, date, surgeon)
```

Consider $Q1$ in Fig. 3 and its translation to XQuery. By default, every column in the relational schema is mapped into an XML element with the same name. Here `CanonicalView()` is a SilkRoute function that represents the canonical view of the relational database.

	RXQuery	Translation to XQuery

$Q1$	table patient	for $Patient in CanonicalView()/patient
		return
		  <patient>
		    <pid> { $Patient/pid/text() } </pid>
		    <name> { $Patient/name/text() } </name>
		    <dob> { $Patient/dob/text() } </dob>
		    <address> { $Patient/address/text() } </address>
		  </patient>

$Q2$	table patient
	  omit name
	  rename dob as @date-of-birth
	where $Patient/dob/text() < 1950
	return
	  <age>
	    { 2005 - $Patient/dob/text() }
	  </age>
		for $Patient in CanonicalView()/patient
		where $Patient/dob/text() < 1950
		return
		  <patient date-of-birth = ‘‘$Patient/dob/text()’’>
		    <pid> { $Patient/pid/text() } </pid>
		    <address> { $Patient/address/text() } </address>
		    <age> { 2005 - $Patient/dob/text() } </age>
		  </patient>

$Q3$	table patient
	return
	  table surgery omit pid
	  where $Patient/pid/text() =
	    $Surgery/pid/text()
		for $Patient in CanonicalView()/patient
		return
		  <patient>
		    <pid> { $Patient/pid/text() } </pid>
		    <name> { $Patient/name/text() } </name>
		    <dob> { $Patient/dob/text() } </dob>
		    <address> { $Patient/address/text() } </address>
		    { for $Surgery in CanonicalView()/surgery
		      where $Patient/pid/text() = $Surgery/pid/text()
		      return <surgery>
		            <sid> $Surgery/sid/text() </sid>
		            <date> $Surgery/date/text() </date>
		            <surgeon> $Surgery/surgeon/text()
		            </surgeon>
		          </surgery>
		  </patient>

Fig. 3. Examples of RXQuery and their translations to XQuery.

Query $Q2$ illustrates the `omit` and the `rename` keywords that omit and/or rename some of these attributes, and the `where` and the `return` clauses that allow the user to restrict which rows in the table are to be exported in XML and to add more subelements to each row. In $Q2$, the `name` column is omitted, the `dob` column is exported as the `@date-of-birth` attribute, rather than the default `dob` element, and an additional element `age` is computed for each row.

RXQuery is especially powerful when specifying complex, nested public views, which is the typical case in practice. $Q3$ is a very simple illustration of this power. Here, the nested subquery is a simple `table` expression, which is expanded automatically by the preprocessor into a complex subquery.

In addition to the features illustrated in the example, RXQuery includes functions, which we have found to be important in specifying complex mappings, since parts of the relational database need to be included several times in the XML document. The `limit n` clause (see Fig. 2) represents a limit on the recursion depth, when the function is recursive: this allows us some limited form recursive XML views over relational data (SilkRoute does not support recursive XML structures).

One measure of effectiveness of RXQuery is its conciseness, since this is correlated to the readability and maintainability of the public views. Fig. 4 reports

PV for CSM:

Public View	Lines	Words	Chars
XQuery(manual)	583	1605	28582
RXQuery(w/o functions)	141	352	4753
RXQuery(w/ functions)	125	303	4159
XQuery(generated)	634	1633	34979

PV for IM:

Public View	Lines	Words	Chars
XQuery(manual)	1383	3419	64393
RXQuery(w/o functions)	381	1178	14987
RXQuery(w/ functions)	151	427	5603
XQuery(generated)	1427	3575	66105

Fig. 4. Two examples of large public views defined in RXQuery: on the CSM database and on the original relational version of the IM (Image Manager) database. The tables show the original, manual definition of the public view in XQuery, the definition in RXQuery without functions, the same with functions, and the resulting, automaticallly generated XQuery.

the number of lines for two public views: for CSM and for the original version of IM (which is in a relational database). The CSM public view became about 5 times smaller, shrinking from 583 lines in XQuery to 125 lines in RXQuery. The IM public view shrank from an original XQuery with 1383 lines of code to an RXQuery expression with only 151 lines. In both examples, the XQuery public view generated automatically by the RXQuery preprocessor was only sightly larger than the original manual public view.

Mapping Other Data Sources to XML. While most data sources can be mapped to XML in a meaningful way, sometimes this is not possible. In such cases we decided to keep the original data model, rather than massaging it to an artificial XML structure. The Foundational Model of Anatomy (FMA) is a rich ontology, which is best represented as a graph, not a tree. We kept its query interface, OQAFMA, which uses StruQL as a query language and allows users to express complex recursive navigation over the ontology graph. For example, the StruQL query below returns all anatomical parts that contain the middle part of the superior temporal gyrus:

```
  WHERE  Y->":NAME"->"Middle part of superior temporal gyrus",
         X->"part"*->Y,
         X->":NAME"->Parent
  CREATE  Concept(Parent);
```

The query computes a transitive closure of the `part` relationship. While the query data model is best kept as a graph, the query answers can easily be mapped back into XML. In our example, the answer returned by OQAFMA is:

```
<results> <Concept> <Ancestor>Neocortex</Ancestor>     </Concept>
          <Concept> <Ancestor>Telencephalon</Ancestor> </Concept>
  . . . . . . . .
</results>
```

Finally, native XML data are queried directly using XQuery. In our case, the Image Manager data (IM) is stored in XML and queried using Galax [13]. The following example finds all images annotated by the *middle part of the superior temporal gyrus*:

```
for $image in document("image_db.xml")//image
where $image/annotation_set/image_annotation/name/text() =
                                    "middle part of the superior temporal gyrus"
return <image>  {$image/oid}  </image>
```

```
ExprSingle   ::= "execute at" <URL>  [ "xquery" { ExprSingle }  |  "foreign" { String } ]
                    ( "handle" <VAR>:<NAME-SPACE> <EXPR> )*
```

Fig. 5. Grammar for XQueryD.

4.2 Distributed XQuery Processing

The standard approach to data integration is based on a mediator, an architecture proposed by Gio Wiederhold [38]. With this approach, a single mediator schema is first described over all sources, and all local sources are mapped into the mediated schema.

We found this approach too heavy duty for our purpose, for three reasons. First, mediators are best suited in cases when the same concept appears in several sources, and the mediated concept is the set union of the instance of that concept at the sources. For example, BioMediator, a mediator-based data integration project for genetic data [32], integrates several sources that have many overlapping concepts: e.g. most sources have a `gene` class, and the mediator defines a global `gene` class which is the logical union of those at the local sources; similarly, most sources have a `protein` concept, which the mediator also unions. By contrast, in XBrain the concepts at the sources are largely disjoint, and the mediated schema would trivially consist of all local schemas taken together, making the mediator almost superfluous.

The second reason is that mediator based systems require a unique data model for all sources, in order to be able to perform fully automatic query translation. They also hide the schema details at sources from the user, allowing inexperienced users to access large numbers of data sources. None of these applies to XBrain: some sources (like FMA) are best kept in their native datamodel, which is not XML, and our sophisticated users are quite comfortable with the details of the source schemas.

Finally, despite fifteen years of research, there are currently no widely available, robust tools for building mediator systems.

Our approach in XBrain is different, and is based on a distributed evaluation of XQuery. All local sources are fully exposed to the users, who formulate XQuery expressions over them.

