

Structural Summaries as a Core Technology for Efficient XML Retrieval

Dissertation

zur Erlangung des akademischen Grades des
Doktors der Naturwissenschaften
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München



von

Felix Weigel

06. November 2006

Diese Arbeit ist als Buch mit dem Titel „Efficient XML Retrieval with Structural Summaries“ im Verlag Dr. Hut erschienen (München 2006, 303 Seiten, ISBN 3-89963-461-6).

This work was published as a book entitled “Efficient XML Retrieval with Structural Summaries” by Dr. Hut Verlag (Munich 2006, 303 pages, ISBN 3-89963-461-6).

Erstgutachter/Primary supervisor:	Prof. Dr. François Bry Ludwig-Maximilians-Universität München
Zweitgutachter/Secondary supervisor:	Prof. Dr. Klaus U. Schulz Ludwig-Maximilians-Universität München
Externer Gutachter/External supervisor:	Prof. Dr. Gerhard Weikum Max-Planck-Institut für Informatik, Saarbrücken
Beginn der Arbeit/Begin date:	01. Oktober 2003
Tag der Abgabe/End date:	06. November 2006
Tag der mündlichen Prüfung/ Date of oral examination:	04. Dezember 2006

Structural Summaries as a Core Technology for Efficient XML Retrieval



Abstract

The *Extensible Markup Language (XML)* is extremely popular as a generic markup language for text documents with an explicit hierarchical structure. The different types of XML data found in today's document repositories, digital libraries, intranets and on the web range from flat text with little meaningful structure to be queried, over truly semistructured data with a rich and often irregular structure, to rather rigidly structured documents with little text that would also fit a relational database system (RDBS). Not surprisingly, various ways of storing and retrieving XML data have been investigated, including *native XML* systems, *relational engines* based on RDBSs, and *hybrid* combinations thereof.

Over the years a number of native XML indexing techniques have emerged, the most important ones being *structure indices* and *labelling schemes*. Structure indices represent the *document schema* (i.e., the hierarchy of nested tags that occur in the documents) in a compact central data structure so that structural query constraints (e.g., path or tree patterns) can be efficiently matched without accessing the documents. Labelling schemes specify ways to assign unique identifiers, or *labels*, to the document nodes so that specific relations (e.g., parent/child) between individual nodes can be inferred from their labels alone in a decentralized manner, again without accessing the documents themselves. Since both structure indices and labelling schemes provide compact approximate views on the document structure, we collectively refer to them as *structural summaries*.

This work presents new structural summaries that enable highly efficient and scalable XML retrieval in native, relational and hybrid systems. The key contribution of our approach is threefold. (1) We introduce *BIRD*, a very efficient and expressive labelling scheme for XML, and the *CADG*, a combined text and structure index, and combine them as two complementary building blocks of the same XML retrieval system. (2) We propose a purely relational variant of BIRD and the CADG, called *RCADG*, that is extremely fast and scales up to large document collections. (3) We present the *RCADG Cache*, a hybrid system that enhances the RCADG with incremental query evaluation based on cached results of earlier queries. The RCADG Cache exploits schema information in the RCADG to detect cached query results that can supply some or all matches to a new query with little or no computational and I/O effort. A main-memory cache index ensures that reusable query results are quickly retrieved even in a huge cache.

Our work shows that structural summaries significantly improve the efficiency and scalability of XML retrieval systems in several ways. Former relational approaches have largely ignored structural summaries. The RCADG shows that these native indexing techniques are equally effective for XML retrieval in RDBSs. BIRD, unlike some other labelling schemes, achieves high retrieval performance with a fairly modest storage overhead. To the best of our knowledge, the RCADG Cache is the only approach to take advantage of structural summaries for effectively detecting query containment or overlap. Moreover, no other XML cache we know of exploits intermediate results that are produced as a by-product during the evaluation from scratch. These are valuable cache contents that increase the effectiveness of the cache at no extra computational cost.

Extensive experiments quantify the practical benefit of all of the proposed techniques, which amounts to a performance gain of several orders of magnitude compared to various other approaches.

Zusammenfassung

Die *Extensible Markup Language (XML)* ist eine weit verbreitete Auszeichnungssprache für hierarchisch strukturierte Textdokumente. Heutzutage finden sich in Dokumentensammlungen, elektronischen Bibliotheken, im Intra- und Internet verschiedenste Arten von XML-Dokumenten: angefangen von Textdaten, deren flache Struktur sich kaum für die Anfrage eignet, über semistrukturierte Dokumente im eigentlichen Sinne, die eine reiche und oft unregelmäßige Struktur aufweisen, bis hin zu eher einheitlich strukturierten Dokumenten mit wenig Text, die ebenso gut in einer relationalen Datenbank gehalten werden könnten. So ist es nicht überraschend, wie viele unterschiedliche Arten es gibt, XML-Dokumente zu speichern, insbesondere *native* Systeme, *relationale* Systeme und *hybride* Ansätze, die beide kombinieren.

Im Laufe der Zeit sind eine ganze Reihe nativer Indizierungsverfahren für XML entstanden, insbesondere *Strukturindizes* und *Numerierungsschemata*. Strukturindizes repräsentieren das *Dokumentenschema*, d. h. die Hierarchie verschachtelter XML-Etiketten (*tags*), in einer einzigen zentralen Datenstruktur. Auf diese Weise können strukturelle Anfragebedingungen, etwa Pfad- oder Baummuster, effizient und ohne Zugriff auf die Dokumente ausgewertet werden. Numerierungsschemata zeichnen die Dokumentknoten mit eindeutigen Kennnummern aus. Aus diesen lassen sich bestimmte Beziehungen zwischen den Knoten (z. B. die Eltern-Kind-Beziehung) herleiten, wiederum ohne Zugriff auf die Dokumente oder auch nur eine zentrale Datenstruktur. Sowohl Strukturindizes als auch Numerierungsschemata stellen eine näherungsweise Sicht auf die Dokumentstruktur dar. Daher bezeichnen wir beide als *Strukturauszug* (*structural summary*).

Die vorliegende Arbeit stellt neuartige Strukturauszüge vor, mit denen XML-Daten in nativen, relationalen und hybriden Systemen auf höchst effiziente und skalierbare Weise durchsucht werden können. Unser Ansatz zeichnet sich in dreifacher Hinsicht aus. (1) Wir führen *BIRD* ein, ein sehr effizientes und ausdrucksstarkes Numerierungsschema, sowie den Text- und Strukturindex *CADG*, und verknüpfen beide Verfahren in einem XML-Anfragesystem. (2) Es wird eine rein relationale Variante von *BIRD* und dem *CADG* namens *RCADG* vorgestellt, die selbst große Dokumentensammlungen sehr schnell durchsucht. (3) Der hybride *RCADG Cache* erweitert den *RCADG* um eine inkrementelle Anfragekomponente auf der Grundlage von zwischengespeicherten Ergebnissen früherer Anfragen. Der *RCADG Cache* bedient sich der im *RCADG* vorhandenen Schemainformationen, um diejenigen Anfragen im Zwischenspeicher zu finden, die alle oder zumindest einige Treffer für eine gegebene neue Anfrage mit wenig oder gar keinem Berechnungsaufwand oder Zugriffen auf die Peripherie liefern können. Mit Hilfe eines Hauptspeicherindex auf dem Zwischenspeicher werden solche wiederverwendbaren Anfrageergebnisse selbst dann schnell gefunden, wenn bereits viele Anfragen gespeichert worden sind.

Es zeigt sich, daß XML-Anfragesysteme hinsichtlich ihrer Effizienz und Skalierbarkeit erheblich von Strukturauszügen profitieren, und zwar in mehrfacher Hinsicht. Die bisher bekannten relationalen Ansätze nutzen die Vorzüge von Strukturauszügen kaum aus. Am Beispiel des *RCADG* wird deutlich, daß sich solche nativen Indizierungsverfahren durchaus auf die XML-Suche in relationalen Datenbanken übertragen lassen. *BIRD* ermöglicht eine schnelle Suche bei nur mäßig erhöhtem Speicherbedarf, anders als manches frühere Numerierungsschema. Soweit bekannt, ist der *RCADG Cache* das einzige Verfahren, das mit Hilfe von Strukturauszügen untersucht, welche Anfrageergebnisse einander enthalten oder überlappen. Darüber hinaus ist uns kein weiterer XML-Zwischenspeicher geläufig, der auch Zwischenergebnisse enthält, die während der Anfrageauswertung ohnehin anfallen. Solche Zwischenergebnisse erhöhen den Wirkungsgrad des Verfahrens, ohne daß dafür zusätzliche Rechenleistung erforderlich wäre.

Nach ausgiebigen Versuchsreihen läßt sich der praktische Nutzen der oben genannten Verfahren auf einen Gewinn von mehreren Größenordnungen im Vergleich zu verschiedenen anderen Ansätzen beziffern.

Meinen Eltern,
denen ich so viel verdanke

For my parents,
to whom I owe so much

Acknowledgements

During the three years of Ph.D. work and before, I have enjoyed the support and encouragement of many great people. First of all I would like to thank my beloved wife Marion, who has helped me in so many different ways, as well as our families and friends, in particular my parents who made all this possible.

Among those who directly contributed to this work, my warmest thanks go to my supervisor and mentor Klaus U. Schulz. For nine years I have benefited from your scientific expertise, open-mindedness, objective judgement and patient guidance, as well as an admirable unifying perspective from the broad vision down to the precise details of ideas. Working with and learning from you was a great pleasure. I am equally grateful to my supervisor and mentor François Bry, who initially aroused my interest in XML, for his strong support and helpful advice during the past five years. With your sense of good ideas and productive research, you have provided me and many others with a stimulating working environment. I would also like to thank Gerhard Weikum for serving as external supervisor, as well as for judicious comments and interesting discussions. Further thanks go to Hans-Peter Kriegel, Hans Jürgen Ohlbach and Christian Böhm for reserving time for the oral examination.

This is also the place to express my gratitude and consideration given that so many colleagues and friends contributed good questions, inspiring ideas and helpful criticism to my work, both at the Centre for Information and Language Processing and at the Department of Computer Science at LMU Munich. I have learned a lot from you over the years. My special thanks go to Holger Meuss, Tim Furche and Dan Olteanu. I also thank Franz Guenther, Levin Brunner, Eduardo Torres-Schumann, Andreas Hauser, Christoph Ringlstetter, Clemens Marschner, Uli Reffle, Annette Gotscharek, Thomas Schäfer and Norbert Eisinger. I have very much enjoyed the insightful discussions with Christoph Koch and Georg Gottlob (then Technical University of Vienna), Thomas Rölleke and Mounia Lalmas (Queen Mary College London), Torsten Grust (Technical University of Munich) and Ralf Schenkel (Max-Planck-Institute for Computer Science Saarbrücken), who also provided me with his XML version of the *Internet Movie Database* collection. Wolfgang Lindner and Martin Sachenbacher (then MIT) shared their experience about postdoctoral studies. Thank you all for your support. Of course this work would not have been possible without the financial support by the German Science Foundation (DFG) through the research grants SCHU 1026/2-3,5. Other work during my Ph.D. was partially funded by the European Union in the context of the REWERSE European Network of Excellence (IST 506779).

Felix Weigel
Munich, December 13th, 2006

CONTENTS

I	Introduction	1
1	XML Retrieval	3
1.1	Motivation	3
1.2	Approaches to Efficient and Scalable XML Retrieval	3
1.3	Goal and Scope of the Thesis	5
1.4	Structure of the Thesis	6
2	Querying XML Documents using Structural Summaries	7
2.1	XML Data Model	7
2.2	XML Query Model	9
2.3	Structural Summaries	10
2.4	The Three-Level Model of XML Retrieval	12
II	Labelling Schemes for XML	15
3	Labelling Schemes for XML and Tree Databases	17
3.1	Overview	17
3.2	Reconstruction and Decision of Query Constraints	19
3.3	Subtree Encodings	21
3.3.1	Interval Encoding	21
3.3.2	Pre-/Postorder Encoding	22
3.3.3	Region Encoding	26
3.4	Path Encodings	27
3.4.1	Full Path Encodings	27
3.4.2	Partial Path Encodings	32
3.5	Multiplicative Encodings	34
3.6	Summary and Discussion	36
4	The BIRD Labelling Scheme	41
4.1	Overview	41
4.2	The Family of BIRD Labelling Schemes	43
4.2.1	Creating BIRD Weights	43
4.2.2	Creating BIRD Labels	44
4.3	Reconstruction of Tree Relations with BIRD	48
4.4	Decision of Tree Relations with BIRD	52
4.5	Handling Document Updates with BIRD	52
4.5.1	Sparse BIRD Labelling	54
4.5.2	Layered BIRD Labelling	54
4.6	Experimental Evaluation	56

4.6.1	Storage Consumption	57
4.6.2	Efficiency of Decision and Reconstruction	59
4.6.3	Efficiency of Query Evaluation	61
4.6.4	Updatability	63
4.7	Summary and Discussion	64
4.8	Optimizations and Open Problems	66
III Index Structures for XML		69
5	Index Structures for Structured Documents	71
5.1	Overview	71
5.2	Inverted Files	71
5.3	Atomic Path Indexing	72
5.3.1	Inverted Path Files	73
5.3.2	Path Bitmaps	73
5.4	Compositional Path Indexing	74
5.4.1	DataGuide	75
5.4.2	IndexFabric	76
5.4.3	Signature File Hierarchy	76
5.4.4	T-Index	77
5.5	Tree and Graph Indexing	77
5.6	Summary and Discussion	78
6	The Content-Aware DataGuide (CADG)	81
6.1	Overview	81
6.2	Materialized Join of Content and Structure	81
6.3	Keyword-Driven Path Matching	82
6.3.1	Signature CADG (SCADG)	82
6.3.2	Inverted-File CADG (ICADG)	83
6.4	Experimental Evaluation	84
6.5	Summary and Discussion	85
IV Relational Storage of XML		87
7	XML Retrieval in Relational Database Systems	89
7.1	Overview	89
7.2	Classification of Storage Schemes	89
7.3	Node Indexing	90
7.3.1	Edge	90
7.3.2	XPath Accelerator	91
7.3.3	STORED	91
7.4	Path Indexing	91
7.4.1	Atomic Path Indexing with XRel	92
7.4.2	Compositional Path Indexing with BLAS	93
7.5	Summary and Discussion	94
8	The Relational CADG (RCADG)	97
8.1	Overview	97
8.2	The RCADG Storage Scheme	97
8.3	BIRD Revisited: Reconstruction and Decision in the RDBS	99
8.4	Query Evaluation with the RCADG	101
8.4.1	Schema-Level Query Rewriting	102

8.4.2	Schema-Level Matching	105
8.4.3	Document-Level Query Rewriting	107
8.4.4	Query Planning	108
8.4.5	Document-Level Matching	112
8.4.6	Computing the Final Query Result	116
8.5	Experimental Evaluation	118
8.5.1	Test Systems	119
8.5.2	Runtime Performance	120
8.5.3	Impact of Query Planning and Optimization	123
8.5.4	Storage Requirements	124
8.6	Summary and Discussion	124
8.7	Optimizations and Open Problems	126
V Caching Techniques for XML		127
9	Caching Techniques for Incremental XML Retrieval	129
9.1	Overview	129
9.2	XML Query Containment and Overlap	130
9.3	Complexity of XML Query Containment	132
9.4	XML Query and Result Caching	132
9.4.1	Incomplete Trees	132
9.4.2	HLCaches	133
9.4.3	Prefix-Based Containment	134
9.4.4	ACE-XQ	135
9.4.5	Caching Based on Access Frequencies	136
9.4.6	Argos	136
9.5	Summary and Discussion	136
10	The RCADG Cache for XML Queries and Results	139
10.1	Overview	139
10.2	Schema Information in the RCADG Cache	140
10.2.1	A Simple Example	141
10.2.2	The General Case	142
10.3	Intermediate Query Results in the RCADG Cache	144
10.4	Exploiting Containment and Overlap with the RCADG Cache	146
10.5	Incremental Query Evaluation with the RCADG Cache	146
10.5.1	Storing Queries in the RCADG Cache	148
10.5.2	Retrieving Cache Contents	148
10.5.3	Deciding Schema-Hit Containment	150
10.5.4	Remainder Query Planning	157
10.6	Experimental Evaluation	159
10.6.1	Cost and Benefit of Evaluating Queries with the RCADG Cache	160
10.6.2	Small-Scale Experiment	160
10.6.3	Large-Scale Experiment	161
10.7	Summary and Discussion	164
10.8	Optimizations and Open Problems	165
VI Conclusion		169
11	Summary and Discussion	171

12 Perspectives and Outlook	175
12.1 Lessons Learnt	175
12.2 Further Applications of Structural Summaries	176
12.2.1 Relevance Ranking	176
12.2.2 User Interaction	177
VII Appendix	179
13 Experimental Set-up	181
13.1 Hardware and Software	181
13.2 Document Collections	181
14 Comparative Performance Evaluation of Five Labelling Schemes	183
Backmatter	186
Index	187
Bibliography	193
List of Figures	203
List of Tables	205
List of Algorithms	207
List of Definitions	209
List of Lemmata	211
About the Author	213

Part I

Introduction

XML Retrieval

1.1 Motivation

The *Extensible Markup Language (XML)* [XML] has by now become widely accepted as the standard markup language for modelling, querying, exchanging and storing a broad range of semistructured data with different characteristics. At the one end of the spectrum, there are text-centric documents with only little explicit structure that is worth querying, such as web pages, Wikis, Blogs, news feeds, e-mail and FAQ archives. At the other end of the spectrum, we have rather database-like XML content with a far more rigid and meaningful structure and little text, such as product catalogues, tax payer's data submitted via electronic forms, bibliography servers, address books, web service descriptions and even scientific sensor data. In between those two extremes, XML is perhaps most commonly used for a wide variety of data which is truly semistructured, having a more or less complex and irregular structure that adds significant information to the rich textual content. Examples are documents in digital libraries or in the database of a publishing house, articles in electronic encyclopedias, on-line manuals, technical documentation in corporate intranets, linguistic databases containing parsed fragments of natural language, and scientific taxonomies or ontologies that formalize domain knowledge in a structured way.

While generic markup languages for semistructured data such as the *Standard Generalized Markup Language (SGML)* [SGML] have been used already for a long time, most notably in document management and publishing, it was only the adoption of XML for the World-Wide Web that has made the semistructured data model so popular for all kinds of businesses and applications. Given a steadily growing entourage of complementary specifications, standards and tools that foster the creation, retrieval and manipulation of large amounts of XML data, the community has long since abandoned the often-cited toy collections of the early days [Bosak 1999] that contained a few kilobytes of manually marked-up poetry, facing today the many gigabytes of real-world XML data in productive systems. In other words, now that such a large number of people using such a large amount of data are convinced that XML is a good choice for their purposes, efficient and scalable retrieval techniques need to be developed in order to prove them right.

1.2 Approaches to Efficient and Scalable XML Retrieval

Trying to tackle new problems with existing solutions is not uncommon and sometimes even the best strategy. Moreover, given that XML is partly used for content which is close to either flat text or completely structured data, it seems natural to find out how far one can get in XML retrieval using traditional Information Retrieval (IR) engines or relational database systems (RDBSs). On the one hand, these two options have the advantage of relying on rather mature technology, including very efficient data structures and algorithms. On the other hand, both approaches suffer from the inherent dichotomy between text and structure that is characteristic of XML, incapable of supporting the two simultaneously to the extent needed. Neither the highly structured relational model nor the unstructured flat-text data model can fully capture a rich XML hierarchy. In order to fit the relational model, the hierarchical, irregular structure of XML

data must first be broken down to tuples with a suitable schema, and then efficiently restored from sets of tuples at runtime. Even simple queries involving nested elements, possibly with interwoven text, are not fully grasped by SQL's string matching capabilities or regular expressions. Analogously, IR systems need a more expressive data model than linear full-text to cope with the hierarchical nature of XML. Early IR-inspired approaches to structured text retrieval, such as *PAT* expressions [Salminen and Tompa 1992] or the *Region Algebra* [Consens and Milo 1994], only partially overlap with today's query languages like *XQuery* [XQuery] and *XPath* [XPath].

Therefore the development of dedicated XML retrieval systems has received much attention. They are most commonly referred to as *native* (i.e., structured-preserving) systems, as opposed to purely *relational* approaches and *hybrid* systems that combine the former two. Native XML retrieval engines¹ are built on top of a tree or graph data model such as the *Document Object Model (DOM)* [DOM] or the *Object Exchange Model (OEM)* [Papakonstantinou et al. 1995]. Native XML systems are designed to capture the nature of the data as closely as possible, unlike relational databases or flat-text IR engines where on the contrary the XML data must be adapted to the nature of the system. A wealth of tree- or graph-specific data structures and algorithms have been devised to this end. They fall into three categories:

1. *structure indices*: index structures for retrieving instances of specific nested tag patterns in the documents², partly inspired by earlier work on query optimization in object-oriented database systems³
2. *labelling schemes*: methods of assigning XML elements unique identifiers that encode certain structural relations (e.g., nesting or document order) between these elements⁴
3. *structural joins*: join algorithms that operate on sets of XML elements to find instances of more or less complex tree or path patterns, such as twigs, in the documents⁵

Structure indices are often called *structural summaries* in the literature. In this work we deliberately generalize this term to subsume not only structure indices, but also labelling schemes. This is to emphasize that both benefit XML retrieval by providing a reference to structural properties of XML elements, which therefore need not be looked up in the documents. More precisely, we view structure indices as *centralized structural summaries*, i.e., global data structures where path patterns in the query can be matched, and labelling schemes as *decentralized structural summaries*, which allow to infer relationships between elements from information that is local to these elements. Since many contributions in the distinct categories are largely complementary, synergies arise from combining structure indices, labellings schemes and join algorithms for XML. In fact, most structural joins have been designed with a specific labelling scheme in mind.

Structural summaries in the above sense are not to be confused with so-called *schema* specifications, i.e., formal definitions of the document structure such as a DTD or XML Schema [XSD1]. These are grammar formalisms for specifying structural constraints that must be satisfied by all documents of a specific type. Although structural summaries also represent the document schema, their purpose is to reflect structural patterns or properties that are currently expressed in the documents. As a consequence, structural summaries may change in response to modifications of the document collection that introduce new structural patterns. By contrast, when adding documents to a collection that conforms to a specific DTD or XML Schema, structural patterns that are not reflected there are dismissed for being invalid with respect to the (fixed) document schema. In this sense the DTD and XML Schema formalisms are *prescriptive*, whereas the structural summaries we deal with here are *descriptive*.

¹Native XML retrieval systems include, e.g., those by McHugh et al. [1997], Naughton et al. [2001], Li and Moon [2001], Barbosa et al. [2001], Fiebig et al. [2002], Jagadish et al. [2002], and Paparizos et al. [2003].

²Structure indices for XML have been proposed, among others, by Goldman and Widom [1997], Milo and Suciu [1999], Cooper et al. [2001], Kaushik et al. [2002b], Jiang et al. [2003], Schenkel et al. [2004] and Qun et al. [2003]. Chapter 5 surveys some of these approaches.

³Index structures for object-oriented databases have been put forward, among others, by Bertino and Kim [1989], Kemper and Moerkotte [1992], Nestorov et al. [1997] as well as Goldman and Widom [1997].

⁴An overview of labelling schemes for XML is given in Chapter 3.

⁵Structural join algorithms have been presented, e.g., by Zhang et al. [2001], Li and Moon [2001], Al-Khalifa et al. [2002], Bruno et al. [2002], Chien et al. [2002], Jiang et al. [2003], Grust et al. [2003], Lam et al. [2003], Chen et al. [2005a], Li et al. [2005] and Lu et al. [2005].

1.3 Goal and Scope of the Thesis

It is true that building a native XML retrieval system from scratch allows one to take full advantage of the aforementioned native data structures and techniques. However, now that scalability and retrieval efficiency are major concerns, storing and querying XML data in an RDBS is particularly tempting because (1) efficient access methods and highly scalable storage methods for relational data have been developed in over thirty years; (2) query planning and optimization in the relational algebra is well-understood; (3) RDBSs are already widely deployed and offer key features for the productive use, e.g., concurrency, transactions and safety. Replicating this functionality in a home-grown native XML database requires much work. On the other hand, so far the benefits of native XML techniques, such as structural summaries, seem hard to reconcile with the rigid and flat relational data model. In fact, almost all approaches to XML retrieval in RDBSs are more or less oblivious of the most efficient indexing and labelling techniques for XML that have been developed over the years.

The goal of this work is to show how innovative use of structural summaries can contribute to very efficient XML retrieval in native, relational and hybrid systems:

- New native structural summaries are proposed whose properties are especially valuable for efficient and scalable XML retrieval.
- These structural summaries are shown to be easily combined with other structural summaries. We jointly integrate them into a hybrid retrieval system, whose performance is thereby significantly improved.
- The same combination of structural summaries is used in a purely relational retrieval system. It turns out that the cost of migrating the native XML retrieval techniques to the RDBS is low, whereas the benefit in terms of retrieval speed can amount to several orders of magnitude.
- We show that structural summaries are also a very effective means to locate reusable data in a cache of XML query results. Adding cache functionality based on structural summaries to our relational retrieval system again improves the performance by orders of magnitude.

The successful use of structural summaries for different purposes in different retrieval contexts illustrates that structural summaries are much more versatile than what is commonly perceived. Especially the benefit of centralized structural summaries for detecting query containment and overlap in XML caching has been largely ignored so far, as it goes far beyond their canonical use as mere structure indices. Also, the tight integration of structural summaries with the relational query engine that we achieve in our system is a novum. It contributes to bridging the apparent gap between native and relational XML retrieval.

Besides the aforementioned applications, structural summaries are also very useful in two other respects, which are only covered in a cursory way by this work (see Section 12 in Part VI). The first application concerns IR-based XML retrieval systems, which face the twofold burden of adapting their storage and relevance-ranking models to documents with a hierarchical structure. Here structural summaries not only help to increase the retrieval efficiency, but also provide fast access to different kinds of structure-specific ranking parameters, like path frequencies etc., that are needed for XML relevance ranking. Earlier work [Weigel et al. 2005a; Weigel et al. 2004b] has studied the benefit of our structural summaries for both tasks in combination with a variety of ranking models from XML Information Retrieval.

Second, it has been repeatedly pointed out in the literature that in addition to being efficient and scalable, XML retrieval systems should also guide human users in their quest for specific parts of the documents, whose structure they may not know a-priori. This challenge is clearly specific to structured document retrieval, and not faced by flat-text IR or current web search engines. Goldman and Widom [1997] recognized early that centralized structural summaries, as global representations of the document schema, play an important role in making users acquainted with the structure of the documents they are querying. They proposed a graphical representation of the document schema and selected samples of element content that users could browse before starting to formulate queries. Elsewhere [Weigel 2006] we argue that this separate schema browsing can be tightly integrated with the actual retrieval process and extended to cover both the structure and the contents of the documents. The goal is to provide users with a highly interactive and intuitive retrieval experience, where the borders between schema browsing, query formulation, query

evaluation and result inspection are largely blurred. Encouraging users to interact with the system in such a way of course makes sense only with a very responsive retrieval engine. Actually this was the initial motivation behind our studies of XML caching techniques, which allow to recognize previously computed query results that can be immediately presented to the user in response to a new query. Since structural summaries also play a role in this task (see above), they actually support intuitive XML retrieval in two respects: on the one hand, by providing users with a graphical representation of the document schema, and on the other hand, by enabling a smooth continuous feedback by the system during the integrated query and browsing process that has just been sketched. Keeping in mind also the third benefit (namely, the support for relevance ranking), one should indeed regard structural summaries as a core technology for various aspects of XML retrieval.

Finally, a few words on the limitations of this work are in order. First, our techniques are based on a tree data model that ignores cross-references in XML documents, which may be specified using either ID/IDREF attributes or XLink [XLink] and XPointer [XPointer] constructs. Especially the labelling schemes we present make the assumption that every node in the document tree (except the root) has exactly one parent node. Second, as others before we deliberately employ a formal query model instead of using XPath or XQuery directly. However, our formalism covers the core features of most XML query languages, including all thirteen XPath axes. Third, while updates of the document collection are discussed at various occasions throughout this work, we assume that all data to be queried is simultaneously stored in the retrieval system at any point in time. In particular, this excludes distributed settings and the retrieval of streamed XML data. Finally, several general database issues that also apply to XML retrieval systems are ignored here. These include, e.g., concurrency and recovery, access control and privacy, and versioning of XML data.

1.4 Structure of the Thesis

As mentioned before, this work focuses on the role of structural summaries for improving the efficiency of XML retrieval systems. The following parts of the thesis cover different aspects of this topic. Part II (page 17) presents various labelling schemes for XML, which summarize the document structure in a decentralized way. Part III (page 71) reviews different index structures that all belong to the class of centralized structural summaries. Part IV (page 89) shows how structural summaries can be used for XML retrieval in relational database systems. Part V (page 129) deals with caching techniques for XML, including a novel approach to detect query containment and overlap with the help of centralized structural summaries. At the end of the thesis, Part VI (page 171) summarizes the contributions made and concludes with a brief outlook on other useful aspects of structural summaries, namely, for enhancing XML relevance ranking and the user interaction in XML retrieval systems. Finally, a short appendix lists further details of the experiments that were carried out as part of this work.

Each of the parts just mentioned comprises two chapters. The following second chapter of the introduction compiles important preliminaries, including the data and query model to be used throughout this work as well as the *Three-Level Model of XML Retrieval* that illustrates the use of structural summaries from an abstract point of view. In Parts II to V, the first chapter contains a compact survey of contributions to the respective aspect of XML retrieval that are representative for that part of the literature. The second chapter in each part then proposes a new approach to the same problem. All new contributions are explained in detail and evaluated empirically in extensive comparative experiments. We also highlight specific weakspots of prior approaches that are addressed by the new solution, as well as open questions that remain to be solved. The two chapters in Part VI contain a short summary and outlook, as mentioned above. The appendix also consists of two chapters, one listing technical parameters of the experimental set-up and another supplying a detailed analysis of our experiments with different labelling schemes.

Querying XML Documents using Structural Summaries

2.1 XML Data Model

As a basis for the XML retrieval techniques to be presented below, it is convenient and common practice to abstract from the XML serialization [XML] and introduce a more formal data model instead. Throughout this work, we regard any (collection of) XML documents as a *document tree* (disregarding cross-references specified with ID/IDREF attributes), which is defined as follows:

Definition 2.1 (Document tree) *Let T be a finite alphabet of tag names. A document tree is a finite ordered node-labelled rooted tree $D = \langle V, r, \text{Child}, \text{NextSib}, \text{tag} \rangle$ where V is the finite and non-empty set of document nodes (elements)¹, $r \in V$ is the root of D , $\text{Child} \subseteq V \times V$ is a binary relation such that $\langle V, r, \text{Child} \rangle$ is an unordered tree with root r , $\text{NextSib} \subseteq V \times V$ is the sibling order relating a child to its immediate right sibling (if any), and $\text{tag} : V \rightarrow T$ assigns to each node $v \in V$ a tag $\text{tag}(v) \in T$. \square*

Figure 2.1 on the following page illustrates a single document both in XML syntax (a.) and as a document tree D (b.). For convenience, each node in D is given a unique *node label* (the number inside each node; ignore the precise labelling scheme for the moment). To keep the data model simple, multiple documents in a collection are modelled as one large tree D consisting of a newly created root r and the individual document trees whose roots are children of r .² In the sequel, $n = |V|$ denotes the cardinality of V . The *Child* relation is assumed to exclude self-edges of the form $\langle v, v \rangle$ and multiple edges between any pair of nodes in V . The sibling order *NextSib* must respect the XML document order [XML]. For any $v, w \in V$, let $\text{distance}(v, w)$ be the number of edges on the unique path connecting v and w . Furthermore, $\text{level}(v) = \text{distance}(r, v)$ denotes the vertical position of v in D (and also the number of v 's ancestors), whereas $h_D = \max_{v \in V} \{\text{level}(v)\}$ is the height of D . Finally, let $\text{size}(v)$ be the number of descendants of v (i.e., nodes in the subtree rooted in v , excluding v itself), and let $\text{pre}(v)$ ($\text{post}(v)$) denote the rank of v in a left-to-right preorder (postorder) traversal³ of D . Note that *pre* coincides with the XML document order.

Besides *Child* and *NextSib*, there are a number of other binary tree relations relevant for XML retrieval. The relations listed in Table 2.1 on page 9 (first column) cover an important fragment of the XPath language [XPath], similar to *Core XPath* as defined by Gottlob et al. [2006]. In particular, all thirteen XPath axes can be expressed in our data model. For *Child*, *NextSib*, *Following* and their inverse relations (*Parent*, *PrevSib* and *Preceding*, respectively), the closest XPath axes are given in the second column of Table 2.1. Similarly,

¹For simplicity, we treat the terms *document node* and *element* as interchangeable in the sequel. XML attributes and namespace nodes are treated analogously to elements, as shown later.

²This is common practice in the literature. Alternatively, document identifiers may be introduced to ensure that any element can be mapped to the unique containing document, if needed.

³Throughout this work we assume that in any depth-first (preorder, postorder, inorder) or breadth-first tree traversal, each node is visited exactly once. A different definition of depth-first traversal is sometimes encountered in the literature: here each node v is visited twice, once before and once after its descendants. The resulting two ranks of v equal the token positions of its opening and closing tags in the XML serialization. In the sequel, we refer to this double-visit depth-first variant as the *combined pre-/postorder traversal* of the document tree.

```
<?xml version="1.0" ?>
```

```
<people>
```

```
  <!-- subtree  $a_1$  -->
```

```
  <person>
```

```
    <name>Jeff Smith</name>
```

```
    <profile>
```

```
      <edu>MSc</edu>
```

```
      <sex>male</sex>
```

```
    </profile>
```

```
  </person>
```

```
  <!-- subtree  $a_2$  -->
```

```
  <person>
```

```
    <name>Jill Lee</name>
```

```
    <profile>
```

```
      <edu>PhD</edu>
```

```
      <sex>female</sex>
```

```
    </profile>
```

```
  </person>
```

```
  <!-- subtree  $a_3$  -->
```

```
  <person>
```

```
    <name>Mae Lee</name>
```

```
    <profile>
```

```
      <sex>female</sex>
```

```
    </profile>
```

```
  </person>
```

```
  <!-- subtree  $a_4$  -->
```

```
  <person>
```

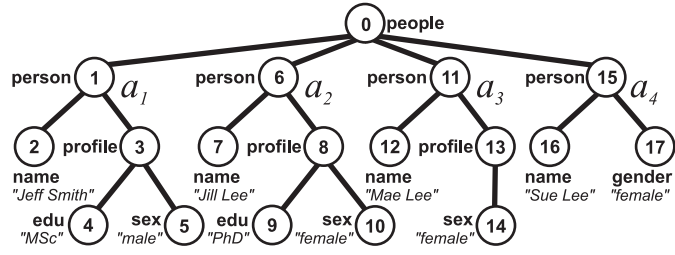
```
    <name>Sue Lee</name>
```

```
    <gender>female</gender>
```

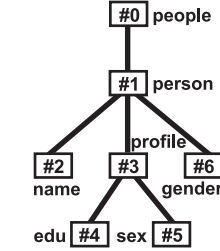
```
  </person>
```

```
</people>
```

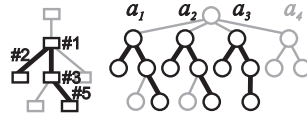
a. XML serialization



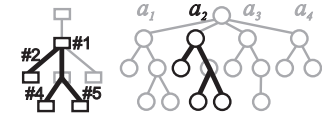
b. document tree D with unique node labels (numbers)



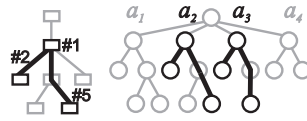
c. schema tree S for D



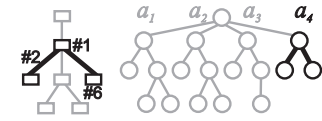
d. χ^Q and its matches in D



e. χ^Q and its matches in D



f. $\chi_1^{Q^n}$ and its matches in D



g. $\chi_2^{Q^n}$ and its matches in D

Figure 2.1: Different representations of a sample XML document (a.–b.) and its structural summary (c.). In d.–g. four schema hits for the queries in Figure 2.2 on page 10 are shown, along with their respective matches in D : d., matches to Q^l (Fig. 2.2 a.); e., matches to Q (Fig. 2.2 b.); f.–g., matches to Q^n (Fig. 2.2 c.).

Self corresponds to the *self* axis in XPath. *Sibling* relates all pairs of children of a given node, regardless of the sibling order. This corresponds to the union of XPath’s *preceding-sibling*, *following-sibling* and *self* axes. Finally, given two nodes $v, w \in V$, $NextElt(v, w)$ ($PrevElt(v, w)$) holds iff w occurs after (before) v in document order.

We also consider proximity variants of *Child*, *NextSib*, *NextElt* and their inverse relations. For any such relation R , let $R_i^j = \bigcup_{i \leq l \leq j} R^l$ where R^l denotes the l -fold composition $R \circ \dots \circ R$ of R . Thus R is equivalent to R_1^1 . For convenience, the symbol “*” acts as a “don’t care” upper bound.⁴ As shorthands, we write R^* for R_0^* and R^+ for R_1^* . For instance, $Child^*$ corresponds to the XPath axis *descendant-or-self* and $Child^+$ to *descendant*. Furthermore, let R^i be a shorthand for R_i^i . Thus $Child^i(v, w)$ holds true iff w is a descendant exactly i levels below v , i.e., iff $Child^+(v, w)$ and $distance(v, w) = i$. Since *Following* and *Preceding* are already closed under composition, there is no natural interpretation of similar proximity variants for these relations. Instead, we define *i-th-Following*(v, w) to capture the semantics of the XPath expression *following::*[i]*, relating v to the i -th member w of the *Following*-image of v (in document order). The reverse counterpart *i-th-Preceding* is defined analogously (in reverse document order).

The remaining XPath axes (namely, *attribute* and *namespace*) are modelled as combinations of

⁴For instance, one may assume that “*” represents any fixed value greater than the total number n of document nodes.

name	description / XPath axis	proximity variant	transitive closure
$Child(v, w)$	child	$Child^t(v, w)$	$Child^+(v, w)$
$Parent(v, w)$	parent	$Parent^t(v, w)$	$Parent^+(v, w)$
$NextSib(v, w)$	following-sibling	$NextSib^t(v, w)$	$NextSib^+(v, w)$
$PrevSib(v, w)$	preceding-sibling	$PrevSib^t(v, w)$	$PrevSib^+(v, w)$
$Following(v, w)$	following	$i\text{-th-Following}(v, w)$	n/a
$Preceding(v, w)$	preceding	$i\text{-th-Preceding}(v, w)$	n/a
$Self(v, w)$	self	n/a	n/a
$Sibling(v, w)$	unordered sibling relation	n/a	n/a
$NextElt(v, w)$	document order	$NextElt^t(v, w)$	$NextElt^+(v, w)$
$PrevElt(v, w)$	reverse document order	$PrevElt^t(v, w)$	$PrevElt^+(v, w)$

Table 2.1: Decidable relations in the document tree.

the binary $Child$ relation and a set $\mathcal{T} = \{Elements, Attributes, Namespaces\}$ of unary relations indicating the type of any node $v \in V$ (element, attribute and namespace node, respectively). For convenience, let $Root = \{r\}$ be the singleton relation containing only the root r of D . Furthermore, as a counterpart to the $level$ function introduced before, we define $Level_i^j \subset V$ as the relation containing all nodes on levels $i \leq l \leq j$, with $Level_0^0 = Root$. Similarly, as a counterpart to the tag function introduced before, we define for each tag $t \in T$ a relation $Tag_t \subset V$ containing exactly the nodes with tag t . These two relations are needed for specifying queries against the document tree (see the next section).

Finally, to model the textual contents of XML documents, we define relations $Contains_k, Governs_k \subset V$ for each k in the set K of keywords occurring in the documents.⁵ Given a node $v \in V$, $v \in Governs_k$ (“ v governs k ”) iff there is a textual occurrence of k somewhere between the opening and the closing tag⁶ of v . By contrast, $v \in Contains_k$ (“ v contains k ”) iff there is a textual occurrence of k somewhere between the opening and the closing tag⁶ of v which is outside the pairs of opening and the closing tags of all descendants of v . Note that in the case of element nodes, government is a necessary but insufficient condition for containment. By contrast, for non-element nodes (which are leaves of D by definition) the two relations coincide. For instance, consider the sample document tree in Figure 2.1 *b*. on the facing page: here the node 25 contains the keyword “*PhD*”. As a consequence, node 25 and all its ancestors in D (i.e., 24, 18 and 0) also govern that keyword. As a matter of fact, the root node 0 in Figure 2.1 *b*. governs a couple of distinct keywords, but contains none. Note, however, that any node is allowed to have both children and textual content. In other words, the data model is flexible enough to capture documents with mixed content.

The type, level, tag, keyword and root relations together make up the set \mathcal{R}_1 of unary relations in D . The relations listed in Table 2.1 constitute the set \mathcal{R}_2 of binary relations in D .

2.2 XML Query Model

Based on the data model introduced in the previous section, we now define a concise query formalism that captures the core features query languages for XML databases, such as XPath.⁷ Contrary to the XPath semantics, the following definition permits queries with multiple result nodes. It also slightly extends the concept of *conjunctive queries* [Gottlob et al. 2006] with tag and keyword disjunctions.

Definition 2.2 (Query) A query Q is a triple $\langle Q_v, Q_c, Q_r \rangle$ where Q_v is a finite and non-empty set of query nodes, $Q_r \subset Q_v$ is a non-empty set of result nodes, and Q_c is a finite and non-empty set of query constraints of the form $R_1(q)$ or $R_2(q, q')$ such that all of the following conditions are satisfied:

⁵The rich data model underlying the XML Schema specification [XSD2] defines a variety of data types for element content. For simplicity, we ignore non-textual data types such as integers, dates, etc. in this work.

⁶Or opening and closing quotes, if v is an attribute or namespace node.

⁷Advanced features of XPath and XQuery [XQuery], such as iteration, functions and data types, are less tightly related to structural summaries and therefore beyond the scope of this work. Conversely, the query model introduced here slightly extends the text search capabilities of these languages.

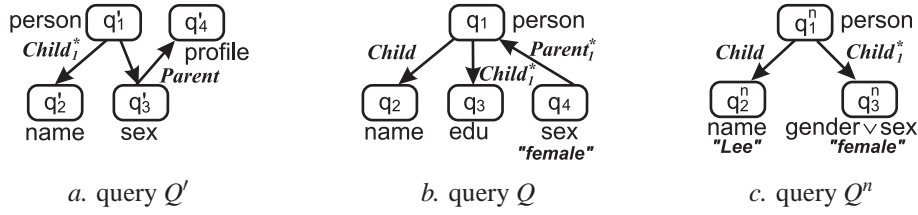


Figure 2.2: Sample queries against the document tree on page 8. All query nodes are regarded as result nodes. In *b.* and *c.*, keyword constraints denote containment. In *c.*, the node q_3^n specifies a tag disjunction.

1. $q, q' \in Q_v$;
2. $R_1 \in \mathcal{R}_1$ is a unary tree relation;
3. $R_2 \in \mathcal{R}_2$ is a binary tree relation;
4. the resulting query graph $\langle Q_v, Q_c \rangle$ is connected (but not necessarily acyclic).

Multiple keyword constraints on the same query node are marked as either conjunctive or disjunctive. Multiple tag constraints on the same query node are implicitly marked as disjunctive. \square

Figure 2.2 illustrates three sample queries against the document tree in Figure 2.1 *b.* on page 8. The following two definitions specify which parts of the document tree D are relevant to a given query against D .

Definition 2.3 (Matching) A matching of a query $Q = \langle Q_v, Q_c, Q_r \rangle$ against D is a mapping $\mu_Q : Q_v \rightarrow V$ such that all of the following conditions are satisfied:

1. $\mu_Q(q) \in R_1$ for each unary constraint $R_1(q) \in Q_c$;
2. $\langle \mu_Q(q), \mu_Q(q') \rangle \in R_2$ for all binary constraints $R_2(q, q') \in Q_c$.

We also write μ instead of μ_Q without ambiguity when Q is clear from the context. For a given matching μ , the μ -image of Q_v is called a match to Q in D . \square

Definition 2.4 (Query result) The result or answer $ans(Q)$ in D for a query $Q = \langle Q_v, Q_c, Q_r \rangle$ is the set of matches (μ -images of Q_v) induced by all matchings μ of Q in D , restricted to Q_r . \square

Unless stated otherwise, we assume $Q_r = Q_v$ for any query Q in the sequel. The answer to Q^n in Figure 2.2 *c.*, e.g., consists of the `person`, `name`, `sex` and `gender` nodes in the subtrees a_2, a_3, a_4 of D in Figure 2.1 *b.* (page 8). The results of all three queries Q' , Q and Q^n in Figure 2.2 *a.–c.* are illustrated on the right-hand side of Figures 2.1 *d., e.* and *f.–g.*, respectively.

2.3 Structural Summaries

While earlier XML test corpora comprised only a few documents of several kilobytes each [Bosak 1998; Bosak 1999], nowadays XML databases must scale up to collections of many gigabytes which cannot be expected to fit main memory. One way to ensure fast query evaluation in such cases is to develop efficient access methods and paging strategies for the secondary storage where the documents reside. For instance, Kanne and Moerkotte [2000] and Fiebig et al. [2002] have gone in this direction. Alternatively, certain query constraints may be matched in the first place against an approximation, or *summary*, of the document tree that is much smaller and can therefore be accessed more efficiently (e.g., in main memory). In a second step, the remaining query constraints are matched directly against those selected parts of the document tree which were recognized as relevant in the first step.

This work investigates the use of various summaries of the document structure, or *schema*, for fast query evaluation. The following general definition of a *structural summary* subsumes labellings schemes,

some of which are presented in Chapters 3 and 4, as well as structure indices for XML, to be discussed in Chapters 5 and 6. We shall see later that combining different structural summaries with each other and with text indices enables highly efficient XML retrieval.

Definition 2.5 (Structural summary) *A structural summary of a document tree D is a compact data structure from which specific structural properties of D can be inferred without access to D itself. Structural summaries can be centralized or decentralized. A typical centralized summary of D is a tree containing information about the set T of tags occurring in D , the levels of nodes with these tags, and the way they are nested. Typical decentralized summaries include labelling schemes that identify an individual node and its tree relations in D using a limited amount of information that is local to that node.* \square

One particular centralized structural summary, which we refer to as *schema tree* throughout this work, is fundamental to many XML index structures. It was introduced as *DataGuide* by Goldman and Widom in 1997. As a preliminary notion, let the *tag path* of any node $v \in V$ be the sequence $/tag(v_0)/\dots/tag(v_j)$ of tags of all nodes $r = v_0, \dots, v_j = v$ on the path from the root r down to node v (i.e., where $Child(v_l, v_{l+1})$ for all $0 \leq l < j$). Let P be the set of distinct tag paths in D . Then the function $\pi : V \rightarrow P$ maps any node $v \in V$ to its unique tag path in D . For instance, in Figure 2.1 b., $\pi(25) = /people/person/profile/edu$.

Definition 2.6 (Schema tree) *The schema tree for a document tree D is the finite rooted unordered node-labelled tree $S = \langle P, \pi(r), Child', tag' \rangle$ whose nodes (schema nodes) are the tag paths in D and whose root is the tag path of the root r in D . The function $tag' : P \rightarrow T$ maps a tag path $p \in P$ to the last tag $t \in T$ in p . For any two tag paths $p_1, p_2 \in P$, $\langle p_1, p_2 \rangle \in Child' \subset P \times P$ iff there exists a tag $t \in T$ such that $p_2 = p_1/t$. If $Child'(p_1, p_2)$, then $Sibling' \subset P \times P$ relates p_2 to all other children of p_1 , if any (recall that S is unordered). Finally, the function $occ : P \rightarrow \mathfrak{P}(V)$ maps a node p in S to the set $occ(p) \subset V$ of nodes in D with the corresponding tag path (its occurrences in D).* \square

Figure 2.1 c. on page 8 shows the schema tree for D in b. Duplicate tag paths in D (such as, e.g., $/people/person/profile$) are represented only once in S . Every schema node is given a unique label (number preceded by “#”), in this case simply its preorder rank in S . Since each distinct tag path in D corresponds to exactly one node in S , we treat both as interchangeable in the sequel. For instance, the tag path $/people/person/profile$ and the node labelled #3 in S are identical. The level of a schema node p is defined as the level of any of its occurrences in D . It is easily verified that this is unambiguous, given that all document nodes with the same tag path reside at the same level in D . By contrast, since an XML element may have both a child element and an attribute with the same name, there may be multiple document nodes with identical tag paths but different types. To distinguish such nodes in S , we assign each schema node a type from the set $\mathcal{T} = \{Elements, Attributes, Namespaces\}$ introduced above. Any document node v is then represented by the unique schema node with the same tag path and type as v .

Definition 2.6 mirrors some of the tree relations introduced before, but on the set P of tag paths rather than on the set V of document nodes as in Section 2.1. Thus $Child'$ corresponds to $Child$ and $Parent'$ to $Parent$, and likewise for $Sibling'$, $Self'$ and the unary constraints. Note that given a pair $v_1, v_2 \in V$ of nodes in D , $Child'(\pi(v_1), \pi(v_2))$ is a necessary, but insufficient condition for $Child(v_1, v_2)$. For instance, although #3 is a child of #1 in S (see Figure 2.1 c.), not all `person` and `profile` nodes in D are parent/child pairs (see Figure 2.1 b.). This results from the approximative nature of the structural summary. Similarly, document order is not captured by the schema tree. Matching these relations against the schema tree can only filter out some parts of the document tree which are guaranteed not to match a given query, while other parts need to be examined by accessing D directly. The following key definitions distinguish query constraints that can be matched against the schema tree S from those which must be checked against the document tree D (or a suitable representation of D):

Definition 2.7 (S-constraint) *The set of S-constraints to be matched against the schema tree comprises*

1. $Parent'$ and $Child'$
2. $Sibling'$
3. $Self'$

4. *type, level, tag and root constraints*

5. Contains_k^l and Governs_k^l (approximate keyword constraints, see Chapter 6)

where $i \leq j \in \mathbb{N}$. □

Definition 2.8 (D-constraint) *The set of D-constraints to be matched against the document tree comprises*

1. *Parent and Child*

2. *PrevSib, NextSib and Sibling*

3. *PrevElt and NextElt*

4. *Preceding and Following*

5. *Self*

6. Contains_k and Governs_k

where $i \leq j \in \mathbb{N}$ and $k \in K$. □

For matching S -constraints against the schema tree, we define $\mu_S : Q_v \rightarrow P$ analogously to μ (see Definition 2.3 on page 10), and call the μ_S -images of Q its *schema hits*. For instance, the query Q^n in Figure 2.2 c. on page 10 has two schema hits, $\chi_1^{Q^n}$ and $\chi_2^{Q^n}$, shown in Figures 2.1 f.–g. (left-hand sides). The first one, $\chi_1^{Q^n}$, consists of the schema nodes #1, #2 and #5 which match the query nodes q_1^n , q_2^n and q_3^n , respectively. The second schema hit, $\chi_2^{Q^n}$, consists of the schema nodes #1, #2 and #6. In the example, Q^n has two schema hits because of the tag disjunction on the query node q_3^n . But even with unambiguous tags, a query involving $*$ proximity bounds as in Child_1^* might have multiple matchings in S . As an example, assume that a new `editedBy` node is added as a child of every document node below the root in Figure 2.1 b. on page 8. Then each of the schema nodes #1–#6 in Figure 2.1 c. would have an additional `editedBy` child. Now if q_3^n specified the single tag constraint `editedBy` instead of the disjunction `gender \vee sex`, Q^n would have six distinct schema hits (where q_3^n would be matched in turn by each of the six new schema nodes).

Figures 2.1 f.–g. also illustrate how each match $a \in \text{ans}(Q^n)$ corresponds to exactly one schema hit of Q^n (namely, the one consisting of the tag paths in a). Given any schema hit χ for a query Q , let $\text{ans}_Q(\chi)$ denote the subset of $\text{ans}(Q)$ corresponding to χ (its *matches* for Q). We drop the subscript to ans when Q is clear from the context. For instance, $\text{ans}(\chi_1^{Q^n}) = \{a_2, a_3\}$ (without the `profile` nodes) and $\text{ans}(\chi_2^{Q^n}) = \{a_4\}$, as shown in Figures 2.1 f.–g. Note that some schema hits of a query Q may have no matches in D . For example, a (hypothetical) schema hit consisting of the nodes #1, #3 and #6 in Figure 2.1 c. would have no matches since there is no `person` node in D with both a `profile` and a `gender` child.

2.4 The Three-Level Model of XML Retrieval

The following *Three-Level Model of XML Retrieval* summarizes the role and the benefit of the document schema in XML retrieval, based on the data and query model introduced before. Queries, schema hits and documents can be viewed as residing on three distinct levels of abstraction which differ both in their relation to the actual XML data and in their physical representation. This is illustrated in Figure 2.3 on the next page. The topmost level (the *query level*) is populated by query expressions as purely intensional⁸ descriptions of some parts of the data (namely, those a user is interested in). Figure 2.3 depicts the three sample queries from Figure 2.2 on page 10; obviously the query level contains an infinite number of other possible expressions, too. Queries are created and manipulated in main memory (although they may of course be stored on disk, e.g., in a query cache as described in Chapter 10). On the bottom level (the *document level*), we have the extensions⁸ of these queries, i.e., their matches in the document tree. As

⁸By *intension* we mean an abstract description of data (e.g., query results) in terms of desired properties (such as the structure and keyword constraints specified by a query), while *extension* denotes some representation of the existing data with such properties.

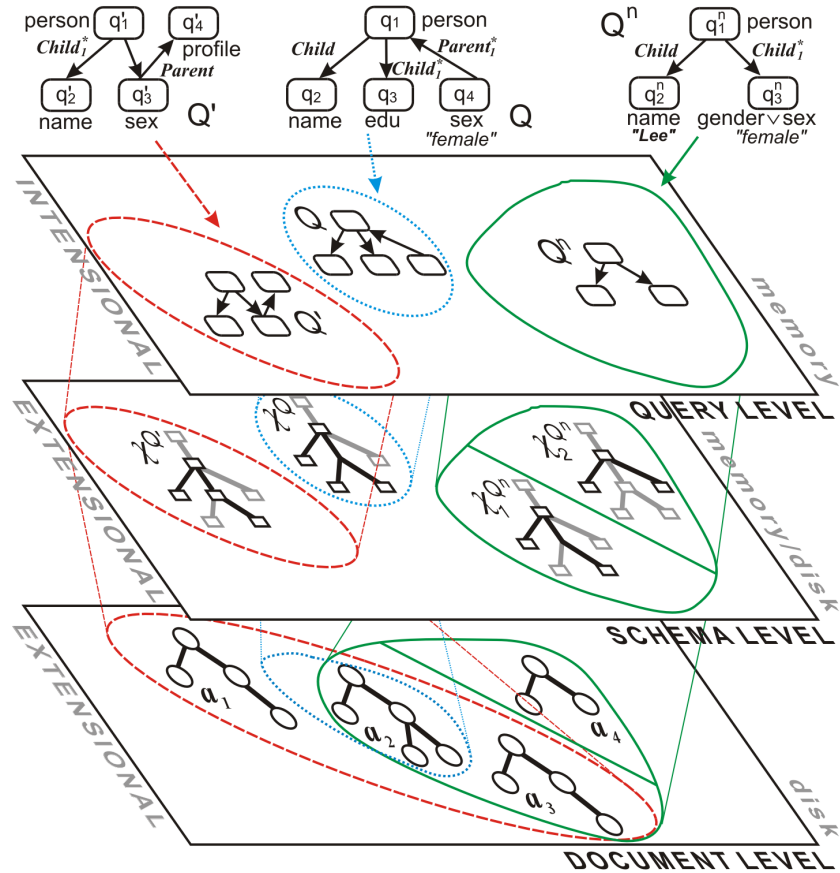


Figure 2.3: The *Three-Level Model of XML Retrieval* relates queries (top level) to their matches in the document tree (bottom level) and the corresponding hits in the schema tree (intermediate level). The sample queries shown here are taken from Figure 2.2 on page 10. The document matches and the schema hits are the same as in Figures 2.1 d.–g. on page 8.

mentioned before, the documents are held in secondary storage. Finally, the schema tree resides on an intermediate level (the *schema level*) between the queries and the documents. It is typically small enough to be kept in main memory, but may also be kept on disk (e.g., when stored in a relational database system, as explained in Chapter 8).

With this Three-Level Model of XML Retrieval and the definitions above in mind, the idea of XML query processing with structural summaries can be rephrased as follows. Given a query Q on the top level in Figure 2.3, we are looking for all matchings μ mapping the query nodes in Q to relevant nodes in the document tree D . However, matching query constraints directly on the bottom level means accessing a large amount of data in secondary storage, which entails expensive I/O and possibly joins. By contrast, given the schema tree S we can match some query constraints (the S -constraints) very efficiently in a first step (*schema matching*). The resulting matchings μ_S select a number of schema hits on the intermediate schema level as a preliminary extension of Q . Each such schema hit χ represents a set $ans(\chi)$ of potential matches, or *candidates*, for Q (recall that the schema tree is only an approximate summary of the document structure). In a second step (*document matching*), the set of candidates is narrowed down to those which also satisfy the remaining query constraints in Q (the D -constraints). In this way we finally obtain the actual query extension $ans(Q)$.

Parts II and III of this work elaborate on the details of this procedure. Among other things, it is shown how labellings schemes, the second type of structural summary, facilitate document matching on the bottom level and thus complement schema matching on the intermediate level. In Part IV, the schema

level is migrated to the relational data model so that both of the lower two layers reside on disk. In Part V the schema-level information is used together with the query intensions on the top level in order to detect containment and overlap of query results on the document level. Finally, at the end of this work, we will come back to the Three-Level Model of XML Retrieval once again, when discussing the benefits of structural summaries for result ranking and user interaction in Part VI.

Part II

Labelling Schemes for XML

Labelling Schemes for XML and Tree Databases

3.1 Overview

The previous chapter has introduced the notion of structural summaries as a compact representation of selected properties of the document tree D . One instance of particular interest is the schema tree that summarizes all tag paths occurring in D in a single central data structure. This chapter deals with a different kind of structural summary that captures tree relations between document nodes. These structural summaries are commonly referred to as *labelling schemes*.¹ Labelling schemes are decentralized summaries in the sense of Definition 2.5 on page 11. In other words, the information about tree relations between specific nodes in D is not stored in a global data structure such as the schema tree, but part of the representation of these nodes. The following definition stresses the decentralized nature of labelling schemes:

Definition 3.1 (Labelling scheme) *A labelling scheme (or tree encoding) for a document tree D is a decentralized structural summary of a specific set of tree relations in D . Each node in D is assigned a (typically unique) node label so that any of these relations between nodes in D can be inferred from their labels, without access to remote parts of D or to a global representation of the entire document tree. \square*

As an example of a most basic labelling scheme, consider the assignment of consecutive integer labels in a preorder traversal of the document tree. Figure 3.1 *b*. on the following page (right-hand side) depicts the preorder labelling for a small XML document shown in Figure 3.1 *a*. (left-hand side). It is easy to see that the node labels (i.e., preorder ranks) encode two of the tree relations introduced in Section 2.1, namely, *PrevElt* and *NextElt* (document order). In the following, let $pre(v)$ denote the preorder rank of a document node v . Given two nodes v and w in D , we have $NextElt_1^+(v, w)$ iff $pre(v) < pre(w)$ and $NextElt_i^j(v, w)$ iff $i \leq (pre(w) - pre(v)) \leq j$, and likewise for *PrevElt*.

With these formulae, a binary constraint $NextElt(q, q')$ in a query against D can be matched through some simple arithmetic calculations on the labels of possible matches to the query nodes q and q' . The next subsection compares different ways to match query constraints by inferring tree relations through the manipulation of node labels. In any case, to take advantage of a particular labelling scheme for the evaluation of XML queries, several conditions must be satisfied:

- The labelling scheme in question must support the efficient matching of at least some of the allowed query constraints.
- At indexing time, node labels must be created and stored persistently for all document nodes.
- During query evaluation, there must be a way to retrieve the node labels of matches to query nodes.
- In dynamic settings where the document contents change over time, the node labels must be kept up to date.

¹Synonyms for the term *labelling scheme* include *naming scheme*, *node identification scheme*, *numbering scheme* (for a numeric representation of tree relations), and *tree encoding* (on tree documents only).

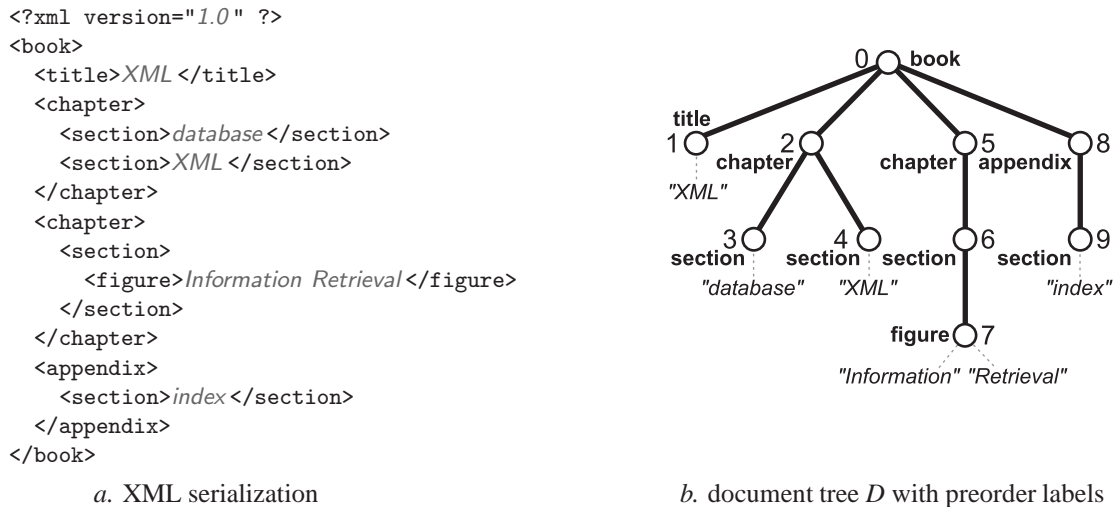


Figure 3.1: Preorder labelling of a sample document tree.

These conditions can be more or less easy to fulfill in a given retrieval system. Choosing a suitable labelling scheme depends on a number of factors:

1. *query language*: Which structural constraints are allowed? How is textual content retrieved?
2. *nature of the data*: How large is the document collection? Are the documents very heterogeneous in structure? Do they change often? If so, is the document structure affected or mainly their textual content?
3. *storage*: How are documents represented? Is it a native, hybrid, or relational system? How much storage space is available?
4. *retrieval*: How are document nodes retrieved? Which index structure are available? Does the system use a centralized structural summary?

Labelling schemes differ greatly in how well they fit a given query language and document collection in the presence of specific storage requirements or retrieval and indexing techniques. The following list includes the most salient properties of labelling schemes that need to be reconciled with the demands and constraints of the retrieval system:

1. *expressivity*: Which tree relations can be inferred from the node labels, and in which way?
2. *efficiency*: How fast is the manipulation of node labels during query evaluation?
3. *storage*: How much space is occupied by the node labels on disk and in memory? What is the average and the maximal label size?
4. *robustness*: How are the node labels updated when documents change? Do local changes affect a large number of labels?

Section 3.2 below rephrases the question of expressivity in a more precise way, introducing two distinct ways of matching non-unary query constraints that are fundamental not only in the context of labelling schemes, but also for all following contributions presented in this work. The rest of this chapter reviews a number of different labelling schemes from the literature and compares them in terms of their expressivity, efficiency for query evaluation, storage demands, and robustness against changes to the document collection. We explain representative approaches from three distinct classes of labelling schemes in detail (see Sections 3.3 to 3.5). The classification is based on fundamental principles underlying the different labelling procedures. The final comparison in Section 3.6 also highlights some open problems and possible optimizations. To illustrate the great diversity of labelling schemes that have been developed over more than twenty years, we explicitly include references to many approaches that are not reviewed here. A more exhaustive survey of labelling schemes for XML and tree database is currently under way [Weigel and Schulz 2007].

3.2 Reconstruction and Decision of Query Constraints

It has been mentioned before that the core functionality of query languages for XML databases, when abstracting from language-specific details, is typically captured by unary predicates (e.g., node tests in XPath) and binary tree relations. Accordingly, the data and query models introduced in Chapter 2 comprise a set of unary and binary tree relations that are used to specify query constraints to be matched against nodes in the document tree. Algorithms for evaluating XML queries of such kind can choose from a spectrum of different strategies, with the following two extreme positions:²

1. We may use the unary query constraints 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 structural joins, which amounts to solving a decision problem for the tree relation specified by the corresponding binary query constraint.
2. Since candidate sets for unselective unary constraints can be very large, we may alternatively fetch only the candidate sets for more restrictive query nodes (e.g., query leaves with selective keywords). Given the matches to these nodes, candidates for other query nodes are computed from their labels in memory, without further I/O taking place. This requires the use of a suitable labelling scheme.

The latter option is particularly interesting for binary relations R that are *functional*, i.e., where the set $R(v)$ of R -successors of any given document node v contains at most one node. Examples of functional relations include *Parent*, *PrevSib* and *NextSib*, as well as any composition of these. The same applies to relations R that are selective in the sense that database nodes typically have only a small set of possible R -successors. These are, e.g., the transitive or reflexive-transitive closures of *Parent*, *PrevSib* and *NextSib*. Given a query containing a constraint $R(q, q')$ on two query nodes q, q' for such a relation R , if we already have a small candidate set for q , then the second option permits to compute all relevant candidates for q' efficiently. Especially when the unary constraints on q' are weak, obtaining a consistent candidate set for q and q' via decision instead might be costly.

In this section we extend the query model presented in Section 2.2 with some additional constraints in order to capture the aforementioned differences in how the matching is realized. For each tree relation R in Table 2.1 on page 9, let $f_R^{Dec} : V \times V \rightarrow \{0, 1\}$ be a binary Boolean function such that for any pair v, w of document nodes, $f_R^{Dec}(v, w) = 1$ iff $R(v, w)$ holds true. To compute $f_R^{Dec}(v, w)$ one obviously needs to know the labels of both v and w . In addition, for each functional tree relation R (e.g., *Parent* or *Parentⁱ*) let $f_R^{Rec} : V \rightarrow V$ be a unary node-valued function that computes exactly the unique R -successor of a given document node. Thus, $f_{Parent^i}^{Rec}(v)$ returns the only element w for which *Parentⁱ*(v, w) holds, namely, the i -th ancestor of v (if it exists). Note that for computing f_R^{Rec} it suffices to know a single node label, rather than two labels as needed for f_R^{Dec} .

In the sequel we refer to the computation of f_R^{Rec} as the *reconstruction* of R and to the computation of f_R^{Dec} as the *decision* of R . Among the many labelling schemes described in the literature, decision is a much more common feature than reconstruction. In fact, a scheme that is able to reconstruct a particular tree relation R (by computing f_R^{Rec}) can also decide R (since $f_R^{Dec}(v, w) = 1$ iff $f_R^{Rec}(v) = w$). Clearly the inverse is not true. Therefore the most expressive labelling schemes are those with reconstruction capabilities (see Section 3.6). Later it will be shown that labelling schemes capable of reconstructing some tree relations indeed tend to expedite the whole evaluation process, compared to schemes that only support decision. The reason is that deciding a tree relation involves the fetching and joining of a second set of nodes (possibly including false positives).

Table 3.1 lists additional query constraints symbolizing the reconstruction of different tree relations. The counterparts of *Parentⁱ*, *PrevSibⁱ* and *NextSibⁱ* from Table 2.1 on page 9 are *parentⁱ*, *prevSibⁱ* and *nextSibⁱ*, respectively. For instance, the function *parentⁱ* is equivalent to $f_{Parent^i}^{Rec}$. Note that these are partial functions because not every document node has an ancestor or sibling at distance i . Furthermore, we consider some functions which do not correspond to any of the binary relations in Table 2.1, but nevertheless

²McHugh et al. [1998] discuss a number of different query evaluation strategies which are more or less close to either of the two extremes. The strategies proposed by Li and Moon [2001], Zhang et al. [2001], Grust [2002], Bruno et al. [2002], Al-Khalifa et al. [2002], and Chien et al. [2002] rely entirely on decision, whereas Bremer and Gertz [2006] or Pal et al. [2004] employ reconstruction.

name	description	domain / range
$parent^i(v)$	i -th ancestor of v (reverse document order)	$V \rightarrow V$
$prevSib^i(v)$	i -th sibling left of v (reverse document order)	$V \rightarrow V$
$nextSib^i(v)$	i -th sibling right of v (document order)	$V \rightarrow V$
$i\text{-th-child}(v)$	i -th child of v (document order)	$V \rightarrow V$
$i\text{-th-ca}(v, w)$	i -th common ancestor of u and v (reverse document order)	$V \times V \rightarrow V$
$lca(v, w)$	lowest common ancestor of u and v	$V \times V \rightarrow V$
$sepLevel(v, w)$	level of the lowest common ancestor of u and v	$V \times V \rightarrow \mathbb{N}$
$distance(v, w)$	number of edges on the path from u to v	$V \times V \rightarrow \mathbb{N}$

Table 3.1: Reconstructible relations in the document tree.

have come up in the literature. The i -th child (from left to right) of an element v is computed by the function $i\text{-th-child}(v)$. Two binary functions reconstruct common ancestors of a given pair of elements. The first one, $lca(v, w)$, returns the lowest common ancestor of v and w (i.e., the last node in document order that is an ancestor of both v and w). The second function, $i\text{-th-ca}(v, w)$, reconstructs a common ancestor of both v and w at a specific distance i . Note that since any two elements are descendants of the document root r , the function lca always reconstructs an existing ancestor whereas the value of $i\text{-th-ca}$ may be undefined for some pairs of elements and a given parameter i . The level of the lowest common ancestor of v and w is computed by the function $sepLevel(v, w)$ (for *separation level* [Peleg 2000]), and $distance(v, w)$ returns their distance as defined in Section 2.1. It is easy to see that $distance(v, w) = level(v) + level(w) - 2 \cdot sepLevel(v, w)$.

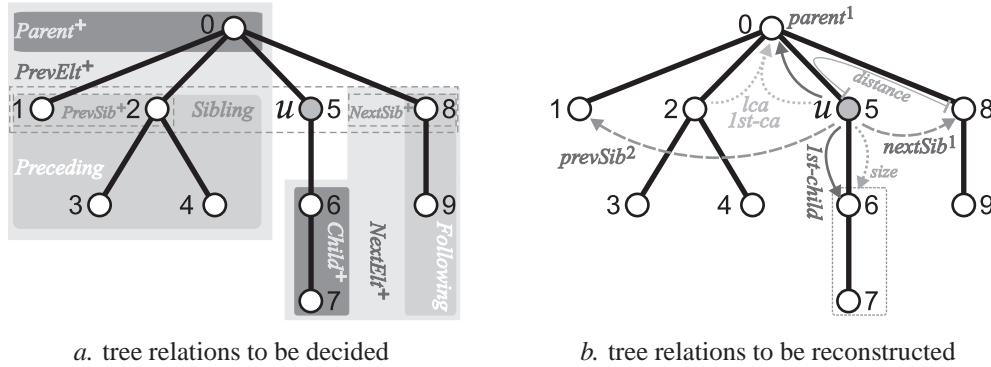

 Figure 3.2: Tree relations involving the node u that are to be decided or reconstructed.

Figure 3.2 illustrates tree relations that might be decided or reconstructed for a fixed node u in the document tree D from Figure 3.1 *b.* on page 18. In Figure 3.2 *a.* (left-hand side), each rectangular area contains the R -image of u for a particular relation R to be decided (i.e., all document nodes standing in relation R with u). For instance, the root of D is part of the $Parent^+$ -image of u . Note how the images of more general relations contain images of more specific ones. Thus the root node is also part of u 's $PrevElt^+$ -image, which contains the $Parent^+$ -image and the $Preceding$ -image of u . Such containment of tree relations is interesting when analyzing the expressivity of labelling schemes. From the observation just mentioned, e.g., one can conclude that a labelling that decides $Parent^+$ and $PrevElt^+$ also decides $Preceding$.

Figure 3.2 *b.* (right-hand side) indicates which nodes in D can be reached by reconstructing selected tree relations using the label of u . For instance, if a labelling scheme is capable of reconstructing $parent^i$, then the label 0 of the parent of u can be obtained from u 's label 5 without access to D . As observed above for decision, support for reconstructing certain relations implies the capability for reconstructing others. For instance, schemes that reconstruct $parent^i(v)$ also reconstruct $lca(v, w)$ and $i\text{-th-ca}(v, w)$, by iterating the ancestor reconstruction of either node and intersecting the resulting node sets.

3.3 Subtree Encodings

The simplest labelling schemes (apart from ordinary pre- or postorder) assign every node v in D a label representing the subtree D_v below v , as captured by the following general definition:

Definition 3.2 (Subtree encoding) *The class of subtree encodings subsumes those schemes where the label of a given document node v in D encodes the position and the extent of the subtree D_v of D that is rooted in v , by means of offsets in the sequence of nodes resulting from traversing (at least part of) the document tree in a specific order.* \square

The most common way to obtain this node sequence is a preorder or postorder or combined pre-/postorder traversal³ of the entire document tree, but not all approaches in this class follow that pattern. While the exact representation of the subtrees varies accordingly, for given nodes v, w in D $Child^+(v, w)$ is always decided by testing whether D_v contains D_w .

The labelling schemes in this class all decide more or less the same set of tree relations, but do not support the reconstruction of the node neighbourhood (see Table 3.2 on page 39). The subtree encodings reviewed below fall into three subclasses: *interval* (or *range*) *encodings* (see Section 3.3.1) label every node v with the interval spanned by the smallest and largest preorder ranks in D_v . The second approach (see Section 3.3.2) uses both pre- and postorder ranks to represent the subtree of a given node. In both cases, only elements are labelled while keyword occurrences are ignored. By contrast, a number of similar encodings for structured text documents (see Section 3.3.3) model subtrees as nested *regions* in the sequence of opening tags, closing tags and keywords forming the XML serialization of the document tree (see Figure 3.1 a. on page 18). A fourth class that is omitted here contains *leaf encodings*, which resemble interval-based schemes to some extent [Weigel and Schulz 2007].

3.3.1 Interval Encoding

Among the earliest labelling schemes that appeared in the literature, *interval encodings* were originally proposed for accelerating the routing in communication networks. Especially the often-cited work by Santoro and Khatib [1985] has inspired a number of simplified variants for structured documents. In Figure 3.3 on page 24 a couple of interval encodings are applied to the sample document tree in Figure 3.1 b. on page 18.

Pre/Max. The scheme in Figure 3.3 a. is sometimes called *Pre/Max*. Here each node v is labelled with the interval $I_v = [pre(v), max(v)]$ where $max(v) = \max\{pre(w) \mid w \in D_v\}$. As shown in the figure, I_v contain the labels of all descendants of v , which allows to decide the $Child^+$ relation as follows: we have $Child^+(v, w)$ iff $pre(w) \in I_v$.⁴ Furthermore, verify that $Following(v, w)$ iff $pre(w) > max(v)$; $NextElt^+(v, w)$ iff $pre(w) > pre(v)$; and likewise for inverse and proximity variants. Kannan et al. [1992] sketch this scheme using postorder ranks.

Order/Size. Note, however, that when inserting new nodes into the document both the lower and the upper bound of certain interval labels will need to be updated. Therefore, nodes are often labelled with their preorder rank and subtree size, from which the interval defined above is easily inferred. This saves the updating of the upper interval bound. The resulting scheme, described by Li and Moon [2001], is commonly referred to as *Order/Size* encoding in the literature. As can be seen in Figure 3.3 b. on page 24, for a given node v with the label $\langle pre(v), size(v) \rangle$, we have $max(v) = pre(v) + size(v)$ and therefore $I_v = [pre(v), pre(v) + size(v)]$. Chien et al. [2002] use Order/Size in a stack-based structural join algorithm.

Extended Preorder. Li and Moon [2001] also put forward a more robust variant of the Order/Size scheme, called *Extended Preorder*, which strives to reduce the impact of node insertions by reserving certain labels for future use. Others refer to this scheme as *durable node numbering* [Chien et al. 2001;

³The combined pre-/postorder tree traversal is explained in footnote 3 on page 7.

⁴Conceptually $Child^+(v, w)$ is decided by testing the interval inclusion $I_w \subset I_v$, but $pre(w) \in I_v$ (or, alternatively, $max(w) \in I_v$) can be checked more efficiently and is equivalent when assuming properly nested interval bounds, as in well-formed XML documents.

Yu et al. 2005]. The idea is simply to leave certain preorder ranks in the Order/Size scheme unassigned during indexing, which may then be used later for nodes newly inserted at the resulting gap positions. Each node v in a tree encoded using Extended Preorder is labelled with a pair $\langle order(v), offset(v) \rangle$ where $offset(v) \geq size(v)$. If $offset(v) = size(v)$ then $order(v) = pre(v)$ and Extended Preorder coincides with the Order/Size scheme in Figure 3.3 b. Greater offset values make the encoding more *sparse*, which means that more labels are available for subsequent node insertions. There is no definitive algorithm in the literature specifying which labels should be reserved for future use in this way. Clearly this depends on the nature of both the documents and the updating workload. Figure 3.3 c. on page 24 illustrates Extended Preorder with uniform gaps of size $s = 5$ between any two (opening or closing) tags in the XML serialization (symbolized by triangular subtrees). In this example, each node v_i in the complete sequence v_0, \dots, v_{n-1} of nodes in document order is labelled with a pair of integers $I_{v_i} = \langle order(v_i), offset(v_i) \rangle$ such that

$$order(v_i) = \begin{cases} 0 & \text{if } i = 0 \\ order(v_{i-1}) + s + 1 & \text{if } Parent(v_i, v_{i-1}) \\ order(v_{i-1}) + offset(v_{i-1}) + s + 1 & \text{otherwise} \end{cases}$$

$$offset(v_i) = \begin{cases} s & \text{if } size(v_i) = 0 \\ order(v_{j-1}) + offset(v_{j-1}) + s - order(v_i) & \text{otherwise} \end{cases}$$

where $j = \max\{l \mid Child(v_i, v_l)\}$ is the index of the rightmost child v_j of v_i .

SPaR. Due to its simplicity and (limited) robustness, Extended Preorder has been adopted in a number of systems. Chien et al. [2001; 2002; 2006] apply the scheme to multiversion document management. Their *Sparse Preorder and Range (SPaR)* labels are pairs $\langle dnn(v), range(v) \rangle$ consisting of a *durable node number* and a *range*, which are exactly the *order* and *offset* components described above.

3.3.2 Pre-/Postorder Encoding

The *Pre/Post* labelling scheme, proposed first by Dietz [1982] and later by Tsakalidis [1984], exploits the characteristic nesting of XML elements to decide the ancestor/descendant relation. Recall that enumerating all nodes in the order of their opening tags is equivalent to a (left-to-right) preorder traversal of the document tree, whereas visiting the nodes in the order of their closing tags yields a postorder traversal. As shown in Figure 3.3 d. on page 24, the Pre/Post scheme labels every node v in D with the pair $\langle pre(v), post(v) \rangle$ of its pre- and postorder ranks in D . It is easy to see that $Child^+(v, w)$ holds iff $pre(w) > pre(v) \wedge post(w) < post(v)$. Intuitively, this is because in the (well-formed) XML serialization, elements nested within v are opened after the opening tag of v and closed before the closing tag of v .⁵

XPath Accelerator. Grust [2002; 2004] arranges the node labels in a two-dimensional *pre/post plane* spanned by the pre- and postorder ranks in D . Figure 3.4 a. on page 25 shows the *pre/post* plane that corresponds to the sample tree D in Figure 3.3 d. on page 24. The root node of D , labelled with the smallest preorder rank and the greatest postorder rank in D , always resides in the upper left corner of the plane. As shown in Figure 3.4 a., $Child^+$, $Following$ and $NextElt^+$ as well as their inverse relations each correspond to a particular area relative to the context node v . For instance, the descendants of $v = \langle 6, 5 \rangle$ all lie in the shaded rectangle whose upper left corner represents v ; compare this to the formal containment test above. Similarly, the Pre/Post scheme decides the other relations as follows: $Following(v, w)$ iff $pre(w) > post(v)$; $NextElt^+(v, w)$ iff $pre(w) > pre(v)$; and analogously for the inverse relations.

The XPath Accelerator engine developed by Grust et al. [2002; 2004] extends Pre/Post with level and other information in order to decide all remaining XPath axes such as `attribute` or `preceding-sibling`. The R-Tree index [Guttman 1984; Böhm et al. 2000] is used to index the points in the resulting multidimensional plane. Grust also explains how to derive a lower bound on $pre(w)$ and an upper bound on $post(w)$

⁵Note that the interval containment test for interval encodings is *not* applicable to the Pre/Post scheme: as can be seen in Figure 3.3 d., there may be inner nodes v for which $pre(v) > post(v)$, such as the shaded node v with the label $\langle 6, 5 \rangle$. Deciding $Child^+$ based on the resulting empty interval $[pre(v), post(v)]$ would not reflect the fact that v does have descendants.

for deciding $Child^+(v, w)$, which restricts the search space spanned by the inequality statement above. This optimization, called *shrink-wrapping*, exploits an interesting property of the Pre/Post encoding:⁶

Lemma 3.3 (Pre/Post level/size dependency) *For any node v in the document tree, $pre(v) - post(v) + size(v) = level(v)$.* \square

Proof. Let a_v, d_v, p_v denote the number of ancestor, descendant, and preceding nodes of v in D , respectively. Note that the number of ancestors of v is equal to its level, i.e., $a_v = level(v)$. Similarly, $d_v = size(v)$ (see Section 2.1). Then we have

$$\begin{aligned} pre(v) &= |\{v' \mid pre(v') < pre(v)\}| &= a_v + p_v \\ post(v) &= |\{v'' \mid post(v'') < post(v)\}| &= d_v + p_v \end{aligned}$$

Figure 3.2 *a.* on page 20 illustrates the last equality in both lines. With the above observations on $level(v)$ and $size(v)$, it follows that

$$\begin{aligned} pre(v) - post(v) + size(v) &= a_v + p_v - (d_v + p_v) + size(v) \\ &= a_v - d_v + size(v) \\ &= level(v) - size(v) + size(v) \\ &= level(v) \end{aligned}$$

\square

For all leaf nodes v' , $size(v') = 0$ and therefore Lemma 3.3 becomes

$$pre(v') - post(v') = level(v') \leq h_D \quad (1)$$

The final inequality follows from the definition of the height h_D of the document tree D (see Section 2.1). To obtain upper and lower bounds on the pre- and postorder ranks of any descendant w of v , consider the leftmost and rightmost leaves w_l, w_r in the subtree rooted in v . (Of course, if no such leaves exist, $Child_v^+(w)$ fails for any node $w \in V$.) Given the position of w_l and w_r , the following facts are obvious:

$$post(w_r) \leq post(v) \quad (2) \quad pre(w_l) \geq pre(v) \quad (3)$$

From (1), (2) and (1), (3), respectively, Grust infers the following upper bound on $pre(w_r)$ and lower bound on $post(w_l)$. Note that since w_l has the smallest postorder rank and w_r the greatest preorder rank among all descendants of v , these bounds also apply to all other descendants w of v :

$$pre(w) \leq pre(w_r) \leq post(v) + h_D \quad (4)$$

$$post(w) \geq post(w_l) \geq pre(v) - h_D \quad (5)$$

Thus with shrink-wrapping, the decision of $Child^+$ is modified as follows:

$$Child^+(v, w) \quad \text{iff} \quad pre(w) \in [pre(v), post(v) + h_D] \wedge post(w) \in [pre(v) - h_D, post(v)]$$

using (4) and (5). To illustrate the benefit of this optimization, Figure 3.4 *a.* on page 25 depicts the restricted area of the *pre/post* plane to be searched for descendants of the node $v = \langle 6, 5 \rangle$. Note that while shrink-wrapping also applies to the *child*, *attribute* and *descendant-or-self* axes, there is no analogue for the *ancestor* axis: for all nodes u such that $Parent^+(v, u)$, the lower bound on $pre(u)$ and the upper bound on $post(u)$ are fixed by the document root which, by definition, has the smallest preorder rank and the greatest postorder rank in D .

While originally the XPath Accelerator engine was based on Pre/Post encoding, subsequent work by Grust et al. [2004] adopted a common variant of region encoding (introduced as Start/End encoding below) which is more favourable to B^+ -Tree indexing and at the same time less sensitive against node insertions. Recently, they adopted the Order/Size scheme [Boncz et al. 2005a] and combined it with a paging strategy [Boncz et al. 2005b] for further reducing the cost of updates. Besides shrink-wrapping, Grust et al. [2003; 2003; 2004] discuss the *Staircase Join* and various other optimizations for the efficient decision of tree relations in the two-dimensional node plane, which aim to reduce the range of index scans.

⁶The proof is omitted in the 2002 and 2004 papers by Grust.

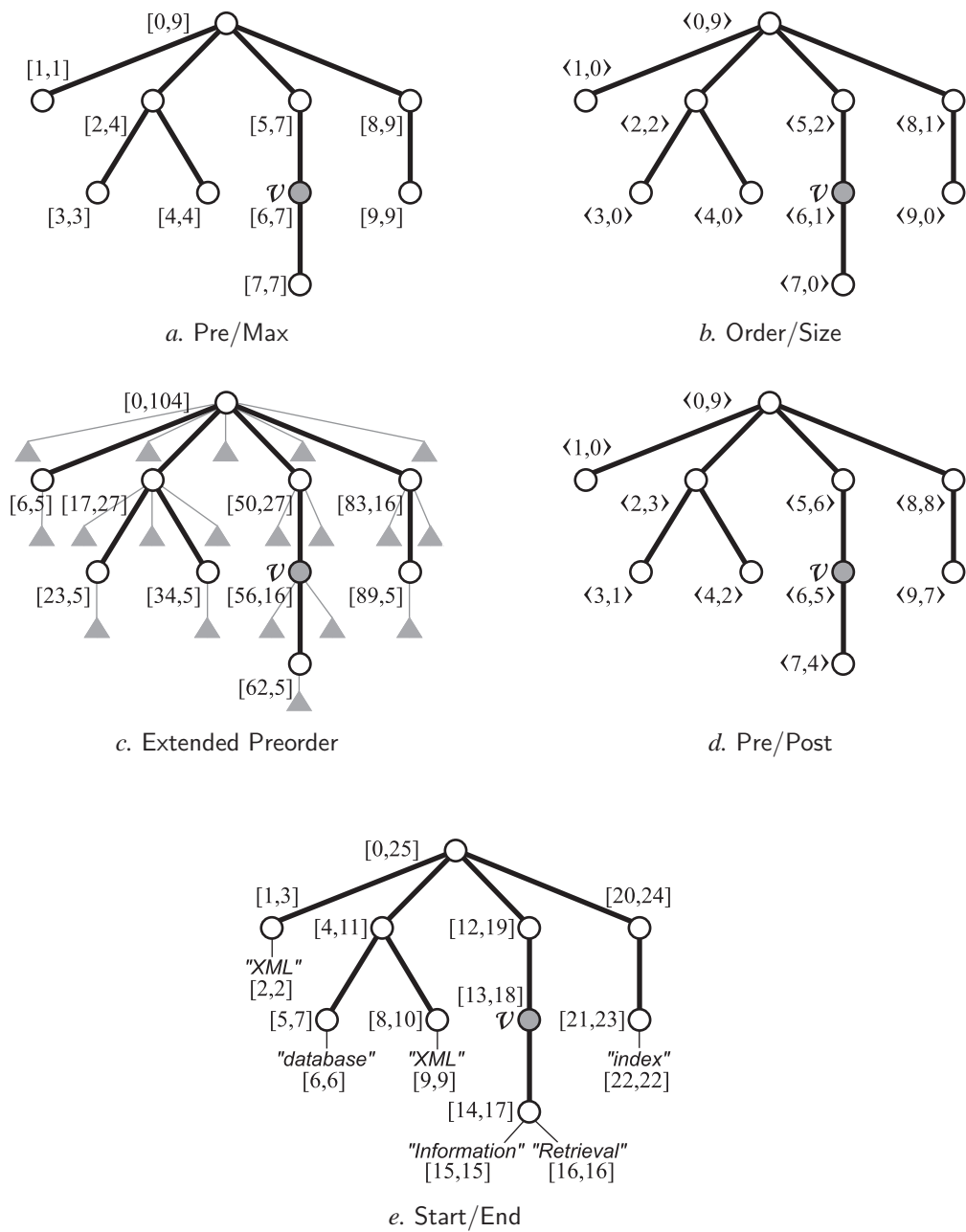


Figure 3.3: Selected subtree encodings applied to the document tree in Figure 3.1 b. on page 18. In c., shaded triangles symbolize subtrees of “virtual” nodes.

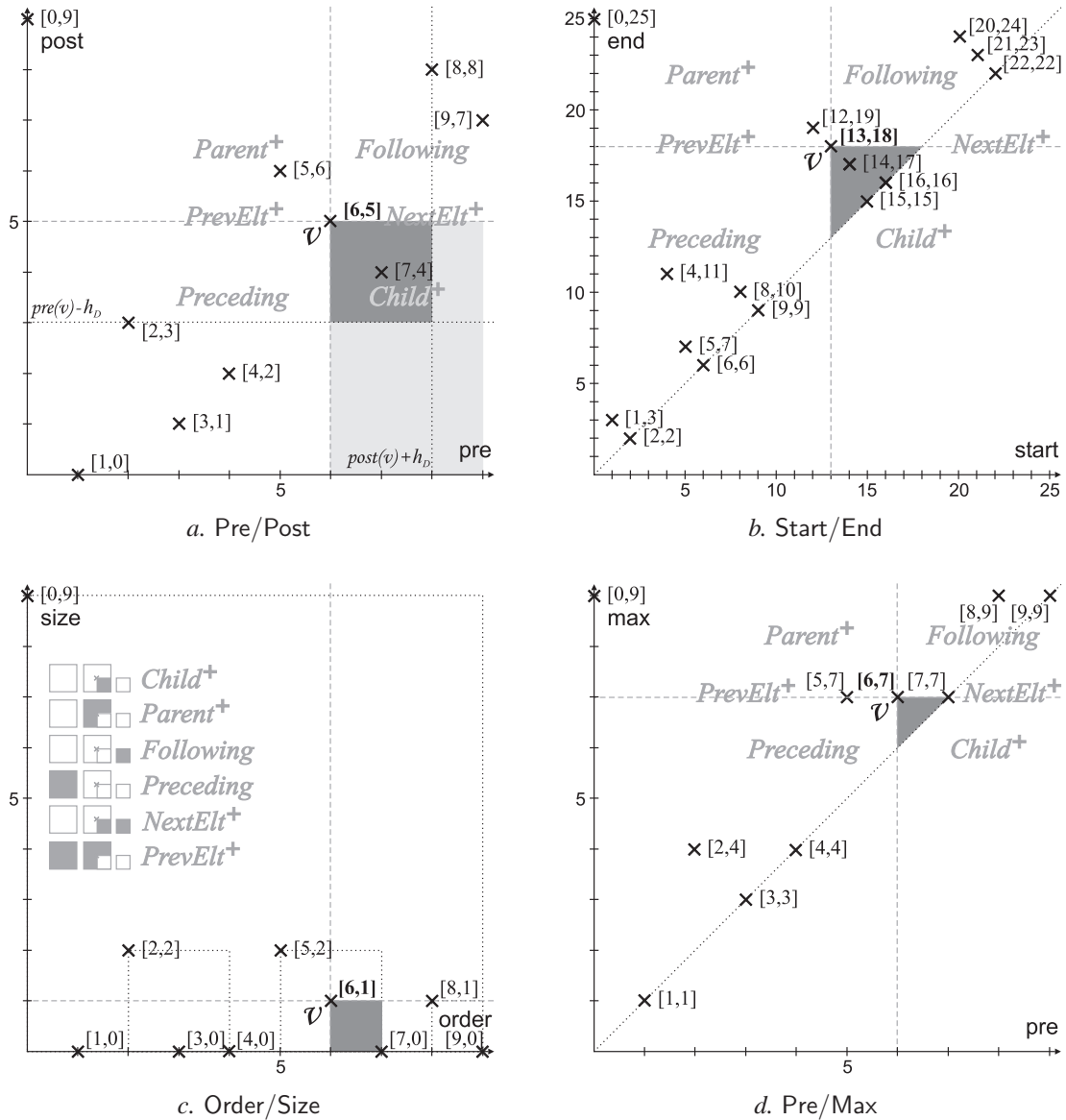


Figure 3.4: Two-dimensional representation of selected trees in Figure 3.3 on the preceding page: *a.* *pre/post* plane for Figure 3.3 *d.*; *b.* *start/end* plane for Figure 3.3 *e.*; *c.* *pre/size* plane for Figure 3.3 *b.*; *d.* *pre/max* plane for Figure 3.3 *a.* Descendants of a particular node v lie in the shaded area: *a.* range covered by $Child^+$ -images of $v = \langle 6, 5 \rangle$ (light/dark: without/with shrink-wrapping); *b.* $Child^+$ -images of $v = [13, 18]$; *c.* $Child^+$ -images of $v = \langle 6, 1 \rangle$; *d.* $Child^+$ -images of $v = \langle 6, 7 \rangle$.

3.3.3 Region Encoding

The third class of subtree encodings subsumes schemes that were originally designed for structured text databases. Modelling elements and text phrases as regions (substrings) in the serialized document, these schemes are most commonly referred to as *region encodings*. Unlike the Pre/Post scheme or the different interval encodings described above, which assign labels to XML elements (rather than tags or text strings) in a depth-first traversal of the document tree D , region encodings are not tied to the notion of a document tree whose nodes cover the whole textual extent of the document. As a matter of fact, early approaches like the *PAT* algebra by Salminen and Tompa [1992] or the *Region Algebra* by Consens and Milo [1994; 1995; 1998] abstract from the various ways to mark up structured documents, assuming the set of (possibly overlapping) regions to be somehow known at indexing time. This makes them amenable also to structured text documents with little or no explicit markup, such as program source code, Wiki documents, BIB_T_E_X files, or plain text that complies with specific formatting conventions.

The text-centric character of region encodings has two immediate consequences. First, since not only elements but also occurrences of keywords (or phrases) are labelled, the encodings are sensitive against changes to the textual contents of the documents. On the other hand, proximity constraints on keyword occurrences (which are a common feature of query languages for text databases) can be checked against the labels. Note, however, that region encodings do not retain information about the element distance in D unlike the tree-based encodings discussed above.

Second, with their notion of independent regions which is more general than the tree model underlying other subtree schemes, region encodings also capture overlapping elements which do not contain one another. Prohibited in well-formed XML, this “improper” nesting is common, e.g., in SGML documents and multi-hierarchical corpora. However, it is easy to verify that both the Pre/Post and the interval encodings described above can be applied to documents with overlapping elements (although this is typically not considered in the post-SGML literature). This observation is consistent with the fact that all three encoding variants are essentially alternative representations of the position and size of a node’s subtree.

Start/End. A number of region encodings have been developed over the years, the main difference lying in the representation of regions and the corresponding procedure for deciding region containment (i.e., *Child*⁺ in our data model). One particularly common scheme, which we henceforth refer to as *Start/End* encoding, labels every node v with the interval $I_v = [start(v), end(v)]$ spanned by the first and last visit to v in the combined pre-/postorder traversal of the document tree. Note that each keyword occurrence is now modelled as a *text node* in its own right.⁷ More formally, for each node v_i in the complete sequence v_0, \dots, v_{n-1} of structure and text nodes in document order,

$$start(v_i) = \begin{cases} 0 & \text{if } v_i \text{ is the root} \\ end(v_{i-1}) + 1 & \text{otherwise} \end{cases}$$

$$end(v_i) = \begin{cases} start(v_i) & \text{if } v_i \text{ is a text node} \\ start(v_i) + size_t(v_i) + 2 \cdot size_s(v_i) + 1 & \text{otherwise} \end{cases}$$

where $size_t(v_i)$ and $size_s(v_i)$ respectively denote the number of text nodes and structures nodes below v_i . The resulting labels are illustrated in Figure 3.3 *e.* on page 24. According to the first case in the definition of $end(v_i)$ above, each keyword occurrence has identical start and end positions, whereas for a structural leaf node v denoting an empty element, we have $end(v) = start(v) + 1$. (Occasionally structural leaves are assigned identical start and end positions, too [Halverson et al. 2003; Chen et al. 2005b].) The second case covers structure nodes without children, and ancestors of either text or structure nodes or both. This is a straightforward generalization of definitions in the literature [Grust et al. 2004] which typically do not consider mixed content. If nodes never have both text and structure nodes as descendants, then

$$end(v_i) = \begin{cases} start(v_i) & \text{if } v_i \text{ is a text node} \\ start(v_i) + size(v_i) + 1 & \text{if } v_i \text{ has text children} \\ start(v_i) + 2 \cdot size(v_i) + 1 & \text{otherwise} \end{cases}$$

⁷Contrast this with similar data models such as *DOM* [DOM] and *InfoSet* [InfoSet], where a text node may contain multiple keyword tokens.

Figure 3.4*b.* on page 25 depicts the two-dimensional *start/end* plane for the document tree in Figure 3.3*e.* on page 24. Note how all text nodes w lie on the diagonal where $start(w) = end(w)$. Below this line the plane is empty since $start(v) \leq end(v)$ for every node v . The effect is the same for Pre/Max in Figure 3.4*d.* on page 25, where the diagonal borderline is marked by the structural leaves (since there are no text nodes in this encoding). Compare this to the *pre/post* and *pre/size* planes on the left-hand side of Figure 3.4 (*a.* and *c.*, respectively), where the whole space is populated. The decision of $Child^+$, $Following$ and $NextElt^+$ described for the Pre/Max encoding above also applies to the Start/End scheme by analogy. Thus we have $Child^+(v, w)$ iff $start(w) \in I_v$ (or, alternatively, $end(w) \in I_v$), and so on.

3.4 Path Encodings

The largest and most diverse class of labelling schemes contains all approaches that create node labels from the paths leading to the nodes they designate, rather than their subtrees as in the previous section:

Definition 3.4 (Path encoding) *The class of path encodings (or prefix encodings) subsumes those schemes where the label of a given node v encodes (at least some of) the nodes on the path from the document root down to v , as a sequence of sibling codes each uniquely denoting an ancestor of v on that path.* \square

We choose the term *sibling code* to emphasize that for each step on the path leading down to v , the label must specify in which subtree (i.e., below which of the sibling nodes lying ahead) the node v is located. The full top-down sequence of sibling codes then uniquely identifies v . Due to their hierarchical nature, the labels of any path encoding respect document order iff the underlying sibling codes do. This is true, e.g., if siblings are simply assigned ascending integers from left to right as with Dewey encoding in Figure 3.6*a.* on page 31. By contrast, schemes that use tag-specific sibling codes cannot decide $NextElt^+$ (see below).