XQueryD: A Distributed XQuery Language. The goal is to allow users to query multiple sources in one query. While this can already be done in XQuery, it supports only the *data shipping* model (through the `document()` function): it fetches all data sources to a single server, then runs the query there. This is a major limitation for many applications, especially when some data sources are very large, or when a data source is only a virtual XML view over some other logical data model. For example, our CSM data source is not a real XML document, but a virtual view over a relational database. If we materialized it, the 8MB relational database becomes a 30MB XML document; clearly, it is very inefficient to fetch the entire data with `document()`. We propose a simple extension to XQuery that allows *query shipping* to be expressed in the language,

```
for $image in document("image_db.xml")//image
let $region_name := execute at "http://csm.biostr.washington.edu/axis/csm.jws"
      xquery { for $trial in PublicView("Scrubbed.pv")/patient/surgery/csmstudy/trial
               where $trial/trialcode/term/abbrev/text()="2"
               return $trial/stimsite/name()
             },
    $surrounding_regions :=
        for $term in $region_name
        return <term> {(execute at "http://csm.biostr.washington.edu/oqafma"
                        foreign  {WHERE Y->":NAME"->"$term",
                                        X->("part")*->Y,
                                        X->":NAME"->Ancestor
                                  CREATE  Concept(Ancestor); }
                        )/results/Concept/text()
                      }
               </term>
where $image/annotation_set/image_annotation/name/text() = $surrounding_regions/text()
return $image/oid/text()
```

Fig. 6. Example of a query in XQueryD.

in addition to data shipping. The language consists of a single new construct added to `ExprSingle`, and is shown in Fig. 5

Example 2. We illustrate XQueryD with one single example. The query in Fig. 6 integrates three sources: the Image database (XML), the CSM databases (relational), and the OQAFMA database (ontology). The query starts at the image database, and iterates over all images. The `execute at` command instructs the query processor to send the subsequent query to a remote site for execution: there are two remote queries in this example. The query returns all images are annotated by an anatomical name that is part of the anatomical region surrounding any language site with error type 2.

We initially implemented XQueryD by modifying Galax to accept the new constructs, which was a significant development effort. Once new versions of Galax were released, we found it difficult to keep up with Galax' code evolution. We are currently considering implementing a translator from XQueryD to XQuery with Webservice calls that implement the `execute` statements. For the future, we argue for the need of a standard language extension of XQuery to support distributed query processing in query shipping mode.

Discussion. There is a tradeoff between the mediator-based approach and the distributed query approach. In XQueryD users need to know the sources' schemas, but can formulate arbitrarily complex queries as long as these are supported by the local source. Adding a new source has almost no cost. A mediator based systems presents the user with a logically coherent mediated schema, sparing him the specific details at each source; it can potentially scale to large number of sources. On the other hand, users can only ask limited form of queries supported by the mediator, typically conjunctive queries (i.e. without aggregates or subqueries), and the cost of adding a new source is high.

4.3 Graphical Query Interface

In order to allow easy access to the integrated sources and to all data processing tools, XBrain needs to allow users to formulate complex queries over the data without necessarily being competent in a query language. Our current approach is to provide the user with (a) a free form for typing XQueryD expressions, (b) a number of predefined XQueryD expressions, which can be modified by the users in free form, and (c) a simple interface that allows users to save query expressions and later retrieve and modify them. This part of the system will be extended in the future with elements of graphical query languages; there is a rich literature on graphical query interfaces, e.g. QBE [40] and XQBE [4].

4.4 Heterogeneous Output Formats

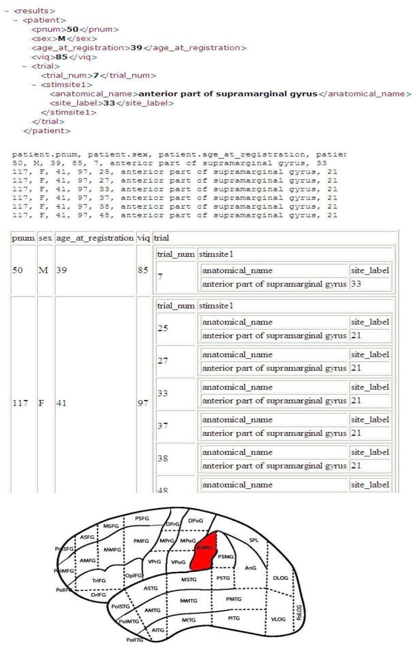
The output of XQueryD is a single XML document describing the results of integrating data from the different data sources. While such a document may be useful for analysis programs or XML-savvy users, it is not the most intuitive. Thus, we allow users to choose alternative output formats, including both common formats such as HTML or CSV (comma separated values for input to Excel), and formats that are specific for an application.

In our approach, the system generates automatically an XSLT program for each XQueryD, and for each desired output format. The XSLT program is simply run on the query's answer and generates the desired output format. We currently support CSV, HTML, and some proprietary formats for image generation tools. To generate the XSLT program, the system needs to know the structure of the XML output, i.e. the element hierarchy and the number of occurrences of each child, which can be `*`, `1`, or `?` (0 or 1). In an early version we computed this structure by static analysis on the query (type inference), but we found that code brittle and hard to maintain and are currently extracting the structure from the XML output: the tiny performance penalty is worth the added robustness. Figure 7 (a) shows four possible output formats for the same output data: in XML format, in CSV format, in HTML format, and as an image.

4.5 Integration of Heterogeneous Visualization Tools

Common output transformations, such as HTML or CSV as noted in the previous section, can be part of a generic integrated application that could be applied to many different problems. However, each biomedical application will have its own special output requirements that may best be addrsseed by independent visualization tools. Our approach to this problem is to create independent tools that can run as web services callable by the XBrain application. We have experimented with such services for a 2-D image visualization tool that accepts XML output from the CSM database and generates an image showing the locations on a 2-D sketch of the brain where specific types of language processing occur.

Such an approach may also be useful for 3-D visualization of query results, using a server-based version of our BrainJ3D visualization tool [26]. Interactive visualization and analysis of the results, which might include new query formation, will require alternative approaches, such as a Web Start application that can create an XQuery for the integrated query system.



```
- <results>
  - <patient>
      <pnum>50</pnum>
      <sex>M</sex>
      <age_at_registration>39</age_at_registration>
      <viq>85</viq>
    - <trial>
        <trial_num>7</trial_num>
      - <stimsite1>
          <anatomical_name>anterior part of supramarginal gyrus</anatomical_name>
          <site_label>33</site_label>
        </stimsite1>
      </trial>
    </patient>
```

```
patient.pnum, patient.sex, patient.age_at_registration, patie
50, M, 39, 85, 7, anterior part of supramarginal gyrus, 33
117, F, 41, 97, 25, anterior part of supramarginal gyrus, 21
117, F, 41, 97, 27, anterior part of supramarginal gyrus, 21
117, F, 41, 97, 33, anterior part of supramarginal gyrus, 21
117, F, 41, 97, 37, anterior part of supramarginal gyrus, 21
117, F, 41, 97, 38, anterior part of supramarginal gyrus, 21
117, F, 41, 97, 48, anterior part of supramarginal gyrus, 21
```

pnum	sex	age_at_registration	viq	trial		
				trial_num	stimsite1	
					anatomical_name	site_label
50	M	39	85	7	anterior part of supramarginal gyrus	33
117	F	41	97	25	anterior part of supramarginal gyrus	21
				27	anterior part of supramarginal gyrus	21
				33	anterior part of supramarginal gyrus	21
				37	anterior part of supramarginal gyrus	21
				38	anterior part of supramarginal gyrus	21
				48		

(a)

```
<results>
  { for $trial in PublicView("Scrubbed.pv")
      /patient/surgery/csmstudy/trial
    where $trial/trialcode/term/abbrev/text()="2"
    return
      <answer>
        <name>
          { $trial/stimsite/name() }
        </name>
        <image-anchor>
          <uri>
            http://csm.biostr.washington.edu/vbm
          </uri>
          <param>
            { $trial/stimsite/name() }
          </param>
          <param>
            gray
          </param>
        </image-anchor>
      </answer>
  }
</results>
```

(b)

Fig. 7. Different query output formats (XML, CSV, HTML, and Image) (a). Query for embedding images in XML files (b).