Creating node labels from paths has consequences regarding space consumption, robustness, expressivity and runtime performance that differentiate path encodings from the subtree encodings described above. First, the size of each label grows with the length of the encoded path ($O(n)$ in the worst case). Therefore path encodings typically take up more space than subtree encodings, whose label size is in $O(\log n)$. Some approaches come with binary encodings of the “raw” sibling code sequences that reduce the label size in practice, even though the asymptotic behaviour is not improved. Second, since a node’s label does not reflect the size of its subtree, path encodings are inherently robust against insertions in certain positions (such as adding children to leaf nodes). Third, from the path to a node v we can tell both the ancestors and the descendants of v , as follows. Since the root path to an ancestor u of v is always a prefix of the root path leading to v , $Child^+(u, v)$ holds iff the sequence of sibling codes that form the label of u is a prefix of the sequence of sibling codes in v ’s label. The same is true for the binary-encoded node labels, provided the codes for any set of siblings are prefix-free⁸. Thus the decision of $Child^+$ boils down to a comparison of bit strings. Furthermore, by removing a suffix of a specific length from the (raw or binary) label of v , we obtain the label of any ancestor of v . For instance, deleting the last sibling code in the label of v produces the label of v ’s parent node. This way path encodings support the reconstruction of $parent^i$, unlike all subtree encodings. As shown later, this makes a fundamental difference in query performance.

Path encodings fall into two subclasses. In contrast to *full path encodings* (see the next subsection) where all ancestors of a node contribute to its label, *partial path encodings* cover only fragments of a node’s root path (see Section 3.4.2). This reduces the label size, but in most cases also the reconstruction capabilities compared to full path encodings.

3.4.1 Full Path Encodings

Dewey. The most well-known path labelling scheme is certainly Dewey encoding⁹, which is used in the Dewey Decimal Classification [DDC] for libraries and was also adopted by the early hypertext search engine *HyTime* [Kimber 1993]. Dewey labels are assigned as follows. First all children of a given node

⁸A set of binary codes is *prefix-free* if none of the codes is a prefix of another code in the set.

⁹The labelling of nodes in a hierarchy with sequences of integers, similar to the section numbers in this paper, is called Dewey encoding after the American librarian Melvil Dewey (1851–1931), who used a restricted variant of this scheme for his Dewey Decimal Classification [DDC].

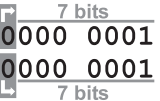
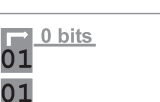
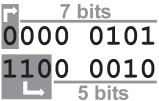
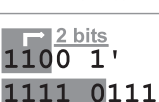
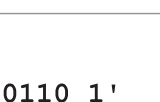
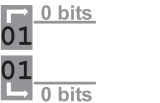
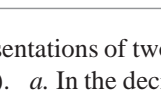
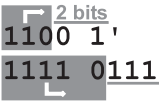
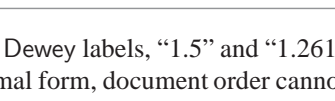
	LEVEL 0	LEVEL 1	BITS	
DEC.	1 1	5 261		<i>a.</i>
FIXED BINARY	'0000 0001' '0000 0001'	0000 0000 0000 0101' 0000 0001 0000 0101'	24 24	<i>b.</i>
BINARY	'0000 0001' '0000 0001'	0000 0101' 0000 0001 0000 0101'	16 24	<i>c.</i>
VARIABLE UTF-8	 '0000 0001'  '0000 0001'	 0000 0101'  1100 0010  0000 0101'	16 24	<i>d.</i>
ORDPATH	 '01'  '01'	 1100 1'  1111 0111 0110 1'	7 15	<i>e.</i>

Figure 3.5: Different representations of two Dewey labels, “1.5” and “1.261”, and their impact on deciding *NextElt*⁺ (document order). *a.* In the decimal form, document order cannot be checked in a lexicographical string comparison, but only numerically after level alignment. *b.* With a fixed number of bits per level, the labels compare lexicographically, but may occupy needless space. Thus the second sibling code of the first label has been left-padded with a zero-byte (8 leftmost bits on level 1). *c.* Levels with a variable number of bits must be aligned (using separators) and left-padded with 0’s (first label, level 1) for comparison at runtime. *d.* UTF-8 byte prefixes (shaded) specify the length and value range of the following suffix (‘0’: 7 bits, values [0..127]; ‘110’+‘10’: 5+6 bits, values [0..2047] where [0..127] is unused). The prefixes of the second byte in each label permit lexicographical comparison without alignment. Prefixes also indicate level boundaries for reconstruction. *e.* The ORDPATH encoding is similar, but more compact for small sibling codes (‘01’: 0 bits, [1..1]; ‘10’: 1 bit, [2..3]; ‘110’: 2 bits, [4..7]; ‘11110’: 8 bits, [24..279]).

are given consecutive sibling codes in ascending order from left to right, starting from 1, as illustrated in Figure 3.6*a.* on page 31. The Dewey label of a node v is then simply the top-down concatenation of all sibling codes on the root path to v , with a dot as separator between every pair of sibling codes.

Notice that this guarantees unique node labels and also allows to decide document-order constraints, provided the labels are not compared as strings (where “1.261” would incorrectly occur before “1.5”) but numerically and level-wise (see Figure 3.5*a.*). As Tatarinov et al. [2002] point out, this can be achieved through two distinct types of binary label encoding: (1) Allocating a fixed number of bits for each level eliminates the need for separators, but may result in excessive label size since the greatest sibling code on a given level causes bits to be wasted in all labels with smaller sibling codes on that level (see Figure 3.5*b.*). (2) The same level may occupy a variable number of bits in distinct labels, depending on the corresponding sibling codes in these labels (see Figure 3.5*c.*). Two given labels for which the number of bits per level is known can be split into corresponding sibling codes to be compared numerically (by dynamically left-padding the shorter sibling code with 0’s, if applicable).

Tatarinov et al. [2002] adopt this second option, using UTF-8 as the variable-size label encoding.¹⁰ Here sibling codes in the range [0..127] occupy one byte whereas larger sibling codes are reserved two bytes. As shown in Figure 3.5*d.*, the sibling code boundaries in the binary label string can be inferred from the leading bits in each byte, which are prefix-free. Hence a query constraint *NextElt*⁺(v, w) could be decided by first cutting the binary labels of v and w into sibling codes and then comparing them pairwise, with left-padding where necessary. But since a one-byte UTF-8 code (which always begins with a 0) is lexicographically smaller than any two-byte code (whose first bit is fixed to 1), labels can even be compared without aligning and padding sibling codes, in a simple bitwise (or bytewise) left-to-right comparison. This speeds up structural joins of large node sets, where document order tends to be checked very frequently.

¹⁰UTF-8 is the byte-oriented encoding form of the Unicode character encoding (see www.unicode.org).

By contrast, deciding and reconstructing tree relations other than $NextElt^+(u, v)$ requires individual sibling codes to be manipulated, which must therefore be located in the (raw or binary) label strings first. For reconstructing $parent^i(v)$, the last i sibling codes are removed from the label of v . Obviously one must know the sibling code boundaries in the binary label to chop off the right bit-string suffix. For i -th-child(v), a new sibling code i is appended to the label. Note, however, that while this yields the label reserved for the i -th child of v , the existence of this child is not guaranteed. The same applies to the i -th right sibling of v , whose label results from incrementing the last sibling code in v 's label by i . Similarly, a decrement produces the label of $prevSib^i(v)$ (if the last sibling code was 1, then v has no left sibling). The labels of ancestors common to two given nodes u, v (e.g., $lca(u, v)$) are obtained by reconstructing and comparing the root paths of u and v . The separation level of u, v (see Section 2.1) is equal to the number of sibling codes in the label of their lowest common ancestor, $lca(u, v)$. Finally, $distance(u, v)$ can be computed from the level information that is inherent to the labels (see Section 3.2).

Deciding query constraints with Dewey labels is done as follows. $Child^+(u, v)$ constraints can be matched by reconstructing the ancestors of u and v and testing for equality, or alternatively, by checking whether u 's label is a prefix of v 's label such that the end of the former coincides with a level boundary in the latter. $Child^i(u, v)$ is decided by comparing u to the result of reconstructing $parent^i(v)$. $Sibling(u, v)$ holds true iff u and v differ only in their last sibling code. For $NextSib^+(u, v)$ v 's last sibling code must be greater than u 's; for $NextSib^i(u, v)$ the difference must be exactly i . $Following(u, v)$ holds true iff $NextElt^+(u, v) \wedge \neg Child^+(u, v)$. Deciding the proximity relations i -th- $Following(u, v)$ and $NextElt^i(u, v)$ would require $size(u)$ to be known, which cannot be reconstructed from Dewey labels (see above).

ORDPATH. The *ORDPATH* scheme by O'Neil et al. [2004] enhances Dewey in two respects. First, the binary label strings are created using a Huffman code [Huffman 1952] designed to reduce the (average and maximum) label length. Note that since Dewey assigns ascending sibling codes from left to right, small sibling codes close to the minimum value, 1, occur much more frequently than greater ones, at least in typical documents where the average node fan-out is low. The binary encoding proposed by O'Neil et al.¹¹ reduces the number of bits used for small sibling codes, at the expense of longer labels for nodes with large sibling codes on their path. Figure 3.6 *b.* on page 31 depicts a sample tree with raw and encoded *ORDPATH* labels. Each sibling code (separated by “|”) is preceded by a length component (terminated by “|”) indicating the number of bits used for the following sibling code value.¹² Values up to 7 take up less space than with UTF-8 (see the value ranges in the caption below Figure 3.5 *e.* on the preceding page); e.g., only 2 bits are needed for the most frequent value 1, compared to 8 bits with UTF-8. For values beyond 24, *ORDPATH* mostly requires more bits than UTF-8.

ORDPATH also comes with an update method, called *caretting-in* by O'Neil et al., which allows for (theoretically) unlimited node or subtree insertions at any position in the document tree, without affecting existing labels. To this end, for a newly labelled document tree only odd sibling codes are used, as shown in Figure 3.6 *b.*, whereas even codes are reserved for future insertions, as follows. There are three cases of insertion to be handled: (1) A node v is inserted as the only child of a former leaf u . Then v 's label is the label of u after appending an additional odd sibling code “1”. For instance, a child to be inserted below node “1.3.1” in Figure 3.6 *b.* would be labelled “1.3.1.1”. (2) A node v is inserted as the new leftmost (rightmost) child of an inner node u . Then v 's label is the label of the former leftmost (rightmost) child w of u after decrementing (incrementing) the last sibling code in w 's label by 2. Note that this may create negative sibling codes, which are also covered by the binary encoding. For instance, a newly inserted left sibling of node “1.3.1” would be labelled “1.3.-1”. (3) A node v is inserted between two adjacent children w and w' of u . Then v 's label is the label of u after appending the even sibling code (*caret*) that falls between w and w' , followed by a new odd code “1”. For instance, two new siblings w'' , w''' between “1.3.1” and “1.3.3” would be labelled “1.3.2.1” and “1.3.2.3”, respectively. Repeated insertions on the same path may create labels with multiple consecutive carets, such as “1.3.2.2.1” for another sibling between w'' and w''' . Note, however, that *ORDPATH* labels always end in an odd sibling code.

¹¹In fact, O'Neil et al. present two alternative encodings (optimizing large or small node fan-out), both of which are skewed toward reducing the length of smaller sibling codes. Details given here, in Figure 3.5 *e.* on the preceding page, and in the experimental evaluation in Chapter 4 apply to the second encoding.

¹²The bit-string separators “|” and “|” are used for illustration purposes only and are not present in physical storage.

It is easy to see that carets do not affect the decision of $NextElt^+(u, v)$ and $Child^+(u, v)$ (and hence, $Following(u, v)$, see above). Thus a bitwise comparison of the children of the node $u = "1.3"$ in the previous example correctly reflects the document order $"1.3.1" < "1.3.2.1" < "1.3.2.2.1" < "1.3.2.3" < "1.3.3"$. Also, all these labels are recognized as belonging to descendants of u , having $"1.3"$ as a common prefix. By contrast, when solving the vertical proximity problems $Child^i(u, v)$ or $parent^i(v)$ (and similarly, i -th-ca(u, v) or $lca(u, v)$) carets must be treated as zero-depth components that do not add to the proximity count i . Similarly, $Sibling(u, v)$ holds true iff u and v differ only in a suffix consisting of zero or more carets followed by the last (odd) sibling code; for $NextSib^+(u, v)$ v 's suffix must be bitwise greater than u 's.

Unlike Dewey, ORDPATH cannot decide $NextSib^i(u, v)$ because caretting-in blurs the distance between siblings. For the same reason, ORDPATH does not support the reconstruction of $prevSib^i(v)$, $nextSib^i(v)$ and i -th-child(v). Note, however, that this decrease in expressivity is compensated for by much greater robustness: with ORDPATH, a node v can be inserted at any position in constant time, whereas with Dewey all following siblings of v and their descendants must be relabelled.¹³ This also outweighs the increased storage consumption of ORDPATH due to the sparse encoding. The overhead compared to Dewey is not measured by the authors but turns out not to be dramatic in our own experiments (see Chapter 4). O'Neil et al. briefly outline how caretting-in can also be applied to a subtree encoding similar to Pre/Max.

Extended Dewey. While ORDPATH skips certain sibling codes to accommodate future node insertions, the *Extended Dewey* scheme by Lu et al. [2005] uses sparse sibling codes that allow to determine all tags on the root path of a node v in D from its label. More precisely, every sibling code in v 's label is mapped to the corresponding tag by a global data structure containing the necessary tag information from D (see below). In particular, for any tag t occurring in D we need to know its *child tags*, i.e., the set t_0, \dots, t_{c_t-1} containing all c_t distinct tags of nodes in D whose parent has the tag t . For instance, the root tag `book` of the sample document in Figure 3.6 c. on the next page has the child tags `title`, `chapter`, `appendix`, hence $c_{\text{book}} = 3$, whereas $c_{\text{chapter}} = c_{\text{section}} = c_{\text{appendix}} = 1$. Each set of child tags is assumed to be ordered in some arbitrary way (a possible order is indicated by the tag subscripts in Figure 3.6 c.).

Given an inner node u in D with tag t and child tags t_0, \dots, t_{c_t-1} ($c_t > 0$), sibling codes are assigned to all children of u from left to right (i.e., in document order), as follows. Any child v of u with tag t_i ($0 \leq i < c_t$) is assigned the smallest free sibling code $s \geq 0$ such that $s \bmod c_t = i$. For instance, let u be the root of the document tree shown in Figure 3.6 c. on the facing page. Besides, let $t = \text{book}$, $t_0 = \text{title}$, and $t_1 = \text{chapter}$, $t_2 = \text{appendix}$. Then the `title` child of u receives the sibling code $s = 0$ in order to meet the condition $s \bmod 3 = 0$ for `title` children of a `book` node. The sibling codes for the two `chapter` children of the root are not consecutive because both must satisfy $s \bmod 3 = 1$. Therefore the first `chapter` node has code 1 and the second has code 4 (the smallest $s > 1$ such that $s \bmod 3 = 1$). The sibling code for the `appendix` child happens to be the next free integer, 5 (for a second `title` child it would be 6 instead). As can be seen in Figure 3.6 c., text nodes are also labelled, using the fixed sibling code -1 . The root label is the empty word ε .

For decoding Extended Dewey labels, Lu et al. use a Finite-State Transducer (FST), as shown in Figure 3.7 a. on page 33 for the sample document D in Figure 3.6 c. on the facing page. The FST reads a sequence of sibling codes and outputs the corresponding tag sequence. There is one state for each distinct label in D , plus one extra state representing textual content (`PCDATA` in Figure 3.6 c.). The initial state, `book`, represents the root label in D . For each inner-node tag t with child tags t_0, \dots, t_{c_t-1} , there is a transition from t to t_i ($0 \leq i < c_t$) which accepts all sibling codes s such that $s \bmod c_t = i$, and outputs t_i . Furthermore, from every state representing a tag there is a transition to the `PCDATA` state which accepts the sibling code -1 and outputs `PCDATA`. To keep Figure 3.7 a. simple, these are shown for the `title`, `section` and `figure` tags only. As an example, the label $"4.0"$ of node v in Figure 3.6 c. on the next page enters the initial state, outputting `book`, then passes through the `chapter` state because $4 \bmod 3 = 1$, and finally reaches the `section` state since $0 \bmod 1 = 0$. The data needed to create the FST for D is obtained from a DTD or other schema, if available, or else collected in a first pass through the documents, before creating the node labels.

¹³Even though ORDPATH supports unlimited node insertions without invalidating existing labels, O'Neil et al. [2004] suggest a periodical relabelling at least for highly dynamic document collections, in order to avoid long chains of even sibling codes caused by repeated caretting-in at the same position in the document tree.

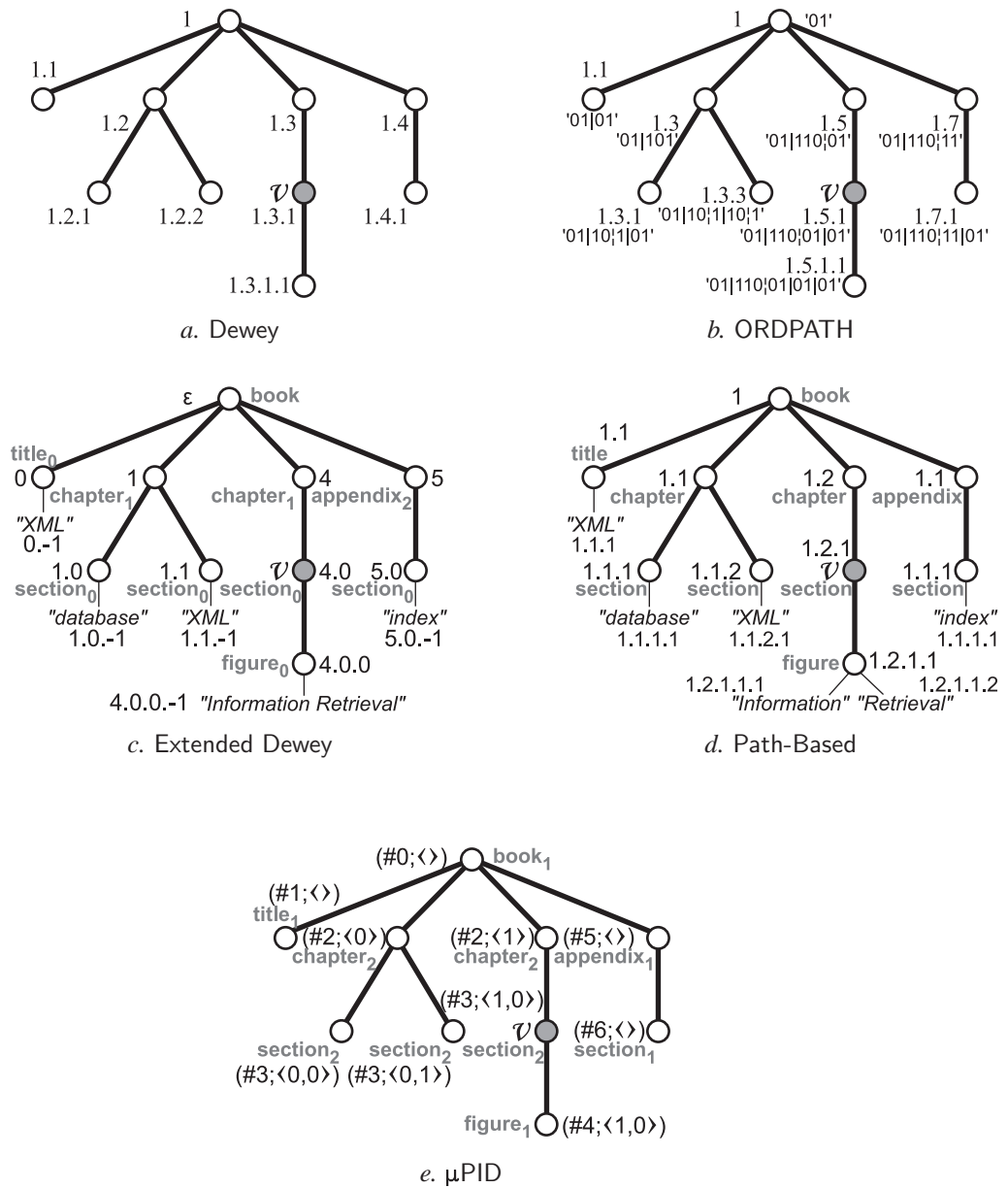


Figure 3.6: Selected path encodings applied to the document tree in Figure 3.1 b. on page 18.

Path-Based scheme. The sibling codes used by the Dewey-based approaches above set up a total order among the children of any node. Therefore they cannot handle overlapping siblings to be found, e.g., in SGML documents. By contrast, the *Path-Based* labelling scheme by Sacks-Davis et al. [1997] establishes a partial sibling order by assigning consecutive left-to-right sibling codes to all children *with the same tag name*.¹⁴ As shown in Figure 3.6*d*. on the preceding page, siblings with different tag names (such as the `title` and `appendix` nodes) may have the same sibling code (“1”) and hence identical labels (“1.1”).

The consequences of using tag-specific sibling codes are threefold. First, the Path-Based scheme supports overlapping siblings as long as they are distinguished by their tag name. Second, the labels do not reflect document order, because of the partial sibling order. For instance, in Figure 3.6*d*. the `appendix` node has a smaller label than its left sibling. Only nodes with the same sequence of tags on their root path can be compared. Third, since the numerical labels alone are not unique (see above), the tags of all ancestors of a given node must be compared when deciding *Child⁺* or *Sibling*: *Child⁺(u, w)* holds true iff *u*’s sequence of sibling codes is a prefix of *w*’s sequence and *u*’s tag path is a prefix of *w*’s tag path. For instance, let *u* be the first `chapter` in Figure 3.6*d*. on the previous page and let *w* be the `section` labelled “1.1.2”, which is directly below *u*. Without the second condition concerning the tag paths of the two nodes, *w* could be mistaken for a descendant of the `title` node or the `appendix` node because both have the same label as *u*, namely, “1.1”.

To make sure that all necessary information for deciding *Child⁺* is available during query evaluation, Sacks-Davis et al. keep the tag paths together with the node labels in an inverted index. In each posting for a given keyword (say, “XML”), the labels of nodes containing that keyword are grouped by their tag path which is stored once for each group. For instance, the “XML” posting for Figure 3.6*d*. on the preceding page would contain two singleton groups, one for the path `/book/title` (containing the label “1.1.1”) and another for the path `/book/chapter/section` (containing the label “1.1.2.1”). Mind the redundant storage of the path prefix `/book`, which can be avoided with other data structures (see the next subsection). Also note that unlike elements, text nodes (each representing a single keyword occurrence) have consecutive sibling codes, in order to support text distance queries. Thus the occurrence of “Retrieval” in Figure 3.6*d*. has the sibling code 2 for being the second keyword occurrence in the `figure` node, not 1 for being the first occurrence of “Retrieval” in that node.

3.4.2 Partial Path Encodings

The μ PID scheme by Bremer and Gertz [2006] relies on tag-specific sibling codes like the Path-Based encoding above, and also has the same expressivity. However, it uses several compression techniques and a binary encoding for reducing the storage consumption, as follows. Given a tag path *p* in *D*, let the *arity* of *p* be the maximum number of siblings with path *p* in *D*. First, Bremer and Gertz observe that when assigning tag-specific sibling codes ≥ 0 , all nodes with a tag path whose arity is 1 have the fixed code 0. For instance, consider the tag path *p* = `/book/chapter/section/figure` that occurs only once in the document tree *D* in Figure 3.6*e*. on the previous page. Since there are never two siblings with the path *p*, the arity of *p* is 1. For convenience, in Figure 3.6*e*. the arity of any tag path is indicated as a subscript to the last tag in that path (thus `figure` has the subscript 1). In fact, nodes whose tag path is either `/book/chapter` or `/book/chapter/section` are the only nodes in *D* whose subscript is greater than 1 (i.e., whose sibling codes are not fixed). All other sibling codes can be omitted from the labels without loss of information (see below), provided that (1) we record during the labelling the tag paths with an arity > 1 and (2) we know the tag path of every node. Recall from the description of the Path-Based scheme above that tag path information is required anyway with tag-specific sibling codes, both for ensuring node uniqueness and for deciding *Child⁺* constraints. The following paragraphs explain how the μ PID scheme realizes the two requirements that enable a very efficient label compression.

In order to avoid the redundant storage of duplicate tag path prefixes, as with the Path-Based scheme, Bremer and Gertz separate the node labels from the tag information and keep the latter in a centralized structural summary (in this case, a DataGuide). The summary for the sample document *D* in Figure 3.6*e*. on the preceding page is shown in Figure 3.7*b*. on the next page. Every DataGuide node has two properties: the arity of the tag path it represents (subscripts in Figure 3.7*b*.), and a unique path label (in this case,

¹⁴This tag-specific sibling coding is called *Same-Sibling Order Encoding* by Tatarinov et al. [2002].

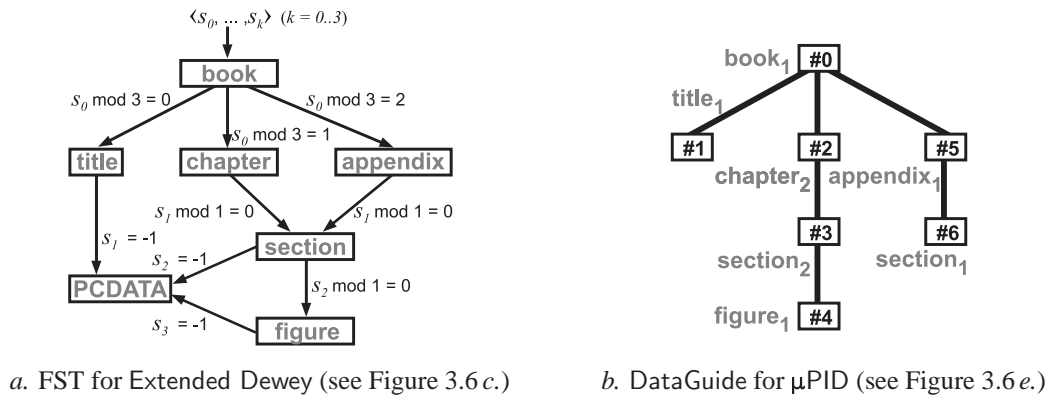


Figure 3.7: Global data structures used by selected labelling schemes in Figure 3.6 on page 31. *a.* The Finite-State Transducer (FST) for the Extended Dewey scheme, applied to the document tree D in Figure 3.6 *c.* The input is a sequence of sibling codes in D determining a unique path in the transducer. The output is the sequence of states (tags) along this path. *b.* Structural summary (DataGuide) for the μ PID-encoded tree D in Figure 3.6 *e.* DataGuide nodes have preorder labels. Tag subscripts specify the maximum number of siblings with a specific tag path in D .

preorder labels are used, although the particular labelling scheme does not matter). The label of a tag path in the DataGuide is referenced by all document nodes with that tag path: the μ PID label of a document node v in D is a pair $(\pi(v), pos(v))$ where the first component is the path label of the unique DataGuide node representing the tag path of v . With upward pointers in the DataGuide, the tags and arities along the entire root path of any document node are thus available without redundant storage. The second label component, $pos(v)$, is the *position path* of v , i.e., the sequence of non-fixed sibling codes on v 's root path. For instance, the node v with the μ PID label $(\#3, \langle 1, 0 \rangle)$ in Figure 3.6 *e.* on page 31 has the position path $\langle 1, 0 \rangle$ and references the DataGuide node #3. Note that the corresponding tag path of v , $/book/chapter/section$, comprises three steps whereas $pos(v)$ contains only two sibling codes. This is because one of the ancestors of v has a fixed sibling code that has been omitted during labelling in order to save space. Obviously, we need to know which code was dropped when using v 's label in decision or reconstruction operations. For instance, we cannot compute $parent^1(v)$ if we ignore whether the second sibling code in $pos(p)$ belongs to v (in which case it has to be removed) or to v 's parent (in which case it must be kept).

To align $pos(v)$ with the tag path represented by $\pi(v)$, the arities of DataGuide nodes are used as follows. From $\pi(v) = \#3$ and its ancestors in the DataGuide we can tell that the sibling code for v 's $book$ ancestor has been omitted: as indicated by the tag subscript 1 to $book$ in Figure 3.7 *b.*, the arity of the tag path $/book$ is 1, therefore the $book$ node in D has the fixed sibling code "0". In other words, the original position path of v before the compression was $\langle 0, 1, 0 \rangle$ (compare this to the Path-Based label $\langle 1, 2, 1 \rangle$ of the same node v in Figure 3.6 *d.* on page 31).¹⁵ Given this information, the (compressed) μ PID label of v 's parent is easily reconstructed as $parent^1(v) = (\#2, \langle 1 \rangle)$, where #2 is the parent of #3 in the DataGuide and $\langle 1 \rangle$ results from deleting the sibling code "0" of v in $pos(v) = \langle 1, 0 \rangle$.

The way μ PID decides $Child^+$ constraints is quite close to the Path-Based scheme. Given the μ PID labels of two nodes u and v , $Child^+(u, v)$ holds iff $pos(u)$ is a prefix of $pos(v)$ and $Child'^+(\pi(u), \pi(v))$ holds true in the DataGuide. Note that the prefix test is applied to the compressed position paths, without the need to restore omitted sibling codes. As a matter of fact μ PID labels are not even manipulated as raw sequences, but as packed bit strings consisting of fixed-length binary sibling codes. No level separators are used since the number of bits needed for the sibling codes of a particular tag path p is fixed to the base 2 logarithm of the arity of p , which is stored in the DataGuide. If $Child'^+(\pi(u), \pi(v))$ holds true, then the sibling codes in $pos(u)$ and $pos(v)$ are prefix-free and the two μ PID labels can be compared in their binary form. Otherwise u does not contain v anyway. Moreover, given the lengths of sibling codes specified

¹⁵Note that the sibling code "0" in the last step of $pos(v)$ is not fixed – e.g., the $section$ node left of v has code "1" – and hence cannot be omitted.

by the DataGuide nodes, the binary label of any node v can be decomposed for reconstruction without extra bits for length components or padding, as in Figure 3.5 on page 28. To sum up, Bremer and Gertz show how the use of a centralized structural summary for storing global document information permits effective compression and fast manipulation of node labels at the same time. In practice μ PID labels are very compact (see the experimental results in Chapter 4), although the worst-case label size is still $O(n)$, as for all path encodings.

3.5 Multiplicative Encodings

Finally, we would like to mention two labelling schemes whose decision or reconstruction capabilities are based on arithmetic properties of node labels in trees with a highly regular structure, such as binary trees or complete k -ary trees. In such a regular tree, labels can be assigned so that specific relations between nodes can be inferred from their labels alone, by simple numeric calculations. The idea is to find a mapping from a given irregularly structured document tree D to a regular tree D_ρ such that some of the arithmetic properties in D_ρ carry over to D :

Definition 3.5 (Multiplicative encoding) *The class of multiplicative encodings subsumes schemes where the label of a given document node v in D numerically encodes certain tree relations involving v , without direct reference to nodes in the neighbourhood of v . To this end the document tree D is (not necessarily physically) mapped to an internal representation D_ρ with certain structural regularities that generally do not hold in D , such as fixed node fan-outs or subtree sizes. These regularities entail arithmetic (typically multiplicative) invariants on the labels of nodes in D_ρ that are in a specific tree relation. The mapping ρ from D to D_ρ is such that tree relations between nodes in D can be decided or reconstructed by exploiting the invariants of their counterparts in D_ρ . \square*

Labelling with multiplicative encodings conceptually involves three consecutive steps. First, the document tree D to be labelled is analyzed in order to determine a suitable internal representation D_ρ of D . In the second step, labels are created for the nodes in D_ρ . Finally, each node in D is assigned the label of its unique counterpart in D_ρ . Note, however, that the labelling of D_ρ and D might happen simultaneously, i.e., steps two and three may be merged. In fact, the multiplicative approaches reviewed below do not even fully represent D_ρ physically in memory or on disk. Once a suitable mapping ρ from nodes in D to nodes in D_ρ is found, they simply traverse D and assign every node the label that it would have in D_ρ .

It has been mentioned above that every multiplicative encoding relies on specific structural regularities in D_ρ to decide or reconstruct tree relations in D . Since in most realistic cases D is not as regularly structured as D_ρ , however, not all desirable properties of the labels in D_ρ carry over to D . Therefore the first of two major challenges in devising a good multiplicative labelling scheme is to find a way of mapping *any* given document tree D to an internal representation D_ρ such that the desired expressivity of the scheme is guaranteed. The second challenge concerns the storage space needed for the resulting labels in D . As will become apparent in the rest of this section, the structural regularity of D_ρ typically causes D_ρ to contain many nodes that have no counterpart in D (so-called *virtual nodes*, a term coined by Lee et al. [1996]). In more formal terms, the mapping ρ from nodes in D to nodes in D_ρ is generally not surjective.¹⁶ In practice this means that a potentially large portion of the range of possible labels is not used in the labelling of D , being reserved for virtual nodes. Conversely, a much larger range of label values may be needed for indexing a given document tree D than when using a less sparse encoding such as, e.g., Order/Size or Pre/Post (see Sections 3.3.1 and 3.3.2, respectively). On the other hand, these labelling schemes are also far less expressive than some multiplicative encodings. The trade-off between expressivity and label size is further discussed in Section 3.6.

Virtual Nodes. The earliest multiplicative labelling scheme we know of has been proposed by Lee et al. in 1996. We refer to it as the *Virtual Nodes* scheme in the remainder of this work. As a very expressive but also very space-hungry approach, it is a good example for the aforementioned trade-off faced by some multiplicative encoding. Besides, the arithmetics behind this scheme is interesting in its own right.

¹⁶By contrast, the mapping ρ clearly is injective because otherwise the node labels in D would not be unique, unless they contain extra components besides the labels from D_ρ .

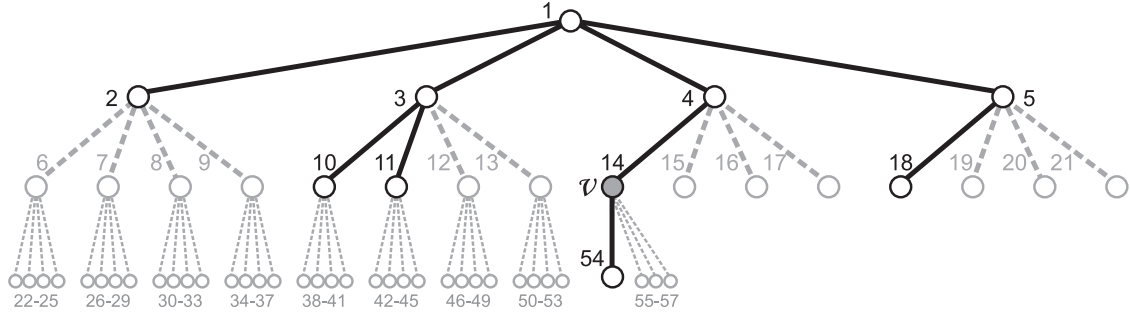


Figure 3.8: The multiplicative encoding Virtual Nodes applied to the document in Figure 3.1 b. on page 18. Shaded circles reached by dashed lines symbolize “virtual” nodes. Note the breadth-first label order.

When labelling a given document tree D using the Virtual Nodes scheme, the shape of the internal representation D_ρ of D depends on the *maximal fan-out* in D , i.e., the greatest number of children of any node in D . In the sequel, let k_D denote the maximal fan-out in D . Once the value of k_D is determined (usually, in a first traversal of the document tree), D is mapped to a k_D -ary tree D_ρ , as shown in Figure 3.8.¹⁷ Intuitively, D_ρ is obtained by adding virtual nodes to D : for each v in D , $\rho(v) = v$; if v does not comply with the definition of a k_D -ary tree, the missing nodes are added as rightmost virtual children to v . The example illustrates that the number of virtual nodes in D_ρ may be exponential in the height of D in the worst case.

The nodes in the k_D -ary tree D_ρ (including virtual nodes) are labelled with consecutive integer numbers ≥ 1 in a left-to-right breadth-first traversal of D_ρ . As shown in Figure 3.8, the nine nodes in D are given labels in the range [1..54], which illustrates the sparseness of the Virtual Nodes scheme. On the other hand, a number of tree relations can be decided and reconstructed using these labels. The following lemmata explain the reconstruction of *i-th-child* and *parent*¹ (the proofs have been omitted by Lee et al. in the 1996 paper):

Lemma 3.6 (Virtual Nodes child reconstruction) *Let D be a document tree with maximal fan-out k_D and let D_ρ be the k_D -ary tree that is used for labelling D with the Virtual Nodes scheme. Besides, let v be a node in D and let $\text{order}(v)$ be the label of v in D_ρ . If v has at least i children in D_ρ ($i > 0$), then the i -th child w of v in D_ρ has the label $\text{order}(w) = k_D \cdot (\text{order}(v) - 1) + i + 1$. \square*

Proof. For any node u in D_ρ , let p_u denote the number of nodes preceding u in a left-to-right breadth-first traversal of D_ρ . First of all, from the way Virtual Nodes labels are assigned, beginning with the label 1, it follows that $\text{order}(u) = p_u + 1$ for any node u in D_ρ . In particular, this is true for the desired child node w of v . So it remains to be shown that $p_w = k_D \cdot (\text{order}(v) - 1) + i$.

Obviously $p_w = p'_w + p''_w$ where p''_w is the number of left siblings of w , which are children of w 's parent v , and p'_w denotes the number of nodes preceding w that are not children of v . Given that w is the i -th child of v , there are $i - 1$ left siblings of w , i.e., $p''_w = i - 1$. The number p'_w is determined as follows. Since node labels are assigned consecutively in a left-to-right breadth-first traversal of D_ρ , all nodes preceding w that are not children of v are either predecessors of v or children of predecessors of v . We know that there are $\text{order}(v) - 1$ predecessors of v in D_ρ (see above) and that each of them has k_D children (none of them is a leaf, otherwise v could not have a child node w by definition of D_ρ as a k_D -ary tree). Hence there are $k_D \cdot (\text{order}(v) - 1)$ children of predecessors of v in D_ρ . However, the only predecessor of v that is not also a child of a predecessor of v is the root of D_ρ . It follows that $p'_w = k_D \cdot (\text{order}(v) - 1) + 1$. We conclude that

$$\begin{aligned} p_w &= p'_w + p''_w \\ &= k_D \cdot (\text{order}(v) - 1) + 1 + i - 1 \\ &= k_D \cdot (\text{order}(v) - 1) + i \end{aligned} \quad \square$$

¹⁷By a *k-ary tree*, we mean a tree in which every node is either a leaf or an inner node with exactly k children, and where no inner node is visited after a leaf node in the left-to-right breadth-first traversal of the tree.

The formula in Lemma 3.6 can be used to reconstruct i -th-child(v) for any node v in D , with an important restriction: the result is correct only for nodes that indeed have at least i children in D . Notice that this condition is even stricter than the one for D_ρ that is mentioned in the lemma. After all, even if the i -th child of v in D_ρ exists, we ignore whether it has a counterpart in D , i.e., it might be a virtual node. In this case the formula computes the label that is reserved for the i -th child of v in D , should it ever exist.

Lemma 3.7 shows how to reconstruct $\text{parent}^1(v)$ for any node v in D except the root (which is easily recognized by its fixed label 1). Higher ancestors are obtained by repeated parent reconstruction. Note that there is no additional restriction on v here because of the way how D is mapped to D_ρ (see Figure 3.8): one can show that for any node v in D_ρ , if v belongs to D then so do all its ancestors.

Lemma 3.7 (Virtual Nodes parent reconstruction) *Let D be a document tree with maximal fan-out k_D and D_ρ the k_D -ary tree that is used for labelling D with the Virtual Nodes scheme. Besides, let v be a node in D and let $\text{order}(v)$ be the label of v in D_ρ . If v is not the root of D_ρ , then the parent u of v in D_ρ has the label $\text{order}(u) = \left\lfloor \frac{\text{order}(v)-2}{k_D} \right\rfloor + 1$. \square*

Proof. As in the proof of Lemma 3.6, we define p_u as the number of nodes preceding u in a left-to-right breadth-first traversal of D_ρ , and similarly for p_v and v . Recall that $\text{order}(u) = p_u + 1$, hence $p_u = \left\lfloor \frac{\text{order}(v)-2}{k_D} \right\rfloor$ remains to be shown. First, assume that v is the leftmost child of its parent u . Then the number of predecessors of v that are not children of u is $\text{order}(v) - 1$. By the same argument as in the proof above, the parents of these nodes are exactly the p_u predecessors of u in D_ρ . Among the $\text{order}(v) - 1$ predecessors of v , the root of D_ρ is the only node that does not have a parent in D_ρ , so we are looking for the p_u parents of $\text{order}(v) - 2$ nodes. Given that exactly k_D children share the same parent, there must be $p_u = \frac{\text{order}(v)-2}{k_D}$ parents.

In general, if v is the i -th child of u ($1 \leq i \leq k_D$), then the number of predecessors of v that are not children of u is $\text{order}(v) - 1 - (i - 1)$, and accordingly $p_u = \frac{\text{order}(v)-2-(i-1)}{k_D}$. However, this is equal to $\left\lfloor \frac{\text{order}(v)-2}{k_D} \right\rfloor$ since $(i - 1) \leq k_D - 1$. \square

PBiTree. Unlike Virtual Nodes, which is fairly expressive, the *Perfect Binary Tree (PBiTree)* encoding by Wang et al. [2003a] only decides Parent^+ constraints. The document tree D is mapped (“binarized”) to a complete binary tree D_ρ ¹⁸ using an injective homomorphism ρ such that for any pair u, v of document nodes, $\text{Parent}^+(v, u)$ in D iff $\text{Parent}^+(v_\rho, u_\rho)$ in D_ρ . As with other multiplicative schemes, this may entail the creation of numerous virtual nodes needed to make D_ρ complete. (Again, the binarization need not take place physically.)

The label of a node v in D is the inorder rank of $\rho(v)$ in D_ρ . In the binary tree with its highly regular structure, ancestor reconstruction is possible. The ancestor of a node $\rho(v)$ at height h in D_ρ is computed as $\text{anc}(\rho(v), h) = 2^h \cdot \left\lfloor \frac{\rho(v)}{2^{h+1}} \right\rfloor + 2^h$. The ancestor reconstruction in D_ρ allows to decide Parent^+ in D , as follows: $\text{Parent}^+(v, u)$ holds true for nodes u, v in D iff $\rho(u) = \text{anc}(\rho(v), h)$ for some h . By contrast, deciding $\text{Parent}^i(v, u)$ for a specific proximity i is impossible because the binarization does not preserve the node levels and distances in the original tree D . To check whether u and v are at a specific distance in D , we would need to know the height of $\rho(u)$ in the binarized tree D_ρ —but there is no way to infer h from i . By the same argument, PBiTree does not support the reconstruction of parent^i in D .

3.6 Summary and Discussion

Labelling schemes are structural summaries that can match D -constraints on individual document nodes in a decentralized fashion, without accessing larger parts of the data. As such they are a fundamental building block for many different structural joins algorithms and a valuable complement to centralized structural summaries like the schema tree introduced in Section 2.3. This section has provided an overview of selected representative labelling schemes for XML documents. As mentioned before, there is a wealth

¹⁸By a *complete k -ary tree*, we mean a tree in which every node is either a leaf or an inner node with exactly k children, and where all leaf nodes are at the same level.

of contributions that have been made in more than twenty years, with a particular raise of activity during the last five to six years. An exhaustive survey is still missing in the literature but currently under way [Weigel and Schulz 2007]. It applies an extended version of the classification above to about thirty distinct labelling schemes (see Table 3.2 on page 39).

To compare and evaluate the various approaches, four characteristic properties and possible optimization goals have been suggested. The *expressivity* of a labelling scheme indicates which D -constraints can be matched using that scheme, and in which way. In this context we have proposed the terms *decision* and *reconstruction* to denote two very different matching techniques that are preferable in distinct retrieval situations. The impact on the matching performance is quantified in Section 4.6 and Section 8.5. Besides, the *runtime performance* of alternative labelling schemes varies for a given decision or reconstruction operation, which can affect the performance of the query processor, too. The *space consumption* on disk and in memory depends on the average and maximal label sizes. Finally, *updatability* (or *robustness*) deals with how well a given scheme can reflect structural or textual modifications of the underlying documents. Node insertions are particularly difficult to handle for certain classes of labelling scheme (see below).

As often, these major characteristics turn out to represent conflicting optimization goals. The rest of this chapter briefly compares the selected approaches above on each of the four fields, in an attempt to highlight the different priorities of the individual labelling schemes (and also classes of schemes) in the trade-off between space consumption, expressivity, efficiency, and robustness.

Space consumption. In the literature on labelling schemes, storage issues are sometimes regarded either from a more theoretical or from a more practical point of view, depending on the intended application scenario and also the community that a particular approach comes from. On the one hand, it is important to explore the asymptotic worst-case bounds on the label size, and much work has been devoted to obtaining ever tighter upper bounds, mostly by experts in the field of Discrete Mathematics (among others, Peleg [2000], Kaplan and Milo [2000], Abiteboul et al. [2001a], Alstrup et al. [2002]). However, some of these labelling schemes are rather complicated and therefore unlikely to be widely deployed. Also the significance of theoretical bounds for practical use is limited: thus some of the highly space-optimized schemes have been reported to perform worse than simpler ones [Kaplan et al. 2002], because the worst-case bounds tend to overestimate the space that is actually consumed when labelling real-world documents.

On the practical side, many techniques for reducing the space consumption have been developed, even though they may not improve the asymptotic behaviour. These techniques target either the average size or the maximal size of the labels that are created for a given document collection, depending on whether individual labels are stored using a variable or a fixed number of bits, respectively. For path encodings with their worst-case label size of $O(n)$, various binary encodings have been put forward, including the skew Huffman codes used by ORDPATH that reserve shorter bit strings for the most frequent sibling codes. Even for the less specialized UTF-8 encoding applied to Dewey and Extended Dewey, Lu et al. [2005] report space savings of up to 50% compared to raw labels. Unlike these approaches, the μ PID scheme does not encode the number of bits used for each sibling code in the labels, but stores this information in a centralized structural summary once for all nodes with the same tag path. This way the label size is reduced considerably. Some more space is saved by omitting codes of singleton siblings in the labels, which effectively compresses the labels. Compared to ORDPATH, μ PID encoding reduces the space consumption by up to 50% with variable-size labels, and even more for fixed-size labels (see Section 4.6 in the next chapter). In some cases the average and occasionally even the maximum size of μ PID labels is smaller than for preorder ranks [Bremer and Gertz 2006; Weigel et al. 2005d].

Extended Dewey, Path-Based and μ PID also exemplify alternative approaches to combining node labels with element tags using some sort of global data structure outside the labels. The Path-Based scheme keeps all distinct tag paths together with the node labels in an inverted index, which causes redundant storage but saves pointers. By contrast, Extended Dewey and μ PID labels reference the corresponding tag paths in a centralized structural summary (FST or DataGuide, respectively). While μ PID uses explicit pointers to DataGuide nodes, Extended Dewey chooses sparse sibling codes representing FST states. Which approach takes up less space depends on the structure of the documents: it affects both the size of the DataGuide (hence the length of the pointers) and the sparseness of the sibling codes. In general the FST has the same number of edges but fewer nodes than the DataGuide. On the other hand, while Extended Dewey supports

recursive DTDs, it cannot handle tags that occur below different tag paths in the documents, unlike μ PID. A modified FST with states representing tag paths rather than singleton tags would grow to the same size as the DataGuide.

Although subtree encodings enjoy a modest upper label size bound of $O(\log n)$, there are differences to observe. For instance, Grust et al. [2004] point out that with a fixed label size, region encodings like Start/End can represent only half as many nodes as Pre/Post. Intuitively, this is because node labels $[start(v), end(v)]$ with $start(v) > end(v)$ are prohibited (see the unused lower part of the two-dimensional label space in Figure 3.4*b*. on page 25). On the other hand, the Order/Size scheme is no sparser than Pre/Post (compare Figures 3.4*a*. and *c*.) but more robust against changes of the document tree (see the paragraph on updatability below).

Expressivity. Table 3.2 on the facing page summarizes the expressivity of the labelling schemes discussed in this chapter. Roughly speaking, one can observe two clusters in the table: one the one hand, there are very expressive schemes like most multiplicative encodings and the path encodings close to Dewey (upper half of Table 3.2). On the other hand, many other approaches are much more focused on specific decision or reconstruction problems. In particular, subtree encodings (lower part of Table 3.2) only decide $Parent^+$ and $NextElt^+$ constraints, but ignore sibling decision as well as any reconstruction besides subtree size. Combined with the fact that they are concise, fast in decision problems and easy to implement, this makes them a good choice for the set-at-a-time identification of ancestor/descendant pairs in structural joins.

The group of path encodings (middle part of Table 3.2) falls into three subgroups each with a different expressivity profile. The most expressive approaches with rich decision and reconstruction capabilities are those based on some form of Dewey encoding (rows down to and including Extended Dewey in Table 3.2). A second cluster subsumes a number of path encodings that are not compatible with document order, which prevents the decision and reconstruction of most horizontal tree relations. These includes schemes with tag-specific sibling codes, like the Path-Based and μ PID labellings, as well as so-called *layered* encodings (marked with an “ ℓ ” in Table 3.2). Layering is discussed in Section 4.5.2 below. Like the layered schemes, the remaining path encodings mostly stem from the graph theory and network routing communities and are mainly designed to minimize the label size and derive tight asymptotic bounds, at the expense of even more restricted expressivity. While guaranteeing a better worst-case label size, they tend to be far too complicated for productive use and are therefore primarily of theoretical interest. Note that due to their top-down approach to node labelling (see Section 3.4), all path encodings ignore the size of specific subtrees in the documents (last column in Table 3.2). The only exception, the *LCA* scheme by Peleg [2000], is in fact a combined path and subtree encoding.