Our approach is as follows (we refer to Fig. 1 below). Every visualization tool must be a Webservice and offer a common interface that accepts some tool specific input parameters and generates an image (jpeg file). The XQueryD expression returns certain subelements in the answer that are *anchors* to visualization Webservices. The user has to explicitly construct these anchors, like in Fig. 7 (b), which returns a set of stimulation sites, each with an image of the brain with the corresponding stimulation site highlighted. The answer to the query contain elements `image-anchor`. When the HTML generator (see Fig. 1) translates the XML document into HTML, it processes the image anchors by calling the appropriate Webservice with the appropriate parameters, then embeds the resulting images in the HTML table.

5 Related Work

Distributed Query Processing. There is a rich literature on distributed query processing and optimization; a survey is in [22]. Our syntactic approach to distributed query is closest in spirit to the Kleisli system [39], and also related to process calculi and their application to database queries [17, 31]. Unlike ubQL [31], we use only one communication primitive, namely the migration operator `execute`, and omit channels and pipelining. A different approach to distributed data is Active XML [1–3]. Here an XML tree may contain calls to Webservices that return other XML fragments. Query evaluation on Active XML naturally leads to distributed execution. XQueryD differs from Active XML in several ways. In Active XML the data need to be modified by inserting Webservice calls and users are unaware of the distributed nature of the data; by contrast, in XQueryD the data do not need to be modified, while queries require detailed knowledge of the sources and their capabilities.

Mapping Relational Data to XML. Mapping relational data to XML has been discussed extensively [8, 12, 14, 33, 34, 37]. In addition, most of the database vendors today offer some support for XML publishing: Oracle [10, 24], SQL Server [25], BEA's Liquid Data [9]. Each mapping language is proprietary, and can only be used in conjunction with that particular product (relational database or query engine). In addition, none offers any shortcuts to defining the mapping: users have to write each piece of the mapping. In contrast RXQuery is more lightweight, and is translated into XQuery, and is specifically designed to be very concise when defining complex mappings.

Other Related Work. For XQuery processing we use the Galax, which is described in [13]. For translating XQuery to SQL we use SilkRoute, whose architecture is described in [11], and which we recently extended significantly to generated optimized SQL code: the extensions and optimizations are described in [29]. Other optimization techniques for the XQuery to SQL translation are discussed in [23]. There is a rich literature on graphical query interfaces, starting with Zloof's QBE [40]. Recent work describes a graphical query interface to XQuery, called XQBE, is in [4].

6 Conclusions

The high complexity in integrating today's biomedical data has two root causes: the fact that the data are increasingly distributed and generated by collaborative environments, and the fact that they are processed, analyzed and visualized by increasingly more complex tools. We have described a framework for integrating data and tools for biomedical data, with a specific application to the University of Washington's Human Brain Project.

Acknowledgments

This work was funded in part by NIH Human Brain Project grant DC02310 and Suciu was partially supported by the NSF CAREER Grant IIS-0092955, NSF Grants IIS-0140493, IIS-0205635, and IIS-0428168, and a gift from Microsoft.

References

1. S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy query evaluation for active XML. In *SIGMOD*, 2004.
2. S. Abiteboul, O. Benjelloun, and T. Milo. Positive active xml. In *PODS*, 2004.
3. S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic xml documents with distribution and replication. In *SIGMOD*, pages 527–538, 2003.
4. 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.
5. J. Brinkley, R. Jakobovits, and C. Rosse. An online image management system for anatomy teaching. In *Proc. AMIA Fall Symposium*, page 983, 2002.
6. J. Brinkley, B. Wong, K. Hinshaw, and C. Rosse. Design of an anatomy information system. *Computer Graphics and Applications*, 19(3):38–48, 1999. Invited paper.
7. J. F. Brinkley, L. M. Myers, J. S. Prothero, G. H. Heil, J. S. Tsuruda, K. R. Maravilla, G. A. Ojemann, and C. Rosse. A structural information framework for brain mapping. In *Neuroinformatics: An Overview of the Human Brain Project*, pages 309–334. Mahwah, New Jersey: Lawrence Erlbaum, 1997. See also `http://sig.biostr.washington.edu/projects/brain/`.
8. M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. subramanian. XPERANTO: publishing object-relational data as XML. In *Proceedings of WebDB*, Dallas, TX, May 2000.
9. M. J. Carey. BEA liquid data for WebLogic: XML-based enterprise information integration. In *ICDE*, pages 800–803, 2004.
10. A. Eisenberg and J. Melton. SQL/XML is making good progress. *SIGMOD Record*, 31(2):101–108, 2002.
11. M. Fernandez, Y. Kadiyska, A. Morishima, D. Suciu, and W. Tan. SilkRoute : a framework for publishing relational data in XML. *ACM Transactions on Database Technology*, 27(4), December 2002.
12. M. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middleware queries. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Santa Barbara, 2001.
13. M. Fernandez and J. Simeon. Galax: the XQuery implementation for discriminating hackers, 2002. available from `http://db.bell-labs.com/galax/`.
14. M. Fernandez, D. Suciu, and W. Tan. SilkRoute: trading between relations and XML. In *Proceedings of the WWW9*, pages 723–746, Amsterdam, 2000.
15. M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. Declarative specification of web sites with strudel. *VLDB Journal*, 9(1):38–55, 2000.
16. J. Funderburk, G. Kiernan, J. Shanmugasundaram, E. Shekita, and C. Wei. Technical note - XTABLES: Bridging relational technology and XML. *IBM Systems Journal*, 42(3):538–, 2003.
17. P. Gardner and S. Maffeis. Modelling dynamic Web data. In *Proceedings of DBPL*, pages 75–84, Potsdam, Germany, 2003.
18. X. Hertzenberg, A. Poliakov, D. Corina, G. Ojemann, and J. Brinkley. X-batch: Embedded data management for fmri analysis. In *Society for Neuroscience Annual Meeting*, page 694.21, San Diego, 2004.
19. K. Hinshaw, A. Poliakov, R. Martin, E. Moore, L. Shapiro, and J. Brinkley. Shape-based cortical surface segmentation for visualization brain mapping. *Neuroimage*, 16(2):295–316, 2002.

20. R. Jakobovits, C. Rosse, and J. Brinkley. An open source toolkit for building biomedical web applications. *J Am Med Ass.*, 9(6):557–590, 2002.
21. S. Koslow and S. Hyman. Human brain project: A program for the new millenium. *Einstein Quarterly J. Biol. Med.*, 17:7–15, 2000.
22. D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.
23. R. Krishnamurthy, R. Kaushik, and J. Naughton. Efficient XML-to-SQL query translation: Where to add the intelligence? In *VLDB*, pages 144–155, 2004.
24. M. Krishnaprasad, Z. Liu, A. Manikutty, J. Warner, V. Arora, and S. Kotsovolos. Query rewrite for XML in oracle XML DB. In *VLDB*, pages 1122–1133, 2004.
25. M. Library. Creating xml views by using annotated xsd schemas, 2005.
26. E. Moore, A. Poliakov, and J. Brinkley. Brain visualization in java3d. In *Proceedings, MEDINFO*, page 1761, San Francisco, CA, 2004.
27. P. Mork, J. F. Brinkley, and C. Rosse. OQAFMA querying agent for the foundational model of anatomy: a prototype for providing flexible and efficient access to large semantic networks. *J. Biomedical Informatics*, 36(6):501–517, 2003.
28. C. Re, J. Brinkley, K. Hinshaw, and D. Suciu. Distributed XQuery. In *Workshop on Information Integration on the Web (IIWeb)*, pages 116–121, September 2004.
29. C. Re, J. Brinkley, and D. Suciu. Efficient publishing of relational data to XML. submitted.
30. C. Rosse and J. L. V. Mejino. A reference ontology for bioinformatics: the foundational model of anatomy. *Journal of Bioinformatics*, 36(6):478–500, 2003.
31. A. Sahuguet and V. Tannen. ubQL, a language for programming distributed query systems. In *WebDB*, pages 37–42, 2001.
32. R. Shaker, P. Mork, J. Brockenbrough, L. Donelson, and P. Tarczy-Hornoch. The biomediator system as a tool for integrating biologic databases on the web. In *Proc. Workshop on Information Integration on the Web, held in conjunction with VLDB*, 2004.
33. J. Shanmugasundaram, , J. Kiernana, E. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *Proceedings of VLDB*, pages 261–270, Rome, Italy, September 2001.
34. 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, Cairo, Egypt, September 2000.
35. Z. Tang, Y. Kadiyska, H. Li, D. Suciu, and J. F. Brinkley. Dynamic XML-based exchange of relational data: application to the Human Brain Project. In *Proceedings, Annual Fall Symposium of the American Medical Informatics Association*, pages 649–653, Washington, D.C., 2003. http://quad.biostr.washington.edu:8080/xbrain/index.jsp.
36. Z. Tang, Y. Kadiyska, D. Suciu, and J. Brinkley. Results visualization in the xbrain xml interface to a relational database. In *Proceedings, MEDINFO*, page 1878, San Francisco, CA, 2004.
37. I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, May 2002.
38. G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, March 1992.
39. L. Wong. The functional guts of the Kleisli query system. In *Proceedings of ICFP*, pages 1–10, 2000.
40. M. M. Zloof. Query-by-example: A data base language. *IBM Systems Journal*, 16(4):324–343, 1977.

Optimizing Runtime XML Processing in Relational Databases

Eugene Kogan, Gideon Schaller, Michael Rys, Hanh Huynh Huu, and Babu Krishnaswamy

Microsoft Corporation
One Microsoft Way
Redmond, WA 98052, USA
{ekogan,gideons,mrys,hanh,babuk}@microsoft.com

Abstract. XML processing performance in database systems depends on static optimizations such as XML query rewrites, cost-based optimizations such as choosing appropriate XML indices, and the efficiency of runtime tasks like XML parsing and serialization. This paper discusses some of the runtime performance aspects of XML processing in relational database systems using Microsoft® SQL Server™ 2005's approach as an example. It also motivates a non-textual storage as the preferred choice for storing XML natively. A performance evaluation of these techniques shows XML query performance improvements of up to 6 times.

1 Introduction

Most relational database management systems are in the process of adding native XML support [13], largely motivated by the necessity of querying and modifying parts of XML documents that cannot easily be mapped to the relational model. The native XML support consists mainly of a native XML data type – based on the SQL-2003 and upcoming SQL-200n standards, XML Schema and XQuery [3] support. Interestingly, all systems take a similar functional approach [13], and, even though some of their underlying physical approaches differ with respect to the various XML storage and indexing techniques, they have similar trade-offs to make in order to efficiently store, retrieve, validate, query and modify their XML.

While XML is a scalar data type for the relational type system, it has to be transformed into the XQuery data model [4] for XQuery evaluation, and the results of XQuery have to be serialized back into scalar form. This involves XML parsing, possible XML validation, and XML generation. XML parsing and validation are considered expensive in database environments [9].

Based on Microsoft SQL Server 2005's implementation, we will present some current state of the art runtime optimization approaches that are used by relational database management systems to address these issues.

Microsoft SQL Server 2005 provides the native XML data type which optionally can be constrained according to a collection of XML schemas and queried using XQuery. Additionally, a modification language based on XQuery can be used to update the XML in place. For more information, see [1] and [7].

SQL Server internally represents the XQuery data model as a row set of XML "information items". During XQuery evaluation this row set is processed by a set of extended relational operators. Operators that transform an XML scalar into the XML info item row set and aggregate the row set into an XML scalar are two major XML-specific extended operators. The former is the more expensive operator since it has to perform XML parsing as well as typing XML info items when processing typed XML.

SQL Server 2005 provides the option to store the XML info item row set pre-shredded to speed-up XQuery processing. This pre-shredded form is called the primary XML index [2]. The XML index allows the relational optimizer to exploit the full power of cost based optimizations when building XML query execution plans.

However, there are multiple important scenarios when an XML instance can't be indexed or the XML indexing cost is prohibitive. In such cases the cost of the XML query evaluation is dominated by the cost of XML parsing and producing the type information for the XML info item row set.

This paper will provide details on the performance optimizations of these runtime operations. The optimizations are in the areas of tuning the serialization format used for the scalar XML storage and integrating XML processing APIs into the database infrastructure. In Section 2 we describe the XML support in Microsoft SQL Server 2005 from the runtime point of view. The information in Section 2 is presented indifferent to the employed XML storage format. Section 3 analyzes the inefficiencies of XML processing with traditional APIs and text XML as the storage format. Section 4 describes binary XML storage format properties as well as some of the solutions and techniques we employed in order to maximize the performance of runtime XML processing. It also shows how they address the performance bottlenecks we identified in Section 3. Section 5 provides the results of some performance evaluations based on XMark [8] and XMach-1 [6] workloads. Related work is discussed in Section 6. Section 7 contains concluding remarks and potential future work.

2 XML Support in Microsoft SQL Server 2005

In this section we briefly describe the main XML features in Microsoft SQL Server 2005 and highlight their execution phase. The XML support in SQL Server 2005 is described in more detail in [1], [2], [7], and [14].

XML is a new data type natively supported in Microsoft SQL Server 2005. It is supported as a variable type, a column type, and a parameter type for functions and stored procedures:

```
DECLARE @xvar XML
CREATE TABLE t(pk INT PRIMARY KEY, xcol XML(xsc))
CREATE PROCEDURE xproc @xparam XML(xsc)
AS SELECT xcol FROM t
```

The *XML(xsc)* syntax indicates that an XML schema collection *xsc* is associated with the type which validates and types the XML data. We refer to schema constrained XML data as "*typed XML*".

SQL Server guarantees that data stored in XML type is a well formed XML fragment or document and provides *XML fidelity* [13]. Additionally, a typed XML instance is guaranteed to validate against its XML schema collection. To support this, the server parses the XML data when it arrives in the database and during data conversions, and checks that it is well formed; typed XML is also validated at that point. Once parsed and validated, the server can write the XML into the target storage unit in a form optimized for future parsing. This format is the subject of discussion in Section 3 and 4. Physically, XML is treated as a large scalar data type and – just like any other large data types (LOB) in SQL Server, – has a 2GB storage limit.

2.1 Querying XML

SQL Server 2005 adds XQuery through methods on the XML type and applies it to a single XML type instance (in this release). There are 4 methods on the XML type that accept XQuery string and map the resulting sequence into different SQL types:

– the *query* method returns XML;
– the *value* method returns SQL typed values other than XML;
– the *exist* method is an empty sequence test;
– the *nodes* method returns a table of references to XML nodes;

Additionally, the *modify* method uses the Microsoft data modification extensions to XQuery in order to modify an XML instance. The *modify* method is not described is this paper. However, it is worth mentioning that its implementation in SQL Server 2005 provides the ability to perform partial updates of the LOB containing the scalar XML data, i.e. a minimally necessary part of XML LOB is modified and logged during updates with *modify()* method instead of changing the whole document.