Finally, the first two of the multiplicative encodings (upper part of Table 3.2) are highly expressive owing to the strong tree regularities they exploit (see Section 3.5). Note that the Virtual Nodes scheme does not recognize document order because it assigns node labels in a breadth-first traversal of the document tree. The BIRD scheme is presented in the next chapter. In contrast to these schemes, PBiTree encoding supports only the decision of $Parent^+$ because of the lossy mapping into its internal tree representation (see Section 3.5). Like path encodings, the multiplicative schemes are incapable of reconstructing *size*, due to their sparseness.

Runtime performance. Labelling schemes not only differ with respect to the choice of tree relations that they decide or reconstruct, but also in how fast this is done. For instance, ancestor reconstruction with Virtual Nodes or Dewey-based schemes may take time linear in the length of the label string (depending on the binary label encoding), whereas μ PID and BIRD compute the ancestor labels at any height in quasi-constant time. The asymptotic behaviour of these labelling schemes when faced with various reconstruction and decision problems is analyzed in earlier work [Weigel et al. 2005c].

Note, however, that assessing the efficiency of a labelling approach is complex, and hardly possible when taking only theoretical properties into account, because many factors can have a more or less tangible influence on the runtime performance of the query engine. Thus our experiments in the next chapter (see Sections 4.6.2 and 4.6.3) show that in practice, using reconstruction rather than decision has a huge impact while the difference between distinct reconstruction methods is often negligible.

labelling scheme		tree relation																
		decision								reconstruction								
multip.	BIRD ^g [Weigel et al. 2005d]	•	•	•	•	•	•	•	•	•	V	•	V	•	•	•	•	
	Virtual Nodes [Lee et al. 1996]	•	•	•	•	•	•	•	•	•	V	•	V	•	•	•	•	
	PBiTree [Wang et al. 2003a]	•																
path	Dewey [Tatarinov et al. 2002]	•	•	•	•	•	•	•	•	•	V	•	V	•	•	•	•	
	Simple Persist. [Cohen et al. 2002]	•	•	•	•	•	•	•	•	•	V	•	V	•	•	•	•	
	ORDPATH [O'Neil et al. 2004]	•	•	•	•	•	•	•	•	•				•	•	•	•	
	Extended Dewey ^g [Lu et al. 2005]	•	•	•	•	•	•	•	•	•				•	•	•	•	
	Path-Based ^g [Sacks-Davis et al. 1997]	•	•	•	•	•	•	•	•	•				•	•	•	•	
	μPID ^g [Bremer and Gertz 2006]	•	•	•	•	•	•	•	•	•				•	•	•	•	
	NCA ^l [Alstrup et al. 2002]	•	•	•	•	•	•	•	•	•				•	•	•	•	
	StatDL ^l [Korman et al. 2004]	•	•	•	•	•	•	•	•	•				•	•	•	•	
	<i>d</i> -Ancestor ^l [Kaplan and Milo 2000]	D	D	•							D			D	D	D,L	D	
	2-Lv. Parent ^l [Kaplan and Milo 2000]	P	P	•							P							
	<i>l</i> -Lv. Parent ^l [Kaplan and Milo 2000]	P	P	•							P							
	Parent [Kannan et al. 1992]	P	P	•	C	C			C	C	P							
	LCA [Peleg 2000]	•	L	L	L	L	•	•	•	•					•	L	L	•
	Simple Prefix [Kaplan et al. 2002]	•	L				•	•	•	•								
Compr. Prefix ^l [Kaplan et al. 2002]	•	L						•	•									
Distance ^l [Peleg 1999]								C	C						L	•		
subtree	Order/Size [Li and Moon 2001]	•	L				•	•	•	•							•	
	Pre/Max [Kannan et al. 1992]	•	L				•	•	•	•							•	
	Pre/Post [Dietz 1982]	•	L				•	•	•	•							L	
	Start/End [Salminen and Tompa 1992]	•	L				•	T	•	T							T	
	Netw. Lab. [Santoro and Khatib 1985]	I	I,L				I	I	•	•							•	
	Intv. Rout. [Gavoille and Peleg 2003]	I	I,L				I	I	•	•							•	
	Extended Pre [Li and Moon 2001]	•	L				•	•	•	•								
	GCL [Clarke et al. 1995]	•	L				•	•	•	•								
	Parenthesis [Abiteboul et al. 2001a]	•	L				•	•	•	•								
	2-Lv. Leaf ^l [Kaplan et al. 2002]	•	L				•	•	•	•								
	2-Lv. Interval ^l [Abiteboul et al. 2001a]	•	L				•	•	•	•								
<i>l</i> -Lv. Interval ^l [Abiteboul et al. 2001a]	•	L				•	•	•	•									
preorder								•	•									

V virtual (might not exist)

C extra label constraints

I inverse unsupported

T w/o text labelling

L with level/height

D distance limit

P parent only

• supported

■ relevant in XPath/XQuery

$i \in \mathbb{N}$

$u, v \in V$

l : layered scheme

g : needs global data

$size$: descendant count

$(l)ca$: (lowest) common ancestor

$sepLevel$: separation level (level of lca)

Table 3.2: Expressivity of different labelling schemes. Symbols in the individual columns indicate whether a specific tree relation is decided or reconstructed by a particular labelling scheme (see the key below the table for the meaning of the symbols). Shaded columns highlight tree relations that are of special interest for XPath or XQuery processors. The table includes many approaches not discussed in this work. For a detailed analysis and comparison of all labelling schemes, see the complete survey [Weigel and Schulz 2007].

Updatability. Modifying a document collection that has been indexed using a specific labelling scheme may affect the existing labels assigned to individual document nodes. The impact of a document update typically depends on any combination of (1) the labelling scheme used (e.g., path versus subtree encoding with or without text labelling), (2) the kind of update (insertion versus removal of a document, element, or text passage), (3) the location of the update (e.g., in a leaf node of the document tree), and (4) the extent of the update (e.g., how many nodes are added at a given position). While the removal of documents, elements or text passages is often handled simply by leaving labels unassigned, adding new content is more critical.

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). Under such circumstances, robustness is a minor concern, whereas storage demands and runtime performance are much more important. A straightforward solution is to reindex the entire document collection from time to time. On the other hand, in dynamic databases whose contents change frequently, like news repositories, auction servers, or flight booking services, such a strategy is clearly infeasible. Here node insertions must be done *incrementally*, i.e., without affecting too many of the existing node labels.

The class boundaries between subtree encodings, path encodings and multiplicative encodings also mark fundamental differences with respect to updatability. As observed by Yu et al. [2005], subtree encodings propagate topological changes bottom-up through the document tree D because the label (i.e., interval or region) of any document node is contained in the label of all its ancestors in D . For instance, if a newly inserted node causes the parent interval to overflow, then this might propagate up to the root of D . The situation is even worse for multiplicative encodings, where an overflow might propagate to disparate regions in D or even the entire tree (e.g., consider Virtual Nodes labelling when the maximal fan-out of D increases).

By contrast, path encodings propagate label updates top-down to all descendants, which inherit a prefix of the path label from their ancestor. As a consequence, path encodings can more easily accommodate an unknown number of future node insertions. Thus Dewey encoding naturally supports the adding of new rightmost siblings (in other words, unordered insertions) without the need for reassigning existing labels, provided labels can occupy a variable number of bits. Other path encodings use placeholder labels, either one below each node (for unordered insertions, as suggested by Kaplan et al. [2002]) or one between any two siblings (for ordered insertions at arbitrary positions in D , as suggested by O’Neil et al. [2004] for ORDPATH).

With fixed-size subtree or multiplicative encodings, the easiest way to prepare for a limited number of node insertions to come is to leave a certain number of labels unassigned during indexing. This technique was proposed by Li and Moon [2001] for Extended Preorder, among others. Note, however, that the size of the gaps must be fixed at indexing time, unlike the placeholders used with path encodings. If variable-size labels are admissible, floating-point rather than integer numbers may be used as interval bounds in order to make subtree encodings more robust. This idea was already hinted at by Santoro and Khatib [1985] and was later adopted by Chien et al. [2001] and Jagadish et al. [2002], too.

Alternatively, the *Relative Region* scheme by Kha et al. [2001] encodes the interval bounds of any node v in D relative to the interval bounds of the parent u of v . Similar encodings have been described by Tatarinov et al. [2002] (*Local Order*) and Sacks-Davis et al. [1997] (*Path-Based*). However, this means that $Parent^+(v, u)$ can no longer be decided from the labels of v and u alone, which spoils the local character of the labelling and may entail extra I/O during the query evaluation.

Kha et al. also suggest defining the interval bounds in terms of byte offsets rather than preorder node ranks. The same idea is pursued by Yoshikawa et al. [2001]. Note that this not only reduces the robustness of the labelling, but also prevents proximity matching in terms of the token distance. In the literature on structural joins for XML retrieval, where region encoding has been most influential (see above), the original representation using token positions is predominant.

The BIRD Labelling Scheme

4.1 Overview

To assess the cost and benefit of labelling schemes for XML, the preceding chapter has surveyed a choice of approaches with different features and weaknesses. We now present a new labelling scheme called *BIRD* (the acronym of *Balanced Index-based numbering scheme for Reconstruction and Decision*) that combines great expressivity and efficiency with modest storage needs and reasonable robustness. BIRD is the first in a sequence of interrelated contributions to be presented throughout this work. Together with the CADG index (Part III), it serves as a building block to the RCADG retrieval engine (Part IV), whose evaluation algorithm draws much of its power from the reconstruction capabilities of the BIRD scheme. The benefits of BIRD also carry over to incremental query evaluation with the RCADG Cache (Part V).

Before explaining BIRD in detail, let us briefly recapitulate on the role of labelling schemes for XML retrieval. Given more and more large collections of XML documents, efficiency and scalability are a major concerns. Matching query constraints directly in the documents is prohibitively expensive especially when using general-purpose storage infrastructure, such as an RDBS or a standard file system. Centralized structural summaries like the schema tree introduced in Chapter 2 partially shift the burden to the schema level, where weaker query conditions (so-called *S-constraints*) can be processed very fast. However, while this typically rules out some false positives, the results of the schema-matching stage must be checked once more on the document level against the exact query constraints (the *D-constraints*). Labelling schemes as decentralized structural summaries can assist in this document-matching stage, supplying information about the tree relations between individual document nodes through their labels. Rephrasing Definition 3.1 on page 17 a little more precisely, one can say that labelling schemes specify conventions for assigning unique labels to the nodes in the document tree that allow to decide or reconstruct specific *D-constraints* efficiently without access to the entire document tree, which saves I/O and possibly join operations.

Following the terminology introduced in the previous chapter, BIRD belongs to the small class of multiplicative encodings (see Section 3.5). Recall from Definition 3.5 on page 34 that unlike subtree or path encodings which derive the label of an element respectively from its descendants or ancestors, multiplicative labelling schemes exploit certain regularities in the document structure to encode tree relations numerically in the labels. In the case of BIRD, each document node is given a fixed *weight* at indexing time, and labels are assigned in such a way that the label of every node is a multiple of its weight. As shown later, this allows to reconstruct the ancestors, siblings and children of any given document node and to decide almost all tree relations in our data model. Thus BIRD is among the few most expressive labelling schemes known to the literature (see Table 3.2 on page 39). In particular, BIRD labels respect the document order, which greatly facilitates sort, join and merge operations on node sets (see Section 4.6). Of course using only labels that are multiples of specific weights leaves many possible label values unused. As other multiplicative labelling schemes, BIRD is in fact a rather sparse encoding that reserves many labels to so-called “virtual” nodes, i.e., additional document nodes that do not exist physically (and hence cannot be queried), but are assigned node labels nonetheless. Their existence is assumed solely for the sake of establishing structural regularities that are not manifest in the original document tree.

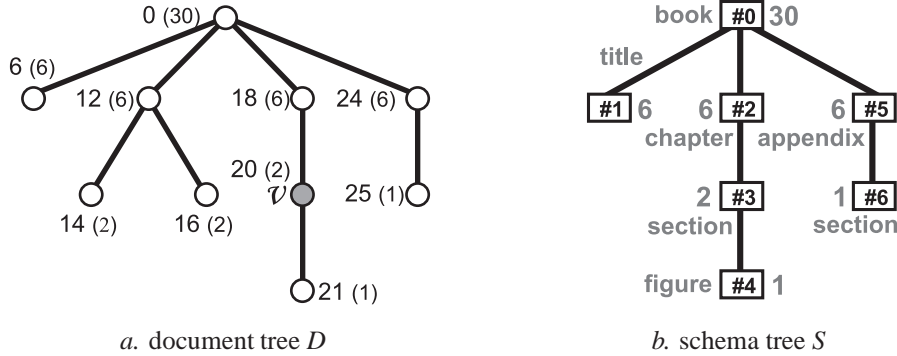


Figure 4.1: The multiplicative encoding BIRD applied to the document in Figure 3.1 b. on page 18.

For illustration, consider the document tree shown in Figure 4.1 a. Each node v is annotated with its BIRD label $\beta(v)$ (in bold face) and with its weight $\omega(v)$ (in parentheses). For instance, the shaded node v in the figure has the label $\beta(v) = 20$ and the weight $\omega(v) = 2$. Weights are not stored with the labels, but in a centralized structural summary such as the schema tree in Figure 4.1 b. All nodes in a. with the same tag path as v have the same weight, which is attached to the corresponding schema node $\pi(v)$ in b. (large numbers). Decision problems for any XPath axis can be solved based on the following observations. First, given two nodes v and w in the document tree, w is a descendant of v iff $\beta(v) < \beta(w) < \beta(v) + \omega(v)$. Thus node 21 is a descendant of v because $20 < 21 < 20 + 2$. To decide $\text{NextSib}^+(v, w)$, we test if $\beta(v) < \beta(w)$ and the two nodes v, w have the same parent node (parents are reconstructed, see below). $\text{Following}(v, w)$ holds true iff $\beta(w) \geq \beta(v) + \omega(v)$. For instance, node 25 follows v since $25 \geq 20 + 2$. Furthermore, given the BIRD label $\beta(v)$ of a node v and the weight $\omega(u)$ of any ancestor u of v (say, its parent node), we can reconstruct the BIRD label of u , which is $\beta(u) = \beta(v) - (\beta(v) \bmod \omega(u))$.¹ The parent of v in Figure 4.1 a., e.g., has the weight 6, hence the label reconstruction yields $20 - (20 \bmod 6) = 20 - 2 = 18$. This briefly illustrates how the BIRD labelling is used to decide the tree relations in our data model for two given document nodes and to reconstruct part of the tree neighbourhood (here, the root path) of a single given document node. The decision and reconstruction of other tree relations is discussed below.

We shall see that the BIRD scheme supports the decision of *all* XPath axes, as well as the reconstruction of all functional XPath axes (i.e., those containing at most one node by definition, such as the `parent` axis). Involving only trivial arithmetic calculations such as those shown above, the decision and reconstruction is very efficient, provided that fast access to the BIRD weights is available. To this end, the weights are stored in a centralized structural summary (e.g., the schema tree introduced in Chapter 1) that is typically small enough to reside in main memory. Matters of storage consumption are discussed below, where we introduce various variants of BIRD labelling schemes that offer distinct compromises between the expressivity of the scheme and the size of the resulting BIRD labels. The storage requirements are also influenced by the choice of the structural summary (see Section 4.8). In this sense, BIRD labelling actually defines a family of possible schemes. Our experimental evaluation (see Section 4.6) shows that BIRD outperforms various other tree labelling schemes in terms of runtime performance and expressivity, and that its storage behaviour and updatability are competitive on document collections up to the gigabyte range.

The next section explains how to create BIRD labels and weights for a given document tree to be labelled. The algorithms described there cover different variants of BIRD encoding. Section 4.3 presents a series of lemmata that show how to reconstruct tree relations from BIRD labels and weights. The decision of query constraints is covered by Section 4.4. These two sections formally prove the expressivity of the BIRD scheme that was claimed earlier (see Table 3.2 on page 39). Handling document updates with BIRD is discussed in Section 4.5, featuring two variants of the scheme that promises increased robustness. Section 4.6 reports on the outcome of our experimental evaluation and comparison of a number of different labelling schemes for XML data. Section 4.7 summarizes the contributions made by the competing schemes. We conclude in Section 4.8 with an outlook on further optimizations and open questions.

¹For integers i, j ($j \neq 0$), let $i \bmod j$ denote the unique integer $l \equiv i \pmod{j}$ such that $0 \leq l < j$.

4.2 The Family of BIRD Labelling Schemes

This section deals with how to create BIRD labels and weights at indexing time. Runtime operations for the decision and reconstruction of query constraints are covered in the next two sections. As mentioned above, a centralized structural summary is used for providing quick access to BIRD weights during query evaluation. In the sequel, we assume that the weights are stored in the schema tree, as shown in Figure 4.1 on the facing page. For a discussion of other structural summaries that can be used with BIRD, see Section 4.8.

Let D be a document tree to be labelled, and let S be its schema tree. As illustrated in the examples above, we have to make sure that for any given node v in D , the weights of v and all its ancestors in D can be obtained from S . Also, it has already been mentioned that for any node v in D , the BIRD label $\beta(v)$ of v must be a multiple of its weight $\omega(v)$. We thus enforce the following two invariants during the labelling:

1. *Weight invariant:* All document nodes with the same tag path have the same BIRD weight. For any node v in D , its weight $\omega(v)$ is stored in the node $\pi(v)$ in S .
2. *Label invariant:* During the labelling of D , the BIRD label $\beta(v)$ of any node v in D is determined to be the smallest unassigned multiple of $\omega(v)$ in document order.

Let $\omega(p)$ denote the BIRD weight stored in a given a schema node p in S . The first invariant states that for every document node v , $\omega(v) = \omega(\pi(v))$. Intuitively, the weight of v must be large enough to subsume the interval spanned by all BIRD labels in the subtree of D that is rooted in v . Note that in general this interval is larger than the number $size(v)$ of nodes in that subtree because v 's descendants are subject to the label invariant above. Also note that distinct document nodes with the same tag path may have different intervals. To comply with the weight invariant, we therefore choose $\omega(\pi(v))$ to be the largest interval for any document node with the same tag path as v . This idea is expressed formally in the weight definition below (see the next subsection).

Labelling a document tree D with BIRD is done in three phases. First, D is traversed once to determine for each document node v the number of children of v , which is later used to determine the aforementioned subtree interval of v . In the second phase, the schema tree S is traversed bottom-up to compute and store the weight $\omega(\pi)$ of every schema node p in S . Finally, in the third phase D is traversed again in document order to assign BIRD weights to all document nodes in D , based on the weights in S .

4.2.1 Creating BIRD Weights

To facilitate the labelling process, we only consider *balanced* variants of the BIRD scheme. Here the weights for schema nodes are unified among all children (or grand-children, etc.) of a given schema node in S .² The degree of balancing is controlled by the parameter b .

Let p denote a node in S , and let $b \geq 1$. By the *b -step ancestor* of p , we mean the unique ancestor of p in S that is reached from p in exactly b parent steps. As a matter of fact, the b -step ancestor of p is defined if and only if $level(p) \geq b$. Since the weight $\omega_b(p)$ of p in a b -balanced BIRD scheme is based on the maximal interval size among the siblings, cousins, grandcousins, etc. of p in S , depending on the value of b , the following definition of *b -equivalent nodes* is needed. Intuitively, two nodes are 1-equivalent iff they are siblings (i.e., share the same parent), 2-equivalent iff they are siblings or cousins (i.e., share the same grandparent), and so on.

Definition 4.1 (b -equivalence) Let S be a schema tree with set of nodes P . The equivalence relations \sim_b ($b \geq 1$) on the set P of nodes in S are inductively defined as follows:

1. for all $p, p' \in P$, $p \sim_1 p'$ iff the 1-step ancestors (i.e., parents) of p and p' are defined and coincide.
2. Let $b \geq 1$. For all $p, p' \in P$, $p \sim_{b+1} p'$ iff $p \sim_b p'$, or the $b+1$ -step ancestors of p and p' are defined and coincide.

If $p \sim_b p'$, we say that p and p' are b -equivalent. By $[p]_b$ we mean the equivalence class of the node p with respect to \sim_b . □

²There also exists an unbalanced variant that produces smaller weights and labels, but has disadvantages in terms of expressivity and memory consumption during label creation (see Section 4.8).

The following two definitions are key to the bottom-up creation of balanced BIRD weights in S :

Definition 4.2 (Child count) Let D be a document tree with set of nodes V , and let $v \in V$ be a node in D . The child count $\text{childCount}(v)$ of v is the number of children of v in D , i.e., $\text{childCount}(v) = |\{w \in V \mid \text{Child}(v, w)\}|$. \square

Definition 4.3 (Balanced BIRD weight) Let D be a document tree with set of nodes V , and let S be the schema tree for D with set of nodes P . Besides, let $b \geq 1$. The b -balanced BIRD pre-weight $\omega'_b(p)$ and the b -balanced BIRD weight $\omega_b(p)$ of a schema node $p \in P$ are recursively defined as follows:

$$\begin{aligned} \omega'_b(p) &:= \begin{cases} \omega_b(p') \cdot \max_{v \in V} \{\text{childCount}(v) + 1 \mid \pi(v) = p\} & \text{iff } p \text{ has a child } p', \\ 1 & \text{otherwise} \end{cases} \\ \omega_b(p) &:= \max_{p' \in P} \{\omega'_b(p') \mid p' \sim_b p\}. \end{aligned}$$

Finally, for every $v \in V$, the b -balanced BIRD weight of v is defined as $\omega_b(v) := \omega_b(\pi(v))$. \square

Note that the maximum operation in the definition of ω_b leads to unified weights for all b -equivalent schema nodes in S (balancing). It also guarantees the well-definedness of pre-weights ω'_b since any two children p', p'' of a schema node p have the same b -balanced weights $\omega_b(p') = \omega_b(p'')$. The final clause conforms to the weight invariant on page 43.

1-balanced weights are also called *child-balanced* weights. If b equals the height h_D of the document tree D , then $\omega_b(p)$ is called the *totally balanced weight* of the schema node p . In the remainder of this chapter, S_b denotes the variant of the schema tree S where the b -balanced weight $\omega_b(p)$ is attached to each schema node p , as illustrated for $b = 1$ in Figure 4.2 on the next page.

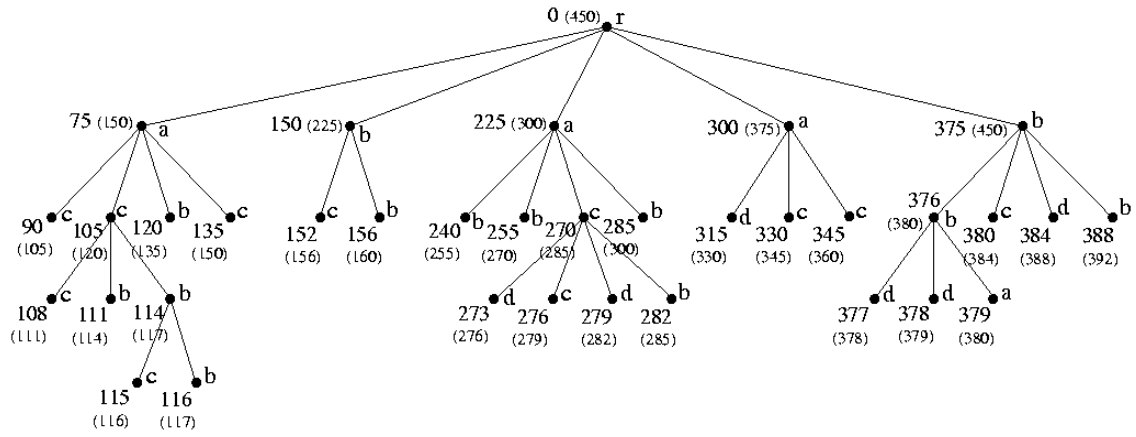
Example: Child-balanced BIRD weights. Consider the document tree D shown in Figure 4.2a. and the corresponding schema tree S_1 in Figure 4.2b. on the facing page, to which the child-balanced BIRD has been applied (i.e., $1 = 1$). Each node p in S_1 is annotated with its child-balanced weight $\omega_1(p)$ and, for convenience, the pre-weight $\omega'_1(p)$ (in parenthesis). If p has children, then Figure 4.2b. also depicts the number $\max_{v \in V} \{\text{childCount}(v) + 1 \mid \pi(v) = p\}$ that is used in Definition 4.3 above (written above the line next to p). Note that only the weights ω_1 are stored physically in the schema tree.

To understand how the depicted pre-weights and weights in S_1 are computed, consider the leftmost path in Figure 4.2b. The weighting procedure runs bottom-up and begins with the leaves $p_1 = /r/a/c/b/c$ and $p_2 = /r/a/c/b/b$, which respectively represent the document nodes 115 and 116 in D (larger numbers³ in Figure 4.2a.). Their pre-weight is fixed to $\omega'_1(p_1) = \omega'_1(p_2) = 1$ according to the first part of Definition 4.3. To compute the final weight $\omega_1(p_1)$ of p_1 , we must determine the greatest pre-weight of all schema nodes that are 1-equivalent to p_1 . Since $[p_1]_1 = [p_2]_1 = \{p_1, p_2\}$, we obtain $\omega_1(p_1) = \omega_1(p_2) = 1$ (second part of Definition 4.3). We next consider the parent node $p_3 = /r/a/c/b$ of p_1 and p_2 , which represents the document nodes 111, 114 and 282 in D . The nodes 111 and 282 have no children, but $\text{childCount}(114) = 2$ (see Figure 4.2a.). Therefore in the calculation of $\omega'_1(p_3)$, the child weight $\omega_1(p_1)$ is multiplied by a factor $2 + 1 = 3$ according to the first part of Definition 4.3. The resulting pre-weight of p_3 is $\omega'_1(p_3) = 3$. In the next step, the final weight $\omega_1(p_3)$ is computed: The bottom-up algorithm has already computed the pre-weight of the siblings $p_4 = /r/a/c/c$ and $p_5 = /r/a/c/d$ of p_3 , which is 1 because they are leaves. The weight of each of the three siblings p_3, p_4 and p_5 is the maximum of their pre-weights, i.e., $\omega_1(p_3) = \omega_1(p_4) = \omega_1(p_5) = 3$ according to the second part of Definition 4.3. On the higher levels, pre-weights and weights are computed in exactly the same way until we reach the root $/r$ of S_1 . Its 1-balanced BIRD weight is 450.

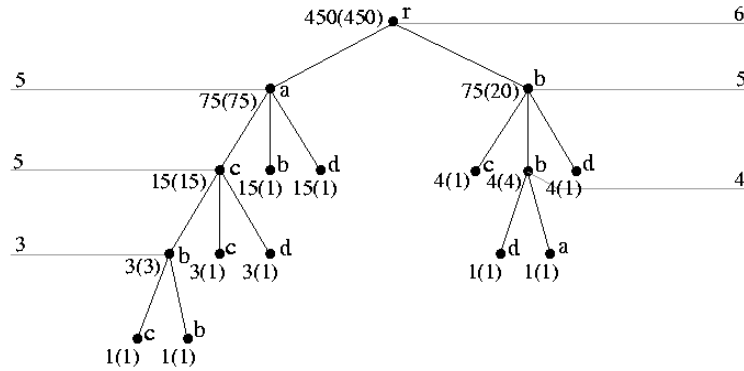
4.2.2 Creating BIRD Labels

We now describe the *b-balanced BIRD scheme*, which assigns an integer $\beta_b(v)$ to each node v in D , given the schema tree S for D that contains b -balanced weights as described above (see Definition 4.3). In the special case where $b = 1$, the scheme is called the *child-balanced BIRD scheme*. If $b = h_D$ represents the height of the document tree, we refer to it as the *totally balanced BIRD scheme*.

³For convenience, the example refers to document nodes by their BIRD labels, although the label computation is only explained later (see the next subsection).



a. document tree D



b. schema tree S_1

Figure 4.2: Child-balanced BIRD labelling ($b = 1$). a. A sample document tree D . For any document node v shown in the figure, the large number denotes the child-balanced BIRD label $\beta_1(v)$ of v , whereas the small number in parentheses denotes the upper bound of v 's subtree interval, i.e., $\beta_1(v) + \omega_1(v)$. b. The 1-balanced schema tree S_1 for D in a. For any schema node p shown in the figure, the large number denotes the child-balanced BIRD weight $\omega_1(p)$ of p , whereas the small number in parentheses denotes the corresponding pre-weight $\omega'_1(p)$ of p . For each non-leaf node p in S , the number $\max_{v \in V} \{childCount(v) + 1 \mid \pi(v) = p\}$ is indicated (see Definition 4.3 on the preceding page). Note that only BIRD labels and weights (i.e., large numbers in a. and b.) are stored physically.

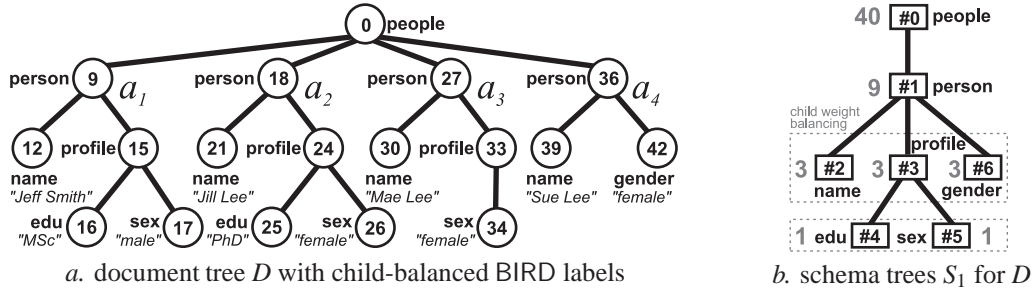


Figure 4.3: Child-balanced BIRD labelling applied to the sample document in Figure 2.1 on page 8.

Definition 4.4 (Balanced BIRD label) Let D be a document tree with root r , and let S_b be the schema tree for D with b -balanced weights as defined above, for a fixed $b \geq 1$. The b -balanced BIRD label $\beta_b(v)$ of a given document node v in D is recursively defined as follows. If $v = r$, then $\beta_b(v)$ is any multiple of $\omega_b(\pi(r))$ (e.g., $\beta_b(v) := 0$). Otherwise let u denote the parent of v in D , and let $\beta_b(u)$ be the b -balanced BIRD label of u . If v is the leftmost child of u , then $\beta_b(v)$ is the smallest multiple of $\omega_b(\pi(v))$ that is greater than $\beta_b(u)$. Otherwise let w be the immediate left sibling of v in D , and let $\beta_b(w)$ be the b -balanced BIRD label of w . Then $\beta_b(v) := \beta_b(w) + \omega_b(\pi(v))$. \square

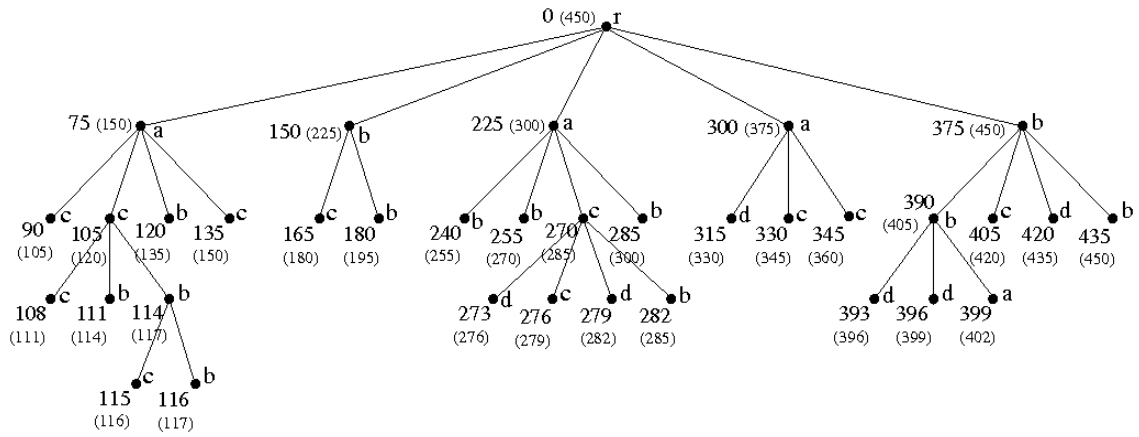
The recursive definition of the BIRD labels above translates naturally into a labelling algorithm that traverses D in pre-left order (i.e., document order) during the third of the aforementioned labelling phases. This ensures that the label invariant on page 43 is observed (see Lemma 4.6 below). Note that Definition 4.4 does not distinguish between specific values of the balancing parameter b . Instead the b -balancing is implied by the weights in S . Therefore exactly the same labelling algorithm is used for creating the labels of any b -balanced BIRD scheme.

Example: Child-balanced BIRD labels. In Figure 4.2 *a.* on the preceding page, each document node v is annotated with its 1-balanced BIRD label $\beta_1(v)$ (large number). The labelling starts with 0 for the root node, and traverses the document tree top-down left-to-right, as described above. Note that for any $b \geq 1$ the BIRD labels and weights are defined in such a way that all labels in the subtree rooted in a document node v are contained in the interval $[\beta_b(v), \beta_b(v) + \omega_b(v)]$. This important relation between labels and weights is established by Lemma 4.5 below. For convenience, the upper bound of the subtree interval is depicted as the small number in parentheses next to each node in Figure 4.2 *a.* Note that only the labels are stored physically, whereas the intervals are calculated from the labels and weights at runtime.

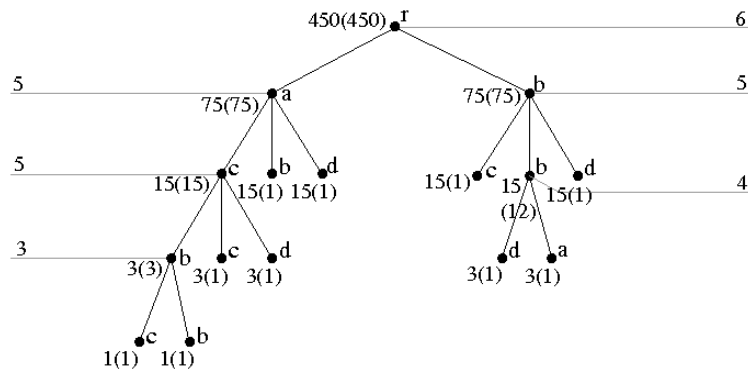
Example: Totally balanced BIRD labels. Figure 4.4 *a.* on the next page shows the same document tree as Figure 4.2 *a.* on the preceding page, but with BIRD labels computed for a balancing parameter of $b = 4$. Note that since D has height $h_D = 4$, the labelling shown in Figure 4.4 *a.* is the totally balanced BIRD scheme for D . Each database node v is annotated with its label $\beta_4(v)$ (large number) and with the upper bound $\beta_4(v) + \omega_4(v)$ of its subtree interval. The 4-balanced BIRD weights for the nodes in D are given as annotations to the corresponding nodes in the schema tree S_4 (see Figure 4.4 *b.*).

Example: Document tree and schema tree from Chapter 2. Figure 4.3 shows the document tree D and child-balanced schema trees S_1 for the XML fragment in Figure 2.1 *a.* on page 8. The document tree in Figure 4.3 *a.* differ from the one in Figure 2.1 *b.* in that preorder labels have been replaced with child-balanced BIRD labels. The schema tree in Figure 4.3 *b.* is the same as in Figure 2.1 *c.*, except that child-balanced BIRD weights have been added to the nodes.

The following two lemmata show that BIRD weights define subtree intervals for the labels of document nodes and their descendants in D . This observation is important because it guarantees the uniqueness of the node labels in D . In addition, it shows that the labelling function β is compatible with the document order $NextElt^+$ in D , in the sense that $NextElt^+(v, w)$ implies $\beta(v) < \beta(w)$ for any two nodes v, w in D .



a. document tree D



b. schema tree S_4

Figure 4.4: Totally balanced BIRD labelling ($b = h_D = 4$). a. A sample document tree D . For any document node v shown in the figure, the large number denotes the totally balanced BIRD label $\beta_4(v)$ of v , whereas the small number in parentheses denotes the upper bound of v 's subtree interval, i.e., $\beta_4(v) + \omega_4(v)$. b. The 4-balanced schema tree S_4 for D in a. For any schema node p shown in the figure, the large number denotes the totally balanced BIRD weight $\omega_4(p)$ of p , whereas the small number in parentheses denotes the corresponding pre-weight $\omega'_4(p)$ of p . For each non-leaf node p in S , the number $\max_{v \in V} \{childCount(v) + 1 \mid \pi(v) = p\}$ is indicated (see Definition 4.3 on page 44). Note that only BIRD labels and weights (i.e., large numbers in a. and b.) are stored physically.

Lemma 4.5 (BIRD label order) *Let D be a document tree with set of nodes V , and let $b \geq 1$. Besides, let v be a node in D , and let v_1, \dots, v_m denote the sequence of all children of v in document order. Finally, let $\omega := \omega_b(\pi(v_1)) = \dots = \omega_b(\pi(v_m))$. Then we have*

$$\beta_b(v) < \beta_b(v_1) < \dots < \beta_b(v_m) < \beta_b(v_m) + \omega \leq \beta_b(v) + \omega_b(v) \quad \square$$

Proof. Clearly $\omega \geq 1$ (see Definition 4.3 on page 44). Hence the inequalities $\beta_b(v) < \beta_b(v_1) < \dots < \beta_b(v_m) < \beta_b(v_m) + \omega$ follow from Definition 4.4 on page 46, and only the final inequality $\beta_b(v_m) + \omega \leq \beta_b(v) + \omega_b(v)$ remains to be proved.

Let $p := \pi(v)$ and let $p' := \pi(v_1) = \dots = \pi(v_m)$. Obviously $\omega = \omega_b(p')$. Since according to Definition 4.4 $\beta_b(v_{i+1}) = \beta_b(v_i) + \omega$ for all $1 \leq i < m$, we have $\beta_b(v_m) + \omega \leq \beta_b(v) + \omega \cdot (m+1)$. Furthermore, since v_1, \dots, v_m are child nodes of v , $\omega \cdot (m+1) \leq \omega \cdot \max_{w \in V} \{\text{childCount}(w) + 1 \mid \pi(w) = p\}$. With the first part of Definition 4.3, it follows that $\omega \cdot (m+1) \leq \omega'_b(p)$, which in turn is no larger than $\omega_b(p)$, according to the second part of Definition 4.3. Putting it all together, we have proved that

$$\beta_b(v_m) + \omega \leq \beta_b(v) + \omega \cdot (m+1) \leq \beta_b(v) + \omega'_b(p) \leq \beta_b(v) + \omega_b(p)$$

The final inequality in Lemma 4.5 follows directly from $\omega_b(p) = \omega_b(v)$. \square

Lemma 4.6 (BIRD labelling function) *Let D be a document tree with set of nodes V and root r , and let $b \geq 1$. Regardless of the initial assignment of $\beta_b(r)$, both of the following statements are true:*

1. For all $v \in V$, $\beta_b(v) \bmod \omega_b(\pi(v)) = 0$.
2. The labelling function β_b is injective and compatible with the document order in D . \square

Proof. The statement 1 follows immediately from Definition 4.4 and statement 2 from Lemma 4.5. \square

The final lemma in this subsection shows how the growth of node labels is limited by the height and branching degree of the document tree:

Lemma 4.7 (BIRD label size) *Let D be a document tree with height h_D , maximal fan-out k_D , set of nodes V and root r . Besides, let $b \geq 1$. Suppose that r is assigned the b -balanced BIRD label $\beta_b(r) := 0$. Then $\beta_b(v) \leq (k_D + 1)^{h_D}$ for all $v \in V$. \square*

Proof. Let S_b be the schema tree for D with b -balanced BIRD weights as defined above. Besides, for any schema node p in S_b let $\text{height}(p)$ denote the height of p in S_b , defined in the obvious way. A simple induction starting from leaves of the schema tree shows that for all p in S_b , $\omega_b(p) \leq (k_D + 1)^{\text{height}(p)}$. Since $\text{height}(\pi(r)) = h_D$, we have $\omega_b(\pi(r)) \leq (k_D + 1)^{h_D}$. The result follows from Lemmata 4.5 and 4.6. \square

4.3 Reconstruction of Tree Relations with BIRD

We now examine the runtime manipulation of BIRD labels during query evaluation. This section deals with the reconstruction of various tree relations from BIRD labels and weights that are stored in a centralized structural summary. Solving decision problems with BIRD is discussed in the next section.

In the sequel, let D be a document tree, and let S_b be the schema tree for D with b -balanced BIRD weights, for a fixed $b \geq 1$. Besides, let π be the mapping from document nodes in D to schema nodes in S_b , as defined in Chapter 2.

Lemma 4.8 (BIRD ancestor reconstruction) *Suppose that for some document node v in D we are given its BIRD label $\beta_b(v)$ and the schema node $p := \pi(v)$ in S_b . Let $i \geq 1$. Then using the weights in S_b we can solve the following tasks without access to D :*

- Decide if there exists an ancestor u of v that is reached from v with exactly (at least) i parent steps.
- In the affirmative case, get the BIRD label $\beta_b(u)$ and the schema node $\pi(u)$ corresponding to u . \square

Proof. Obviously, v has an ancestor u that can be reached with exactly i parent steps iff p has such an ancestor, p' , in S_b . By traversing the root path of p in S_b upwards, we may decide this question, finding p' in the affirmative case. By Lemma 4.6 on the facing page, $\beta_b(u)$ is a multiple of $\omega_b(p')$. It follows from Lemma 4.5 that $\beta_b(u)$ is the greatest multiple of $\omega_b(p')$ that is smaller than $\beta_b(v)$. As mentioned before, the BIRD label of u can be calculated as $\beta_b(u) = \beta_b(v) - (\beta_b(v) \bmod \omega_b(p'))$. \square

Lemma 4.9 (BIRD child reconstruction) *Suppose that for some document node v we are given its BIRD label $\beta_b(v)$ and the schema node $p := \pi(v)$ in S_b . Let $i \geq 1$. Then using the weights in S_b we can compute the BIRD label $\beta_b(v_i)$ of the i -th child v_i of v , assuming that this child exists, without access to D .* \square

Proof. From S_b we fetch the uniform weight $\omega := \omega_b(p')$ of any child p' of p . By Definition 4.4 on page 46, if $i = 1$ then $\beta_b(v_i)$ is the smallest multiple of ω that is larger than $\beta_b(v)$, and for $i > 1$ we have $\beta_b(v_i) = \beta_b(v_1) + \omega \cdot (i - 1)$. \square

Note that in general, we cannot directly compute the schema node $\pi(v_i)$ that corresponds to the i -th child v_i of v , unless we have further information (e.g., in the schema tree S_b we would need the tag of v_i). In any case, however, that we know the weight of v_i since $b \geq 1$, i.e., we use a child-balanced labelling scheme.

Lemma 4.10 (BIRD left-sibling reconstruction) *Suppose that for some document node v we are given its BIRD label $\beta_b(v)$ and the schema node $p := \pi(v)$ in S_b . Let $i \geq 1$. Then using the weights in S_b we can solve the following tasks without access to D :*

- Decide if v has exactly (at least) i siblings that precede v in document order.
- If v has at least i preceding siblings, get the number $\beta_b(v_i)$ of the i -th preceding sibling v_i of v . \square

Proof. We may assume that v has a parent node u (otherwise v has no siblings). Let $\beta_b(u)$ denote its BIRD label, calculated as described in Lemma 4.8 on the facing page. Besides, let $\omega := \omega_b(p)$. By Lemma 4.5 on the preceding page, v has at least i preceding siblings iff $\beta_b(u) < \beta_b(v) - i \cdot \omega$. From Definition 4.4 on page 46, it follows that v has exactly i preceding siblings iff $\beta_b(v) - (i + 1) \cdot \omega \leq \beta_b(u) < \beta_b(v) - i \cdot \omega$. If the i -th preceding sibling v_i of v exists, it has the BIRD label $\beta_b(v_i) = \beta_b(v) - i \cdot \omega$. \square

As for the i -th child (see above), we cannot directly compute the schema node corresponding to the i -th left sibling v_i of v , unless we have further information. The nodes v_i and v have the same weight since $b \geq 1$.

Lemma 4.11 (BIRD right-sibling reconstruction) *Suppose that for some document node v we are given its BIRD label $\beta_b(v)$ and the schema node $p := \pi(v)$ in S_b . Let $i \geq 1$. Then using the weights in S_b we can compute the number $\beta_b(v_i)$ of the i -th right sibling v_i of v , assuming that it exists, without access to D .* \square

Proof. Similar to Lemma 4.10 above. \square

Note that while we can decide whether a given document node v has a specific ancestor or left sibling (see Lemmata 4.8 and 4.10, respectively), the same is not true for children and right siblings (see Lemmata 4.9 and 4.11, respectively). Intuitively, this is because the only way to find out about the existence of such a node is to reconstruct the BIRD label that it would have if it existed, and then to check whether the assumption that this label is indeed assigned to the node in question causes a conflict, e.g., because there is no corresponding schema node or because there is overlap with the subtree interval of nodes whose label is already known. However, since the labelling is done in document order, these cases are easy to detect for nodes on reverse axes but much harder for nodes on forward axes (in the XPath terminology). As far as Lemmata 4.9 and 4.11 are concerned, we can only check if the child or right sibling to be reconstructed would fit into the subtree interval $[\beta_b(v), \beta_b(v) + \omega_b(v)[$ of the assumed parent node v . But it is impossible to decide whether it actually exists or whether it is only a virtual node.

The following lemma summarizes the reconstruction capabilities of balanced BIRD schemes, which are also listed in the first row of Table 3.2 on page 39.

$parent^i(v)$ parent	We proceed as in Lemma 4.8 on page 48.
i -th-child(v) child	We proceed as in Lemma 4.9 on the previous page.
$prevSib^i(v)$ preceding-sibling	We proceed as in Lemma 4.10 on the preceding page.
$nextSib^i(v)$ following-sibling	We proceed as in Lemma 4.11 on the previous page.
i -th-ca(v, w)	Starting from v , we visit all nodes on the root path of v bottom-up. Ancestors of v are reconstructed iteratively using the procedure described in Lemma 4.8 on page 48. For each node u on v 's root path (including v itself), we decide if $Child^*(u, w)$ holds true (see Lemma 4.14 in Section 4.4 below), until either the i -th decision test succeeds or the root of D is reconstructed. In the first case, the last reconstructed ancestor of v equals i -th-ca(v, w). Otherwise the value of i -th-ca(v, w) is undefined.
$Ica(v, w)$	The value of $Ica(v, w)$ is computed as i -th-ca(v, w) for $i = 1$ (see above).
$sepLevel(v, w)$	Let $u := Ica(v, w)$, computed as described above, and let $p' := \pi(u)$ be the schema node corresponding to u . It is easy to see that either $p' = p$ or p' is obtained during the ancestor reconstruction as described in Lemma 4.8 on page 48. The separation level $sepLevel(v, w)$ of v and w is the level of p' in the schema tree.
$distance(v, w)$	Let $level(v)$ and $level(w)$ be the levels of v and w in D , respectively. Furthermore, let $sepLevel(v, w)$ be the separation level of v and w , which is computed as described above. Then $distance(v, w) = level(v) + level(w) - 2 \cdot sepLevel(v, w)$.

Table 4.1: Relations reconstructible using any b -balanced BIRD scheme where $b \geq 1$. Given the BIRD label $\beta_b(v)$ of a document node v as well as the schema node $p = \pi(v)$ holding the weight corresponding to v , all binary functional relations are reconstructible without access to the document level. Analogously, given $\beta_b(v)$, p and the BIRD label $\beta_b(w)$ of a second document node w , all ternary functional relations are reconstructible without access to D . For $distance$, the level of w must be known, too. For each reconstruction problem $f(v)$ or $f(v, w)$, the reconstruction procedure is sketched as part of the proof of Lemma 4.12, and the corresponding XPath axis is given, if applicable, with v as context node. For example, $parent^i(v)$ denotes the i -th ancestor of node v , which is on the parent axis.

Lemma 4.12 (BIRD reconstruction) *Let D be a document tree, and let S_b be the schema tree for D that contains the b -balanced weights of the nodes in D , for a fixed $b \geq 1$. Suppose we are given the BIRD label $\beta_b(v)$ of the document node v in D and the schema node $p = \pi(v)$ in S_b that corresponds to v . Let f be any of the following functional relations: $parent^i$, i -th-child, $prevSib^i$, $nextSib^i$. Then, using the weights in S_b we can reconstruct the value of $f(v)$ without access to D .*

Furthermore, assume that in addition to $\beta_b(v)$ and p we are also given the BIRD label $\beta_b(w)$ of a second document node w in D . Now let R be any of the following relations: i -th-ca, Ica , $sepLevel$. Then, using the weights in S_b we can reconstruct the value of $f(v)$ without access to D .

Finally, assume that in addition to $\beta_b(v)$, $\beta_b(w)$ and p we are also given the level $level(w)$ of w in D (e.g., because $\pi(w)$ is known, too). Then, using the weights in S_b we can reconstruct the distance $distance(v, w)$ of v and w in D without access to D . \square

Proof. See Table 4.1. \square

The remainder of this subsection discusses some properties specific to totally balanced BIRD labelling. An attractive feature of totally balanced BIRD labelling is the following.

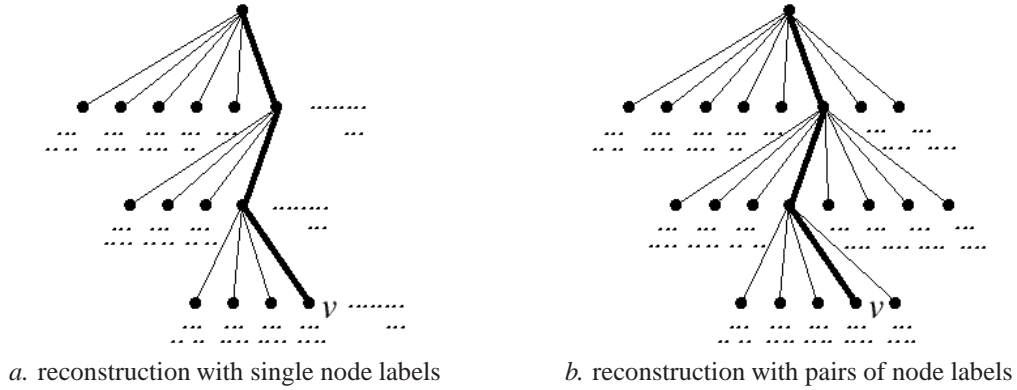


Figure 4.5: Reconstruction of nodes in a document tree D of height h_D that is labelled with the totally balanced BIRD scheme. *a.* The part of D that can be reconstructed given the label $\beta_{h_D}(v)$ and the weight $\omega_{h_D}(v)$ of the node v in D . *b.* The part of D that can be reconstructed from v using pairs of BIRD labels (preorder and inverse postorder).