The following example extracts the ISBN and constructs a list of AuthorName XML elements for every book in the XML variable @x:

```
SELECT
  xml_ref.value('@ISBN','NVARCHAR(20)') isbn,
  xml_ref.query('for $a in Authors/Author return
                  <AuthorName>
                    {data(Name/First), data(Name/Last)}
                  </AuthorName>') author_name_list
FROM @x.nodes('/Book') AS tbl(xml_ref)
```

Each individual XQuery expression is compiled into an XML algebra operator tree. At this stage SQL Server performs annotations of the XML operator tree with type information, XQuery static typing, and various rule-based optimizations like XML tree simplifications and XPath collapsing. Next, the XML operator tree is mapped into an extended relational operator tree and grafted into the containing SQL relational operator tree. Finally, the relational optimizer performs cost-based optimizations on the resulting single extended relational operator tree. For more details on mapping XQuery to relational operators see [14].

The extended relational operators, that evaluate XQuery, process row sets where each row represents an XML node with type information, i.e. a typed XML information item or *XML info item*. The XML hierarchy and document order are captured in this row set with the use of a special key called ORDPATH [2], [5]. The ORDPATH allows the efficient calculation of the relationship between two XML nodes in the XML tree. Some of the important XML-specific extensions of to the relational operators are those that shred XML scalars into XML info item row sets and aggregate the row sets into XML scalars: the XML Reader and the XML Serializer.

The XML Serializer operator serializes the info item row set into an XML scalar. XML well-formedness is checked by the XML Serializer and it throws dynamic errors in cases such as serialization of a top level attribute or occurrences of duplicate attributes in an element.

The XML Reader operator is a table-valued function that parses an XML scalar and produces an XML info item row set generating the ORDPATH keys in the process. On a typed XML instance, the XML Reader also has to generate typed values and provide the XML info item type information. Note that while parsing XML from XML Reader operator it is not necessary to check if the input XML is well formed or valid to its schema collection since well-formedness and validity are enforced when instantiating and modifying the XML scalar.

Except for trivial XQuery expressions, an XQuery compilation normally results in multiple XML Reader operators in the query plan. Thus, the XML instance can be shredded multiple times during the XQuery evaluation. This makes the XML Reader performance critical for overall XQuery performance when the XML is not indexed.

2.2 Indexing XML

Indexing XML [2] is the main option to increase XQuery performance in SQL Server 2005. Users can create a primary XML index and optional secondary XML indexes. A primary XML index is a materialized row set of XML info items with XML names tokenized. It contains the primary key of the base table to allow back joins. This allows query optimizer to use the primary XML Index instead of the XML Reader in the query plan.

One of the main performance benefits from the XML index comes from the ability to create secondary XML indexes to speed up XPath and value look-ups during query execution. Secondary XML indexes allow the query optimizer to make cost-based decisions about creating *bottom-up* plans where a secondary index seek or scan is joined back to the primary XML index which is joined back to the base table containing XML columns.

The XML Reader operator populates the primary XML index in XML insertion and modification query plans. Thus, performance of the XML Reader is important for the indexed XML column modification performance. It is especially visible since the XML index modification is done on top of the XML Reader as a partial update.

Even though the XML index is the best tool for boosting XQuery performance, it is an overhead on storage space and on data modification performance. There are XQuery scenarios where creating the XML index may not be practical or beneficial:

- In scenarios where XML is used as the transport format for complex data. XML is often used as the variable or parameter type. XQuery is then applied in order to extract the values from the XML instance. XML indexes cannot be created on variables and parameters. The goal is to have a big range of similar queries performing well enough so that no index needs to be created.
- When XML instances in a table are predominantly located using a table scan or a non-XML index, such as an index on a relational column or full text index, the benefits of XML indexes are limited. Additionally, if XML instances are small or the XQuery expressions are simple and access a large number of nodes in the XML instances, XQuery on the XML blob should yield the result equally fast.

If XML instances are queried in such scenarios, the cost of XML index creation and maintenance could be avoided, making performance of parsing the XML scalar into the XML info item row set important.

3 Performance Challenges of Textual XML Storage Format

Here and in Section 4 we evaluate textual and binary XML storage format options based on the following goals for our runtime performance evaluation:

- to make XML parsing as fast as possible since many XML processing scenarios depend on that;
- to make XML parsing scalable with respect to the size of XML, different ratios of XML mark-up vs. text data, the number of attributes for an element, etc.
- to make the performance of XQuery on typed XML column without XML index comparable with XQuery on untyped XML column without XML index;
- to make XML serialization/generation scale and perform well.

Between the performance and scalability of XML parsing and serialization, XML parsing clearly commands higher priority because of its importance for a bigger number of XML processing scenarios. Note that XML scalability in a database server also means that XML processing should require only limited amount of memory in order to not hurt overall server performance and throughput.

The naïve choice for native XML storage format is to store its original textual form. Advantages are the ability to use a standard parser and a standard generator for all XML parsing and generation needs, and the ability to send XML content with no conversion to any client.

SQL Server 2005 leverages the fastest native-code text XML parser available from Microsoft. It is a pull model XML parser that provides a light weight COM-like interface. SQL Server uses it to build an XML Validating Reader that can produce typed XML info set items if the XML data is associated with a schema collection. It is optimized for performance and scalability and has deterministic finite state automata objects for typed XML elements cached in a global server cache.

In the remainder of this section we will discuss performance challenges and bottlenecks that result from the textual XML storage choice. Section 4, will present our solutions to the outlined problems. We will also use the textual XML storage as the base line for our performance evaluation in Section 5.

3.1 XML Parsing and Generation Performance Bottlenecks

Here we enumerate most of the CPU-intensive operations during XML parsing using traditional text XML parsers as well as operations that may require large amounts of memory:

- XML character validity checks and attribute uniqueness checks that are done as part of the XML well-formedness check may require large amounts of memory to store unbounded number of attribute names as well as add to the CPU load. Since XML well-formedness is guaranteed for the XML type, such checks should only be necessary when converting other types to XML or during operations that generate an XML type and not during the conversion to the XML info item row set.

- The XML parser has internally to keep all in-scope XML prefix mappings at any point during the document processing. In the general case, the XML parser either needs to buffer in memory all attributes and XML namespace declarations for an element or, if the input stream is seekable (as it mostly is in server scenarios), it has to do two passes through the attributes in order to resolve all namespace prefixes to namespace URIs before returning the element node.
- Entity resolution, attribute whitespace normalization, and new line normalization in text nodes may require substantial CPU and memory resources depending on the implementation approach. What's important for our investigation is that if an attribute value or text node needs to be accessed by the caller of the XML parser as a single value then such a value needs to be copied and buffered either in memory or on disk if it is large.
- Many of the above listed operations involve memory allocations and de-allocations. Memory management during XML parsing is one of the most expensive operations in terms of CPU utilization. Besides, larger memory consumption decreases query throughput.
- Textual XML verboseness adds to CPU utilization when checking well-formedness and I/O overhead when reading and writing the XML to and from disk.
- XML can be provided in many different language encodings that in the XQuery data model are all mapped to Unicode. While encoding translation is one of most expensive operations during XML parsing, it can be dealt with by storing the XML in the target Unicode encoding (UTF-16 for SQL Server), so that the translation has to occur only once.
- Document Type Definition (DTD) processing can be very expensive both in terms of CPU utilization and memory consumption. We are not going into details of DTD processing here since it has very limited support in SQL Server 2005 – only internal entities and default attributes are supported and only with an explicit conversion flag. DTD is always stripped during population of XML type instances.

In the case of XML serialization/generation the most expensive operations are well-formedness check and entitization. Also, having an API with XML attributes as second class objects require buffering attributes during element construction which may not scale well in database server scenarios.

3.2 Typed XML Processing Overhead

When using text XML as the XML type storage format the XML Reader operator uses the Validating Reader for producing the typed XML info item row set. We expect the XML instance to be valid according to its schema collection, but we still have to go trough the validation in order to annotate the info items with type information as well as provide typed values.

The Validating Reader introduces performance overhead on top of the basic XML parser when evaluating XQuery on a typed XML. This overhead is normally greater than the XML parsing overhead. For example, converting a 4.5MB Microsoft Word

2003 document in XML format to XML typed with a fairly simple WordML schema (available for download through http://www.microsoft.com/downloads/details.aspx?FamilyId=FE118952-3547-420A-A412-00A2662442D9&displaylang=en) takes 2.2 times longer than converting the same document to untyped XML. More complex schema validation that also involves more type conversions can introduce bigger overhead on shredding typed XML. We didn't invest into performance evaluation of various XML schemas concentrating instead on a solution that would minimize the number of XML validations required.

The XQuery evaluation in SQL Server 2005 requires that the XML Readers flow rows representing the simple type or the simple content element with their typed value "pivoted" in the same row. Since the XML parser and the Validating Reader return XML nodes in XML document order such "pivoting" in turn requires the buffering of attributes, comments, and processing instructions preceding the simple type/content element value text node. The buffering is done in an in-memory buffer that is spilled to disk if the amount of data buffered goes above a threshold. Note that the "pivoting" is required for shredding typed XML for both XQuery evaluation on the fly and for the XML index.

4 Using Optimized Representations to Store XML

In the previous section, we assumed that XML type instances are stored in their textual format. The idea of all of the runtime performance optimization we describe below is to do all the processing-expensive steps of XML parsing and validation only once – when populating the XML scalar. The XML scalar representation should then use an optimized representation that – together with a custom parser of that representation – will allow us to provide a more efficient XML Reader operator that can avoid many of the steps that traditional XML parsers go through.

After analyzing the textual XML runtime performance bottlenecks we also want to avoid buffering large or unbounded amounts of data in memory and have minimal data copying by implementing XML shredding and generation in a streaming way.
Based on that, we perform the following three runtime optimizations:

- choosing a binary XML format as the storage format for XML type,
- revising the XML parsing and generation interfaces,
- introducing techniques to avoid copying larger values returned from the XML parser.

Below we describe each of the solutions and how they addressed the performance bottlenecks we described in Section 3. These solutions take advantage of the fact that the LOB storage in SQL Server 2005 is optimized for random access so the XML parser and generator operate on seekable streams with efficient seeks when working with both on-disk and in-memory XML LOBs.

4.1 Binary XML as XML Type Serialization Format

For the XML type representation SQL Server uses a binary XML format that preserves all the XQuery data model [4] properties. The binary XML format has to be self-contained so it can be sent to the clients that support the format without any preprocessing, i.e. the format should not rely on the server metadata like the XML schema or the XML name token dictionaries. Also, the requirement to have efficient partial XML BLOB updates limits XML structure annotations that could be exploited in the format. Finding an optimized layout of the XML BLOB on-disk data pages (like it is done in Natix [12] and System RX [17]) that would require additional processing upon retrieval is not a goal - XML index is an option for optimal XQuery execution performance, and support for XML navigation APIs like Document Object Model (DOM) is also not required.

Describing the Binary XML storage format of SQL Server 2005 is beyond the scope of this paper. Instead, we will focus on the properties of the format that allow building a fully streamable binary XML parser and how they address the performance bottlenecks.

- XML attribute values and text nodes can be stored with typed values in the SQL Server 2005 binary XML representation. Primitive XML schema types are supported as well as most of SQL types. This allows avoiding performing type conversions when serializing the typed XML info item row set into a XML BLOB, and also allows skipping data conversions when shredding XML BLOB into a XML info item row set as well as when validating XML instances after modification.

  Note that typed values do not necessary require less space than their string serialization; for example, the UTF-16 string "1" takes less space than the 8-byte integer 1. Also note that the binary XML format does not generally preserve string value formatting – typed values are serialized in their XML Schema canonical form when binary XML is converted to text XML.
- XML name strings as well as values of string types are stored in the target API code page – UTF-16, – so no code page conversion is required when generating or parsing the binary XML representation.
- All variable length values are prefixed with their length so that the binary XML parser can seek over such value when parsing XML from a seekable stream allowing the caller to access such value only if needed. We'll explain the idea in more detail in Section 4.3.
- All XML qualified names are represented as sets of local name, namespace URI, and namespace prefix and stored tokenized. Tokens can be declared anywhere in binary XML BLOB before the token is first used. This allows streaming binary XML generation.

  Tokenization of the XML QNames allows the binary XML parser to avoid supporting a XML namespace prefix resolution interface since namespace URI is returned with every element or attribute info item without pre-scanning all attributes of an element and without keeping the list of visible XML namespaces in memory. Note that typed values derived from the QName type are stored tokenized as well. Supporting namespace prefix resolution is still required for cases where the XML QNames are not stored typed such as in the case of XPath expressions stored as strings in XSLT.

  QName tokenization can lead to a significant space compression for XML documents where the same QNames are used repeatedly.
- The binary XML format supports format-specific extensions that are not part of the XML content and only visible to binary XML parsers. Like XML processing

instructions such extensions can be ignored by a XML parser that does not process those. They are ignored when the XML type is converted to string.

These format-specific extensions are used in the server space for two reasons.

1. Add element/attribute integer type identifiers needed for info item type annotations in XML info item row set. Together with supporting typed values this allows shredding the typed XML into XML info item row set without the validation step.
2. Annotate simple type/content elements with offsets to their typed values in the binary XML BLOB so the expensive typed value "pivoting" described in Section 3.2 can be done with two seeks in binary XML stream instead of buffering the attributes, comments, and processing instructions preceding the element value. Note that for simple content elements the trick with the offset makes partial update of a typed XML BLOB more expensive if an attribute, comment, or processing instruction of such element is modified – the offset of the value may need to be adjusted.

When generating binary XML, character entitization (such as transforming < into &lt;) and string conversions of typed XML values become unnecessary. However, QName tokenization may add to CPU load during binary XML generation to the degree that that binary XML generation may become more expensive than the equivalent text XML generation. This can be mitigated by caching QName tokens in cases where the set of QNames is known statically, like when formatting a row set as XML or serializing typed XML (the latter case is not currently optimized in SQL Server 2005).

The QName tokenization performance overhead of generating the binary XML representation is outweighed by performance benefits for binary XML parsing. The binary XML format with the above properties allows the parser to be fully streamable and free from all the XML parsing bottlenecks we listed in Section 3.1.

However, it introduces the need to maintain an in-memory QName token look-up array in order to resolve tokens to XML QNames during the parsing process. This array can require a large amount of memory. To work around this issue, the binary XML format requires XML generators to insert a special binary token into the binary XML stream that signals to the readers that the QName token look-up array should be freed and new tokens are declared for QNames used afterwards. This command for flushing the token array must be inserted when the total size of all QName string or total number of defined QNames reach some threshold. This way there's always a preset limit on the memory consumption of the QName look-up array.

Note that SQL Server 2005's binary XML format may be used in other areas than the SQL Server Engine. Even though we don't document the binary XML format in more detail in this paper, the properties of the format listed above are all the important binary XML features the server takes advantage of – XML QName tokenization, typed values, prefixing variable length types with data length, element/attribute type annotations.