Lemma 4.13 (BIRD totally balanced reconstruction) *Let D be a document tree of height h_D . Besides, let p be a schema node in S_{h_D} with a child p' . Then $\omega_{h_D}(p)$ is a multiple of $\omega_{h_D}(p')$.*

Furthermore, let $\beta_{h_D}(v)$ be the BIRD label of some document node v in D with children v_1, \dots, v_m in document order, and let $\omega := \omega_{h_D}(v_1) = \dots = \omega_{h_D}(v_m)$ be the balanced weight of the children of v . Then we have $\beta_{h_D}(v_i) = \beta_{h_D}(v) + i \cdot \omega$ for all v_i ($1 \leq i \leq m$). \square

Proof. The first statement is simply a consequence of the fact that all schema nodes at the same level of the schema tree are assigned the same weight by ω_{h_D} . By Definition 4.3, each balanced pre-weight ω'_{h_D} on the parent level is a multiple of this weight. Hence the same holds for the maximum of the pre-weights, which produces the weight on the parent level. The second statement follows easily. \square

Note that Lemma 4.13 generally does not apply to labelling schemes that are not totally balanced, i.e., where $b < h_D$. Figure 4.2 on page 45 illustrates this for $b = 1$ and $h_D = 4$. As a counterexample for the first statement in the lemma, consider the schema node $/r/b$ in Figure 4.2 *b.*, whose weight 75 is not a multiple of the weight 4 of its children in S_1 . In the document tree D shown in Figure 4.2 *a.*, the leftmost child 375 of the root and its four children illustrate that the second statement in Lemma 4.13 does not apply.

The following is a simple consequence of Lemma 4.13. Given a node v with the totally balanced BIRD label $\beta_{h_D}(v)$, the label $\beta_{h_D}(w)$ of any descendant w of v , specified in the form “ w is the i -th child of the \dots of the j -th child of v ”, can be computed without access to D , using the totally balanced BIRD weights stored in S_{h_D} . Note, again, that for $b < h_D$ we cannot guarantee the existence of this node without accessing D .

From the totally balanced BIRD label $\beta_{h_D}(v)$ of a node v in D we can reconstruct the weight $\omega_{h_D}(\pi(v))$, given the list of the uniform weights of all levels of the schema tree S_{h_D} . In fact $\omega_{h_D}(\pi(v))$ is the largest weight ω stored in our list such that $\beta_{h_D}(v) \bmod \omega = 0$. (As a by-product, the level of v is also obtained this way.) Hence, for $b = h_D$ Lemmata 4.8, 4.9 and 4.10 can be refined in the sense that we do not need to know the schema node p corresponding to v .

The higher the balancing degree b , the fewer nodes in the structural summary are needed for storing weights. For $b = h_D$, an h_D -tuple of weights suffices for the reconstruction of node labels. In special cases, however, it might be convenient to store the weights redundantly in all nodes of the summary. This is true, e.g., when using the schema tree which serves both as weight index and as path index during query evaluation.

The results obtained for the totally balanced enumeration scheme are summarized in Figure 4.5 *a.* Given the BIRD label $\beta_{h_D}(v)$ of a document node v , we immediately know how many ancestors u of v there are, and we can compute the labels $\beta_{h_D}(u)$ of all these ancestors without accessing the document level. Furthermore we can deduce the number of preceding left siblings w for each of these nodes as well as their labels $\beta_{h_D}(w)$. In the remaining regions of the tree (indicated by small dots in Figure 4.5 *a.*) we

know the label reserved for each node, yet we cannot decide which labels correspond to existing nodes and which ones are unassigned. This picture can be generalized through the use of a symmetric second labelling based on an *inverse postorder traversal* of the document tree. The inverse postorder behaves like a “right-to-left preorder”. If each document node is assigned a pair of labels according to a preorder (\rightarrow) and a “right-to-left preorder” (\leftarrow) traversal of D , then for a given node v with the labels $\langle \beta_{h_D}^{\rightarrow}(v), \beta_{h_D}^{\leftarrow}(v) \rangle$ we can compute the number of preceding and following siblings of v as well as their respective label pairs.

4.4 Decision of Tree Relations with BIRD

In this section we explain how to decide specific tree relations without access to the document level, using the BIRD labelling scheme. In the sequel we assume that a b -balanced BIRD scheme is applied to the document tree D , where $b \geq 1$. As before, let S_b denote the schema tree for D that contains the b -balanced BIRD weights, and let π be the mapping from document nodes in D to schema nodes in S_b , as defined in Chapter 2.

The first row in Table 3.2 on page 39 lists the set of tree relations that can be decided using balanced BIRD labelling. Comparing this to Table 2.1 on page 9 reveals that a large subset of the relations in our data model is covered. In fact, the only relations that cannot be decided by BIRD (besides *Self*, which is trivial to decide) are *i-th-Following* and *NextEltⁱ* as well as their reverse counterparts. Unlike the other proximity relations (e.g., *Parentⁱ* or *NextSibⁱ*), deciding any of these requires knowledge about the size of specific subtrees of D , which is spoilt by the sparseness of the BIRD labels (as for all multiplicative and most path encodings, by the way).

In the following we constructively prove that all other tree relations can be decided using BIRD. The following sixteen relations⁴ are mentioned explicitly: *Child*, *Child⁺*, *Child^{*}*, *NextSib*, *NextSib⁺*, *NextSib^{*}*, *Following*, the respective inverse relations, *Sibling*, and *Self*. For any such relation R and two document nodes v, w in D , we write $D \models R(v, w)$ iff in D the relation R holds between v and w i.e., if $f_R^{Dec}(v, w) = 1$. For instance, $D \models \text{Child}(v, w)$ iff w is a child of v in D .

The following lemma shows that using any balanced BIRD scheme, a superset of all XPath axes⁵ is decidable without any I/O operation. The XPath axes corresponding to the aforementioned relations are given in Table 4.2 on the facing page.

Lemma 4.14 (BIRD decision) *Let D be a document tree, and let S_b be the schema tree for D that contains the b -balanced weights of the nodes in D , for a fixed $b \geq 1$. Suppose we are given*

- the BIRD label $\beta_b(v)$ of the document node v in D ,
- the schema node $p = \pi(v)$ in S_b that corresponds to v ,
- the BIRD label $\beta_b(w)$ of a second document node w in D .

*Let R be any of the following relations: *Child*, *Child⁺*, *Child^{*}*, *Parent*, *Parent⁺*, *Parent^{*}*, *NextSib*, *NextSib⁺*, *NextSib^{*}*, *PrevSib*, *PrevSib⁺*, *PrevSib^{*}*, *Sibling*, *Following*, *Preceding*, *Self*. Then, using the weights in S_b we can decide if $D \models R(v, w)$ (or if $D \models R(w, v)$) without access to D . \square*

Proof. See Table 4.2 on the next page. \square

4.5 Handling Document Updates with BIRD

In this section we sketch two different update strategies to illustrate that the BIRD scheme is not only appropriate for static document collections, but capable to adapt to different kinds of dynamic data. The second strategy below is also interesting from a theoretical point of view since it generalizes the update technique of path encodings (see Section 3.4).

⁴Further proximity variants such as *Parent^j* are handled similarly. They are also included in an alternative presentation of BIRD’s decision and reconstruction capabilities, to be introduced in Chapter 8.

⁵We do not consider the *attribute* and *namespace* axes here, which can be treated similarly to the *child* axis, see Section 2.1.

$D \models \text{Child}(v, w)$ child	We check if p has any child, say, p' , using S_b . In the negative case, w is not a child of v . In the positive case let $\omega := \omega_b(p')$. Then $D \models \text{Child}(v, w)$ iff $\beta_b(w)$ is a multiple of ω and $\beta_b(v) < \beta_b(w) < \beta_b(v) + \omega_b(p)$. The weights $\omega_b(p')$ and $\omega_b(p)$ are obtained from S_b .
$D \models \text{Child}^+(v, w)$ descendant	We retrieve $\omega_b(p)$ using S_b . Then $D \models \text{Child}^+(v, w)$ iff $\beta_b(v) < \beta_b(w) < \beta_b(v) + \omega_b(p)$.
$D \models \text{Child}^*(v, w)$ descendant-or-self	The relation holds iff $D \models \text{Child}^+(v, w)$ or $\beta_b(v) = \beta_b(w)$.
$D \models \text{Parent}(v, w)$ parent	We proceed as in Lemma 4.8 on page 48, with $i = 1$, and compare the resulting BIRD label to $\beta_b(w)$.
$D \models \text{Parent}^+(v, w)$ ancestor	We iterate the procedure in Lemma 4.8 for $i = 1$ until reaching either w (positive result) or a node u where $\beta_b(u) < \beta_b(w)$ (negative result).
$D \models \text{Parent}^*(v, w)$ ancestor-or-self	The relation holds iff $D \models \text{Parent}^+(v, w)$ or $\beta_b(v) = \beta_b(w)$.
$D \models \text{NextSib}(v, w)$	We obtain $\omega_b(p)$ and p 's parent p' from S_b and compute the label $\beta_b(u)$ of the parent u of v in D (see Lemma 4.11 on page 49). $D \models \text{NextSib}(v, w)$ holds iff $\beta_b(w) = \beta_b(v) + \omega_b(p)$ and $\beta_b(w) < \beta_b(u) + \omega_b(p')$.
$D \models \text{NextSib}^+(v, w)$ following-sibling	We obtain $\omega_b(p)$, p' and $\beta_b(u)$ as above (see $D \models \text{NextSib}(v, w)$). $D \models \text{NextSib}^+(v, w)$ holds iff $\beta_b(w) - \beta_b(v)$ is positive and a multiple of $\omega_b(p)$ and if $\beta_b(w) < \beta_b(u) + \omega_b(p')$.
$D \models \text{NextSib}^*(v, w)$	The relation holds iff $D \models \text{NextSib}^+(v, w)$ or $\beta_b(v) = \beta_b(w)$.
$D \models \text{PrevSib}(v, w)$	We proceed as in Lemma 4.10 on page 49, with $i = 1$, and compare the resulting BIRD label to $\beta_b(w)$.
$D \models \text{PrevSib}^+(v, w)$ preceding-sibling	We obtain $\omega_b(p)$ and p 's parent p' from S_b and compute the label $\beta_b(u)$ of the parent u of v in D (see Lemma 4.8 on page 48). $D \models \text{PrevSib}^+(v, w)$ holds iff $\beta_b(v) - \beta_b(w)$ is positive and a multiple of $\omega_b(p)$ and if $\beta_b(u) < \beta_b(w)$.
$D \models \text{PrevSib}^*(v, w)$	The relation holds iff $D \models \text{PrevSib}^+(v, w)$ or $\beta_b(v) = \beta_b(w)$.
$D \models \text{Sibling}(v, w)$	The relation holds iff $D \models \text{PrevSib}^+(v, w)$ or $D \models \text{NextSib}^+(v, w)$ or $D \models \text{Self}(v, w)$ (see below).
$D \models \text{Following}(v, w)$ following	The relation holds iff $\beta_b(v) + \omega_b(p) \leq \beta_b(w)$, by Lemma 4.5 on page 48 and Lemma 4.6 on page 48. The weight $\omega_b(p)$ is obtained from S_b .
$D \models \text{Preceding}(v, w)$ preceding	The relation holds iff $\beta_b(w) < \beta_b(v)$ and w is not an ancestor of v . The latter problem is decided as described above (see $D \models \text{Parent}^+(v, w)/\text{ancestor}$).
$D \models \text{Self}(v, w)$ self	The relation holds iff $\beta_b(v) = \beta_b(w)$.

Table 4.2: Relations decidable using any b -balanced BIRD scheme where $b \geq 1$. Given the BIRD labels $\beta_b(v)$ and $\beta_b(w)$ of two document nodes v, w as well as the schema node $p = \pi(v)$ holding the weight corresponding to v , all relations are decidable without access to the document level. For each decision problem $R(v, w)$, the decision procedure is sketched as part of the proof of Lemma 4.14 on the preceding page, and the corresponding XPath axis is given with v as context node. For example, $\text{Child}(v, w)$ means w is a child of v and therefore on the `child` axis.

4.5.1 Sparse BIRD Labelling

It has been mentioned before that as a multiplicative encoding, BIRD labels virtual nodes, causing certain labels to be left unassigned. This amount of unused labels typically grows with the balancing factor b . For instance, reconsider the document tree in Figure 4.2 on page 45 that is labelled using the child-balanced BIRD scheme ($b = 1$). Here 75 labels are reserved for the subtree rooted at the node with the label 150 (the second child of the root, in document order) although the subtree contains only two nodes. This is because the node 150 inherits the weight 75 via child balancing from its left sibling, 75, whose subtree is much larger. When inserting nodes in the subtree below node 150, the odds are that the corresponding labels are still unassigned such that no relabelling is necessary. In other words, the sparse encoding makes BIRD inherently robust against a certain amount of node insertions in specific positions. The same phenomenon is exploited by Li and Moon [2001] for their Extended Preorder labelling, and also applies to other approaches such as region encodings (see Section 3.3).

Of course, inserting a node in a subtree whose label space is exhausted causes an overflow. As a result, the weights not only of the overflowing node, but also of its siblings in the schema tree change (again due to child balancing). This update may propagate up through the schema tree and thus spoil the weights of all other document nodes in the worst case. Because overflows cause a periodical relabelling of the entire document collection, the update strategy just described is applicable only when the data is known to remain reasonably homogeneous over time, with only little difference in the size of subtrees below the same tag path. To reduce the overflow risk further, one may also deliberately leave some extra labels unassigned, as suggested by Li and Moon, at the expense of an increased overall label size (see below).

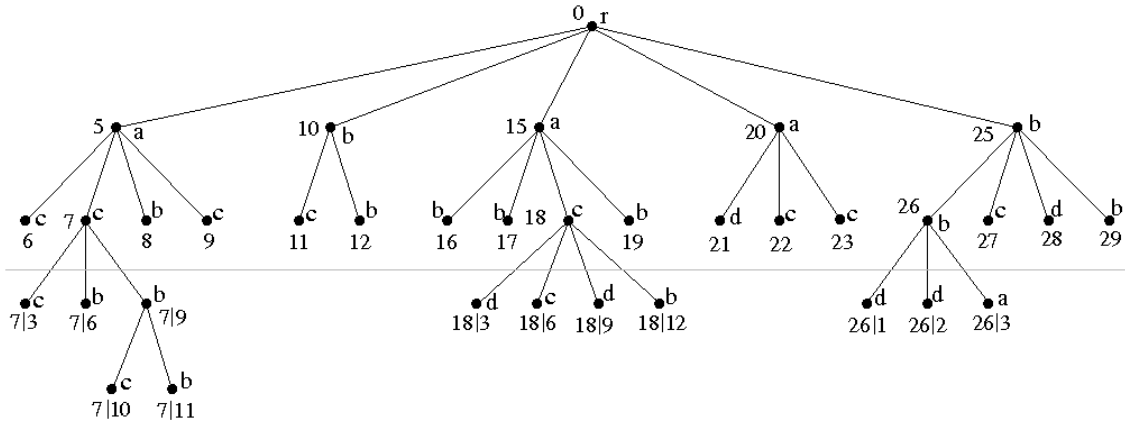
In many applications node insertions do not occur at arbitrary positions in the document tree, but only at the end of the collection (i.e., after the last node in document order). This further reduces the overflow risk. As a special case, consider collections of bibliographic data like *DBLP* [DBLP] or the large *Internet Movie Database (IMDb)* [IMDb] (see also Section 13.2 in the appendix), where the bulk of insertions happen when adding entire documents (e.g., in the case of *IMDb*, new files describing movies, actors, directors, or producers). This does not alter the nodes in existing documents (unless the new document changes the weights of one or more tag paths due to balancing, in which case the labels of at least all nodes with that path throughout the database are affected). Hence for such collections of more or less homogeneous documents with updates at the document level only, incremental updates are not mandatory. Section 4.6 below provides experimental evidence for this.

4.5.2 Layered BIRD Labelling

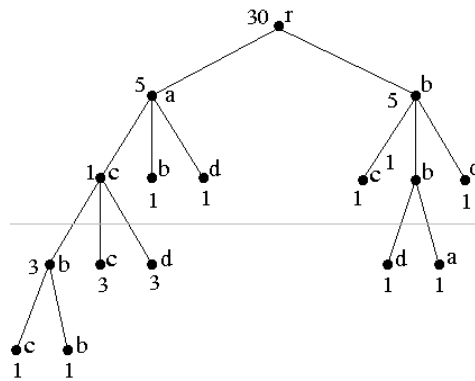
We now sketch a second strategy for decoupling existing labels from labels assigned to newly added nodes. This strategy is henceforth referred to as *layering*.⁶ The idea is to partition the document tree into a hierarchy of horizontal regions, or *layers*, which are then labelled independently. The complete (*layered*) label of a given document node v on layer ℓ_i ($i \geq 0$) is then composed of v 's label on layer ℓ_i as well as the labels of selected ancestors of v on higher layers in the hierarchy. As an extreme case, consider again the path encodings presented in Section 3.4 that label any document node v with a sequence of sibling codes each representing the position of an ancestor of v on a specific document level. Path encodings such as Dewey can be regarded as special layered schemes where each layer corresponds to one level in the document tree, so that each component in the layered label of v is just the sibling code of the ancestor of v on the corresponding document level. It has been observed before that when using path encoding, the label of v stays the same no matter how many nodes are inserted into the subtree rooted in v . This is precisely because all layers (i.e., document levels in this case) are labelled separately. The same idea can be generalized so that multiple levels of the document tree are subsumed by the same layer and therefore represented by the same component of a layered node label, as follows.

Figure 4.6a. on the next page depicts the same document tree D as Figure 4.2a. on page 45, but with two layers ℓ_0 and ℓ_1 that cover the five levels of D . Consequently, the layered BIRD labels consist of two components. The upper layer, ℓ_0 , covers the three topmost levels in D . Nodes on these levels

⁶A formal unified definition of what we call *layering* is given in the aforementioned survey of labelling schemes [Weigel and Schulz 2007]. There we also show that many variations of the same technique have been proposed independently before, mostly for reducing the maximal node label size.



a. layered document tree D



b. layered schema tree S_1

Figure 4.6: Child-balanced Layered BIRD labelling with two layers, ℓ_0 (top) and ℓ_1 (bottom). a. A sample document tree D . For any document node v on the upper layer ℓ_0 , the layered label is simply the child-balanced BIRD label $\beta_1(v)$ of v that results from labelling only the upper part of D . Nodes on the lower layer ℓ_1 inherit the label of their lowest ancestor on layer ℓ_0 (first label component, before “|”). In addition, subtrees on the lower layer ℓ_1 are labelled independently, again with the child-balanced BIRD scheme (second label component, after “|”). b. The 1-balanced schema tree S_1 for D in a. For any schema node p in S_1 that represents document nodes on layer ℓ_i ($i \in \{0, 1\}$), the child-balanced BIRD weight $\omega_1(p)$ of p is shown that results from labelling only subtrees of D on layer ℓ_i . Note that schema nodes representing leaves on either layer in D have the minimal weight, 1 (according to Definition 4.3 on page 44), even if they are not leaves in S_1 .

have as first label component ordinary child-balanced BIRD labels and as an implicit second component 0 (omitted in the figure). By contrast, document nodes on the lower layer, ℓ_1 , inherit the first label component from their lowest ancestor on the upper level ℓ_1 , while the second component results from independently labelling their respective subtree on ℓ_1 , again with the child-balanced BIRD scheme. For instance, consider the node 7 on the upper layer in Figure 4.6a. (left-hand side). All descendants of node 7 reside on the lower layer, and therefore have 7 as their first label component. The second component of their labels is independent of the upper-layer component, which allows to handle node insertions gracefully. For instance, any number of children may be added below the node 7 (with layered labels “7|12”, “7|15”, . . . , according to the child-balanced BIRD scheme on ℓ_1) without affecting the labels of any nodes on ℓ_0 , or any of their descendants on the lower layer. In fact, overflows may only occur inside a document subtree on a given layer (e.g., if a right sibling of node “7|11” has to be added). But since any number of subtrees is allowed on any layer, the Layered BIRD scheme still supports arbitrary many insertions (though not at all positions in the document tree).

The BIRD weights on each layer are easy to determine using the bottom-up procedure described in Section 4.2.1. A new layer is introduced in the scheme as soon as a suitable position for future node insertions is reached (e.g., right above the `movie` level in the *IMDb* collection). Theoretically any number of layers may be created, up to the extreme case where each document level is on a different layer, and the Layered BIRD scheme coincides with Dewey. Layering also helps to prevent individual weights from growing too large: when the desired upper bound is reached, the current layer is closed, and weighting restarts with a leaf value of 1. In fact, any layer may even span only part of a level in the document tree, and different tag paths may cross a different number of layers. Thus the labels of two document nodes v and w on the same level in D need not even consist of the same number of components: e.g., v may be part of a much richer subtree requiring more layers than w . The exact number and position of the layer boundaries in the schema tree determines both the size of the resulting Layered BIRD labels and the positions in D where unlimited insertions are supported. Like path encodings, the Layered BIRD scheme likely benefits from a suitable binary encoding of the node labels (see Section 3.4.1) for storing the variable-sized layered labels in a compact form.

Finally, all decision and reconstruction operations on ordinary BIRD labels are easily adapted to the layered variant. As a matter of fact, in each such operation only one component of a Layered BIRD label needs to be manipulated as in the unlayered case, whereas all other components are either removed from the label or simply ignored. For instance, in order to reconstruct the i -th ancestor $u := \text{parent}^i(v)$ of a document node v in D , one first goes up i levels in the schema tree, starting from $\pi(v)$, to determine the weight of u . Whenever a layer boundary is crossed during the bottom-up traversal, the corresponding component in the layered label of v is removed. The label of u on the target layer is computed from the corresponding component in v 's label as usual, for the remaining number $j \leq i$ of levels covered by that layer. All higher-layer label components remain unchanged.

Assume, e.g., that v is the node “7|10” in Figure 4.6a. on the previous page. If $i = 1$, then no layer boundary is traversed, and the label of the parent u of v is “7|9”, because $(10 - (10 \bmod 3)) = 9$. For $i = 2$, the boundary from ℓ_1 to ℓ_0 is crossed. Hence the second component in the layered label “7|10” of v is removed. The label of node u on layer ℓ_0 is “7” since $7 - (7 \bmod 1) = 7$, according to the ordinary BIRD reconstruction on ℓ_0 . (Note that here u is the lowest ancestor of v on the upper layer, whose label is inherited.) Similarly, all higher ancestors of v are reconstructed: $\text{parent}^3(\text{“7|10”}) = \text{“5”}$ because $7 - (7 \bmod 5) = 5$, and $\text{parent}^4(\text{“7|10”}) = \text{“0”}$ because $7 - (7 \bmod 30) = 0$. Verify in Figure 4.6a. that the nodes on the root path of $v = \text{“7|10”}$ in D are indeed “7|9”, “7”, “5” and “0”.

For deciding $\text{Child}^+(u, v)$, we check whether the relation holds for the label components of u and v on u 's layer and whether all preceding components are equal in both layered labels. Comparing nodes according to the document order is done component-wise in top-down direction, as with path encodings.

4.6 Experimental Evaluation

This section reports on our experimental evaluation and comparison of the following four labelling schemes: BIRD (non-layered child-balanced, i.e., $b = 1$), ORDPATH by O’Neil et al. [2004] (see Section 3.4.1), μ PID by Bremer and Gertz [2006] (see Section 3.4.2) and Virtual Nodes by Lee et al. [1996] (see Sec-

tion 3.5). We applied each scheme to the three document collections *Cities*, *DBLP* and *XMark 1100* (see Section 13.2 in the appendix), which differ considerably in size and structural complexity (in terms of the number and length of the tag paths occurring in the documents). We implemented the four schemes to be compared as described in the original literature. In line with the analysis of labelling schemes in the previous chapter, the following optimization goals are examined: storage consumption (see Section 4.6.1); runtime performance, both for individual reconstruction and decision operations (see Section 4.6.2) and for entire queries (see Section 4.6.3), and updatability (see Section 4.6.4). Differences in the expressivity of all schemes are discussed in Section 4.7.

As testbed we use the native XML database X^2 [Meuss et al. 2005; Meuss et al. 2003; Meuss 2000]. X^2 is implemented in Java and uses a RDBS back-end (*PostgreSQL*) where the XML documents are stored in relational form (details are explained in Chapter 6). During the query evaluation, X^2 manipulates trees in main memory, which are restored from sets of document nodes fetched from the RDBS. All tests are carried out sequentially on the same machine, whose performance characteristics are listed in Section 13.1 of the appendix (Test Environment A). The database cache of the RDBS is disabled. Apart from the processes for X^2 and the RDBS, the test computer is idle during the experiments.

4.6.1 Storage Consumption

The storage consumption of the four labelling schemes on all test document collections are given in Tables 4.3 *a–c*. on page 58. The first three columns after the scheme name contain the minimum, maximum, and average number of bits used for a single node label, respectively. The remaining columns list the storage needed for all labels together, both as an absolute value in MB (kB for *Cities*) in columns five and seven, and relative to the corresponding result obtained for preorder labelling (columns six and eight), which is the baseline in our experiments. The relative values are computed on bit counts, whereas the absolute values are rounded to the nearest MB (kB for *Cities*).

We apply two different methods to compute the total storage consumed by a given labelling scheme. On the one hand, we sum up the exact bit counts needed for the labels, assuming that labels can be stored with variable size. This produces the absolute (relative) values in the fifth (sixth) column, which follow the average label sizes in column four. On the other hand, it is perhaps more realistic to assume that when stored in the database, all labels assigned to nodes in the same document collection take up the same space. The total storage taken up by such fixed-size labels is the product of the maximum label size, as given in column three, and the total number of nodes in the collection (see Section 13.2). The resulting values appear in columns seven (absolute) and eight (again relative to the values obtained for preorder labelling).

We found that 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 Table 3.2 on page 39). When assigning fixed-size labels BIRD reduces the space consumption by nearly a factor 2 for ORDPATH and between 2.2 and 4.5 for Virtual Nodes. The reason is that for BIRD the maximum label size is much closer to the average size than for ORDPATH and Virtual Nodes, which therefore incur a significant storage overhead for fixed-size labels. For variable-size labels this factor decreases, but BIRD labels still are clearly smaller than those of other schemes.

As the only approach (except preorder) with smaller labels than BIRD, the μ PID scheme optimizes storage at the expense of expressivity, as shown in Table 3.2 on page 39. Remarkably, μ PID occupies less space than the preorder scheme in our experiments, at least when assuming variable-size labels. In the underlying trade-off between expressivity and space consumption, the μ PID scheme chooses an intermediate position between schemes with high expressivity and storage consumption, such as Virtual Nodes, on the one hand and schemes with low expressivity and storage consumption, such as the subtree encodings in Section 3.3, on the other hand.

In further experiments with more deeply nested, text-oriented document collections, such as the *INEX* benchmark corpus [INEX] that consists of extremely heterogeneous and layout-polluted research papers, we observed that on average BIRD labels grow larger than ORDPATH labels (97 versus 60 bits; Virtual Nodes 78 bits), whereas their maximum size is still smaller than that of ORDPATH (98 versus 135 bits; Virtual Nodes 217 bits). One reason is that with a maximum path length of 17, the multiplicative effect on the label size (see Lemma 4.7 on page 48) becomes more dominant. What is worse, in heterogeneous collections such as *INEX* the child-balancing blows up the weights of tag paths that lead to subtrees which

scheme	ID size (bits)			total storage (kB)			
	min.	max.	avg.	variable ID size		fixed ID size	
				absolute	% pre	absolute	% pre
BIRD	1	24	22	104	161	113	150
ORDPATH	2	49	33	151	232	223	305
—w/o caretting-in	2	41	27	123	189	186	255
Virtual Nodes	1	58	37	168	261	264	363
μ PID	1	14	11	50	78	64	88
preorder	1	16	14	65	100	73	100

a. *Cities*

scheme	ID size (bits)			total storage (MB)			
	min.	max.	avg.	variable ID size		fixed ID size	
				absolute	% pre	absolute	% pre
BIRD	1	37	36	25	170	25	161
ORDPATH	2	53	37	26	186	36	240
—w/o caretting-in	2	52	36	25	179	35	233
Virtual Nodes	1	95	37	25	174	64	413
μ PID	1	28	21	14	99	19	122
preorder	1	23	21	14	100	15	100

b. *DBLP*

scheme	ID size (bits)			total storage (MB)			
	min.	max.	avg.	variable ID size		fixed ID size	
				absolute	% pre	absolute	% pre
BIRD	1	44	43	113	188	113	177
ORDPATH	2	86	48	124	207	221	345
—w/o caretting-in	2	77	43	111	185	198	309
Virtual Nodes	1	198	81	210	350	508	794
μ PID	1	29	20	54	90	74	116
preorder	1	25	23	60	100	64	100

c. *XMark 1100*Table 4.3: Storage consumption of different labelling schemes on the three document collections *Cities*, *DBLP* and *XMark 1100* (see Section 13.2 in the appendix for details).

greatly vary in size, thus causing many labels to be reserved for virtual nodes. Obviously, this could be avoided if equal weights were assigned to nodes with a similar number of descendants, rather than with equal tag paths. The corresponding structural summary holding the weights clearly would differ significantly from the schema tree. But as mentioned in Section 4.1, BIRD can be combined with other index structures providing efficient access to the weights, as long as specific requirements are met (see Section 4.8 below).

Preliminary experiments show that for the *INEX* collection, the maximum label size may thus be reduced to 64 bits, i.e., below the performance-critical boundary discussed in the next section, although the resulting weight index is huge. The exact size of the labels as well as of the structural summary serving as weight index depends on which document nodes share the same weight, i.e., are regarded as equivalent in terms of their subtree sizes. The finer the underlying equivalence relation, the better the weights reflect the actual subtree sizes, but the more nodes are needed in the structural summary to represent those weights. Methods to optimize this trade-off between label size and weight index size remain to be developed.

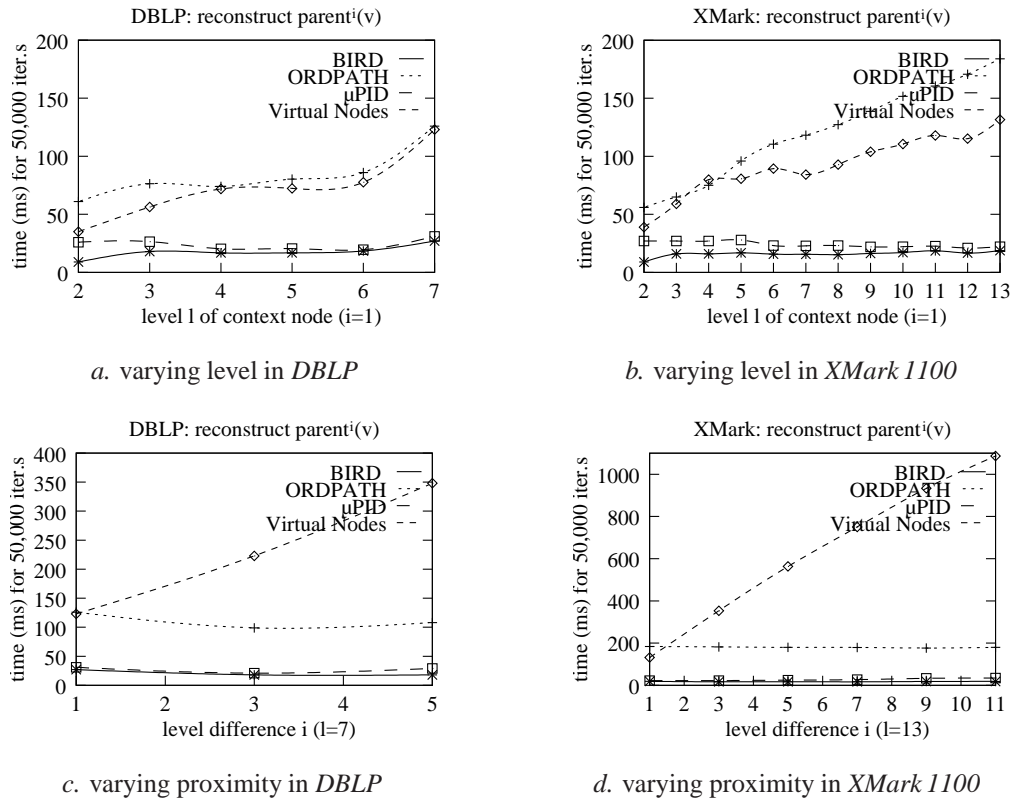


Figure 4.7: Efficiency of ancestor reconstruction. *a.*, *b.* Reconstruction of $parent^i$ from different levels l , for fixed proximity i . *c.*, *d.* Reconstruction of $parent^i$ from a fixed level l , for different proximities i .

4.6.2 Efficiency of Decision and Reconstruction

The first set of runtime experiments measure the efficiency of decision and reconstruction with different labelling schemes. Figures 4.7 and 4.8 plot the computation time needed for various reconstruction and decision problems on the *DBLP* and the *XMark 1100* collection. Results for *Cities* are not shown, but reveal similar tendencies. All four schemes (excluding preorder, for obvious reasons) 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. Note that this subsumes all necessary operations including, e.g., access to the schema tree for BIRD or μ PID and label comparison during decision.

Reconstruction. Figures 4.7 *a.*, *b.* show the time needed to reconstruct the parents of nodes at different levels (abscissa). For *DBLP* (*a.*) and *XMark 1100* (*b.*), μ PID is almost as fast as BIRD, whereas ORDPATH and Virtual Nodes are slower by at least a factor 4. On *XMark 1100*, the difference between BIRD and ORDPATH is up to one order of magnitude. Clearly the performance of both BIRD and μ PID is independent of the level of the source node. For ORDPATH, the computation time grows with the depth of the source node. The reason is that ORDPATH bit strings must be parsed top-down (i.e., from left to right) down to the level of the source node. The deeper the source node is located in the document tree, the longer the parsing takes. We observe the same effect for Virtual Nodes on *DBLP* and *XMark 1100* although in theory its ancestor reconstruction works in constant time (see below). Presumably the representation of numbers of arbitrary size, needed for Virtual Nodes here because of the sheer length of the node labels, creates an overhead for arithmetic operations on label values. Since breadth-first labels grow larger on deeper levels, this explains why the performance of Virtual Nodes degrades in Figures 4.7 *a.* and *b.* The effect is not observed for the *Cities* collection where the Virtual Nodes labels take up at most 64 bits (not

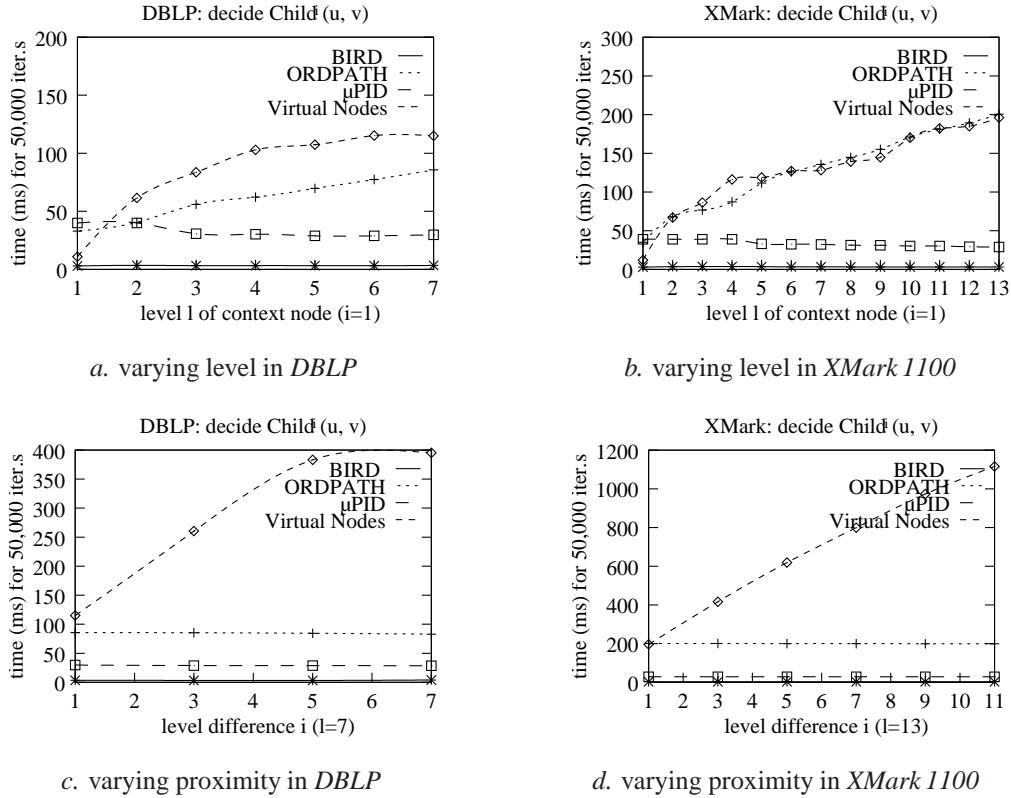


Figure 4.8: Efficiency of ancestor decision. *a.*, *b.* Decision of $Parent^i$ from different levels l , for fixed proximity i . *c.*, *d.* Decision of $Parent^i$ from a fixed level l , for different proximities i .

shown in the figure).

Figures 4.7 *c.* and *d.* on page 59 illustrate the orthogonal situation: here $parent^i$ is reconstructed from source nodes at a fixed level in the tree (level 7 for *DBLP*, level 13 for *XMark 1100*), with varying distance i (abscissa). As in Figures 4.7 *a.* and *b.*, BIRD and μ PID are significantly faster than ORDPATH and Virtual Nodes (nearly one order of magnitude; mind the different scales in *c.* and *d.*) and reveal no dependency on the number of levels to be traversed. Both schemes climb up a path in the schema tree and then directly reconstruct the desired node label, which takes practically constant time. By contrast, the Virtual Nodes scheme reconstructs all ancestors iteratively and therefore suffers from a linear degradation for bigger distances i . ORDPATH's bit shift operations are indifferent to proximity.

Decision. The plots in Figure 4.8 are based on a similar setting as those in Figure 4.7 on the previous page, but this time for the decision of the $Child^i$ relation. We observe the same dependencies on the level of the source node and the distance to the target node as in the reconstruction tests. BIRD is as fast as for reconstruction (3 ms for 50,000 iterations), whereas μ PID is one order of magnitude slower. On *DBLP*, BIRD outperforms ORDPATH and Virtual Nodes by a factor 30 and 40, respectively (up to 100 for Virtual Nodes with a level difference of 7). On *XMark 1100*, the difference is nearly two orders of magnitude (up to 400 for Virtual Nodes with a level difference of 13).

The complete report [Weigel et al. 2005c] on our experiments covers a more detailed analysis of the asymptotic behaviour of different labelling schemes, including the ones evaluated here, when faced with specific reconstruction and decision problems.

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

a. DBLP

QID	SCHEME	PATH JOIN STRATEGY		
		ALWAYS	FIRST	NEVER
Q0	BIRD	617	597	4817
	ORDPATH	1534	1535	12343
	μ PID	662	577	5320
	Virtual Nodes	1723	5760	295084
	Preorder	23925	7569	20613
Q1	BIRD	2634	2591	6745
	ORDPATH	6248	6293	20068
	μ PID	2908	2855	8231
	Virtual Nodes	6913	636649	4749424
	Preorder	92430	97455	188456
Q2	BIRD	14385	14072	19529
	ORDPATH	37149	36355	49827
	μ PID	14919	14978	20668
	Virtual Nodes	36854	65589	82331
	Preorder	567524	13799	18282
Q3	BIRD	30	37	9957
	ORDPATH	98	86	25733
	μ PID	36	42	11102
	Virtual Nodes	86	89	228493
	Preorder	1047	1057	14521

b. XMark 1100

Table 4.4: Efficiency of query evaluation with different labelling schemes on the two document collections *DBLP* and *XMark 1100* (see Section 13.2 in the appendix).

4.6.3 Efficiency of Query Evaluation

Experimental set-up. To quantify to what extent the differences in decision and reconstruction speed observed in Section 4.6.2 affect the overall performance for entire queries, we evaluated a couple of sample tree queries using the same labelling schemes as in the previous section, both against the *DBLP* and the *XMark 1100* collection. The four test queries run against each collection are shown in Tables 14.1 *a.* and 14.1 *b.* on page 184 in the appendix, respectively. 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 (occasionally fewer for some long-running queries) were then averaged. Tables 4.4 *a.* and *b.* list the total evaluation times (without profiling). A second set of runs of the same queries was carried out to measure the contribution of individual query stages. Chapter 14 in the appendix contains a detailed analysis of this additional experiment, including the complete profiling results (see Tables 14.2 *a.* and 14.2 *b.* on page 185).

Due to the restricted tree query language supported by the retrieval system X^2 , the test queries only involve the decision of $Child^i$ and the reconstruction of $parent^i$. Note that all query nodes are result nodes, i.e., an answer to a query comprises the matches to all nodes in the query tree, not just one focussed node as in XPath. The same evaluation algorithm is used for all labelling schemes; just the reconstruction, decision, and comparison operations vary. The only exception is that schemes which do not preserve preorder (i.e., μ PID and Virtual Nodes) cannot benefit from certain optimizations (see below). As a baseline, we use preorder labels with brute-force reconstruction and decision: reconstructing the i -th ancestor of a node requires i look-ups in a parent/child table in the RDBS that maps the preorder label of any node to the preorder label of its parent node.

In order to estimate the benefits of reconstruction operations (which are not supported by all labelling schemes, see above), we implemented and tested the three *path join strategies* *ALWAYS*, *FIRST*, and *NEVER* which differ in their use of reconstruction of $parent^i$. Details about the strategies are given elsewhere [Weigel et al. 2005c]. In short, *ALWAYS* means that the matches of any branching node in the query tree are joined with those of its child nodes by reconstructing the ancestors of the child matches and testing whether they are contained in the branching node's set of matches. Since our retrieval engine X^2 evaluates queries bottom-up, the first child of any branching query node does not undergo the path join (which would fail for the empty set of parent matches), but simply propagates its matches up to the parent node

by reconstruction. The same is true for the second strategy, *FIRST*, which treats only subsequent children differently. Here the path join decides for each pair of matches to the branching node and its child node whether the $Child^+$ relation holds. No test for set containment is needed, and schemes respecting document order may benefit from optimizations saving the decision for some pairs of nodes, using common structural join algorithms. The third strategy, *NEVER*, does not take advantage of reconstruction at all, not even for the first child of a given branching node. Instead of propagating matches upward in the query tree, all nodes in the documents with a path matching the path of the branching node are retrieved and then joined with the matches of its first child query node by deciding $Child^+$. Subsequent children are handled as described for the *FIRST* strategy.

Summary. The following key results sum up the outcome of our experiments (again, see Chapter 14 in the appendix for additional details of the analysis):

Result 1 *The BIRD labelling scheme performs best for virtually all queries and path join strategies, both on the DBLP and the XMark 1100 collection.* □

The overall performance in all tests against the *DBLP* and *XMark 1100* collections is given in Tables 4.4 *a.* and *b.* on page 61. Each of the three rightmost columns corresponds to one of the three path join strategies explained above. BIRD almost always outperforms the other schemes, beaten only once by μ PID (*DBLP*: Q3 *FIRST*; *XMark 1100*: Q0 *FIRST*) and twice by preorder (*DBLP*: Q0 *FIRST* and *NEVER*; *XMark 1100*: Q2 *FIRST* and *NEVER*). The most efficient schemes compared to BIRD are μ PID (*DBLP*: factor ≤ 1.6 ; *XMark 1100*: factor ≤ 1.2) and ORDPATH (*DBLP*: factor ≤ 1.6 ; *XMark 1100*: factor ≤ 3.3). In terms of absolute numbers, the greatest difference between BIRD and μ PID is 1.2 seconds on *DBLP* and 1.5 seconds on *XMark 1100*. ORDPATH is on *DBLP* up to 0.6 seconds slower and on *XMark 1100* up to 30 seconds. The distance to Virtual Nodes is considerable (*DBLP*: factor ≤ 58 ; *XMark 1100*: factor ≤ 704 compared to BIRD). In extreme cases, Virtual Nodes is one order of magnitude slower than the baseline, preorder, and even more compared to the other schemes, especially when reconstruction is disabled (e.g., Q1 *NEVER* in Table 4.4 *b.*). The exact performance differences vary dramatically with the time spent on label comparisons (see also the following results). In terms of absolute numbers, the greatest difference between BIRD and Virtual Nodes is more than one hour. As could be expected, brute-force reconstruction and decision with preorder labels is usually very slow, especially when other schemes benefit from extensive use of in-memory reconstruction. Evaluation with preorder labels takes up to 40 times or 10 minutes longer than with BIRD labels.

Result 2 *The efficiency of label comparisons has a greater impact on the overall performance than reconstruction and decision, and can be affected by the label size.* □

A detailed profiling of different evaluation ingredients (see Chapter 14) proves that most of the query evaluation time is spent on comparing node labels, both during decision and, most prominently, when manipulating the sets of potential matches fetched or reconstructed before. While decision and reconstruction contribute up to one second to the total evaluation time, label comparison easily takes two orders of magnitude longer. Accordingly, the time spent on reconstruction and decision differs by one second or less among the schemes (ignoring cases where Virtual Nodes must perform far more decision operations than the other schemes, see Result 4), whereas the efficiency of label comparison can make a difference of 20 seconds and more. As the difference between Virtual Nodes and the other schemes on *DBLP* shows, the size of the labels can have a huge impact on the performance of all label operations (most notably, the frequent comparisons): as the only scheme whose labels do not fit the native 64-bit data types provided by most high-level programming languages, Virtual Nodes suffers from a considerable overhead even for the strategy *ALWAYS* (a second handicap of Virtual Nodes for the other two strategies is subsumed under Result 4). ORDPATH is subject to the same effect on *XMark 1100* where its labels grow larger than 64 bits, too. While the impact of the label size depends on the underlying computer architecture as well as the data structures used, schemes exceeding a certain label size will always incur some runtime overhead, not to speak of the disk space they occupy.

Result 3 *Reconstruction is of paramount importance to efficient query evaluation because it saves label fetching and comparison.* □

The comparison of the three path join strategies *ALWAYS*, *FIRST* and *NEVER* also clearly shows that reconstruction is key to efficient query evaluation. Performance decreases dramatically for all schemes and almost all queries when reconstruction is disabled (strategy *NEVER*, as opposed to *FIRST* and *ALWAYS*). The fact that the huge overhead incurred by *NEVER* is mainly due to label comparisons rather than node fetching illustrates that our results do not only apply to native retrieval systems like X^2 but also, perhaps to a lesser extent, to other engines where fetching is cheaper (such as purely relational systems). BIRD, ORDPATH and μ PID prefer *FIRST* with its mixture of reconstruction and decision, owing to their efficient decision techniques. Virtual Nodes, by contrast, suffers from a massive join overhead for this strategy, caused by the breadth-first order of its labels (see Result 4). With its different join algorithm, *ALWAYS* brings Virtual Nodes a little closer to the other three schemes.

Result 4 *Labelling schemes preserving document order benefit greatly from path join optimizations.* □

The path join strategies involving decision, i.e., *FIRST* and *NEVER*, locate ancestor/descendant pairs in sets of matches to two given query nodes. Processing these label sets in document order has the advantage that not all possible label pairs (i.e., the full Cartesian product) need to be checked, which may save many decision (and, consequently, comparison) operations, as explained in the complete report on the experiments [Weigel et al. 2005c]. Obviously schemes like BIRD, ORDPATH and preorder benefit from this optimization whereas Virtual Nodes, whose labels are assigned in a breadth-first traversal of the document tree, typically must decide ancestorship for many more label pairs. The resulting overhead explains why for *FIRST* and *NEVER*, Virtual Nodes is far less competitive than for *ALWAYS*. The μ PID scheme, although violating the document order between arbitrary nodes, is also amenable to the optimization provided that only sets of nodes with the same tag path are joined (because among these nodes, the document order is preserved). Since our test system X^2 always retrieves and joins nodes belonging to the same schema node, this condition is satisfied and μ PID can be handled as if it were fully compatible with document order.

4.6.4 Updatability

In Section 4.5.1 an update scenario is outlined where more and more documents are successively added to a collection that was originally labelled with a deliberately sparse BIRD scheme. The question is how often an overflow of some of the weights established during the last indexing occurs, which triggers a relabelling of the entire collection (recall that no layering is available in this setting). To answer this question empirically for a large collection of real-world data, we carry out the following experiment.

An XML version of the *Internet Movie Database (IMDb)* (8.4 GB on disk, see Section 13.2 for details) is labelled with two variants of the child-balanced BIRD scheme that differ in their degree of sparseness. Nearly 2,000,000 documents are indexed consecutively in chunks of 1,000 documents (about 4-6 MB per chunk). Figure 4.9 on the following page shows BIRD's overflow behaviour and space consumption as more and more documents are added. In a first experiment, no future insertions are anticipated, i.e., the weight of a given tag path is always just as large as it must be to accommodate the greatest known subtree below that path ("BIRD" in Figure 4.9 a. and 4.9 b.). We then label the collection once again, this time reserving extra labels for 100 potential child node insertions below any overflowing node during the weight computation ("BIRD + 100" in Figure 4.9 a. and 4.9 b.).