4.2 XML Parsing and XML Generation API Improvements

To take advantage of the properties of the binary XML format we need to revise the pull model XML parser and generator APIs. The API improvements include:

– added typed value support instead of chunked character data; no value chunking is supported for typed values; it does not result in the intermediate value buffering – more details in section 4.3;
– added info item type identifier (specific to the server); the generator and the parser add/retrieve it through binary XML extension tokens;
– stopped supporting XML namespace prefix resolution interface unless specifically requested by the caller for custom QName resolution;
– attributes are returned/accepted as first class XML items, i.e. instead of returning list of attributes with the element event there's a new attribute info item event; there's no need in buffering all attributes for a given element in order to support multiple iterations through the attribute array;
– for all QNames, including values of types derived from QName type, the improved API supports passing those as a triplet of local name, namespace URI, and namespace prefix; the binary XML generator also allows callers to access tokens assigned to QNames and then pass only a QName token if the QName is written multiple times.

Taken together with the binary XML format properties, these API improvements allowed creating a fully streaming binary XML parser and generator. Note one exception to the streaming behavior when generating typed XML and when parsing typed XML for XQuery evaluation or for XML index: seeks are required in order to annotate simple type/content elements with offsets of their values, and when retrieving the value with the element info item.

4.3 Dealing with Large Values

With text XML parsing either the parser or the calling code have to aggregate chunks of single value into a single scalar value. This is required because of entity resolution and various value normalizations that the text XML parser has to do. Such value aggregation from multiple chunks can be expensive since in database server scenarios it has to be disk-backed in order to avoid allocating large amounts of memory.

The binary XML format contains all values in the form returned by the XML parsing API. Specifically, the binary XML format has value lengths implicitly (through value type) or explicitly present in the binary XML stream before the value. Since LOBs in SQL Server are passed as references, we can refer to a fragment of a binary XML BLOB without requiring a value copy.

An object called a Blob Handle is used when LOB is transported within the database server boundary. A Blob Handle is a universal reference to a large object within the server. It can contain a reference to on disk storage containing LOB data, or inlined LOB data for smaller data sizes, or a reference to LOB data dynamically constructed in memory. The usage of Blob Handles ensures that large amounts of data are not moved unnecessarily when LOBs are passed as parameters or flowed in the query processor during query execution.

A new class of Blob Handles is introduced that can refer to a fragment in another LOB and use such Fragment Blob Handles when the source LOB is not mutable. Fragment Blob Handles are used by the binary XML parser for returning string and binary values larger than a preset copy threshold. The implementation of the Fragment Blob Handle allows retrieving the value from the parser input buffer if it is still there (when the value is retrieved from the same thread as the parser is on).

This technique allows the binary XML parser to skip over large string or binary values when working on a seekable stream. If further XQuery processing filters out XML nodes with large values then the disk pages containing these large values do not have to be fetched. This improves performance and scalability of XQuery on XML instances with large string/binary values as well as XML with high data to mark-up ratio.

The above described techniques lowered memory management expenses and allowed building binary XML parser and generator that do not allocate memory after construction except for adding entries to the QName token look-up table. We took advantage of the fact that the QName-token mappings are allocated one by one and freed as a whole and used an incremental memory allocator based on a buffer chain instead of using a more expensive heap allocator.

5 Performance Evaluation

This section reports the SQL Server 2005 runtime performance improvements measured on a set of XML query benchmarks and some additional tests. We evaluated the performance of XQuery execution in the database server environment and not the performance of XML parsing and generation in isolation. All the tests were run with a warm database cache in single user mode so the results reflect mostly CPU cost of the query where XML BLOB processing, in turn, takes the most part. For our experiments we instrumented the sever code so we are able to switch between text and binary XML as the XML type storage format and parser type. The text XML parser was wrapped into a class that implements the new binary XML parser API. Textual XML was stored after it went through text-to-text XML conversion while instantiated so it was in UTF-16 format with any DTD stripped, and all values normalized and reserialized with minimally necessary entitization. So, during text XML parsing from XML Reader operator parts of the most expensive operations were not performed.

The tests were run on a 4-way Intel Xeon 1.5GHz machine with 2GB RAM. The performance differences are large enough to fall into the realm of statistical significance.

5.1 XMark Benchmark for Untyped and Typed XML

We tested our XML storage and runtime improvements on the XMark test suite. XMark [8] is an XQuery benchmark that models an online auction and represents data-centric XML scenario. We chose an XMark scaling factor 0.5 adopted for relational database use so that one large XML instance was shredded into smaller XML instances put into 5 tables representing the data model entities. [2] gives more information on the XMark modifications for SQL Server 2005. We did two runs – one for an XMark database where the XML columns are untyped and one for an XMark database containing XML columns typed according to the XMark schema. The results are presented in Table 1.

Table 1. Performance gain on XMark scale factor 0.5 comparing untyped and typed Binary XML storage format against text format

	Avg gain	Gain Range	Avg Binary XML compression rate
Untyped	2.0	1.53 – 2.48	1.10
Typed	4.7	2.65 – 6.40	1.02

We didn't provide performance gains for individual queries since the results were distributed rather evenly around the avarage.

The way one big XML instance is shredded into many small ones in our XMark database is less benefitial for binary XML compression rate since XML QNames are practically not repeated in XML instances. Typed XMark database got practically no compression since the binary XML included type annotations. Therefore, we can consider the XMark test suite as relatively less beneficial for our runtime XML improvements.

The larger performance gains when running the queries against typed binary XML format can be attributed to skipping XML validation during XML info item row set generation by XML Reader operator. Compared to the untyped XML the typed XML performed 13% slower on average. This can be attributed to the overhead of parsing the type annotations.

5.2 XMach-1 Benchmark and Additional Measurements

We also ran our performance comparison on a more document-centric workload like XMach-1 [6]. We ran 5 out of 8 data retrieval queries – queries 3, 4, 6, 7, and 8, – that were ported to the SQL Server 2005 XQuery implementation.

The queries were run on a set of XML instances totaling 30.8MB in UTF-16 text XML. In binary format the total size of the XML instances was 26.2MB – showing a compression rate of 1.18. The average performance gain was 3.05 with gains of individual queries ranging from 2.77 to 3.31.

We measured raw parsing performance on a basic 2.6GHz Intel Pentium 4 machine with 1GB RAM in a single user load. We formatted a 20000 row table containing customer data as XML in an element-centric manner. This 18MB UTF-16 XML instance with highly repeated XML tags yielded a 2.4 compression rate when stored as binary XML. We used XQuery *count(/*/*)* where practically all the time is spent in parsing the XML. The query on the binary XML performed 2.77 times faster than the same query on the text XML LOB.

The same table formatted in attribute-centric manner resulted in 12.8MB UTF-16 text XML with a 1.77 compression rate when converted to binary XML (binary XML representations of the element-centric and the attribute-centric formatting were nearly of the same size – 7.43MB and 7.23MB). Similar XQuery counting attributes in the document (*count(/*/@*)*) performed 4.17 times faster on the binary XML than on the text XML. The reason of the higher performance gain is that the query on the binary XML performed 1.62 times faster after switching from element-centric to attribute-centric formatting while the performance gain for the text XML was only 1.08. That shows that with all the optimization we described binary parsing cost in this case is largely dominated by the number of calls into the parser, i.e. by the number of info items returned; note that one attribute event when parsing of the attribute-centric formatting corresponds to 3 parser events in case of parsing the element-centric formatting. The text parsing, while also returning fewer info items, had to perform all the expensive attribute list processing we mentioned in 3.1.

6 Related Work

In [9] Nicola et al. analyze cost of SAX parsing and XML schema validation in database scenarios. While their findings are generally in line with our analysis made in this paper we analyzed a pull model XML parser and went into greater detail about its cost. We took the analysis further as a motivation for changes needed in XML storage format, APIs, and integration with the rest of database system in order to decrease cost of XML parsing. We also reported results of a real world commercial implementation of the improvements.