Each point in the plot in Figure 4.9 a. (left-hand side) illustrates how many times at least one weight in the schema tree must be changed while adding another 100,000 documents, thus causing a relabelling of the collection. The two large peaks at the beginning indicate that the ordinary BIRD weights become reasonably stable only after indexing the first 400,000 documents, or 20% of the data. Up to that point, a large number of overflows occur in the first experiment (dashed line). However, this improves significantly when applying the extra-sparse encoding (solid line). Note that in these early stages of the evolution of the collection, relabelling is much cheaper than later on, after many documents have been added. In the sequel, the need for relabelling dwindles rather quickly, especially for BIRD + 100 which triggers only one more weight update before adding 1,300,000 documents without any overflow.

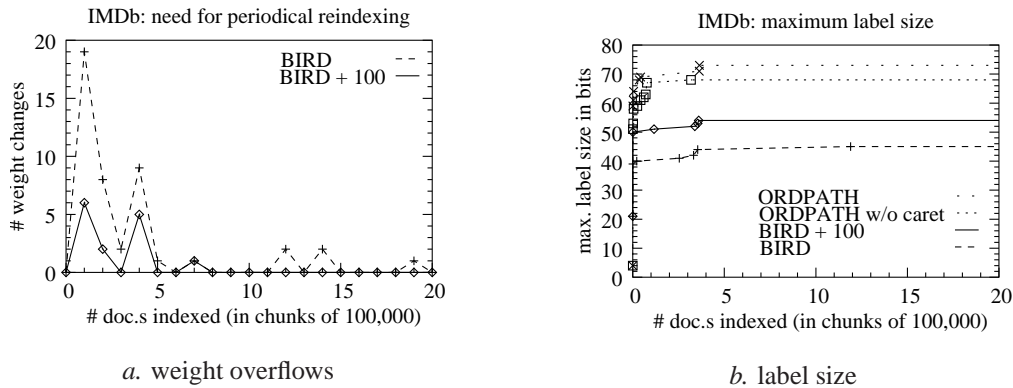


Figure 4.9: Effect of updates to the *IMDb* collection with different labelling schemes. *a.* Number of weight overflows that occur while labelling a chunk of 100,000 documents with child-balanced BIRD schemes of different sparseness. *b.* Growth of the maximum label size for different labelling schemes while adding documents of the collection.

In Figure 4.9*b.* we observe an early saturation of the label sizes (the maximum was mostly reached after indexing less than 20% of the documents) and a very low overall space consumption for BIRD (at most 45 bits per label, for a collection of more than 83,000,000 nodes). Obviously reserving extra labels to increase the robustness of the scheme is not expensive in terms of storage: the greatest BIRD label in the extra-sparse encoding (“BIRD + 100”, at most 54 bits per label) still occupies far less than 64 bits, a critical boundary in our runtime experiments (see Section 4.6.3 above). Although with a height of five the document tree for the *IMDb* collection is fairly shallow, ORDPATH labels grow rapidly beyond the 64-bit line (maximum label size 73 bit). This is true even for a variant of ORDPATH with smaller labels (“ORDPATH w/o caret” in Figure 4.9*b.*; maximum label size 68 bit). Here the sparse encoding (*caretting-in*) for future updates is disabled, at the expense of limited updatability. The resulting ORDPATH variant is similar to Dewey, but enjoys binary ORDPATH encoding. However, the labels are still considerably larger than with either variant of the BIRD scheme.

4.7 Summary and Discussion

This chapter has introduced the *Balanced Index-based numbering scheme for Reconstruction and Decision (BIRD)*. BIRD is a multiplicative labelling scheme optimized towards fast query evaluation through efficient reconstruction and decision of query constraints. Experiments show that BIRD scales up well to large collections containing gigabytes of XML documents, in terms of both the runtime performance and the space occupied by the node labels. We have also sketched several variants of BIRD labelling that target distinct optimization goals. Thus *b-balanced BIRD* with $b > 1$ extends the reconstruction capability beyond *i-th-child*, to descendants on deeper levels. As an extreme case, the *totally balanced* labelling reconstructs large parts of the document tree while minimizing the number of distinct weights to be stored in a structural summary. On the other hand, increasing the balancing parameter b causes the node labels and weights to grow larger unless the structure of the documents is extremely homogeneous.

A decrease in space efficiency is also the price to pay for greater updatability. We have sketched two ways to make BIRD more robust against node insertions: a sparse variant reminiscent of Extended Preorder [Li and Moon 2001] that deliberately reserves labels for future nodes to be added; and the *Layered BIRD* labelling which replaces singleton labels with top-down label sequences, similar to path encoding. The sparse BIRD scheme performs quite well on the large *IMDb* collection, preventing many weight overflows with only a very modest storage overhead. However, deeply nested and heterogeneous collections such as *INEX* are still much of a challenge to the scalability of BIRD.

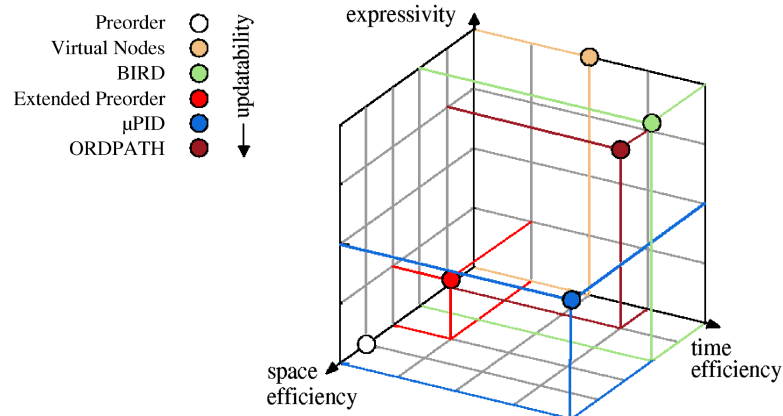


Figure 4.10: Visualization of the positions that different labelling schemes occupy in the trade-off between expressivity, runtime efficiency, storage consumption, and updatability. Each of the first three criteria is represented by a distinct dimension in the three-dimensional trade-off space shown. Updatability is symbolized by the colours of the points in the plot: darker colours indicate a more robust labelling scheme. Time and space efficiency reflect our experimental results, not the theoretical worst-case complexity.

Comparison of labelling schemes. Figure 4.10 presents a tentative visualization of the trade-off between the different optimization goals mentioned before. The idea is to position BIRD and its competitors in the form of a ranking along distinct axes that respectively represent expressivity (vertical), space efficiency (left) and runtime performance (right). Intuitively spoken, one can see that the various approaches head in different directions to solve the problem of “good” XML labelling. Let us briefly highlight the characteristics of each labelling scheme, symbolized by its position in the trade-off space. Preorder labelling (white) is at the lower end of the expressivity and performance dimensions, but of course very space-efficient. Extended Preorder (red, see Section 3.3.1) gains a little expressivity (and hence, runtime efficiency) through the use of a second label component, which doubles the label size. Compared to preorder and Extended Preorder, μ PID (blue, see Section 3.4.2) adds important reconstruction capabilities but lacks support for deciding document order; still we assume that the benefit of the former outweighs the downside of the latter (see below for a short discussion on how to rank the different criteria). In terms of time and space efficiency, the performance of μ PID is unsurpassed in our experiments. ORDPATH (brown, see Section 3.4.1) is more expressive than μ PID (most notably because it respects document order), but less time- and space-efficient. Finally, Virtual Nodes (yellow, see Section 3.5) and BIRD (green) are the most expressive schemes tested. While BIRD is as fast as μ PID and has smaller labels than ORDPATH, Virtual Nodes is fairly inefficient in both respects.

However, notice that the above representation of trade-offs has the following limitations. First, the important criterion of updatability is not represented in a geometric fashion, unlike the other three optimization goals just mentioned. Instead, darker colours in Figure 4.10 indicate a more robust approach. Obviously, ORDPATH is most advanced in terms of updatability. Second, the three-dimensional trade-off space shown in Figure 4.10 is topological, but not metric. In other words, the relative position of two approaches to each other indicates which one is better in terms of a specific criterion, but it does not indicate how much. Finally, the topology in the two horizontal dimensions (i.e., time and space efficiency) is based on our experimental results (see Sections 4.6.1, 4.6.2 and 4.6.3 above), not on the theoretical complexity of the underlying problems. Otherwise, BIRD would be close to Virtual Nodes in the storage dimension because the labels of both schemes grow exponentially in the height of the document tree D (the base being the maximal fan-out in D , see Lemma 4.7 on page 48). Similarly, μ PID would be close to ORDPATH in the storage dimension because despite the good compression rate achieved by μ PID, its worst-case label size is still linear in the number of nodes in D .

Weighting comparison criteria. As the discussion above illustrates, the decision which labelling scheme to use for a particular application depends on a number of different criteria and factors to be weighted against each other. Most prominently, the importance of robustness depends on whether the document collection to be labelled is frequently updated and if so, in which way (see Section 3.6). Similar constraints and preferences may apply to the storage available, the runtime performance e.g. on large collections, and the support for handling specific tree relations. For instance, the fact that in general μ PID and Virtual Nodes labels do not reflect the document order can be an important disadvantage especially for the evaluation of XPath and XQuery, whose semantics strongly build on node sets being sorted in document order. Lack of support for document order deeply affects the evaluation algorithm and seriously limits the use of most common structural join algorithms. However, if it is guaranteed that at any time during the query evaluation only labels are compared that belong to elements with the same tag path, then the μ PID scheme may actually be a good choice, because μ PID labels of such nodes do respect document order (see Section 3.4.2).⁷

Further criteria to be taken into account when choosing a suitable labelling scheme include the indexing performance (e.g., how many traversals of the document tree are needed for labelling), specific mappings to physical storage [Bremer and Gertz 2006] or other labelling schemes [Wang et al. 2003a], or whether global data structures such as the schema tree or an FST can be used [Gavoille and Peleg 2003; Peleg 1999]. Also, manipulating node labels in a restricted environment (such as standard SQL without user-defined extensions) may be an issue (see Chapter 7). For instance, some approaches require full regular expressions [Yoshikawa et al. 2001] or bitwise parsing [O’Neil et al. 2004], which may or may not be supported by the runtime environment.

As a general finding, however, the experiments in Section 4.6 show that the ability of a labelling scheme to reconstruct certain query constraints (most notably, $parent^i$) is key to efficient XML query evaluation. This is confirmed in different settings by Christophides et al. [2003] and by Lu et al. [2005]. Consequently, while the subtree encodings reviewed in Section 3.3 produce small node labels that can be used in structural joins to decide $Child^+$ constraints, they are usually outperformed by schemes like BIRD that exploit the power of reconstruction. We empirically support this claim in further experiments to be presented later (see Chapter 8), where BIRD competes with the Pre/Post labelling (see Section 3.3.2) in a relational environment. The same effect can be expected for other schemes with reconstruction support, e.g., Dewey or ORDPATH. As the use of ORDPATH in a commercial RDBS [O’Neil et al. 2004] shows, these approaches are of great practical interest. The plain Dewey scheme is easy to implement and fairly robust, but needs of course a binary label encoding to prevent excess label size. ORDPATH is particularly attractive due to its support for unlimited updates, which in a highly dynamic setting will outweigh by far the loss of a little expressivity and space efficiency.

4.8 Optimizations and Open Problems

Layered BIRD and unbalanced BIRD labelling. The comparison and experimental evaluation of multiple labelling schemes above has shown that the child-balanced, non-layered BIRD scheme is highly efficient and expressive. The practical performance and benefit of the Layered BIRD labelling outlined in Section 4.5.2 remains to be evaluated. As a matter of fact there is also an *unbalanced* variant of BIRD labelling, which emerges naturally when fixing a balancing factor of $b = 0$. Additional work omitted here shows that the unbalanced labelling scheme creates labels and weights that are smaller and less likely to be affected by node insertions. Intuitively, this is explained by the fact that without balancing fewer document nodes are forced to have the same weight and hence labels that are multiples of a specific number. While a weight overflow in any balanced BIRD scheme invalidates the weights and labels of all document nodes that are represented by a sibling, cousin, . . . of the schema node causing the overflow, the unbalanced BIRD labelling restricts this to those elements with exactly the same schema node.

However, without balancing certain tree relations such as *i-th-child* or *nextSibⁱ* can no longer be reconstructed. Furthermore, the creation of unbalanced labels turns out to be more complex than in the balanced case. In particular, the memory consumption during labelling is probably prohibitively high because for

⁷In fact we exploit this feature, to the benefit of μ PID, in our experiments with the X^2 system, whose query kernel processes node sets that were fetched for specific tag paths.

each element visited in the first pass through the document tree, the sequence of tags of its children must be recorded, rather than only the number of children as in the current labelling procedure. This issue would need to be solved before the unbalanced BIRD scheme might become a more space-efficient and robust alternative to the balanced BIRD labelling described above.

Structural summaries of document subtrees. By contrast, there are other ways how the BIRD scheme could be optimized to obtain labels that are smaller and more robust against modifications of the document tree (most notable, node insertions at arbitrary positions). As suggested by the position of BIRD in the trade-off space in Figure 4.10 on page 65, these are the major challenges faced by our approach. A possible technique for reducing the size of BIRD labels and weights has been hinted at in Section 4.6.1. There we sketched an alternative structural summary which is different from the schema tree that we used as weight index throughout this chapter. Currently all document nodes with the same tag path are assigned the same weight, as stated by the first invariant on page 43. Obviously this may cause many labels to be reserved for virtual nodes, namely, when some document nodes with a given path have a large subtree (and hence, a large weight) while other document nodes with the same tag path would only need a much smaller weight. The sample document in Figure 4.3 *a.* on page 46 illustrates this effect: although the node with the BIRD label 36 (the rightmost child of the document root) has only two children, which would require a BIRD of 3 (see Section 4.2.1), the actual weight of the node 36 is 9. The reason is that other document nodes with the same tag path as node 36 (namely, its three siblings 9, 18 and 27) all have larger subtrees which do not fit a weight of 3.

It therefore seems promising to decouple the weights from the tag paths by using a structural summary in which every node represents element with a similar subtree size, rather than elements with the same tag path. As a matter of fact, BIRD can be used with a variety of structural summaries covered by Definition 2.5 on page 11. The only restriction is that the *Child* relation on document nodes must be preserved by the structural summary in the obvious sense, so that ancestor weights are available when reconstructing *parent*^{*i*}. Clearly this is true for the schema tree: recall from Section 2.3 that given two document nodes u and v with respective tag paths $\pi(u)$ and $\pi(v)$, if we have a D -constraint $Child(u, v)$ in the document tree then the corresponding S -constraint $Child^i(\pi(u), \pi(v))$ holds true in the schema tree. An open question is which other structural summaries could be used that satisfy the above condition and at the same time treat elements as equivalent that have subtrees of a similar size or structure. Note that this could not only help to decrease labels and weights, but also make BIRD labelling more robust: after all, weight changes caused by overflows would no longer propagate to all document nodes with the same tag path, regardless of their subtree size. Instead only nodes with a specific sort of subtree would be affected. Depending on how heterogeneous the document structure is, this may mean that many node labels that are currently invalidated for no reason remain unchanged.

Part III

Index Structures for XML

Index Structures for Structured Documents

5.1 Overview

This chapter surveys existing techniques for indexing both the structure and the textual contents of XML documents. The various table- or tree-shaped data structures presented here are all instances of *centralized structural summaries* (see Definition 2.5 on page 11). As such they could in principle be complemented by decentralized summaries as those discussed before (see Chapters 3 and 4). From the wealth of centralized approaches to capturing the structure of XML documents, only a few representative indexing schemes can be reviewed in the scope of this thesis. For a more detailed survey, the reader is referred to earlier work [Weigel 2002].

5.2 Inverted Files

The most basic document indices are *inverted files* (also called *inverted lists*). These table-like index structures are standard in Information Retrieval on “flat” documents (i.e., documents without markup) but have also been used for semistructured data like XML documents, either stand-alone or in combination with more complex structure indices (see below).

A typical inverted file is shown in Figure 5.1 *a.* on the following page. It indexes the textual contents of the document tree D in Figure 2.1 *b.* on page 8, as follows: each row in the table (or *posting* in the file) maps a unique keyword $k \in K$ (left column) to the places where k occurs in the documents (right column)—much in the same way as the keyword index in the backmatter of this thesis. In the example, each keyword occurrence is given as the unique node label of the containing element; multiple occurrences of k in the same element are not distinguished. However, depending on the underlying data and query model, the index could be either coarser (identifying only the documents where k occurs, as in flat-text retrieval) or more fine-grained (indicating the exact position of k 's occurrences in a given element, as needed when evaluating queries with text distance constraints). In addition, the physical organization of the postings may vary; e.g., the table shown in Figure 5.1 *a.* could also be in first normal form. In Information Retrieval typically not all distinct keywords are indexed, the most frequent ones (so-called *stop words* like conjunctions and prepositions) being left out to keep the index smaller. Finally, keywords are often normalized (e.g., by stemming and conversion to lower-case) in order to map all morphological and orthographical variants of a term to the same set of occurrences.

Inverted files are also used to index tag occurrences in structured documents. Figure 5.1 *b.* on the following page depicts such a tag index for the document tree D in Figure 2.1 *b.* on page 8. Each distinct tag $t \in T$ is mapped to the set of nodes with tag t in D . Note that the two inverted files in Figures 5.1 *a.–b.* together support simple queries against D . For instance, to select all `name` nodes containing the keyword “Lee”, one would look up “lee” in the first table and `name` in the second one, and then intersect the two resulting node sets. This produces the query result $\{21, 30, 39\}$ which is correct, as can be verified in Figure 2.1 *b.*

"female"	26,34,42
"Jeff"	12
"Jill"	21
"Lee"	21,30,39
"Mae"	30
"male"	17
"MSc"	16
"PhD"	25
"Smith"	12
"Sue"	39

a. inverted text file

edu	16,25
gender	42
name	12,21,30,39
people	0
person	9,18,27,36
profile	15,24,33
sex	17,26,34

b. inverted tag file

/people	0
/people/person	9,18,27,36
/people/person/name	12,21,30,39
/people/person/profile	15,24,33
/people/person/profile/edu	16,25
/people/person/profile/sex	17,26,34
/people/person/gender	42

c. inverted path file

"female"	/people/person/profile/sex /0/18/24/26 /0/27/33/34
	/people/person/gender /0/36/42
"Jeff"	/people/person/name /0/9/12
"Jill"	/people/person/name /0/18/21
⋮	⋮
⋮	⋮

d. inverted text/path file

Figure 5.1: Inverted files for the document tree in Figure 2.1 b. on page 8.

By contrast, locating the occurrences of an entire tag path $p = /t_0/\dots/t_j$ (where $j > 0$) in D with an inverted tag file is cumbersome and often inefficient. All tags t_l ($0 \leq l \leq j$) in p must be looked up separately in the index, which produces $j + 1$ node sets. For each $j + 1$ -tuple $\langle v_0, \dots, v_j \rangle$ in the look-up result one must then check whether $Child(v_l, v_{l+1})$ holds true for all $0 \leq l < j$. Tuples for which this fails do not represent element paths in D . For instance, let p be the tag path `/people/person/name` in Figure 2.1 b. on page 8. Looking up the three tags `people`, `person` and `name` in the table in Figure 5.1 b. produces the tuples $\langle 0, 9, 12 \rangle$ and $\langle 0, 9, 21 \rangle$, among many others. The *Child* test reveals that the first tuple is indeed an occurrence of p in D , whereas the second is not a valid element path (because 21 is not a child of 9).

The *Child* test is a special case of a so-called *structural join* [Zhang et al. 2001; Al-Khalifa et al. 2002; Bruno et al. 2002; Chien et al. 2002], where two node sets are compared to find all pairs of nodes in a particular tree relation (most commonly, *Child* or *Child*⁺). Many relations can be decided efficiently for a given node pair when a suitable labelling scheme is available (see Chapters 3 and 4). But even if sophisticated algorithms are used, joining large node sets may be expensive in terms of runtime. In this case the size of the node sets to be joined depends on how often the individual tags in the path occur in the documents. Consider the tag path `/people/person/name` again, and assume there are only few person names in the data, but many `name` nodes occur below other tags, such as `/people/group/name` or `/people/relation/name` and so on. Then the *Child* join will involve a large set of `name` nodes, most of which are not part of the query result (for not being children of `person` nodes). This is because the tag index fails to capture information about the nesting of tags.

A second drawback of indexing singleton tags rather than tag paths is that the number of structural joins needed to rule out invalid tuples grows with the length of the query path—even when the matches to most query nodes are not needed to answer the query. In the example above, unless the `people` and `person` nodes have been explicitly marked as result nodes, the desired answer is just a list of `name` nodes (which of course must have a `person` and a `people` ancestor, but we do not need to know their node labels). An index locating all nodes reached by a specific tag path without touching the ancestors of these nodes can save many structural joins. Such path index structures are presented in the next two sections.

5.3 Atomic Path Indexing

To reduce the number of structural joins needed for matching tag paths, index structures have been proposed that map an entire tag path to the set of nodes D with that path. Each tag path is represented as a single

	/people	/people/person	/people/person/name	/people/person/profile	/people/person/profile/edu	/people/person/profile/sex	/people/person/gender
"female"	0	0	0	0	0	1	1
"Jeff"	0	0	1	0	0	0	0
"Jill"	0	0	1	0	0	0	0
"Lee"	0	0	1	0	0	0	0
"Mae"	0	0	1	0	0	0	0
"male"	0	0	0	0	0	1	0
"MSc"	0	0	0	0	1	0	0
"PhD"	0	0	0	0	1	0	0
"Smith"	0	0	1	0	0	0	0
"Sue"	0	0	1	0	0	0	0

Figure 5.2: Two-dimensional path bitmap for the document tree in Figure 2.1 *b.* on page 8.

text string (like those used for illustration throughout this text) that contains all the tags on that path in top-down order. Note that prefixes shared by distinct tag paths are duplicated in their respective strings: e.g., the common prefix of `/people/person/name` and `/people/person/profile` is stored redundantly. We refer to this as *atomic path indexing* since tag paths are treated as monolithic objects (rather than sequences of tags, as with the *compositional* path representation described below).

5.3.1 Inverted Path Files

A simple way to index tag paths is to put them in an inverted file, either as keys or as values. Figure 5.1 *c.* on the facing page depicts a table similar to the one in *b.*, but with entire tag paths in the first column. Given such an index, query paths involving only *Child* steps can be easily matched. In fact, the paths could be represented physically as a B^+ -Tree or a Trie [Fredkin 1960] to accelerate the look-up. By contrast, query paths with *Child*⁺ or *Child*^{*} steps or missing tag constraints require special string-matching techniques that allow to ignore steps in the indexed tag paths. Details are given in Section 7.4.1 for XRel, an atomic path index by Yoshikawa et al. [2001].

To match both tag paths and keyword constraints without having to intersect node sets looked up in separate text and path indices (such as those in Figures 5.1 *a.* and *c.*), Sacks-Davis et al. [1997] combine both into a single table, shown in Figure 5.1 *d.* This inverted text/path file differs from the original inverted text file in two respects. First, the occurrences in the second column are no longer singleton node labels, but sequences of labels representing element paths in the documents. For instance, while the posting for “*female*” in Figure 5.1 *a.* contains the node label 26, among others, the corresponding posting in Figure 5.1 *d.* contains the element path `/0/18/24/26` instead. Second, the occurrences in a given posting are grouped by distinct tag paths, which are stored with each group. In Figure 5.1 *d.*, the “*female*” posting comprises two groups: the first one contains two occurrences of the tag path `/people/person/profile/sex` (namely, the element paths leading to nodes 26 and 34), whereas the second group contains only one occurrence of another tag path, `/people/person/gender`. Again, special string-matching techniques are needed to handle query paths with descendant steps or tag wildcards, as mentioned above.

5.3.2 Path Bitmaps

In flat-text Boolean Information Retrieval, keyword occurrences in documents are traditionally indexed using a two-dimensional bitmap called the *document/term matrix*. Imagine the bitmap as a table with one column for each document and one row for each distinct keyword (*term*) occurring in these documents. Given any combination of a keyword *k* and a document *d*, the bitmap value $\langle k, d \rangle$ in the corresponding

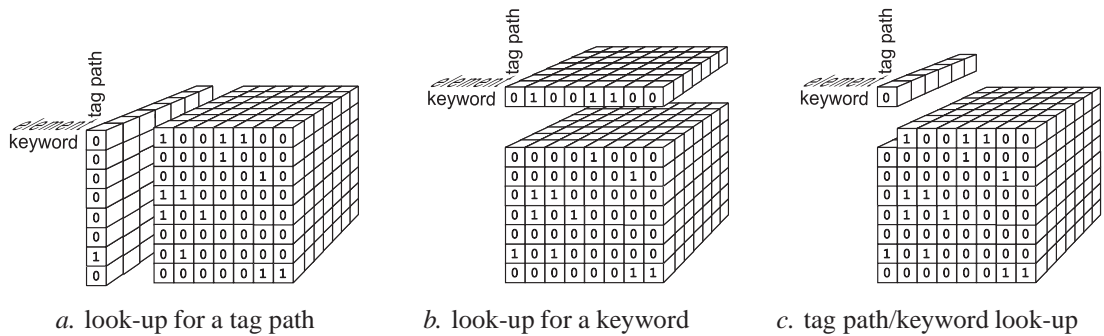


Figure 5.3: Different operations on a three-dimensional path bitmap (BitCube).

table cell indicates whether or not k occurs in d (values 1 and 0, respectively).

The same idea can be applied to structured documents by reserving a single column for each distinct tag path, rather than each document. Figure 5.2 on the preceding page depicts the resulting two-dimensional bitmap for the document tree D in Figure 2.1 *b.* on page 8. Note that since all combinations of keywords and tag paths are materialized, the bitmap is generally sparse. For instance, in the two leftmost columns no bit is set because the `people` and `person` nodes in D do not contain text. By contrast, from the third column we can tell that six distinct keywords occur in `name` nodes. Bit vector operations such as conjunction and disjunction permit to test simple Boolean keyword constraints against tag paths: e.g., the only tag path leading to occurrences of both “*female*” and “*male*” is `/people/person/profile/sex` (bitwise conjunction of the “*female*” and “*male*” rows in Figure 5.2). However, there is no way to determine from the bitmap whether there is any single node in D that contains these two keywords together, because no pointers to individual occurrences of keywords and/or tag paths are given. In terms of the Three-Level Model of XML Retrieval introduced in Section 2.4, the two-dimensional bitmap only indexes information on the schema level, but not the document level.

It seems natural to add a third dimension to the bitmap which captures information on the document level. The *BitCube* proposed by Yoon et al. [2001] is such a three-dimensional bitmap, consisting of a keyword axis, a document axis and an element (or element path) axis. As with the document/term matrix above, we substitute tag paths to documents in order to have the full schema information reflected in the index structure. Thus for any triple $\langle k, p, v \rangle$ consisting of a keyword k , a tag path p and a document node v , a bitmap value of 1 indicates that node v with path p contains an occurrence of k . Like the two-dimensional path bitmap, the *BitCube* may be extremely sparse. For instance, a bitmap value of 0 is stored for every tuple $\langle k', p, v \rangle$ consisting of p and v and any keyword k' that does *not* occur in v .

Figure 5.3 illustrates different ways to look up information in the *BitCube*. In *a.*, a vertical slice of the cube is read which contains all values $\langle k, p, v \rangle$ for a fixed tag path p . This basically produces a tag path-specific inverted text file (compare this to Figure 5.1 *a.* on page 72). Analogously, a horizontal slice of the *BitCube*, as shown in Figure 5.3 *b.*, corresponds to a keyword-specific inverted path file (see Figure 5.1 *c.*), or a single posting in the combined text/path file by Sacks-Davis et al. (see Figure 5.1 *d.*). Finally, a combination of both operations produces a vector containing all elements (or, alternatively, element paths) that have a specific tag path and contain a specific keyword, as depicted in Figure 5.3 *c.*

5.4 Compositional Path Indexing

A variety of path indices have been proposed which are more or less close to the schema tree introduced in the previous chapter (see Definition 2.6 on page 11). As illustrated in Figure 2.1 *c.* on page 8, each distinct tag path occurring in the document tree D is materialized as a sequence of nodes in the schema tree S , rather than an atomic string value.¹ The most obvious advantage of this *compositional* path representation is that prefixes shared by multiple tag paths are stored only once. For instance, the same `people` and

¹In fact, given the alphabet T of tag symbols, the schema tree S can be viewed as a Trie [Fredkin 1960] created from a set of words over T that represent all distinct tag paths.

person nodes in S are part of the tag paths `/people/person/name` and `/people/person/profile`, among others. For highly heterogeneous collections where the schema tree grows large, this decrease in redundancy – compared to atomic path indexing – can save some space. Another benefit of compositional path indexing for query evaluation against a recursive schema is discussed later (see Section 7.4.1).

Note that since the schema tree does not capture document-level information (recall the Three-Level Model of XML Retrieval illustrated in Figure 2.3 on page 13), additional pointers are needed to locate the occurrences of tag paths in the data. Besides, in order to match keyword constraints, the textual contents of the documents need to be indexed, too. In the sequel we review a couple of alternative ways to realize this.

5.4.1 DataGuide

The perhaps best-known compositional path index for semistructured data is the *DataGuide*, developed in 1997 by Goldman and Widom for the *Lore* retrieval system [McHugh et al. 1997]. For tree data, the DataGuide looks exactly like the schema tree S shown in Figure 2.1 c. on page 8. As mentioned before, in all but very few artificial cases S is small enough to fit main memory.² Thus schema matching in the DataGuide is done by following paths in a memory-resident tree structure, typically starting from the root node. Query paths with unspecific tags or with steps involving *Child*⁺ or *Child*^{*} cause backtracking in S since multiple matches might be found. For instance, the XPath query `/people/person/profile/*` matches two tag paths in S , represented by the nodes #4 and #5 in Figure 2.1 c. on page 8, respectively.

To locate occurrences of tag paths in the documents, Goldman and Widom combine the DataGuide with an inverted path file similar to the one shown in Figure 5.1 c. on page 72. The only difference is that in the left column of the table, the tag paths are represented by the numbers of the corresponding DataGuide nodes, rather than strings. Thus the first row maps the tag path #0 to element 0, the second row maps #1 to elements 9, 18, 27, 36, and so on (the tag path numbers correspond to DataGuide nodes in Figure 2.1 c. on page 8). Together the two index structures allow to match query paths where only the leaf node is a result node, as in the XPath expression `/people/person/name` which returns only `name` nodes. Note that the DataGuide does not provide matches to higher nodes on the query path (e.g., the corresponding `person` nodes).

Combining the DataGuide with an inverted text file like the one shown in Figure 5.1 a. on page 72 permits to match path queries with keyword constraints. For instance, to obtain all elements in D that have the tag path `/people/person/name` and contain the keyword “*lee*”, one would proceed in four steps:

1. Search the given tag path in the DataGuide in Figure 2.1 c. on page 8. This selects the schema node #2 (in this case, a singleton node since the query path comprises only *Child* steps with no tag wildcards).
2. Look up the schema node #2 in the inverted path file (see Figure 5.1 c. on page 72). This produces the element set {12, 21, 30, 39} as matches to the structure part of the query.
3. Look up the given keyword in the inverted text file in Figure 5.1 a. on page 72. This produces the element set {21, 30, 39} as matches to the text part of the query.
4. Compute the intersection of both element sets (if the keyword constraint specified government rather than containment, a structural join of the two sets would be needed instead). This yields {21, 30, 39} as the query result.

Note that steps 2 and 3 are independent of each other and could therefore be executed in reverse order. Step 4 can be expensive for large node sets, especially when a structural join is required (see above).

A number of other compositional path indices for XML have been proposed, most of which resemble the DataGuide to some extent. The remainder of this section briefly reviews a few characteristic approaches. For a more detailed survey and comparison, see [Weigel 2002]. Further XML index structures have been proposed by Chung et al. [2002], Kaushik et al. [2002a], Shin et al. [1998], Wang et al. [2003b] as well as Rao and Moon [2004], among others.

²In theory the schema tree S can grow as large as the document tree D , but only if no tag path occurs twice in D . Typically even highly heterogeneous tree collections such as *Treebank* [Treebank] or *INEX* [INEX] contain considerable structural redundancy. Examples of graph documents where the DataGuide has exponential size are given in previous work [Weigel 2002].

5.4.2 IndexFabric

The *IndexFabric* by Cooper et al. [2001] aims to eliminate step 4 above, where the results of two separate index look-ups for a tag path and a keyword are joined to produce the final query answer. To this end, the inverted path and keyword files are combined with the schema tree S into one large disk-based tree structure, the *IndexFabric*, as follows. For a given tag path p in the documents, let K_p be the set of distinct keywords occurring in any element with path p . If K_p is not empty, then new branches are added below p in S which represent the keywords in K_p as a Trie [Fredkin 1960]. The nodes in these additional branches represent sets of elements with path p that contain a particular keyword in K_p . This way not only the query path, but also the keyword constraints can be matched by following paths in the *IndexFabric*.

For instance, for the tag path $p = /people/person/name$ in the document tree D (see Figure 2.1 b. on page 8), we have $K_p = \{“jeff”, “jill”, “lee”, “mae”, “smith”, “sue”\}$. The *IndexFabric* for D would therefore contain (among others) a path $/people/person/name/l/e/e$ representing the elements 21, 30 and 39 (which have the tag path $/people/person/name$ and contain the keyword “lee”, see above). Similarly, $/people/person/name/s/m/i/t/h$ would represent 12, and $/people/person/name/s/u/e$ would represent 39. Note that these two paths in the *IndexFabric* would share a prefix of four steps, including the s node: like tag paths, keywords below the same tag path are also represented in a compositional fashion (namely, as a Trie) to reduce redundancy.

In terms of the Three-Level Model of XML Retrieval (see Figure 2.3 on page 13), the *IndexFabric* combines information from both the schema level (the tag paths) and the document level (the path and keyword occurrences). Clearly the resulting index structure is too large to be held in main memory. Cooper et al. therefore propose a paging strategy for partitioning the *IndexFabric* on disk in order to restrict the number of page faults during index look-ups. Besides, to save disk space all non-branching parts of paths in the *IndexFabric* are contracted, which reduces the number of nodes in the tree. However, the compressed *IndexFabric* only indexes leaf nodes in D or nodes which contain keywords, and therefore fails to answer certain path queries.

5.4.3 Signature File Hierarchy

The *Signature File Hierarchy* by Chen and Aberer [1998; 1999] pursues a different strategy to alleviate the burden of joining path and keyword occurrences in step 4 (see page 75). Recall that since query paths with $Child^+$ or $Child^*$ steps and/or unspecified labels may have more than one match in the schema tree S , entire subtrees of S must be searched in a backtracking procedure. In the course of this search, multiple schema nodes may be selected, all of which undergo the occurrence look-up in the inverted path file (step 2). However, it might happen that some schema node p does not contribute any occurrences to the query result, because no element with the tag path p contains the query keyword. In this case the occurrences of p are in vain fetched from the inverted path file in step 2 and intersected with element sets from the inverted text file in step 4.

In order to rule out such false positives early during the matching, Chen and Aberer use a well-known Information Retrieval technique to give approximate hints as to which tag paths in S have occurrences that contain a specific query keyword, as follows. Assume that each keyword $k \in K$ occurring in the documents is mapped to a bit string with a fixed length and a fixed number of bits set. This bit string is called the *keyword signature* of k . For any node v in the document tree D , let K_v be the set of distinct keywords occurring in v . A keyword signature for the node v is then created by superimposing the signatures of all keywords in K_v by bitwise disjunction. Note that given the keyword signature of v and the signature of any keyword $k \in K$, the following implication holds: if v contains k , then all bits set in the signature of k are also set in v 's keyword signature (we also say that the signature of v *qualifies* for the signature of k). However, the inverse is in general not true: even if v 's signature qualifies for k 's signature, v may not contain k , since the bit patterns in the signatures of distinct keywords contained by v may happen to overlap and together cover all bits that are set in the signature of k .

These observations can help to avoid needless look-ups and joins in steps 2 and 4. From the contraposition of the implication above, it follows that given the signatures of a set of document nodes and a query keyword k , we may recognize for some (though perhaps not all) nodes that they surely do *not* contain k (namely, those whose signatures contain unset bits that are set in k 's signature). To exploit this during path

matching, Chen and Aberer create a Signature File Hierarchy by annotating the schema tree with keyword signatures and information from the inverted path file, as follows. First, every keyword $k \in K$ to be indexed or queried is mapped to a fixed-length signature (usually the same signature will be recreated from k whenever needed, so that the mapping need not be stored physically). Each schema node p in S holds a *signature file* listing all elements v with the tag path p , along with their keyword signatures. These are created by merging the signatures of the keywords they contain, as described in the previous paragraph.

A signature file serves two purposes: on the one hand, it locates the elements with a particular tag path, thus replacing the inverted path file. On the other hand, it provides an approximate summary of the keywords contained in these elements. Given the signature of the query keyword k and the signature file of a schema node p visited during path matching in S , we may recognize that no element listed in p 's signature file contains k —by examining the signature file, without access to any keyword index. This would save us from joining p 's occurrences with other element sets in vain in step 4. Note that this method is inexact in the sense that occurrences of p with the right bits set in their signature might still be false positives. Hence the subset of p 's occurrences whose signatures look promising for k cannot be used as-is, but must be joined with the look-up result for k , as before. However, occurrences whose signature does not qualify can be safely ignored, without altering the query result.

The path occurrences in the signature files add document-level information to the schema tree, which is therefore unlikely to reside in main memory. For instance, applied to the *IMDb* collection comprising more than 80 million document nodes (see Chapter 13), the contents of all signature files together would easily take up some 640 MB (assuming 32-bit signatures and 32-bit element node labels). As a remedy, Chen and Aberer [1999] suggest storing the keyword signatures for each tag path in a Trie rather than a flat list, which avoids the redundant storage of shared bit prefixes (but of course requires some extra space for the Trie structures).

5.4.4 T-Index

With the *T-Index*, Milo and Suciu [1999] have introduced a family of index structures for tree- or graph-shaped documents, that are all tailored to tag paths of a specific structure, described by a *path template* (hence the name T-Index). The template, to be fixed by the database administrator before creating the index, specifies which tag paths (or fragments thereof) are indexed and which ones are ignored. Depending on the given path template, a T-Index may capture more or less of the document structure than the DataGuide. Textual contents of the documents can be indexed with an inverted keyword file, as with the DataGuide.

Milo and Suciu discuss two particular variants of the T-Index that are of general interest. The *1-Index* covers all tag paths starting from the document root. When dealing with tree documents, the 1-Index looks exactly like the DataGuide. The *2-Index* locates all pairs of ancestor and descendant elements that are linked by a specific sequence of tags. For instance, given a 2-Index for the document tree in Figure 2.1 *b*. on page 8, it would be possible to look up all pairs of nodes $\langle u, v \rangle$ where there exists a third node w such that $Child(u, w)$, $Child(w, v)$, $tag(w) = \text{profile}$ and $tag(v) = \text{edu}$. In the example, these are the node pairs $\langle 9, 16 \rangle$ and $\langle 18, 25 \rangle$. Note that the 2-Index allows to retrieve paths and path fragments anywhere in the documents, not necessarily starting at the root. This saves the search and backtracking needed with the DataGuide or 1-Index when matching query paths whose first step involves the descendant axis, such as `//profile/edu`.

Of course, the 2-Index incurs a heavy storage penalty, being quadratic in the size of the document tree in the worst case. More selective path templates may reduce the index size by ignoring less frequently queried paths. Thus a restricted 2-Index might cover only path fragments of a specific length or with specific tags. However, tuning the T-Index in this way requires a thorough knowledge of both the schema and the query workload.

5.5 Tree and Graph Indexing

The data model introduced in Section 2.1 regards XML documents as trees, deliberately restricting them to the nesting structure their elements. However, index structures have been proposed that take into account cross-links, which can be specified using either XML's ID/IDREF attributes or external mechanisms such

as XLink [XLink] or XPointer [XPointer]. For instance, the *Hopi* index by Schenkel et al. [2004; 2005] supports path queries with descendant steps and tag wildcards against arbitrary graphs. The DataGuide and T-Index presented above are also applicable to documents with cross-links.

The key problem here is how elements that are reached by multiple distinct tag paths should be represented in the schema tree. One solution, adopted by the T-Index, is to treat all elements having the same set of tag paths as occurrences of the same schema node. Thus every schema node represents a set of tag paths, rather than a single path as in the tree case. This preserves the unique mapping from elements to schema nodes and ensures that the 1-Index on graph documents grows lineary with the number of document nodes. However, the schema tree may now contain path duplicates because the sets of tag paths represented by distinct schema nodes are not necessarily disjoint. This causes backtracking during path matching even for queries without tag wildcards and *Child*⁺ or *Child*^{*} steps.

An alternative approach, taken by the DataGuide, is to let each schema node represent exactly one tag path as before, which means that elements reached by multiple tag paths are indexed redundantly. On the one hand, this avoids the extra backtracking incurred by the T-Index. On the other hand, the DataGuide may grow exponentially in the worst case, due to the redundant indexing of elements. However, document collections which cause exponential growth tend to be extremely artificial and are unlikely to occur in practice [Weigel 2002].

A graph document model also entails important difficulties for the use of decentralized structural summaries such as the labelling schemes discussed in Chapter 3. Since any document node may be related to any other regardless of the hierarchical nesting of elements, it is much harder to encode specific tree relationships such as *Child* or *Child*⁺ in a local fashion. The most powerful labelling schemes for XML are therefore restricted to tree documents.

Schenkel [2004] argues that a judicious choice of how to index a given document collection depends on a number of parameters including, e.g., the collection size, the query workload and, for graph documents, the structure of the cross-links. For instance, some parts of the collection may be entirely tree-shaped while others are heavily connected through cross-references. The *FliX* framework by Schenkel provides methods to partition a heterogeneous collection of cross-linked documents, based on different parameters, in order to let each part of the collection benefit from the most appropriate indexing technique. This could be a step towards the semi-automatic selection of structural summaries (both centralized and decentralized) based on the monitoring of data and query statistics, as offered by some commercial relational database systems (e.g., IBM's *DB2*).

5.6 Summary and Discussion

For any index structure in whichever data model, there are at least three possible (and often conflicting) optimization goals:

1. *runtime performance*: To what extent does the index accelerate query evaluation?
2. *storage consumption*: How much space does the index structure take up on disk or in memory?
3. *robustness*: How do changes to the indexed data affect the index?

Besides these general questions, there are additional issues specific to the XML data model. The survey of centralized structural summaries in this chapter, albeit brief and by no means exhaustive, has highlighted some of the key problems to be taken into account when indexing XML data:

4. *path representation*: Are tag paths atomic or compositional?
5. *content and structure*: How does the index combine keyword and path occurrences?
6. *backtracking*: How are unspecific query paths matched?

The following is a short discussion of these issues with respect to the different approaches presented above.

The runtime performance of XML query evaluation with a given index structure depends not only on how fast it locates elements that satisfy some part of the query (e.g., a tag, tag path, or keyword constraint),

but also on how many separate index look-ups and joins are needed to compute the whole query result. As mentioned before, the flat inverted files discussed in Section 5.2 support only look-ups for individual tag or keyword constraints. Each conjunction of two constraints entails the intersection or structural join of (possibly large) sets of elements. Hence the overhead for matching complex queries with branching paths can be considerable. Different algorithms have been proposed to expedite joining; in particular, so-called *holistic twig joins* [Bruno et al. 2002] strive to reduce the size of intermediate results by matching multiple tag or keyword constraints simultaneously. However, since inverted files only cover simple constraints, the initial node sets to be joined may still be large.

This problem is addressed by various path indexing techniques which allow to match the leaf of an entire query path including multiple tag constraints at once, without structural joins. Matches to nodes higher on the path are either materialized in the index, as with the inverted text/path file by Sacks-Davis et al., or reconstructed on-the-fly with a suitable labelling scheme (see Chapters 3 and 4). Major differences exist concerning the representation of tag paths in the index. Atomic path indices like the inverted path or text/path files presented in Section 5.3 store tag paths as strings, thereby duplicating shared path prefixes. This redundancy not only increases the index size, but also makes it harder to handle changes to the path structure (e.g., when a subtree in a document is moved). Compositional path indices like the DataGuide and its variants avoid this redundancy by organizing tag paths in a tree structure, similar to a Trie [Fredkin 1960]. Goldman and Widom [1997] show how to update the DataGuide incrementally in time linear in the number of nodes changed.

In any case, matching unspecific query paths that may have multiple matches in the schema requires an additional effort: for atomic path indexing, substring matching or regular expressions are needed, whereas compositional indices must be searched with backtracking. Moreover, multiple schema matches entail additional look-ups in the inverted files as well as additional joins. The Signature File Hierarchy uses keyword signatures as a heuristic means to avoid needless joins. However, the space overhead in the schema tree can be considerable. Exact methods that materialize entire tree relations (e.g., $Child^+$ as with the 2-Index) are unlikely to scale up to tens of gigabytes. Here an alternative are labelling schemes that encode such tree relations locally. However, this introduces additional caveats concerning updates (see Chapter 3).

Another important question is how to combine path and keyword indexing in order to enable fast look-ups of both without blowing up the index size. The straightforward approach sketched for the DataGuide – i.e., separately look up structure and contents, then join the results – again entails the manipulation of potentially large node sets. Keeping both tag and elements in a single table like the inverted text/path file optimizes combined look-ups, at the expense of a larger index size because the same tag path is indexed repeatedly for distinct keywords. This could be problematic at least for atomic path indices, where the entire path string is literally duplicated. Path bitmaps like the BitCube further aggravate the problem by materializing all possible combinations of tag paths and keywords, rather than only those which actually occur in the documents. Compressing the resulting sparse bitmaps could recuperate wasted space, but would also introduce a runtime overhead for decompression during the index look-up.

One viable approach is taken by the IndexFabric, which materializes all existing keyword/tag path combinations in a large Trie on disk and creates additional main-memory structures for fast access to the right disk pages. However, this involves compression techniques that prevent the IndexFabric from answering all queries. The next chapter presents a different solution with full support for the XML query model above: it is an enhanced DataGuide that features (1) combined structure and keyword indexing on disk, (2) a compositional schema representation in main memory, and (3) efficient keyword-driven pruning during path matching.

The Content-Aware DataGuide (CADG)

6.1 Overview

This chapter presents the *Content-Aware DataGuide (CADG)*, a compositional centralized structural summary based on the DataGuide, which is optimized for the efficient evaluation of queries with combined path and keyword constraints. The attribute “content-aware” is meant to emphasize that unlike pure path indices like the original DataGuide, the CADG combines content and structure matching during all steps of the retrieval process. In particular, it allows to prune branches of the schema tree which are irrelevant with respect to a given set of query keywords, in order to avoid needless path look-ups and backtracking during path matching. Moreover, the join of elements with a specific tag path and keyword occurrence is materialized on disk, which significantly reduces the need for joins of large element sets at runtime.

Together with the BIRD labelling scheme (see Part II), the CADG is the basis for the two other main contributions of this work, namely, the relational query evaluation with the RCADG index (see Part IV) and the incremental query processing with the RCADG Cache (see Part V). Besides, the CADG has also been combined with ranking techniques for structured documents [Weigel et al. 2005a] (see Part VI). In the following, some technical details that are not relevant to this work are omitted for simplicity. A more exhaustive presentation and evaluation of the CADG can be found in earlier work [Weigel et al. 2004a; Weigel 2003].

6.2 Materialized Join of Content and Structure

The previous chapter has highlighted several ways to combine the content and the structure of XML documents to be indexed. This problem is indeed of paramount importance for the efficient evaluation of combined tag path and keyword queries. In this respect, the main drawback of the DataGuide setting described above is that content and structure information are rigorously separated into two different data structures (namely, the inverted keyword and tag path files). This way keywords and tag paths taken from the same query must be looked up independently, as if all their occurrences were equally relevant to the query. Only in the last step of the retrieval process (see page 75) content and structure are brought together again, in a join of potentially large element sets that is computed at runtime.

The experiments with the DataGuide and the inverted files below show that the content/structure join is often a bottleneck during the query evaluation. The CADG avoids this by materializing this join at indexing time: the inverted keyword and tag path files are replaced with a single *element table* containing all triples $\langle p, k, v \rangle$ where a document node v with the tag path p contains the keyword k . Figure 6.1 on the following page depicts the element table for the document tree D in Figure 2.1 *b.* on page 8. Tag paths, keywords and elements are stored in the *pid*, *key* and *eid* columns, respectively. Note how the labels of schema nodes in Figure 2.1 *c.* on page 8 act as foreign keys to the *pid* column in the element table in Figure 6.1. In general the element table is larger than the sum of the two inverted files, for two reasons. First, while every pair $\langle p, v \rangle$ of a tag path p and one of its occurrences v is stored once in the inverted path file, the same pair

pid	key	eid
#0	""	0
#1	""	9
#1	""	18
#1	""	27
#1	""	36
#2	""	12
#2	""	21
#2	""	30

pid	key	eid
#2	""	39
#2	"Jeff"	12
#2	"Jill"	21
#2	"Lee"	21
#2	"Lee"	30
#2	"Lee"	39
#2	"Mae"	30
#2	"Smith"	12

pid	key	eid
#2	"Sue"	39
#3	""	15
#3	""	24
#3	""	33
#4	""	16
#4	""	25
#4	"MSc"	16
#4	"PhD"	25

pid	key	eid
#5	""	17
#5	""	26
#5	""	34
#5	"female"	26
#5	"female"	34
#5	"male"	17
#6	""	42
#6	"female"	42

Figure 6.1: The CADG element table for the document tree in Figure 2.1 b. on page 8.