There's a number of publications on binary XML format and compression techniques including [15], [16], plus materials of the W3C Workshop on Binary Interchange of XML Information Item Sets [11]. While being excellent source of ideas none of them enumerates binary XML features that are particularly important for serialization format in a database server environment where XML stream is seekable but has to allow efficient partial updates, where parsing performance and memory consumption are the priorities, and where I/O is easier to scale than CPU power. Bayardo et al. in [10] analyzed use of various binary XML features on XML parsing for stream based XML query processing. Our paper validates that only the most basic binary XML features are requested in database servers. We also listed the necessary XML parsing API performance improvements and the ways to integrate with the database server.

7 Conclusion and Future Work

We took a detailed look at the runtime optimization side of XML processing in relational database systems based on the XML processing framework in Microsoft SQL Server 2005 and showed that generally XML support as a native data type can be highly efficient and scalable.

We analyzed performance bottlenecks of XML parsing in database server environment and set the goal of implementing a streaming XML shredding and generation that do not require buffering large or unbounded amounts of data in memory, have minimal data copying, and avoid unnecessary well-formedness and validation checks. We identified that changing the XML scalar storage format from text XML to a binary XML format is necessary for building a fully streaming XML parser and identified the necessary properties of the binary format. We also listed improvements of the XML parsing and generation API required in order to take advantage of the binary XML format so there's no need for the parser to buffer any parts of XML in memory. Building a custom XML parser allowed avoiding well-formedness and validation checks when evaluating XQuery expressions – all checks are done when instantiating the XML instance. We measured XML query performance gain in the range of 1.5 to 4.17 for untyped XML when using the binary XML format and the new APIs.

We didn't specifically measured performance gains from using BLOB fragment references to avoid large value copies. While this optimization benefits all queries with large values, it can have a big impact for XML queries that filter out nodes with large values based on XPath or other predicates. In such cases the performance gain from skipping the retrieval of some of disk pages containing the LOB can be very large in corner cases with very small markup-to-values ratio and large values like in case of storing audio and video data in XML format.

Parsing typed XML for XQuery evaluation does not require full XML validation and only requires adding type annotations and "typing" values. Using the binary XML format allows to completely avoid XML validation during XQuery evaluation and consequently brings even bigger performance gains for XQuery on typed XML BLOBs – we measured a 5x average for the XMark schema.

As SQL Server always validates XML coming from the client side, a conversion from text to binary XML on the way into the server space is not really an overhead. However, when sending XML to clients that do not support the binary XML format the server has to perform binary-to-text XML conversion. Even though SQL Server performs such conversion by streaming text XML directly into network buffers, it is still a considerable overhead comparing to the simple retrieval of an XML BLOB.

For future releases we envision the binary XML features to remain basic so they do not require more CPU utilization. Typed XML processing may benefit from keeping QName token definitions for QNames present in XML schema as metadata and only adding them when sending binary XML to components that do not have access to SQL Server metadata. Another idea that can benefit typed XML parsing without adding considerable CPU utilization is to package all statically known singleton properties (attributes and elements) for a typed element into a record-like structure, thus avoiding storing mark-up for such properties.

The current LOB storage on the SQL Server Storage Engine level is B^+-trees optimized for random seeks on offset in BLOB. We think that changing the XML BLOB storage format to trees that are optimized for seeks on XML node ID (ORDPATH) can bring the biggest performance gain in XML runtime area. An ability to bind to such a tree as both scalar and a row set would make creating Primary XML index unnecessary – secondary XML indexes can be built on top of such an XML BLOB storage.

Acknowledgements

The authors would like to thank their colleagues Istvan Cseri, Goetz Graefe, Oliver Seeliger, Dragan Tomic, Shankar Pal, Jinghao Liu, Mike Rorke, Derek Denny-Brown for their support and discussions on XML runtime performance improvements.

References

1. M. Rys: XQuery in Relational Database Systems. XML 2004 Conference
2. S. Pal, I. Cseri, G. Schaller, O. Seeliger, L. Giakoumakis, V. Zolotov: Indexing XML Data Stored in a Relational Database. VLDB Conference, 2004

3. Edited by S. Boag, D. Chamberlin et al.: XQuery 1.0: An XML Query Language. W3C Working Draft 04 April 2005, http://www.w3.org/TR/xquery/
4. Edited by M Fernández, A. Malhotra et al.: XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft 4 April 2005, http://www.w3.org/TR/xpath-datamodel/
5. PE. O'Neil, EJ. O'Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels, SIGMOD Conference, 2004
6. E. Rahm, T. Böhme: XMach-1: A Multi-User Benchmark for XML Data Management. Proc. VLDB workshop Efficiency and Effectiveness of XML Tools, and Techniques, 2002
7. S. Pal, M. Fussell, I. Dolobowsky: XML support in Microsoft SQL Server 2005. MSDN Online, http://msdn.microsoft.com/xml/default.aspx?pull=/library/en-us/dnsql90/html/sql2k5xml.asp
8. AR Schmidt, F. Waas, ML Kersten, MJ Carey, I. Manolescu, and R. Busse. XMark: A benchmark for xml data management. In VLDB, pages 974-985, 2002
9. M. Nicola and J. John , XML Parsing: A Threat to Database Performance. CIKM 2003
10. R. J. Bayardo, V. Josifovski, D. Gruhl, J. Myllymaki: An Evaluation of Binary XML Encoding Optimizations for Fast Stream Based XML Processing. WWW 2004 Conference
11. Report From the W3C Workshop on Binary Interchange of XML Information Item Sets. http://www.w3.org/2003/08/binary-interchange-workshop/Report
12. CC. Kanne, G. Moerkotte: Efficient Storage of XML Data, ICDE 2000
13. M. Rys, D. Chamberlin, D. Florescu et al.: Tutorial on XML and Relational Database Management Systems: The Inside Story, SIGMOD 2005
14. S. Pal, I. Cseri, O. Seeliger, M. Rys, G. Schaller, P. Kukol, W. Yu, D. Tomic, A. Baras, C. Kowalczyk, B. Berg, D. Churin, E. Kogan: XQuery Implementation in a Relational Database System. In proceedings of VLDB 2005 Conference
15. WY. Lam, W. Ng, P. Wood, M. Levene: XCQ: Xml Compression and Querying System. Proc of WWW 2003 Conference
16. B. Martin, B. Jano: WAP Binary XML Content Format. W3C NOTE 24 June 1999, http://www.w3.org/TR/wbxml/
17. K. Beyer, R.J. Cochrane et al: System RX: One Part Relational, One Part XML. SIGMOD 2005

Panel: *"Whither XML, ca. 2005?"*

This year, XML will be 9 years old (7 years as a standard). About 6 years ago, the data management community began to adopt it as an interchange and representation format. Some leading database researchers have seen the format as merely an inefficient re-invention of old ideas; others have embraced it as the future and the bridge between databases and "the outside world" (including text). In recent years, we have seen a great deal of work on bridging XML and relations, on developing and customizing languages from XML (building from a variety of foundations, including relational, semi-structured, object, functional, and imperative), on adding ranked retrieval into XML query languages, on using XML as an interchange format, even on using XML as the basis of component-oriented software architectures.

It seems an appropriate time to step back and ask a number of questions like: What have we learned about XML's relevance in today's world? What problems have we solved? What are the critical problems that should be the focus of tomorrow's research?

We have invited a panel of distinguished researchers and practitioners, who will attempt to address these issues and identify important open research challenges. The list of panelists includes Peter Buneman (U. Edinburgh), H.V. Jagadish (U. Michigan), Jayavel Shanmugasundaram (Cornell U.), Daniela Florescu (Oracle), and Tova Milo (Tel Aviv U.).

This panel is conducted as a joint event of XSym and DBPL (Tenth International Symposium on Database Programming Languages).

Author Index