$\langle p, v \rangle$ occurs repeatedly in the element table, once for each distinct keyword that v contains. Besides, it is convenient to store an additional entry $\langle p, v, "" \rangle$ for each pair $\langle p, v \rangle$ and the empty keyword "" such that occurrences of tag paths can be efficiently looked up without any specific keyword in mind.

For instance, given the tag path $p = /people/person/sex$ represented by the schema node #5 in Figure 2.1 c. on page 8, the element table in Figure 6.1 locates either all occurrences of p (entries $\langle \#5, "", v \rangle$ for any v , i.e., 17, 26 and 34) or only the subset of occurrences of p that contain the keyword "male" (entries $\langle \#5, "male", v \rangle$ for any v , i.e., only 17), whatever is need for answering the query. In the second case, the use of the element table saves one look-up in the inverted text file and one content/structure join, compared to the DataGuide evaluation procedure sketched on page 75.

6.3 Keyword-Driven Path Matching

Another problem faced by path indices is that unselective query paths involving $Child^+$ steps or missing tag constraints can have multiple matches in the index (see Section 5.4 above for an example). In compositional path indices like the DataGuide, these matches are found through backtracking in the schema tree. In the worst case, the whole schema tree must be scanned in this way. Even though this does not entail I/O operations since the schema tree is memory-resident, it may cause some overhead in the case of structurally diverse document collections like *Treebank* [Treebank] or *INEX* [INEX], whose DataGuide contains tens of thousands of nodes. More importantly, however, every schema node selected during path matching causes a separate look-up in the inverted path file, which in turn may produce a set of elements to be joined with look-up results from the inverted text file. Reconsider the sequence of steps for query evaluation with the DataGuide (see page 75): only during the join in step 4 it becomes clear which elements satisfy both the structural and the textual query constraints—after all the I/O for the table look-ups is done.

Unlike the DataGuide (or IndexFabric or T-Index), the CADG allows to skip during path matching branches of the schema tree that represent parts of the documents where the query keyword does not occur, and that therefore cannot contribute to the query result anyway. As an example, consider the XPath query `//person//*[contains(., "male")]` and the schema tree in Figure 2.1 c. on page 8. To answer this query with the DataGuide procedure, we would first look up all schema nodes below #1 (the `person` node) in the inverted path file. The following intersection of the resulting five element sets with the inverted text file posting for "male" would reveal that only node 17 satisfies the query. By contrast, with the CADG the path matching could be restricted to the schema node #5 right away, so that only a single element set would be fetched from the path file in step 2 and intersected in the last step. This of course requires some keyword-specific information to be available on the schema level. In terms of the data model introduced in Chapter 2, the CADG allows to match approximate keyword constraints $Contains'_k$ and $Governs'_k$ for keywords $k \in K$ during path matching (see Section 2.3). In the following we outline two alternative ways to do this (for details see [Weigel et al. 2004a; Weigel 2003]).

6.3.1 The Signature CADG (SCADG)

The first possibility to realize a keyword-driven, or content-aware, path matching is inspired by the Signature File Hierarchy. Recall from Section 5.4.3 that here each schema node keeps a signatures file containing

the elements it represents, and for each element a keyword signature that indicates its textual contents in an approximate manner. The problem is that for tag paths which occur frequently in the documents, the list of occurrences to be scanned and signatures to be compared is long. Besides, all information from the inverted path file must be held in memory. The *Signature CADG (SCADG)* remedies this by merging all keyword signatures in the same signature file into a tag path-specific *containment signature*, much in the same way as the signature for a single element is created from the signatures of the keywords it contains. Thus each node in the schema tree S stores only one signature instead of a whole signature file, which reduces the index size and the number of signatures to be compared during path matching.

The path matching procedure for containment constraints is similar to the one sketched for the Signature File Hierarchy before. During step 1, each schema node p matching a query node with a containment constraint for a keyword k is examined to check whether its containment signature qualifies for the signature of k . If this is not the case, p is ignored, i.e., its occurrences are not looked up and do not take part in subsequent joins. Of course merging multiple keyword signatures into a single containment signature (by bitwise disjunction) may render the content representation even less precise than with the Signature File Hierarchy. However, this only affects the number of false positives that might be overlooked during path matching, whereas the final query result is exact (as with the Signature File Hierarchy).

A second signature attached to each schema node p in S indicates which keywords are governed by the elements with the tag path p . This *government signature* is created by merging the containment signatures of all descendants of p in S . Obviously, if the government signature of p does not qualify for any keyword signature in the query, then there is no point in searching p 's subtree in S for nodes with promising containment signatures. Thus government signatures allow to prune entire subtrees of the schema tree and to ignore their nodes during the look-up and join steps. This applies even very early during step 1, when matching nodes higher on the query path which perhaps do not specify keyword constraints themselves (such as the *person* node in the sample query above).

6.3.2 The Inverted-File CADG (ICADG)

A second variant of the CADG pursues the same goals as the SCADG, but with different means. Unlike the SCADG, the *Inverted-File CADG (ICADG)* does not annotate the schema tree in order to lift some content information up to the schema level. Instead, all tag paths that lead to elements containing query keywords are looked up in the element table before the path matching begins. Imagine these keyword-relevant tag paths as highlighted in the schema tree, indicating which parts of the document schema must be examined (from a keyword-only point of view) and which ones can be safely ignored. In fact, since only the leaves of the paths are stored in the element table, we need to decide efficiently for any schema node visited during path matching whether it is an ancestor of such a keyword-relevant leaf. To this end, the Pre/Max labelling schemes introduced in Chapter 3 is applied to the schema tree. Using this interval labelling, ancestorship can be decided in constant time for any two schema nodes.

For instance, reconsider the sample query `//person//*[contains(.,"male")]` and the schema tree S in Figure 2.1.c. on page 8. A quick look-up for “male” in the element table identifies #5 as the only keyword-relevant schema node. Thus the path leading from #0 to #5 in S should be highlighted, indicating that the other branches leading to #2, #4 and #6 can be ignored. This is achieved by comparing the Pre/Max labels of the schema nodes visited during path matching in S to the labels of keyword-relevant nodes fetched beforehand. In the example, the relevant schema node has the label [#5, #5] (being a leaf of the schema tree). Starting from the root of S with the interval [#0, #6], we proceed since [#5, #5] \subset [#0, #6]. Similarly, [#5, #5] \subset [#1, #6] for the child of the root. However, in the following the intervals [#2, #2], [#4, #4] and [#6, #6] do not contain [#5, #5]. Therefore the only keyword-relevant path in S leads from #1 via #3 to #5.

A minor technical issue concerns the robustness of the ICADG against modifications of the document structure. It has been noted above that the labels of schema nodes in S act as foreign keys to the element table. However, when a new tag path appears in the document collection, the schema node labels may need to be reassigned according to the Pre/Max scheme. Since changing foreign keys to the large element table could entail massive disk I/O, every schema node is given an extra identifier that is used as foreign key instead of the preorder rank of the node. This artificial key value remains constant over time and thus preserves the foreign key relation regardless of the current shape of the schema tree.

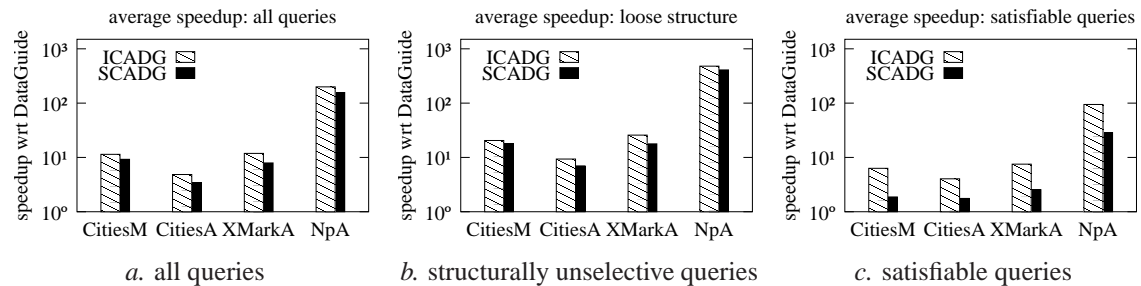


Figure 6.2: Runtime performance gain of the ICADG and SCADG, compared to the DataGuide.

6.4 Experimental Evaluation

The following summarizes the most salient results of the exhaustive experiments that were carried out for the original work on the CADG [Weigel 2003]. The experimental set-up is as follows. Three different index structures have been implemented and integrated with the X^2 retrieval system for XML [Meuss et al. 2005; Meuss et al. 2003; Meuss 2000]: on the one hand, the SCADG and ICADG as main-memory tree structures backed by the element table on disk, and on the other hand, the DataGuide in main memory with the inverted text and tag path files as tables on disk. All three tables are kept in a relational database system, with the following columns indexed: the element table has one B^+ -Tree on the path and keyword columns and another B^+ -Tree on the keyword column alone. The inverted text file is indexed by a B^+ -Tree on the keyword column. The inverted tag path file is indexed by a B^+ -Tree on the path column. The SCADG uses 64-bit signatures.

With this setting, three different document collections have been indexed, whose characteristics are summarized in the appendix (see Section 13.2). *Cities* is very small, with a fairly homogeneous and non-recursive structure, whereas *XMark 29*, a synthetically generated corpus [XMark], is structurally slightly more diverse and contains recursive paths (e.g., *parlist* elements may contain other *parlist* elements). The highly recursive and heterogeneous *NP* collection comprises half a gigabyte of syntactically analyzed German noun phrases [Oesterle and Maier-Meyer 1998]. Both manually written and automatically generated query sets have been evaluated against the three collections, resulting in the following four test suites: *CitiesM* contains 90 hand-crafted queries against the *Cities* collection. *CitiesA* (639 queries), *XMarkA* (192 queries) and *NpA* (571 queries) consist of synthetic queries against the *Cities*, *XMark 29*, and *NP* collections, respectively. All test suites contain both satisfiable and unsatisfiable queries (50% each). Detailed properties and a classification of the queries according to various selectivity measures are given in [Weigel et al. 2004a; Weigel 2003]. Only path queries have been processed in this experiment so as to minimize dependencies on the underlying evaluation strategy and join algorithms employed by the X^2 system.

All tests have been carried out sequentially on the same computer hosting both X^2 and the RDBS back-end (technical details are listed in the appendix, see Test Environment B in Section 13.1). To prevent artefacts due to the file system cache, each query has been processed once without taking the results into account. The following three iterations of the same query were then averaged. Figure 6.2 shows the performance results for three selected subsets of the queries in each test suite: while plot *a*. covers all evaluated queries, plot *b*. in the middle narrows down to unselective queries with mostly *Child*⁺ steps and few tag constraints. Finally, plot *c*. covers all satisfiable queries. Each plot depicts, on a logarithmic scale, the *average speedup* of the SCADG and ICADG over the DataGuide, i.e., the proportion of the CADG's evaluation time to the DataGuide's evaluation time.

In a nutshell, the experiments show that (1) the CADG is considerably faster than the DataGuide especially on large collections and (2) the ICADG always performs a little better than the SCADG. The ICADG beats the DataGuide by a factor 5 to 200 on average, depending on the document collection. Not surprisingly, the speedup increases for poorly structured queries (see Figure 6.2 *b*.), where the potential for subtree pruning is higher. The ICADG evaluates structurally unselective queries against the large *NP* collection 479 times faster than the DataGuide on average. Further statistics show that in this setting one out of two queries are evaluated by two orders of magnitude faster than with the DataGuide [Weigel et al. 2004a].

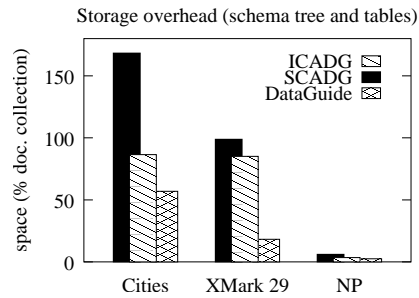


Figure 6.3: Storage consumption of the ICADG, SCADG and DataGuide, relative to the collection size.

For queries with more selective keywords, the speedup again increases by 10-20% on average, and up to 30% for the ICADG. Yet the content awareness pays off even for unselective keywords. Summing up, the CADG performs best on queries with selective keywords and little structure constraints. In practice this is an important class of queries, given that most users are accustomed to web search engines and therefore tend to focus on keyword constraints, especially when they are not familiar with the document schema.

The chart in the Figure 6.2 *c.* on the facing page focuses on the subset of satisfiable queries in each test suite, which makes up about 50%. While the ICADG's average speedup still reaches 4-7 for the smaller test suites (versus 5-12 for all queries in *a.*) and two orders of magnitude for *NpA*, the SCADG performs only twice as good as the DataGuide on the *Cities* and *XMark 29* collections. On *NP* it beats the DataGuide by one order of magnitude (average speedup 28). The reason why the SCADG performs worse in Figure 6.2 *c.* is that this experiment does not include the queries that the SCADG answers particularly fast: obviously it excels at filtering out unsatisfiable queries, especially those with non-existing keywords which it rejects immediately during path matching. In practice this might be a valuable feature, as users are unwilling to accept long response times when there is no result in the end. The ICADG is a little slower here because it recognizes non-existing keywords only after a look-up in the element table.

Figure 6.3 plots the storage consumption of the ICADG, SCADG and DataGuide, respectively. The chart shows that again both CADGs are most effective for large corpora such as the *NP* collection. The ICADG grows to 87% (2.4 MB) and the SCADG to 168% (4.6 MB) of the size of the *Cities* collection in the database (DataGuide 1.6 MB). However, this storage overhead is reduced considerably for *XMark 29* and completely amortized for *NP* (ICADG 3% (21 MB), SCADG 6% (36 MB), DataGuide 3% (15 MB)). Note that the size measures of the SCADG include an extra table containing the signatures for all distinct keywords in the collection. Without this table, the overhead compared to the ICADG is negligible. Further experiments including stop words and unstemmed morphological keyword variants have not substantially changed the results.

6.5 Summary and Discussion

The experiments above clearly show the benefit of the CADG's materialized content/structure join and keyword-driven path matching, which come at a relatively low cost in terms of storage. Note that the results reported here only apply to a hybrid setting, where the structural summary is kept in memory and the rest and the rest of the index structure resides in a relational database system. Chapters 7 and 8 explore a different situation where the index is stored entirely on disk and the whole evaluation process takes place inside the RDBS.

Part IV

Relational Storage of XML

XML Retrieval in Relational Database Systems

7.1 Overview

The indexing approaches presented in Chapters 5 and 6 mostly target native or hybrid retrieval systems where at least some part of the index structure is held in main memory (typically, a centralized structural summary such as the schema tree). However, faced with very large document collections where scalability and retrieval efficiency are major concerns, storing and querying XML data entirely inside a relational database system (RDBS) seems particularly promising because (1) highly efficient access methods for relational data have been developed for over thirty years and (2) query planning and optimization in the relational algebra is well-understood. Besides, nowadays there is a great choice of mature relational databases, some of them freely available, that are already widely deployed and offer many features which are favourable to a productive use. These include, e.g., concurrency, transactions, safety and recovery, as well as sophisticated index structures and algorithms for query planning and optimization.

Consequently, a variety of relational storage schemes for XML have emerged, which are either generic in nature or rely on a fixed schema (e.g., a given DTD or XML Schema [XSD1]). All these approaches have in common that they “shred” the hierarchical XML data into tuples to be stored in the flat data model of the RDBS. On the one hand, possibly expensive joins are necessary to restore part of the original node hierarchy at query time. On the other hand, the resulting tables can be efficiently indexed and searched with the common operators of the relational algebra. This chapter reviews several alternative approaches to XML retrieval in an RDBS, highlighting their respective strengths and weaknesses for different kinds of documents and queries. One particularly interesting question here is in how far existing native XML indexing techniques, like the ones described in the preceding chapters, can be adapted for use in the relational setting.

7.2 Classification of Storage Schemes

A recent survey by Krishnamurthy et al. [2003] provides a comprehensive overview, terminology and classification of a large number of research contributions dealing with XML and RDBSs. First, *storage schemes* are contrasted with *publishing techniques*, whose aim is not to store XML in the relational data model but to make relational data accessible as if it were XML. (The latter are not tightly related to this work and therefore ignored in the sequel.) Relational storage schemes for XML are further differentiated according to the database schema they use for shredding XML data. Approaches in the first class derive a suitable relational schema for each document collection from a given DTD or other prescriptive XML schema, and are therefore called *schema-based* by Krishnamurthy et al.¹ (Yoshikawa et al. [2001] refer to them as *structure-mapping approaches*.) For instance, Schmidt et al. [2000] suggest storing all elements with the

¹Schema-based approaches include work by Shanmugasundaram et al. [1999], Harding et al. [2000], Schmidt et al. [2000], Bohannon et al. [2002], Runapongsa and Patel [2002], Chen et al. [2003], Balmin and Papakonstantinou [2005] and Chebotko et al. [2005].

same tag path together in a separate table. The number of tables needed thus depends on the structural diversity of the documents. The second class is *schema-oblivious* in the sense that all sorts of XML documents, whatever their structure may be, are stored in the same set of tables, designed to fit the XML data model as closely as possible while allowing for efficient query evaluation.² (These storage schemes are therefore called *model-mapping approaches* by Yoshikawa et al.)

Note that schema-oblivious storage schemes may well index the structure of the documents (in fact, we will come to know such schemes in this chapter and the next one). But the document schema does not affect the number of tables and columns used. On the one hand, this means that all elements are stored in a predefined set of tables, which may therefore become large and need appropriate indexing. Besides, keeping all data in a small number of tables might slow down the parallel access by multiple threads. On the other hand, schema-oblivious storage has a number of advantages: (1) No prescriptive schema is needed to index a new collection of XML documents. If desired, a structural summary can be created on the fly while indexing the documents. (2) During query evaluation, only a small fixed number of tables is accessed. There is no need to compute the union of results retrieved from distinct tables. (3) The storage scheme is robust against schema evolution. For instance, no additional table is needed when a new tag path appears in the documents. These issues advocate a schema-oblivious approach in the course of this work. The following brief review of some existing storage schemes therefore covers mainly schema-oblivious works.

7.3 Node Indexing

An obvious way to shred an XML document tree D into relations is to represent each document node in D as a tuple in a *node table*, with enough information to restore specific tree relations through selfjoins of the node table. For instance, if the tuple representing a document node v contains the unique node labels of v and its parent in D , then all parent/child pairs in the documents can be obtained through an equijoin of the node table on the two label columns. Textual contents are either included in the node table or stored in one or more additional tables. We refer to this kind of relational XML storage as *node indexing* schemes in the sequel. Three such schemes are outlined in this section.

7.3.1 The Edge Scheme

The *Edge* scheme by Florescu and Kossmann [1999] uses a node table with five columns that essentially materialized the *Parent* relation. Each document node v is represented as a quintuple containing the unique node label of v , the node label of v 's parent, v 's tag name, the position of v among its siblings (if any) and a flag indicating whether or not v has textual contents. The actual content values are stored in a separate *content table* mapping node labels to strings.³ While matching *Child* steps in a query path is easy with the Edge scheme – a simple equijoin of the node table as sketched above –, handling *Child*⁺ steps is only possible through recursive SQL queries [Krishnamurthy et al. 2003]. Keyword containment constraints entail joins of the node table with the content table. For government constraints again recursive SQL queries would be needed.

Florescu and Kossmann also describe two variants of the Edge scheme that aim to expedite access to relevant tuples in the node table and avoid joins with the content table. First, the node table may be partitioned into a separate table for all nodes with the same tag. Second, further columns may be added to the node table in order to store the attributes of an element and their text values. This is known as *inlining*. However, since not all elements have the same attributes, the resulting node table may contain many null values. Both the partitioning and the inlining turn the storage scheme into a schema-based approach, with the pros and cons listed above.

²Schema-oblivious approaches have been proposed, among others, by [Deutsch et al. 1999], Yoshikawa et al. [2001], Grust [2002], Jiang et al. [2002], Tatarinov et al. [2002], DeHaan et al. [2003], Harding et al. [2003], Chen et al. [2004], Pankowski [2004] and Chen et al. [2005a].

³Actually the Edge scheme is a little more involved, capturing different data types in distinct type-specific content tables.

<i>pre</i>	<i>post</i>	<i>parent</i>	<i>tag</i>	<i>type</i>
0	17		people	Element
1	4	0	person	Element
2	0	1	name	Element
3	3	1	profile	Element
4	1	3	edu	Element
5	2	3	sex	Element
6	9	0	person	Element
7	5	6	name	Element
8	8	6	profile	Element
9	6	8	edu	Element
10	7	8	sex	Element
11	13	0	person	Element
12	10	11	name	Element
13	12	11	profile	Element
14	11	13	sex	Element
15	16	0	person	Element
16	14	15	name	Element
17	15	15	gender	Element

Figure 7.1: Node table of the XPath Accelerator scheme for the document tree in Figure 2.1 b. on page 8.

7.3.2 The XPath Accelerator Scheme

The *XPath Accelerator* scheme by Grust et al. [2002; 2004] also materializes the *Parent* relation, but adds information for handling *Child*⁺ steps and type constraints. Each document node v is represented as a quintuple in the node table which contains the pre- and postorder ranks of v , the preorder rank of v 's parent as well as v 's tag name and node type. Figure 7.1 shows the node table for the document tree in Figure 2.1 b. on page 8. *Child* steps are matched through an equijoin of the node table, as with the Edge scheme. For handling *Child*⁺ steps, Grust et al. takes advantage of the Pre/Post labelling scheme (see Chapter 3). Recall from Section 3.3.2 that given two elements u and v , $Child^+(u, v)$ holds iff $pre(u) < pre(v)$ and $post(v) < post(u)$. This decision procedure translates directly into a predicate for a selfjoin of the node table. Thus the XPath Accelerator efficiently matches *Child*⁺ steps without recursive SQL queries.

Grust et al. show that all XPath axes can be decided through joins with different predicates on the node table columns. In terms of the query model specified in Section 2.2, any query with m query nodes is matched in an m -fold selfjoin of the node table. To expedite the joins, several optimization have been proposed, including the *Staircase Join* [Grust et al. 2003], a new join operator to be integrated into the RDBS kernel, and *shrink-wrapping*, a method to decide *Child*⁺ steps with a more restrictive predicate.

7.3.3 The STORED Scheme

Deutsch et al. [1999] describe a mixed semistructured/relational storage scheme that is at the boundary between schema-based and schema-oblivious approaches. It makes use of data mining techniques for semistructured data [Wang and Liu 1998] in order to devise a relational schema that captures the most regularly structured part of the documents. The remaining data is collected in a so-called *overflow graph* that is not stored in the RDBS, but in a separate database for semistructured data. The creation of the overflow graph may benefit from a prescriptive schema, but does not depend on it. If a document changes, newly inserted data that does not conform to the relational schema is added to the overflow graph.

Queries against the original data in the documents are translated into separate queries to be evaluated by the RDBS and the semistructured database, respectively. Regular path expressions are allowed, but the translation into SQL expressions is non-trivial. Besides, the mediation between the two database systems may cause performance issues. Although in principle any kind of semistructured data can be handled by the STORED scheme, it clearly targets documents with a rather regular structure.

7.4 Path Indexing

Above we raised the question whether path indexing techniques for native XML retrieval could be exploited in a relational retrieval setting as well. Chapter 5 has highlighted two advantages of indexing entire tag paths rather than only individual elements:

1. Path indices allow to match simple query path expressions with fewer joins.
2. Matching tag paths rather than singleton tags provides more selective search conditions, which simplifies index look-ups and reduces the size of intermediate results to be joined.

A third plus is especially relevant to query planning and relevance ranking:

3. Path-specific information (e.g., the node type or statistics about the keyword distribution) need not be stored redundantly for all elements with a given tag path, but only once in the path index.

These observations apply to native or hybrid retrieval systems just as well as to XML retrieval in RDBSs. However, among the many relational storage schemes cited above, few preserve information about entire tag paths or at least fragments thereof. The remainder of this section reviews two such schemes. Similar to the native approaches presented before, they represent tag paths either in atomic or compositional form.

7.4.1 Atomic Path Indexing with XRel

The *XRel* scheme by Yoshikawa et al. [2001] resembles XPath Accelerator to some extent (see Section 7.3.2), but extends the database schema in order to capture schema-level information, as follows. XRel consists of three tables that index tag paths, elements and textual contents, respectively.⁴ Each distinct tag path is represented as a string which is given a unique integer identifier called *path ID*. A *path table* with two columns, *pathexp* and *pathid*, materializes the mapping from path strings to path IDs. The path ID is a foreign key to the other two tables containing document nodes and their contents, respectively. Document nodes are labelled using region encoding (see Section 3.3.3), a labelling scheme similar to Pre/Post that can efficiently decide the *Child*⁺ relation. Each document node v is represented in the node table as the quadruple consisting of v 's start and end position (according to the region encoding) as well as the path ID of v 's tag path and an integer indicating the position of v among its siblings, if any. Similarly, the textual contents of any element v are represented as a tuple in the content table – recall that region encoding treats every text value as a node in its own right – that consists of start and end positions, the path ID of v and the text value to be indexed.

Path queries without keyword constraints are processed in a join of the path and node tables, as follows: relevant tag paths are looked up in the path table (using string matching, see below), and the selected path IDs act as foreign keys to retrieve their occurrences in the node table. As explained in Chapter 5 for native path indices, this means that matching a whole query path of length m (more precisely, retrieving matches to its leaf node) requires just one join of the path and node tables, in contrast with the m -fold selfjoin of the node table needed with the node indexing schemes above. If the query specifies a keyword containment constraint, the path table is joined with the content table instead. However, this way only the position of the matching text value is retrieved, not the containing element itself (this would require another join with the node table).

Tree queries are first divided into path expressions whose leaves are result nodes or branching nodes or leaf nodes in the tree pattern. For instance, the XPath query $Q_3 = /people//person[name]//edu$ is divided into the query paths $/people//person$, $/people//person/name$ and $/people//person//edu$. Then the occurrences of these query paths are retrieved as just described, through multiple joins with the node table. Matches to the entire tree pattern are filtered out during the join by extra predicates that decide the *Child*⁺ relation for the individual occurrences of distinct query paths, using their region-encoded node labels. In the case of Q_3 , e.g., this might rule out name children of person nodes for which no edu descendant could be found.

To understand the look-up of query path expressions in the path table, assume the table contains, among others, the three distinct tag paths $p_1 = /people/person/name$, $p_2 = /people/person/profile/name$ and $p_3 = /people/person/lastname$ as strings in the *pathexp* column. A path query without tag wildcards and *Child*⁺ steps could be matched simply by an equality predicate on the *pathexp* column in the path table. For instance, a suitable predicate for the query $Q_1 = /people/person/name$ in SQL syntax would be $pathexp = '/people/person/name'$, which would correctly select p_1 but not p_2 and p_3 .

⁴The presentation of the XRel scheme here is slightly simplified in order to fit the XML data model from Section 2.1.

Now consider another query $Q_2 = /people/person//name$ that involves a $Child^+$ step. A naïve selection predicate on the path table would be *pathexp* like `'/people/person/%name'`, using SQL's wildcard % for matching any (possibly empty) sequence of characters in a string. However, this would not only match p_1 and p_2 but also p_3 , which is wrong. This is because using %, one cannot distinguish between tag names and their delimiters. Note that *pathexp* like `'/people/person%/name'` would be incorrect too, selecting other tag paths such as, e.g., `/people/personnel/name`. Finally, *pathexp* like `'/people/person/%/name'` would correctly rule out p_3 , but fail to select p_1 .

To handle queries like Q_2 , Yoshikawa et al. replace each delimiter “/” in a tag path with the two-character sequence “#/”. This way the beginning and end of tags in a query path can be marked up independently. For instance, Q_2 is matched using the predicate `#/people#/person#/name`. It is easy to verify that this matches $p_1 = \#/people#/person#/name$ and $p_2 = \#/people#/person#/profile#/name$, but excludes $p_3 = \#/people#/person#/lastname$, as desired. However, more complex path queries such as `/people/*/name` or `/people/**/name` require regular expressions.⁵

The atomic indexing of tag paths as strings in XRel's path table has a number of disadvantages. First, string matching on a large path table can be slow when the selection predicate is a regular expression or a suffix pattern beginning with the % wildcard. Second, the path table contains many duplicates of path prefixes because every tag path is stored in its entirety, from root to leaf. However, query formalisms like XQuery, XPath or the one introduced in Section 2.2 specify path expressions in fragments rather than as root-to-leaf patterns. For instance, the XPath query Q_3 above contains three path fragments (namely, `/people//person`, `name` and `edu`) from which the XRel processor must first restore the query paths `/people//person`, `/people//person/name` and `/people//person//edu` to be looked up in the path table. Third, matching tree queries like Q_3 with XRel sometimes produces many false hits on the schema level that are only discarded during the join with the node table. For instance, when looking up the above query paths for Q_3 in XRel's path table, there is no way to select only those tag paths which refer to the same person node: `/people/faculty/person`, `/people/staff/person/name` and `/people/students/person/edu` are all valid matches to the three query paths, although they do not belong to the same schema hit. Needlessly retrieving and joining their respective occurrences from the node table sometimes slows down the query evaluation considerably (see the experiments in the next chapter). For recursive document collections, this can even lead to false query results. A sample query illustrating this issue and the corresponding SQL code for the XRel scheme are given in the next chapter.

7.4.2 Compositional Path Indexing with BLAS

The *Bi-Labeling Based System (BLAS)* by Chen et al. [2004] is so far the only relational storage scheme for XML we are aware of that represents (suffixes of) tag paths in a compositional manner. The name of the approach alludes to the fact that there are two different kinds of labels, *D-labels* for elements and *P-labels* for tag paths, which are used to match structural query constraints on the document and schema levels, respectively. D-labels are simply integer intervals following region encoding, as with the XRel scheme above. P-labels are generated on the fly during indexing and query evaluation for any tag path suffix encountered in a document or query. A P-label is an integer interval denoting the set of all possible tag paths which share a specific suffix. For instance, the P-label for the tag path suffix `/person/name` represents all possible tag paths `.../person/name`. In particular, each root-to-leaf tag path p (a special case of a path suffix) is assigned a P-label that is stored with each occurrence of p in the node table, similar to the path ID used by XRel above.

The idea is to choose P-labels in such a way that given the P-label P of any tag path suffix in the query, one can easily retrieve all elements with that tag path suffix by inspecting their P-labels in the node table. To this end, the labelling ensures that for any two tag path suffixes s and s' with P-labels P_s and $P_{s'}$, respectively, P_s contains $P_{s'}$ (as an interval) iff s is a suffix of s' . Otherwise P_s and $P_{s'}$ are disjoint. For instance, the P-label for the suffix `/name` contains the P-label for `/person/name` which in turn contains the P-label for `/people/person/name`. Thus a query path $s = //person/name$ can be matched by selecting all tuples in the node table whose P-label is contained in P_s . This would include, e.g., elements reached by `/people/person/name`, but not those below `/people/person/profile/name`

⁵Regular expressions are not part of the SQL-92 standard [SQL2], but included in SQL:1999 [SQL3].

(whose P-label is disjoint with P_s).

P-labels, D-labels and textual contents of elements are all stored together, i.e., there is no separate path table as with XRel. Chen et al. suggest using a separate node table for all elements with the same tag name, similar to the Edge scheme above. Each element v is represented as a tuple consisting of v 's D-label (i.e., its start and end positions in the documents), the P-label of v 's tag path as well as the level of v and its textual content, if any. The P-labels are created on-the-fly for all tag paths encountered during indexing, based on schema statistics like the total number of distinct tags and the height of the document tree. (In this sense BLAS uses a schema-based storage scheme.)

Similarly, when a query Q comes in, P-labels are created for all tag path suffixes in Q . The tag path suffixes in Q are obtained by extracting all sequences of consecutive non-branching *Child* steps from the query path expressions. For instance, the tree query $Q_3 = /people//person[name]//edu$ is cut into four path suffixes, namely, `/people`, `/person`, `/name` and `/edu`. Both the “//” symbol denoting a *Child*⁺ step and XPath predicates indicating a branch act as breakpoints for dividing path expressions into suffixes. These suffixes are looked up as P-labels in the node tables. The resulting four sets of `people`, `person`, `name` and `edu` nodes are then combined through structural joins on their D-labels, in order to filter out those quadruples which indeed form a subtree with the specified structure.

The example above illustrates that path suffixes without *Child*⁺ steps are generally less selective than the original query paths (e.g., compare the four suffixes that BLAS extracts from Q_3 to the three rooted query paths used by XRel above). To obtain more selective look-up predicates, Chen et al. propose two optimizations. First, longer path suffixes can be created for children of a branching query node: in Q_3 , e.g., we can use `/person/name` instead of `/name` because the `person` and `name` nodes are connected through a *Child* step. This might reduce the number of `name` nodes participating in the structural joins. However, the technique does not apply to the `edu` node in Q_3 , because of the descendant step. Thus BLAS still tolerates even more false hits on the schema level than XRel, despite its compositional path representation. Since only path *suffixes* are matched in the first place, there is no way to select only `edu` nodes below a specific `person` node in the schema tree, or even below any `person` at all, let alone to rule out combinations of `person`, `name` and `edu` nodes that do not belong to the same schema hit.

The second optimization makes use of schema information in a DTD (if available) to *unfold* (i.e., instantiate) path expressions like `/people//person//edu` in Q_3 into a set of root-to-leaf paths without *Child*⁺ steps and tag wildcards. This way few look-ups for unselective path suffixes in the node table are replaced with many look-ups for very selective rooted tag paths, in a sort of query expansion. Note that the idea is similar to the path matching that XRel performs through string matching in the path table and that native systems realize by traversing the schema tree. However, with prescriptive schema information as specified by DTDs, the query expansion proposed by Chen et al. is likely to produce many tag paths that do not occur in the documents. For recursive DTDs the unfolding does not even terminate unless a maximum length for the resulting tag paths is fixed. Finally, the unfolding with BLAS seems to happen outside the RDBS, and it is not explained how this could be best done in the relational model.

7.5 Summary and Discussion

Given that today's relational database technology is efficient, scalable, mature and widely deployed, the prospect of seamlessly integrating XML retrieval with RDBSs is particularly tempting. The literature abounds with different ways to store and query XML data as tuples. While many approaches depend on DTDs or other specifications of the document structure to choose a database schema, and some use labelling schemes as decentralized structural summaries of tree relations between individual tuples, very few relational storage schemes leverage the benefit of indexing schema information with a centralized structural summary. Systems that only index singleton elements with their tags, but not paths (as with the Edge scheme) must often join large node sets to find out that only few candidates are actually part of the query result. Sophisticated join algorithms have been developed as a compensation (like the *Staircase Join* by Grust et al. [2003] for XPath Accelerator). But still experimental results such as the ones reported by Chen et al. [2004] or those presented in the next chapter show that path indexing can speed up query evaluation in RDBSs just as much as in a native or hybrid environment.

However, it makes a difference how exactly the schema information is represented. Most observations

made in Chapter 5 for native path indexing also apply to relational systems. On the one hand, atomic path indices like XRel do prevent irrelevant elements from being retrieved and joined in certain cases, but their string representation of tag paths is redundant, awkward to match and of limited use for branching path expressions and recursive document collections. By separating document-level and schema-level information into two distinct tables, XRel can match schema constraints without accessing the full document data, but the resulting path information is often not precise enough to pick exactly the relevant elements in the node table. On the other hand, the compositional path representation of BLAS is quite compact, but produces even more false positives on the schema level than XRel and also requires query preprocessing outside the RDBS (for creating P-labels and unfolding query paths). Moreover, BLAS stores and compares both schema-level and document-level information in node tables, which means larger index scans during schema matching and more I/O needed for updates when the document structure changes.

The next chapter shows how to avoid these shortcomings to make relational XML retrieval benefit even more from path indexing with a centralized structural summary. The *Relational CADG (RCADG)* presented below is based on a compositional path representation which is simpler and more precise than BLAS. It builds on the interval labelling of schema nodes described for the ICADG [Weigel 2003] in Section 6.3.2. As a matter of fact, this approach is dual to BLAS in the following sense. In the ICADG, the interval label of a schema node represents all rooted tag paths with a common prefix. The interval of a longer tag path is contained in the intervals of shorter ones with the same prefix. For instance, the interval for `/people/person` contains the one for `/people/person/name`. By contrast, the P-labels used by BLAS represent sets of tag path suffixes. For the purpose of analogy, they may be regarded as interval-labelled nodes of a modified schema tree containing all *inverse* (i.e., leaf-to-root) tag paths or path suffixes in the documents. The examples above illustrate how indexing path prefixes rather than suffixes can reduce the number and size of intermediate results to be joined. The next chapter explains how the RCADG takes advantage of this observation.

The Relational CADG (RCADG)

8.1 Overview

This chapter introduces the *Relational CADG (RCADG)*, a new time- and space-efficient approach to XML retrieval in relational database systems. The aim of this work is to bring together sophisticated XML indexing techniques and the mature and highly optimized relational technology in order to get the best from both worlds. The RCADG builds on much of the work presented so far, most prominently: the BIRD labelling scheme explained in Chapter 4, a decentralized structural summary with powerful decision and reconstruction capabilities, and the CADG index presented in Chapter 6, a centralized structural summary that combines the schema tree in main memory with a materialization of the content/structure join on disk. The main contributions of the RCADG are (1) a relational storage scheme for the CADG and (2) query planning, translation and evaluation algorithms that together

1. leverage the full schema matching precision of the CADG in an RDBS,
2. preserve its compositional path representation to rule out many false schema hits early,
3. exploit the power of BIRD reconstruction to avoid needless disk I/O and joins of large intermediate results,
4. enable query planning and optimization based on path and keyword selectivity statistics and an analysis of reconstructible relations in the query, and
5. exploit standard relational techniques as much as possible.

The rest of this chapter discusses these issues in more detail. The next section explains the relational storage scheme used by the RCADG and outlines the query evaluation process from an intuitive point of view. Section 8.3 briefly reviews the child-balanced BIRD encoding introduced in Chapter 4, focusing on how to realize decision and reconstruction in the RDBS. Based on these preliminaries, Section 8.4 describes the nuts and bolts of XML retrieval with the RCADG, including query planning and rewriting as well as the generation of SQL code for query matching on the schema and document levels. Section 8.5 reports the results of comparing our implementations of the RCADG, XPath Accelerator and XRel schemes with the original CADG. Section 8.6 provides a quick wrap-up of the RCADG's contributions compared to the related work reviewed in the previous chapter. The last section mentions some remaining issues and open questions.

8.2 The RCADG Storage Scheme

This section describes a relational database scheme for storing the Content-Aware DataGuide (CADG) in an RDBS. As described in Chapter 6, the CADG consists of two data structures, the schema tree and the

<i>pid</i>	<i>parid</i>	<i>maxid</i>	<i>tag</i>	<i>type</i>	<i>level</i>	<i>weight</i>
#0		#6	people	Element	0	45
#1	#0	#6	person	Element	1	9
#2	#1	#2	name	Element	2	3
#3	#1	#5	profile	Element	2	3
#4	#3	#4	edu	Element	3	1
#5	#3	#5	sex	Element	3	1
#6	#1	#6	gender	Element	2	3

Figure 8.1: The RCADG path table for the schema tree in Figure 2.1 c. on page 8.

element table. Since the latter is ready to be stored in an RDBS without further modification, only the schema tree must be migrated to the relational data model. To preserve the compositional representation of tag paths in the CADG, the schema tree should not be stored as a list of path strings, as with the XRel scheme (see Section 7.4.1). Moreover, schema-level and document-level information should be kept separate rather than in one large table, as the one used by the BLAS scheme (see Section 7.4.2).

A straightforward relational representation of the schema tree S is the path table shown in Figure 8.1. The idea is to “shred” S into tuples each representing a single schema node. To decide the $Child$ and $Child^+$ relations in S efficiently, each schema node is labelled according to the Pre/Max encoding (see Chapter 3). The result is similar to applying one of the node indexing schemes mentioned in Section 7.3 to the schema tree (rather than the document tree as proposed there). Each schema node p in S is stored as a tuple

$$\langle pid, parid, maxid, tag, type, level, weight, \dots \rangle$$

that consists of at least the seven fields listed in Table 8.1:

field	description
<i>pid</i>	the preorder rank of p in S (assuming an arbitrary sibling order)
<i>parid</i>	the preorder rank of p 's parent node in S (null for the root of S)
<i>maxid</i>	the greatest preorder rank of any node in the subtree of S rooted in p
<i>tag</i>	the tag name of p in S (the last tag in the tag path corresponding to p)
<i>type</i>	the node type of p in S
<i>level</i>	the level of p in S
<i>weight</i>	the BIRD weight of p (see Section 8.3 below)

Table 8.1: Mandatory fields in the RCADG path table for a given node p in the schema tree S .

Further fields may be added to store tag-path specific information, e.g., statistics for query planning and result ranking or keyword signatures for keyword-driven schema matching (see Section 5.4.3). Examples for such optional fields are given in Table 8.2:

field	description
<i>csig</i>	the containment signature of p in the SCADG (see Section 6.3.1)
<i>gsig</i>	the government signature of p in the SCADG (see Section 6.3.1)
<i>elts</i>	the number of elements with the tag path p
<i>keys</i>	the number of distinct keywords contained in elements with the tag path p

Table 8.2: Selected optional fields in the RCADG path table for a given node p in the schema tree S .

For reasons of clarity, the descriptions in this chapter assume three minor simplifications of the actual path table as it is implemented in our RCADG-based XML database. First, we henceforth consider an RCADG path table consisting only of the mandatory fields in Table 8.1, unless stated otherwise. Second, the string values in the *tag* and *type* columns of Figure 8.1 are given only for illustration purposes. In fact this information is encoded numerically, using a unique mapping from tag names or node types to integer values. Finally, the actual path table has an additional field containing update-robust foreign keys to the element table, as explained for the ICADG (see Section 6.3.2).

For XML retrieval with the RCADG, the path table on disk replaces the schema tree in main memory, which is no longer needed. The element table is the same as for the CADG (see Section 6.2). The RCADG-based retrieval system evaluates a given XML query Q by (1) translating Q into a sequence of SQL statements involving joins of the path and element tables, (2) running these queries in the RDBS to obtain the query result as a set of element tuples (matches), and (3) returning the answer in a suitable form (e.g., by extracting the XML representation of the query matches from the original documents or generating it on the fly). In a first phase, schema matching takes place through an m -fold selfjoin of the path table, where m is the number of query nodes in Q . This produces a preliminary result table containing all schema hits for Q (schema hits are introduced in Section 2.3). Schema-level matching takes advantage of the Pre/Max labels in the path table to decide the ancestorship of schema nodes. In the second phase the schema hits are matched on the document level in repeated joins of the most recent intermediate result with the element table. Successively partial matches to schema hits are either completed or discarded, until all query constraints have been processed and the last result table contains the final query result. Document-level matching benefits specifically from BIRD reconstruction and decision. The following sections explain all steps of this procedure in detail.

8.3 BIRD Revisited: Reconstruction and Decision in the RDBS

In Chapter 4 it has been shown how the reconstruction and decision capabilities of the BIRD labelling scheme can accelerate the query evaluation. BIRD decides and reconstructs many tree relations in the document tree using simple arithmetic computations on numeric element labels and tag path weights. To take advantage of BIRD for the RCADG, these computations must be performed inside the RDBS during query matching on the document level (the second of the abovementioned retrieval phases). More precisely, reconstruction and decision formulae are part of the joins of intermediate result tables with the element table, in the form of either join predicates or projection clauses (the WHERE and FROM parts of a SQL query, respectively) or both.

Figures 8.2 and 8.3 on the following page list rules to deduce suitable join predicates and projection clauses for matching binary query constraints on the document level with child-balanced BIRD labels. In the remainder of this chapter we refer to these rules as *document matching rules*, in contrast to several other types of rule to be introduced later. In each rule, the upper part represents “input” or preconditions, i.e., constraints that are either given in the query or have been deduced through query rewriting (see below) or by applying other rules. The lower part represents “output” or postconditions, i.e., deduced conditions on BIRD labels and weights to be used in the joins, or further constraints to be processed. The join expressions are given in a formal notation as for the relational algebra. Translations of sample expressions into SQL can be found in the next section. Suffice it to say here that the reconstruction and decision formulae in Figures 8.2 and 8.3 on the next page are all simple enough to be expressed in plain SQL [SQL2] without user-defined functions. They involve only comparison operators ($<$, $<=$, $=$), arithmetic operators ($+$, $-$, the multiplication $*$ and the modulo operator $\%$) as well as Boolean operators (AND, OR).

Figure 8.2 on the following page summarizes BIRD’s decision capabilities in eleven rules of the abovementioned form. In each such rule with a binary constraint $R(v_0, v_1)$ in the upper part, the lower part specifies predicates for selecting elements v_1 in a join with the element table such that $R(v_0, v_1)$ is satisfied for a given element v_0 . Both v_0 and the BIRD weight of its tag path $\pi(v_0)$ are assumed to be known. For instance, consider the first rule $DM_{Child_0}^{Dec}$, which states that the nodes in the subtree rooted in a particular element v_0 are exactly those elements whose BIRD label is greater than or equal to v_0 ’s label but smaller than v_0 ’s label plus its weight. Applied to element $v_0 = 24$ in Figure 4.3 a. on page 46, whose tag path $\pi(v_0) = \#3$ has the weight 3 (see Figure 4.3 b. on page 46), the rule $DM_{Child_0}^{Dec}$ selects as v_0 ’s descendants (including v_0) all elements v_1 where $24 \leq v_1.eid < 24 + 3$, i.e., the elements 24, 25 and 26 in Figure 4.3 a. This mirrors exactly BIRD’s decision procedure for $Child_0^*$ that is described in Chapter 4.

In the same way most other binary relations listed in Table 2.1 on page 9 can be decided (except *i-th-Following*, *NextElt_i^j* and their inverses, which are not supported by BIRD), using the corresponding rules in Figure 8.2. Note that constraints of the form $Parent_i^j(v_0, v_1)$ or $Child_i^j(v_0, v_1)$ are rewritten into $Child_0^*(v_0, v_1)$ and $Parent_0^*(v_0, v_1)$ earlier during the evaluation, after matching their proximity bounds on

$$\begin{array}{c}
 \frac{Child_0^*(v_0, v_1)}{v_1.eid \geq v_0.eid \quad \wedge \quad v_1.eid < v_0.eid + \pi(v_0).weight} (DM_{Child_0^*}^{Dec}) \\
 \\
 \frac{Parent_0^*(v_0, v_1)}{v_1.eid \leq v_0.eid \quad \wedge \quad v_1.eid > v_0.eid - \pi(v_1).weight} (DM_{Parent_0^*}^{Dec}) \\
 \\
 \frac{NextSib_1^*(v_0, v_1) \quad \exists v_2 : Parent(v_0, v_2)}{v_1.eid > v_0.eid \quad \wedge \quad v_1.eid < v_2.eid + \pi(v_2).weight \quad \wedge \quad v_1.eid \bmod \pi(v_0).weight = 0} (DM_{NextSib_1^*}^{Dec}) \\
 \\
 \frac{NextSib_i^j(v_0, v_1)}{NextSib_1^*(v_0, v_1) \quad \wedge \quad \frac{v_1.eid - v_0.eid}{\pi(v_0).weight} \geq i \quad \wedge \quad \frac{v_1.eid - v_0.eid}{\pi(v_0).weight} \leq j} (DM_{NextSib_i^j}^{Dec}) \\
 \\
 \frac{PrevSib_1^*(v_0, v_1) \quad \exists v_2 : Parent(v_0, v_2)}{v_1.eid < v_0.eid \quad \wedge \quad v_1.eid > v_2.eid \quad \wedge \quad v_1.eid \bmod \pi(v_0).weight = 0} (DM_{PrevSib_1^*}^{Dec}) \\
 \\
 \frac{PrevSib_i^j(v_0, v_1)}{PrevSib_1^*(v_0, v_1) \quad \wedge \quad \frac{v_0.eid - v_1.eid}{\pi(v_0).weight} \geq i \quad \wedge \quad \frac{v_0.eid - v_1.eid}{\pi(v_0).weight} \leq j} (DM_{PrevSib_i^j}^{Dec}) \\
 \\
 \frac{Following(v_0, v_1)}{v_1.eid \geq v_0.eid + \pi(v_0).weight} (DM_{Following}^{Dec}) \qquad \frac{NextElt_1^*(v_0, v_1)}{v_1.eid > v_0.eid} (DM_{NextElt_1^*}^{Dec}) \\
 \\
 \frac{Preceding(v_0, v_1)}{v_1.eid \leq v_0.eid - \pi(v_1).weight} (DM_{Preceding}^{Dec}) \qquad \frac{PrevElt_1^*(v_0, v_1)}{v_1.eid < v_0.eid} (DM_{PrevElt_1^*}^{Dec}) \\
 \\
 \frac{Self(v_0, v_1)}{v_1.eid = v_0.eid} (DM_{Self}^{Dec})
 \end{array}$$

 Figure 8.2: BIRD document matching rules for deciding binary tree relations ($v_0, v_1, v_2 \in V$).

$$\begin{array}{c}
 \frac{Parent_0^*(v_0, v_1)}{v_1.eid = v_0.eid - (v_0.eid \bmod \pi(v_1).weight)} (DM_{Parent_0^*}^{Rec}) \\
 \\
 \frac{PrevSib_i^i(v_0, v_1) \quad \exists v_2 : Parent(v_0, v_2) \quad \wedge \quad v_0.eid - i \cdot \pi(v_0).weight > v_2.eid}{v_1.eid = v_0.eid - i \cdot \pi(v_0).weight} (DM_{PrevSib_i^i}^{Rec}) \\
 \\
 \frac{NextSib_i^i(v_0, v_1) \quad \exists v_2 : Parent(v_0, v_2) \quad \wedge \quad v_0.eid + i \cdot \pi(v_0).weight < v_2.eid + \pi(v_2).weight}{v_1.eid = v_0.eid + i \cdot \pi(v_0).weight} (DM_{NextSib_i^i}^{Rec}) \\
 \\
 \frac{Self(v_0, v_1)}{v_1.eid = v_0.eid} (DM_{Self}^{Rec})
 \end{array}$$

 Figure 8.3: BIRD document matching rules for reconstructing binary tree relations ($v_0, v_1, v_2 \in V, i \in \mathbb{N}$).

the schema level (see Section 8.4.2). Therefore no further rules for these relations are needed at this stage.

The four decision rules for *NextSib* and *PrevSib* in Figure 8.2 expect as input the BIRD label and weight not only of v_0 , but also of its parent v_2 . Similarly, the rule $DM_{Preceding}^{Dec}$ can only be applied when given the weight of the elements v_1 to be selected. The evaluation procedure presented in the next section makes sure that (1) during schema-level matching all required weights are extracted from the path table and (2) during document-level matching only rules for constraints $R(v_0, v_1)$ are applied for which v_0 (and possibly v_2) is known, either from previous joins with the element table or through reconstruction (see the next paragraph).

Figure 8.3 on the facing page lists the most important reconstruction rules for the BIRD scheme. For any constraint $R(v_0, v_1)$ in the upper part of a reconstruction rule, the singleton R -image of v_0 can be computed from the BIRD label and weight of v_0 as specified in the lower part of that rule. Like in the decision case, some reconstruction rules require as additional input the label and weight of v_0 's parent v_2 or the weight of the element v_1 to be reconstructed, which are retrieved earlier during the evaluation process. For instance, to match a query constraint $Parent_1^1$, which is rewritten to $Parent_0^*$ after schema-level matching, the rule $DM_{Parent_0^*}^{Rec}$ can only be applied when both v_0 and the corresponding parent weight $\pi(v_1).weight$ are given. For $v_0 = 24$ in Figure 4.3 *a.* on page 46, e.g., we know from schema-level matching that v_1 's tag path $\pi(v_1) = \#1$ has the weight 9 (see Figure 4.3 *b.* on page 46). Thus the BIRD label of v_1 is reconstructed as $v_1.eid = 24 - (24 \bmod 9) = 18$ according to the rule $DM_{Parent_0^*}^{Rec}$. Note that apart from the trivial reconstruction of the *Self* relation (rule DM_{Self}^{Rec} in Figure 8.3), only ancestors and preceding siblings at an arbitrary, but fixed distance i can be reconstructed with the RCADG. Other reconstructible relations (see Table 3.1 on page 20) are not considered here.

8.4 Query Evaluation with the RCADG

```

1 // evaluateQuery: RCADG query evaluation from scratch
2 // → Q: the query to be evaluated
3 procedure evaluateQuery (Q: query)
4   // schema-level matching
5   ↗ call rewriteSchemaLevel (Q) // see Section 8.4.1
6   ↘ call matchSchemaLevel (Q) // see Section 8.4.2
7   // document-level matching
8   ↗ call rewriteDocLevel (Q) // see Section 8.4.3
9   P := call createPlan (Qv, Qc) // see Section 8.4.4
10  for all steps s ∈ P do
11    call matchDocLevel (Q, s) // see Section 8.4.5
12  end for
13  // projection to result nodes
14  call createResult (Q) // see Section 8.4.6
15 end procedure

```

Algorithm 8.1: RCADG query evaluation from scratch. The input is a query $Q = \langle Q_v, Q_c, Q_r \rangle$ to be evaluated, where Q_v is the set of query nodes, Q_c the set of query edges and Q_r the set of result nodes in Q .

The top-level evaluation procedure for the RCADG is given in Algorithm 8.1. As mentioned before, query evaluation is divided into two phases. In phase 1, the query constraints are processed on the schema level (lines 5 to 6), typically at a negligible join cost since the path table is rather small. Initially some rewriting attempts to minimize the query and prepares it for evaluation with the RCADG (line 5). The remaining query constraints are then translated to a single SQL statement expressing a selfjoin of the path table (line 6). This produces a first intermediate result on the schema level consisting of tuples of schema nodes that together form a matching to the entire query graph (the *schema hits*, as defined in Section 2.3). The schema hits are stored as rows in a temporary table with columns for the labels (*pid*) and weights

sid	p1	p2	p3	p4	w1	w2	w3	w4
γ^Q	#1	#2	#4	#5	9	2	1	1

a. intermediate result Q_{-s_0} after phase 1

sid	p1	p2	p3	p4	w1	w2	w3	w4	e4	e1
γ^Q	#1	#2	#4	#5	9	2	1	1	26	18
χ^Q	#1	#2	#4	#5	9	2	1	1	34	27

b. intermediate result Q_{-s_1} after step s_1 in phase 2

sid	p1	p2	p3	p4	w1	w2	w3	w4	e4	e1	e2
γ^Q	#1	#2	#4	#5	9	2	1	1	26	18	21
χ^Q	#1	#2	#4	#5	9	2	1	1	34	27	30

c. intermediate result Q_{-s_2} after step s_2 in phase 2

sid	p1	p2	p3	p4	w1	w2	w3	w4	e4	e1	e2	e3
γ^Q	#1	#2	#4	#5	9	2	1	1	26	18	21	25

d. intermediate result Q_{-s_3} after step s_3 in phase 2

e1	e2	e3	e4
18	21	25	26

e. final result $ans(Q)$ after phase 2

Figure 8.4: RCADG result tables for the query Q in Figure 2.2 *b.* on page 10. The SQL code for computing the results comprises five statements, given in Figures 8.6, 8.12 and 8.17 (see pages 105, 113 and 117, respectively). Four intermediate result tables are created, one in phase 1 (*a.*) and three in phase 2 (*b.–d.*). These tables may be stored persistently to build up a query result cache (see Chapter 10). The final answer to Q (*e.*) is extracted from the result table in *d.* It is visualized as subtree a_2 in Figure 2.1 *e.* on page 8.

(*weight*) of all schema nodes in a tuple. Such a result table is shown in Figure 8.4 *a.* for the sample query Q in Figure 2.2 *b.* on page 10.

In the second evaluation phase (lines 8 to 12 in Algorithm 8.1), further rewriting removes query constraints that have been fully catered for on the schema level (line 8). Then the query is matched on the document level, based on the intermediate schema-level result from phase 1. Step by step occurrences of tag paths in the schema hits are retrieved from the element table, checked against the query constraints, and possibly added to the next intermediate result table (see Figures 8.4 *b.–d.*). Where applicable, reconstruction is used to avoid expensive joins with the element table. First a query plan is created (line 9 in Algorithm 8.1) that specifies which binary constraints shall be reconstructed and in which order the others shall be decided. This also determines which occurrences are obtained through joins with the element table, and in which order. Each step in the constraint-solving plan comprises a number of joins of the most recent result table with the element table, together with reconstructions and decisions, and produces another table with the updated intermediate results. A final projection of the last intermediate table onto all distinct matches to the result nodes in the query (line 14) produces the final answer (see Figure 8.4 *e.*). The rest of this section explains all evaluation stages in detail.

8.4.1 Schema-Level Query Rewriting

Before the actual matching takes place, the query is preprocessed in order to eliminate unneeded query nodes and redundant constraints. Note that the underlying query rewriting is designed for the RCADG and BIRD and by no means exhaustive. However, the rules below are generic and hence applicable to other retrieval scenarios, too. We only sketch a couple of basic rewriting rules here. A thorough analysis of minimization techniques for XML queries is beyond the scope of this thesis.¹

¹XML query rewriting and optimization in different application scenarios has been studied, among others, by McHugh and Widom [1997], Amer-Yahia et al. [2001], Zhang et al. [2001; 2002], Ramanan [2002], Olteanu et al. [2002], Pilar [2002], Grust et al. [2003], Flesca and Furfaro [2003] and Jagadish et al. [2004].

It has already been mentioned that query rewriting is triggered twice during the evaluation procedure, namely, once before schema-level matching (phase 1) and once again before document-level matching (phase 2). The different rules that are applicable in either phase are described here and in Section 8.4.3, respectively. It is not hard to prove that all these rules preserve the query semantics. Note that during the rewriting, any binary query constraint $R_i^j(q, q')$ and its inverse $(R^{-1})_i^j(q', q)$ are treated as interchangeable. In the sequel, let $\mathcal{R}_{prox} = \{Parent, Child, NextSib, PrevSib, NextElt, PrevElt\}$ be the set of binary proximity relations (see Section 2.1). The following query rewriting rules are used by the procedure *rewriteSchemaLevel* at the beginning of retrieval phase 1 (see Algorithm 8.1 on page 101):

Adding query edges. The source and target node of any *PrevSib* or *NextSib* edge (with or without proximity bounds) are linked to the same parent node in the query graph. If exactly one of the two sibling nodes has an outbound $Parent_1^1$ edge (or inbound $Child_1^1$ edge), then another $Parent_1^1$ edge is added that links the other sibling to the same parent. Otherwise, if both siblings are connected to different nodes via $Parent_1^1$ (or $Child_1^1$) edges, then these two nodes are linked by a *Self* constraint. If neither sibling is involved in a parent/child constraint, two $Parent_1^1$ edges to a single new parent node are created.

For instance, consider the query in Figure 8.5 *a.* on the following page that contains two binary query constraints $NextSib(q_6, q_5)$ and $Parent(q_5, q_4)$, among others. Since every match to q_6 is necessarily a sibling of a match to q_5 and therefore has the same parent node, we can safely add a new constraint $Parent(q_6, q_4)$ (see Figure 8.5 *b.*). Without the $Parent(q_5, q_4)$ edge in the original query in *a.*, a new query node q_7 would be created with two edges $Parent(q_5, q_7)$ and $Parent(q_6, q_7)$. Conversely, if q_6 were linked to an existing query node q_7 by a *Parent* edge, then a new edge $Self(q_7, q_4)$ would be added instead.

The purpose of this treatment of sibling nodes is twofold. First, it allows to infer further selection predicates from the newly added *Self* edges. Moreover, making implicit parent/child relations explicit through additional *Parent* edges allows to take these relations into account during query planning (see Section 8.4.4). In particular, the query planner might opt for matching the explicit *Parent* constraint at a certain point during phase 2 in order to apply one of the BIRD document matching rules for deciding or reconstructing the sibling relation afterwards. Recall from the previous section that these rules require the respective parent nodes to be known (see rules $DM_{PrevSib_1^*}^{Dec}$, $DM_{PrevSib_i^j}^{Dec}$, $DM_{NextSib_1^*}^{Dec}$, $DM_{NextSib_i^j}^{Dec}$ in Figure 8.2 and rules $DM_{PrevSib_i^j}^{Rec}$, $DM_{NextSib_i^j}^{Rec}$ in Figure 8.3 on page 100).

Merging query nodes. All (new or existing) binary query constraints are analyzed to *merge* query nodes which must have the same set of matches. For instance, all neighbours reached from a given query node by $Parent_i^j$ edges ($i \in \mathbb{N}$) are merged. The same applies to all other functional query constraints, i.e., $NextSib_i^j$, $NextElt_i^j$, $PrevSib_i^j$, $PrevElt_i^j$, and *Self*. The unary constraints involving two nodes to be merged must be compared in order to reconcile their tag names, node types, levels and keywords. If this is impossible (e.g., when a query node representing only elements and another query node representing only attributes are linked via a *Self* edge), the query is rejected as unsatisfiable.

proximity	first	second	result
lower	$R_i^{\dots}(q_0, q_1)$	$R_{i'}^{\dots}(q_0, q_1)$	$R_{\max\{i, i'\}}^{\dots}(q_0, q_1)$
upper	$R_{\dots}^j(q_0, q_1)$	$R_{\dots}^{j'}(q_0, q_1)$	$R_{\dots}^{\min\{j, j'\}}(q_0, q_1)$
	$R_{\dots}^j(q_0, q_1)$	$R_{\dots}^*(q_0, q_1)$	$R_{\dots}^j(q_0, q_1)$
	$R_{\dots}^*(q_0, q_1)$	$R_{\dots}^*(q_0, q_1)$	$R_{\dots}^*(q_0, q_1)$

Table 8.3: Adjustment of proximity bounds when merging of overlapping binary query constraints ($i, i' \geq 0$: lower bounds; $j, j' \geq 0$: upper bounds; *: unspecified upper bound). Two query edges of type $R \in \mathcal{R}_{prox}$ from q_0 to q_1 (first, second) are replaced with a single query edge of type R from q_0 to q_1 (result).

Merging overlapping query edges. Each pair of edges of type $R \in \mathcal{R}_{prox}$ that *overlap*, i.e., share the same source and target node, is replaced with a single edge of type R . The upper and lower proximity bounds of this new edge are determined as specified in Table 8.3. For instance, when two edges $Parent_i^*$, $Parent_{i'}^*$, with different lower proximity bounds i, i' connect the same pair of query nodes, the resulting

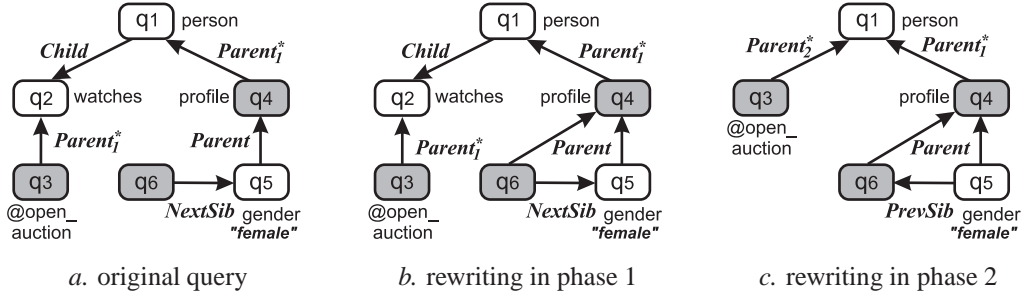


Figure 8.5: RCADG query rewriting. *a.* A sample query against the *XMark* benchmark collection [XMark]. Result nodes are shaded. *b.* The query in *a.* after the first rewriting for matching on the schema level (phase 1). *c.* The query in *a.* after the second rewriting for matching on the document level (phase 2).

merged edge inherits the stricter condition (i.e., the greater lower bound), as specified in the first row in Table 8.3. Note that this rule applies regardless of the upper bounds of the two edges (symbolized by the ellipsis "..."), which are covered by one of the other three rules.

Collapsing transitive query edges. Unselective query nodes cause many tuples in the element table to be selected and joined, and should therefore be eliminated whenever possible. In particular, queries may specify unselective nodes which are neither part of the answer nor needed for path joining. To eliminate such useless parts of the query, we remove all intermediate nodes between two edges of type $R \in \mathcal{R}_{prox}$ in the same direction from the query and replace the two edges with a single direct connection of type R , unless the intermediate node satisfies any of the following conditions: (1) it has unary constraints to be matched; (2) it is also reached by other than the two edges in question; or (3) it is a result node. In all remaining cases the node contributes neither selection predicates nor join predicates nor projection clauses to the SQL queries to be created.

For instance, consider the sample query in Figure 8.5 *a.*. The query node q_2 has two inbound edges, $Child(q_1, q_2)$ and $Parent_1^*(q_3, q_2)$. The $Child$ edge is equivalent to the inverse constraint $Parent(q_2, q_1)$, as mentioned above. Thus q_2 is an intermediate node between two $Parent$ edges in the same direction. However, it cannot be removed because of its tag constraint, which is essential to the query semantics. Without the tag constraint, q_2 would be removed and the two query edges to q_1 and q_3 would be replaced by a single edge linking q_3 to q_1 directly. Table 8.4 summarizes the rules for *collapsing* two edges in this way, i.e., replacing them with a single transitive one that links the source of the inbound edge directly to the target of the outbound edge. As shown in the table, lower and upper proximity bounds add up, if specified (otherwise “don’t care” symbols prevail). Again the ellipsis “...” is a placeholder for ignored bounds to which appropriate rules in Table 8.4 apply in turn.

proximity	inbound	outbound	result
lower	$R_i^{\dots}(q_0, q_1)$	$R_{i'}^{\dots}(q_1, q_2)$	$R_{i+i'}^{\dots}(q_0, q_2)$
upper	$R^j_{\dots}(q_0, q_1)$	$R^j_{\dots}(q_1, q_2)$	$R^{j+j'}_{\dots}(q_0, q_2)$
	$R^j_{\dots}(q_0, q_1)$	$R^*_{\dots}(q_1, q_2)$	$R^*_{\dots}(q_0, q_2)$
	$R^*_{\dots}(q_0, q_1)$	$R^j_{\dots}(q_1, q_2)$	
	$R^*_{\dots}(q_0, q_1)$	$R^*_{\dots}(q_1, q_2)$	

Table 8.4: Adjustment of proximity bounds when collapsing transitive binary query constraints ($i, i' \geq 0$: lower bounds; $j, j' \geq 0$: upper bounds; *: unspecified upper bound). Two query edges of type $R \in \mathcal{R}_{prox}$ from q_0 to q_1 (inbound) and from q_1 to q_2 (outbound) are replaced with a single query edge of type R from q_0 to q_2 (result).

```

CREATE
  TABLE Q_s0 AS                                -- create new result table
SELECT
  createSchemaHitID() AS sid,                  -- create fresh schema hit ID
  PT1.pid AS p1, PT2.pid AS p2,              -- add schema node labels
  PT3.pid AS p3, PT4.pid AS p4,
  PT1.weight AS w1, PT2.weight AS w2,       -- add schema node weights
  PT3.weight AS w3, PT4.weight AS w4
FROM
  PathTable PT1, PathTable PT2, PathTable PT3, PathTable PT4 -- selfjoin of path table
WHERE
  PT1.tag = 'person' AND PT2.tag = 'name' AND -- match tag constraints
  PT3.tag = 'edu' AND PT4.tag = 'sex' AND
  PT1.type = 'Element' AND PT2.type = 'Element' AND -- match node type constraints
  PT3.type = 'Element' AND PT4.type = 'Element' AND
  PT2.parid = PT1.pid AND                    -- decide Child(q1,q2)
  PT3.pid > PT1.pid AND PT3.pid <= PT1.maxid AND -- decide Child*(q1,q3)
  PT1.pid < PT4.pid AND PT1.maxid >= PT4.pid -- decide Parent*(q4,q1)

```

Figure 8.6: SQL code for evaluating the query Q in Figure 2.2 *b*. on page 10 on the schema level (phase 1). A selfjoin of the path table produces the first intermediate result table containing a single schema hit (see Figure 8.4 *a*. on page 102). The table has fields for a schema-hit identifier (sid , created on the fly by the function `createSchemaHitID()`) as well as for the schema node labels (p) and BIRD weights (w) of all nodes in the schema hit. Each pair of columns p_i and w_i corresponds to the query node q_i in Q .

8.4.2 Schema-Level Matching

After the initial rewriting, all S -constraints in the query are matched on the schema level in an m -fold selfjoin of the path table, where m is the number of nodes in the rewritten query. The outcome of this join is a first intermediate result table containing all schema hits for the query, which will later be joined with the element table (see Section 8.4.5). Figure 8.4 *a*. on page 102 shows the schema-matching result for the query Q in Figure 2.2 *b*. on page 10. The sample query has only one schema hit, χ^Q , which is shown as a tree in Figure 2.1 *e*. on page 8. This schema hit occupies one row in the result table, with a unique identifier created on the fly (column sid in Figure 8.4 *a*.). The remaining fields are the labels and weights of all nodes in the schema hit (columns p and w in Figure 8.4 *a*., which correspond to the pid and $weight$ fields from the path table in Figure 8.1 on page 98). For query planning, statistical information about the schema node may be taken from the path table, too (omitted in Figure 8.4 *a*.).

Generating SQL code. The procedure `matchSchemaLevel` called in line 6 of Algorithm 8.1 on page 101 is responsible for generating and executing the SQL code that creates the first result table during phase 1. The sample code corresponding to the table in Figure 8.4 *a*. is given in Figure 8.6. The `CREATE`, `SELECT` and `FROM` clauses are easily derived from the given query Q to be evaluated, as follows. The result table is called Q_{s_0} (denoting step 0 of the evaluation of Q). Since Q has four nodes (see Figure 2.2 *b*. on page 10), four instances of the path table (called `PathTable` here) are joined, and the result is projected onto the path labels and weights as described above. The call to `createSchemaHitID()` in the `SELECT` part is a placeholder for generating fresh identifiers for rows in the result table, which are needed when reusing cached query results (see Chapter 10). It can be realized using, e.g., a system-specific autocounter function.

To generate the `WHERE` part of the statement in Figure 8.6, one must (1) choose those constraints in Q that shall be matched on the schema level and (2) translate these constraints to suitable join conditions. These two tasks are guided by *schema adaptation rules* and *schema matching rules*, respectively.

Adapting query constraints to the schema level. The schema adaptation rules for the RCADG are listed in Figure 8.7 (for unary constraints) and Figure 8.8 on the following page (for binary constraints). Rule SA_0 in Figure 8.7 states that the tag, root, level and type constraints on any query node apply directly to the schema nodes that match this query node (because these are S -constraints, as defined on page 11). By

$$\frac{R(q_0) \quad R \in \mathcal{R}_1 \setminus \{\text{Contains}_k, \text{Governs}_k\}}{R(\pi(v_0))} \text{ (SA}_0\text{)} \quad \frac{R(q_0) \quad R \in \{\text{Contains}_k, \text{Governs}_k\}}{R'(\pi(v_0))} \text{ (SA}_1\text{)}$$

Figure 8.7: RCADG schema adaptation rules for unary query constraints ($q_0, q_1 \in \mathcal{Q}_v$; $v_0, v_1 \in V$; v_l denotes a document node matching q_l).

$$\frac{R(q_0, q_1) \quad R \in \{\text{Parent}, \text{Child}, \text{Self}\}}{R'(\pi(v_0), \pi(v_1))} \text{ (SA}_2\text{)} \quad \frac{R(q_0, q_1) \quad R \in \{\text{NextSib}, \text{PrevSib}, \text{Sibling}\}}{\text{Sibling}'(\pi(v_0), \pi(v_1)) \wedge \neg \text{Root}(\pi(v_1))} \text{ (SA}_3\text{)}$$

$$\frac{R(q_0, q_1) \quad R \in \{\text{Following}, \text{Preceding}, \text{NextElt}\}}{\neg \text{Root}(\pi(v_1))} \text{ (SA}_4\text{)}$$

Figure 8.8: RCADG schema adaptation rules for binary query constraints ($q_0, q_1 \in \mathcal{Q}_v$; $v_0, v_1 \in V$; v_l denotes a document node matching q_l). Proximity bounds are preserved.

$$\frac{\text{Tag}_{t_0}(p) \quad \cdots \quad \text{Tag}_{t_m}(p) \quad t_0, \dots, t_m \in T}{p.\text{tag} = t_0 \vee \cdots \vee p.\text{tag} = t_m} \text{ (SM}_{\text{Tag}}\text{)} \quad \frac{R(p) \quad R \in \mathcal{T}}{p.\text{type} = R} \text{ (SM}_{\text{Type}}\text{)}$$

$$\frac{\text{Level}_i^i(p)}{p.\text{level} = i} \text{ (SM}_{\text{Level}_i^i}\text{)} \quad \frac{\text{Level}_i^j(p)}{p.\text{level} \geq i \wedge p.\text{level} \leq j} \text{ (SM}_{\text{Level}_i^j}\text{)} \quad \frac{\text{Root}(p)}{\text{Level}_0^0(p)} \text{ (SM}_{\text{Root}}\text{)}$$

$$\frac{\text{Contains}'_{k_0}(p) \quad \cdots \quad \text{Contains}'_{k_m}(p) \quad k_0, \dots, k_m \in K}{-(\sigma(k_0) \theta \cdots \theta \sigma(k_m)) \sqcup p.\text{csig} = \top} \text{ (SM}_{\text{Contains}'_k}\text{)}$$

$$\frac{\text{Governs}'_{k_0}(p) \quad \cdots \quad \text{Governs}'_{k_m}(p) \quad k_0, \dots, k_m \in K}{-(\sigma(k_0) \theta \cdots \theta \sigma(k_m)) \sqcup p.\text{gsig} = \top} \text{ (SM}_{\text{Governs}'_k}\text{)}$$

Figure 8.9: RCADG schema matching rules for unary query constraints ($p \in P$; $i < j \in \mathbb{N}$). In the last two rules, σ is the signature creation function; \top denotes a keyword signature with all bits set; $\sqcap, \sqcup, -$ are bit string operators for bitwise conjunction, disjunction and inversion, respectively; and $\theta = \sqcup$ or $\theta = \sqcap$ depending on whether the keyword constraints on p are marked as conjunctive or disjunctive, respectively.

$$\frac{\text{Child}'(p_0, p_1)}{p_1.\text{parid} = p_0.\text{pid}} \text{ (SM}_{\text{Child}'}\text{)} \quad \frac{\text{Child}'_0^*(p_0, p_1)}{p_1.\text{pid} \geq p_0.\text{pid} \wedge p_1.\text{pid} \leq p_0.\text{maxid}} \text{ (SM}_{\text{Child}'_0^*}\text{)}$$

$$\frac{\text{Child}'_i^i(p_0, p_1)}{\text{Child}'_0^*(p_0, p_1) \wedge p_1.\text{level} = p_0.\text{level} + i} \text{ (SM}_{\text{Child}'_i^i}\text{)} \quad \frac{\text{Child}'_i^j(p_0, p_1)}{\text{Child}'_0^*(p_0, p_1) \wedge p_1.\text{level} \geq p_0.\text{level} + i \wedge p_1.\text{level} \leq p_0.\text{level} + j} \text{ (SM}_{\text{Child}'_i^j}\text{)}$$

$$\frac{\text{Sibling}'(p_0, p_1)}{p_1.\text{parid} = p_0.\text{parid}} \text{ (SM}_{\text{Sibling}'}\text{)} \quad \frac{\text{Self}'(p_0, p_1)}{p_1.\text{pid} = p_0.\text{pid}} \text{ (SM}_{\text{Self}'}\text{)}$$

Figure 8.10: RCADG schema matching rules for binary query constraints ($p_0, p_1 \in P$; $i < j \in \mathbb{N}$). The rules $\text{SM}_{\text{Parent}'}, \text{SM}_{\text{Parent}'_0^*}, \text{SM}_{\text{Parent}'_i^i}$ and $\text{SM}_{\text{Parent}'_i^j}$ are omitted for simplicity. They are symmetric to the rules $\text{SM}_{\text{Child}'}, \text{SM}_{\text{Child}'_0^*}, \text{SM}_{\text{Child}'_i^i}$ and $\text{SM}_{\text{Child}'_i^j}$, respectively. Note how $\text{SM}_{\text{Child}'_0^*}$ exploits the Pre/Max labels of schema nodes. For a lower proximity of 1, replace $p_1.\text{pid} \geq p_0.\text{pid}$ with $p_1.\text{pid} > p_0.\text{pid}$ ($\text{SM}_{\text{Child}'_1^*}$).

contrast, keyword constraints can only be matched approximately on the schema level (rule SA_1), and only if the path table contains keyword signatures (the *csig* and *gsig* fields mentioned in Section 8.2). Similarly, *Parent*, *Child* and *Self* constraints in the query translate to $Parent'$, $Child'$ and $Self'$ constraints on schema nodes, according to rule SA_2 in Figure 8.8. Rule SA_3 specifies that the only the unordered sibling relation can be matched in the schema tree and that the root node does not have siblings. The other binary tree relations cannot be matched on the schema level at all, but at least we can infer from *Following*, *Preceding* and *NextElt* constraints in the query that the schema root cannot match their target node (rule SA_4). This is because its only occurrence, the document root, is the first node in document order (and hence not in the *Following*- or *NextElt*-image of any other element) as well as the ancestor of all other elements (and hence not in their *Preceding*-image).

Matching query constraints on the schema level. The schema matching rules in Figures 8.9 and 8.10 on the preceding page determine how to translate unary and binary constraints on schema nodes (as produced by the schema adaptation rules) to join conditions on the path table. The first two rules in Figure 8.9 simply match tag and type constraints against the *tag* and *type* columns in the path table. Likewise, level constraints with fixed proximity bounds translate to an equality check on the *level* column, while level ranges translate to a range check (rules $SM_{Level_i^i}$ and $SM_{Level_j^j}$, respectively). Rule SM_{Root} is justified by the fact that the root is the only node at level 0. Keyword constraints are handled by rules $SM_{Contains'_k}$ and $SM_{Governs'_k}$, if applicable, which respectively check containment and government signatures in the path table. The bit string manipulation in the lower parts of both rules reflects the way keyword signatures are compared (see Section 5.4.3).² Translating these conditions involves the SQL operators for bitwise conjunction (“&”), disjunction (“|”) and inversion (“~”). Thus the keyword-driven schema matching with keyword signatures, as described in Section 6.3.1 for the SCADG, can be mimicked in the path table of the RCADG.

Finally, the rules in Figure 8.10 translate binary constraints on schema nodes that result from the schema adaptation rules in Figure 8.8. The parent/child relation requires a mere equality comparison on the *pid* and *parid* fields of the schema nodes (rule $SM_{Child'}$). The $SM_{Child'_0^*}$ rule exploits the Pre/Max labels in the *pid* and *maxid* columns of the path table to decide ancestorship between two schema nodes, like the ICADG described in Section 6.3.2. The proximity variants $SM_{Child'_i^i}$ and $SM_{Child'_i^j}$ additionally check the level difference of the two schema nodes. Rule $SM_{Sibling'}$ states that two schema nodes are siblings if they have the same parent in S . The *Self'* relation on schema nodes is checked through a simple comparison of their unique labels in the *pid* column (rule $SM_{Self'}$).

8.4.3 Document-Level Query Rewriting

As shown in Algorithm 8.1 on page 101, the second retrieval phase begins with another round of query rewriting. This time the goal is to eliminate parts of the query that contribute only S -constraints, which are not matched on the document level. The following query rewriting rules are used by the procedure *rewriteDocLevel* called in line 8 of Algorithm 8.1:

Removing query nodes. In phase 2, after all S -constraints have been fully processed on the schema level, some query nodes are no longer needed. As in phase 1, this applies to nodes that contribute neither selection predicates nor join predicates nor projection clauses to the SQL queries to be created for document-level matching. In particular, we remove every query node which satisfies all of the following conditions: (1) it is not a result node; (2) it has no keyword constraint; and (3) it is connected to the rest of the query graph by a single *Self* or inbound *Parent* or outbound *Child* edge (with any proximity bounds, if applicable).

It is easy to verify that this preserves all information which is needed to match the query on the document level. For instance, consider the query Q' in Figure 2.2 a. on page 10. Assuming that the node q'_4 is not a result node, it can be safely ignored during document-level matching because it contributes only tag (and perhaps level) constraints, which have already been processed on the schema level (see the next

²Earlier work on the SCADG [Weigel et al. 2004a] explains the choice of the bit string operator θ for keyword conjunctions and disjunctions in Figure 8.9 on the facing page.

section). As a matter of fact, in the schema hits retrieved during phase 1 all tag paths matching q'_3 are of the form `//person//profile/sex`. Therefore there is no need to match q'_4 on the document level. Of course, if q'_4 were reached by, say, a *NextSib* or *NextElt* edge rather than a *Parent* edge, the rule would not apply since sibling constraints are not fully matched on the schema level.

Collapsing transitive query edges. A similar argument concerns intermediate nodes in a chain of two transitive *Parent* or *Child* edges in the query. The two edges can be collapsed and the intermediate node removed unless it is a result node or has keyword constraints. This collapsing rule for phase 2 differs in two respects from the one described in Section 8.4.1 for phase 1: on the one hand, it applies only to *Parent* and *Child* edges; on the other hand, it covers even intermediate nodes with tag, type or level constraints because for *Parent* and *Child* these have been fully catered for on the schema level.

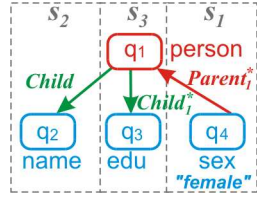
For instance, consider the query shown in Figure 8.5 *a*. on page 104 again. In phase 1, the two edges $Parent_1^*(q_3, q_2)$ and $Child(q_1, q_2)$ (which is equivalent to $Parent(q_2, q_1)$) could not be collapsed (see Figure 8.5 *b*. on page 104) because the intermediate node q_2 has a tag constraint. In phase 2, however, the tag constraint on q_2 has become dispensable since all matches to q_3 have a tag path of the form `//person/watches//open_auction` anyway. Hence q_2 is removed from the query and the two adjacent edges are replaced with a single edge $Parent_2^*(q_3, q_1)$ (see Figure 8.5 *c*.). Note the lower proximity bound 2 resulting from the adjustment rules in Table 8.4 on page 104.

8.4.4 Query Planning

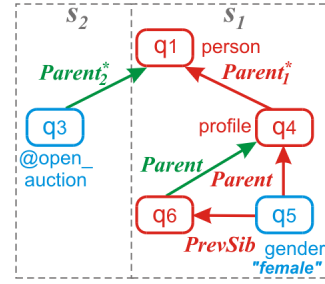
As mentioned before, the document-level matching is performed stepwise according to a query plan created immediately after the second query rewriting. The query plan determines in which order the query nodes are matched, either through joins with the element table or through reconstruction of binary query constraints. The planning goals are (1) to avoid as many joins with the element table as possible by exploiting the full power of BIRD reconstruction, and (2) to minimize the number of tuples in intermediate results by probing the element table with the most selective constraints first (e.g., rare query keywords). Obviously conflicts may arise between these two goals (see below). To simplify the understanding of the main idea, this section describes an algorithm that produces a single query plan for a given query, based on and a naïve but effective optimization strategy. The next subsection explains how exactly to realize joins, reconstruction and decision in the RDDBS. For now we are only concerned with methods to arrange these steps for efficiently evaluating a given query.

Query plans. A *query plan* P is a sequence of *evaluation steps*. Each evaluation step s is a triple $s = \langle Join_s, Rec_s, Dec_s \rangle$ where $Join_s$ is a set of query nodes to be matched in step s through joins with the element table, and Rec_s and Dec_s are sets of binary constraints to be reconstructed and decided in step s , respectively. Figure 8.11 on the next page depicts two sample query plans involving joins, reconstruction and decision. As illustrated in Figure 8.11 *a*., the plan P^Q in *b*. matches the query Q from Figure 2.2 *b*. on page 10 in three steps. In the first step, s_1^Q , $Join_1 = \{q_4\}$ means that matches to query node q_4 are retrieved by joining the schema-matching result from phase 1 (Figure 8.4 *a*. on page 102) with the element table (Figure 6.1 on page 82). Given the BIRD weights in the intermediate result, the ancestors of these elements that match q_1 can be reconstructed on the fly, without the need for another element-table join. This is indicated by the red edge in Figure 8.11 *a*., and specified as $Rec_1 = \{Parent_1^*(q_4, q_1)\}$ in Figure 8.11 *b*. By contrast, $Dec_1 = \{\}$ since the two *Child* constraints in Q cannot be reconstructed with BIRD. The intermediate result table produced by step s_1^Q is shown in Figure 8.4 *b*. on page 102.

In the next step, s_2^Q , this table is joined with the element table to obtain the matches to q_2 ($Join_2 = \{q_2\}$). Now the $Child(q_1, q_2)$ edge in Q can be decided ($Dec_2 = \{Child(q_1, q_2)\}$), which yields the intermediate result in Figure 8.4 *c*. on page 102. Similarly, step s_3^Q joins q_3 and decides $Child_1^*(q_1, q_3)$. This produces the result table in Figure 8.4 *d*. which contains all matchings to Q (since all query constraints have been matched in the three evaluation steps). The query plan P^Q thus specifies a way to answer Q using three look-ups in the element table, compared to four look-ups needed without structural summaries (e.g., see Sections 7.3.1 and 7.3.2). The second query in Figure 8.11 *c*. illustrates a case where the benefit of the RCADG is even greater. As shown in the query plan in Figure 8.11 *d*., this five-node query can be matched



a. the query Q from Figure 2.2 b. on page 10



c. the query from Figure 8.5 c. on page 104

plan $P^Q = \langle s_1^Q, s_2^Q, s_3^Q \rangle$

step $s_1^Q = \langle Join_1, Rec_1, Dec_1 \rangle$
 $Join_1 = \{q_4\}$
 $Rec_1 = \{Parent_1^*(q_4, q_1)\}$
 $Dec_1 = \{\}$

step $s_2^Q = \langle Join_2, Rec_2, Dec_2 \rangle$
 $Join_2 = \{q_2\}$
 $Rec_2 = \{\}$
 $Dec_2 = \{Child(q_1, q_2)\}$

step $s_3^Q = \langle Join_3, Rec_3, Dec_3 \rangle$
 $Join_3 = \{q_3\}$
 $Rec_3 = \{\}$
 $Dec_3 = \{Child_1^*(q_1, q_3)\}$

b. plan for the query Q in a.

plan $P = \langle s_1, s_2 \rangle$

step $s_1 = \langle Join_1, Rec_1, Dec_1 \rangle$
 $Join_1 = \{q_5\}$
 $Rec_1 = \{$
 $Parent(q_5, q_4),$
 $Parent_1^*(q_4, q_1),$
 $PrevSib(q_5, q_6)$
 $\}$
 $Dec_1 = \{Parent(q_6, q_4)\}$

step $s_2 = \langle Join_2, Rec_2, Dec_2 \rangle$
 $Join_2 = \{q_3\}$
 $Rec_2 = \{\}$
 $Dec_2 = \{Parent_2^*(q_3, q_1)\}$

d. plan for the query in c.

Figure 8.11: RCADG query plans for the queries in Figure 2.2 b. on page 10 (a.–b.) and Figure 8.5 c. on page 104 (c.–d.). In a. and c., blue colour indicates query nodes matched through joins with the element table, red colour stands for nodes and edges matched through reconstruction, and green colour highlights query edges matched through decision.

in two steps with only two look-ups in the element table. Compared to the schema-less approaches mentioned above, BIRD reconstruction saves three index look-ups and possibly much I/O in this example. The positive effect of such query planning on the runtime performance is also reflected in the experimental results (see Section 8.5).

Planning algorithm. The two query plans in Figure 8.11 are produced by the procedure *createPlan* that is called in line 9 of Algorithm 8.1 on page 101. The pseudocode for *createPlan* is given in Algorithm 8.2 on the following page. The procedure accepts as input a set M_v of query nodes to be matched and another set M_c of query edges between these nodes. When evaluating from scratch, *createPlan* is called for all query nodes and edges in a given query Q , i.e., $M_v = Q_v$ and $M_c = Q_c$.³ First an empty query plan P is created (line 7). The set K_v initialized in line 9 keeps track of all query nodes that are matched during the execution of the plan, either by element-table joins or by reconstruction.

The outermost **for** loop (lines 12–48) of *createPlan* examines each of the given query nodes in M_v to determine which ones shall be matched through a join with the element table and which ones can then be reconstructed on the fly. For each node $q \in M_v$ to be joined, a new evaluation step s is added to P and $Join_s$ is initialized with q (line 17). Then the **for** loop in lines 20–36 examines nodes q' and edges c in a breadth-first traversal of the subgraph of Q that is reachable from q , using a queue M'_v . Inbound edges $R(q', q)$ may be replaced with their equivalent outbound inverse $(R^{-1})(q, q')$, as during query rewriting. Edges that are

³Different parameters may be given to *createPlan* when a query cache is used, see Chapter 10.

```

1 // createPlan: RCADG query planning
2 //  $\rightarrow M_v$ : the set of query nodes to be matched
3 //  $\rightarrow M_c$ : the set of query edges to be matched
4 //  $\leftarrow$  the query plan to be devised
5 procedure createPlan ( $M_v, M_c$ )
6 // create an empty query plan
7  $P :=$  a new empty query plan
8 // remember "known" nodes that have been matched meanwhile ( $K_v \subset M_v$ )
9  $K_v := \emptyset$ 
10 // for all nodes in  $M_v$ , reconstruct as many edges in  $M_c$  as possible
11 // start with nodes that are selective and support much reconstruction
12 for all  $q \in M_v$  in a suitable order do
13 // join  $q$  in a new step  $s$  unless it is already known
14 if  $q \in K_v$  then next in loop end if
15  $K_v := K_v \cup \{q\}$ 
16  $s :=$  a new empty evaluation step
 $\rightarrow$  17  $Join_s := \{q\}$ 
18  $P := P \cup \{s\}$ 
19 // starting from  $q$ , follow all reconstructible edges in  $M_c$ 
 $\uparrow$  20 for  $M'_v := \{q\}$  while  $M'_v \neq \emptyset$  do
21  $q' :=$  call removeFirst ( $M'_v$ )
22 for all edges  $c$  leaving  $q'$  do
23 if  $c \notin M_c$  then next in loop end if
24  $q_t :=$  the target node of  $c$ 
25 if  $q_t$  was reconstructed from  $q$  then
26 next in loop
27 end if
28 if  $q_t \in K_v$  or  $q_t \notin M_v$  then
 $\rightarrow$  29  $Dec_s := Dec_s \cup \{c\}$ 
30 else if  $c$  is reconstructible then
 $\rightarrow$  31  $Rec_s := Rec_s \cup \{c\}$ 
32  $M'_v := M'_v \cup \{q_t\}$ 
33  $K_v := K_v \cup \{q_t\}$ 
34 end if
35 end for
 $\downarrow$  36 end for
37 // use extra steps to match keywords of nodes reconstructed in step  $s$ 
 $\uparrow$  38 if  $Rec_s \neq \emptyset$  then
39 for all  $c \in Rec_s$  do
40  $q_t :=$  the target node of  $c$ 
41 if  $q_t$  has keyword constraints then
42  $s :=$  a new empty evaluation step
43  $Join_s := Join_s \cup \{q_t\}$ 
44  $P := P \cup \{s\}$ 
45 end if
46 end for
 $\downarrow$  47 end if
48 end for
49 // return the query plan
50 return  $P$ 
51 end procedure

```

Algorithm 8.2: Query planning with the RCADG. The input consists of two sets M_v, M_c of nodes and edges in a query to be matched. The output is a suitable query plan P for evaluating the given query constraints.

not in the given set M_c of edges to be matched are ignored (line 23; this does not happen when evaluating Q from scratch). Edges to nodes that have been matched before are decided (line 29) since in this case the current intermediate result contains already pairs of matches to both endpoints of the edge. Edges to yet unmatched nodes in M_v that can be reconstructed are added to Rec_s (line 31). Each target node q_t of such an edge is appended to the queue M'_v , so that in the end all chains of reconstructible edges from q to nodes in M_v are reconstructed in the current step s . The check in line 25 makes sure that each edge is matched only once. Non-reconstructible edges to nodes in M_v are ignored at this stage. Note, however, that they will be either decided or reconstructed later, when their target node is visited as node q in a subsequent iteration of the outermost **for** loop. The time and space needed for query planning with *createPlan* is linear in the size of the given set M_c of query edges.

For instance, reconsider the query shown in Figure 8.11 *c.* on page 109. In the first iteration of the outermost loop in *createPlan*, a new step s_1 is created for $q = q_5$, therefore $Join_1 = \{q_5\}$ (line 17 in Algorithm 8.2). In lines 20–36, the nodes q_6 , q_4 and q_1 in M_v are visited and the corresponding edges are added to Rec_1 (line 31). Only the edge from q_6 to q_4 is decided because these two nodes have just been matched through reconstruction (line 29). The edge $Parent_2^*(q_3, q_1)$ cannot be reconstructed from q_1 and is therefore ignored in the first step. However, since $q_2 \in M_v$, one of the subsequent iterations will start from q_2 and match this edge—through decision, because q_1 has already been matched in step s_1 (line 29). All other nodes in M_v (q_1 , q_4 and q_6 in Figure 8.11 *c.*) are immediately skipped in the outer loop (line 14).

As shown in the next subsection, keyword constraints are easily handled when matching a query node through a join with the element table. However, a little extra treatment is needed when the matches to a query node with keyword constraints are obtained through reconstruction rather than an element-table join. For example, if the query node q_4 in Figure 8.11 *c.* had a keyword constraint $Contains_k(q_4)$ for some keyword $k \in K$, then the plan in Figure 8.11 *d.* would be incorrect because matches to q_4 are never looked up in the element table to see whether they really contain an occurrence of k . Therefore in lines 38–47 of Algorithm 8.2, all query nodes matched through reconstruction in the current evaluation step s are examined once again. Those with keyword constraints are scheduled for an extra join with the element table in subsequent steps (line 43). In the above example, we would have to add a third step s_3 to the query plan in Figure 8.11 *d.* with $Join_3 = \{q_4\}$, $Rec_3 = \{\}$ and $Dec_3 = \{\}$. Note that although the benefit of reconstruction on the runtime performance is reduced in such cases, this plan is still preferable to one where q_4 is matched through a join right from the start, for two reasons. First, reconstructing q_4 allows to reconstruct q_1 in the first step, too, which saves one join (since q_1 does not have a keyword constraint). Second, looking up q_4 in the element table requires only an equality condition on the *eid* column because the matches to be checked are already known from the previous step. By contrast, deciding the *Parent* edge from q_5 to q_4 involves a range condition, which is less efficient.

Planning strategies. Note that the number of possible reconstructions often depends crucially on the choice of the next query node to be joined. For instance, an alternative query plan for the query in Figure 8.11 *c.* that matches q_1 through a join in the first step cannot reconstruct the $Parent_1^*(q_4, q_1)$ edge and therefore needs more than two element-table joins to answer the query. One method to reduce the number of joins in a query plan is to sort the given set M_v of query nodes in such a way that nodes which allow more edges to be reconstructed are processed first. To this end we compute for each node $q \in M_v$ its *reconstruction count*, i.e., the total length of all reconstructible paths leaving q (not shown Algorithm 8.2). The nodes in M_v are then sorted in the order of descending reconstruction counts before the actual planning begins. As a consequence, the outermost **for** loop in Algorithm 8.2 on the facing page (lines 12–48) collects reconstructible edges in a greedy manner, which is perhaps not optimal but certainly an efficient and quite effective strategy (see the experiments Section 8.5). Although the reconstruction counts could probably be computed in time linear in the number of query edges in M_c , even a simple repeated traversal of the query graph with distinct start nodes runs sufficiently fast, despite its quadratic time complexity in M_c .

However, there are more possible planning goals besides minimizing the number of element-table joins. Most importantly, the size of intermediate results to be joined can be reduced by matching first those query nodes that have selective constraints such as, e.g., infrequent keywords or tag paths or rare combinations of both. To take advantage of the most selective query constraints, statistical information about the distribution of tag paths and keywords in the documents is needed. Section 8.2 has introduced the optional *elts* and *keys*

columns in the path table as basic selectivity estimates. These can easily be integrated with the planning algorithm, similar to the reconstruction count above. More sophisticated planning could also use cardinality estimates for combined structure and keyword constraints. Yet this is outside the scope of this work.

Of course, when pursuing multiple planning goals at once conflicts may arise, e.g., when highly selective query nodes have a low reconstruction count and vice versa. The following preliminary solution to this problem reconciles both reconstruction and selectivity optimization while giving keyword constraints the priority. Currently we simply distinguish query nodes in M_v with any keyword constraint from those without, and sort the two resulting subsets of M_v separately in the order of descending reconstruction counts, as described above. Any node with a keyword constraint is thus visited before all nodes without keyword constraints as node q in the outermost **for** loop of *createPlan* (lines 12–48 in Algorithm 8.2). This suffices to produce query plans like the ones illustrated in Figures 8.11 *a.* and *c.*, for example.

Intermediate results. The procedure *createPlan* in Algorithm 8.2 on page 110 produces query plans with only a single element-table join per evaluation step. In other words, in any given query plan P we have $\forall s_i \in P : |Join_{s_i}| = 1$. However, the planning algorithm is easily adapted to allow for multiple joins in the same step, which reduces the number of steps and hence of intermediate result tables to be created. Since subsequent intermediate results for the same query usually overlap considerably (e.g., consider the tables in Figures 8.4 *b.–d.* on page 102), permitting multiple joins per evaluation step helps to save storage in the RDBS. However, intermediate results play an important role for the incremental query evaluation based on cached queries, as explained in Chapter 10. In a caching scenario, it makes sense to evaluate queries in small steps with many intermediate results, because this allows to compare and reuse the answers to previous queries at a fine granularity.

Both approaches have been successfully applied in the two scenarios. The experiments with query evaluation from scratch in Section 8.5 are based on a slightly modified version of *createPlan* that allows multiple joins in each evaluation step. By contrast, for the incremental evaluation in Chapter 10 we will assume one join per step, which increases the effectiveness of the query cache at the expense of a higher space consumption. For simplicity, the following description of the query matching on the document level adopts the single-join approach, too.

8.4.5 Document-Level Matching

Once a query plan P has been devised for the query Q to be answered, the evaluation steps $s_i \in P$ ($i \geq 1$) are translated and matched on the document level one by one. Each step s_i produces a new intermediate result table Q_{-s_i} by joining the previous result table $Q_{-s_{i-1}}$ with the element table (recall from Section 8.4.2 that Q_{-s_0} denotes the schema-matching result computed during phase 1). Figure 8.4 on page 102 depicts the sequence of result tables produced during the evaluation of query Q in Figure 2.2 *b.* on page 10. The document-level matching discussed here comprises the three tables in Figures 8.4 *b.–d.*, which correspond directly to the three evaluation steps in the query plan P^Q shown in Figure 8.11 *b.* on page 109.

The first step in P^Q , s_1^Q , adds matches to the query nodes q_4 and q_1 (columns p_4 and p_1 in Figure 8.4 *b.*, respectively) by joining q_4 and reconstructing the edge $Parent_1^*(q_4, q_1)$ in Q . Note that there are two distinct pairs matching the two query nodes ($q_4 = 26, q_1 = 18$ versus $q_4 = 34, q_1 = 27$). In the document tree D in Figure 4.3 *a.* on page 46, these are the pairs of **person** and **sex** nodes in the subtrees a_2 and a_3 of D , respectively. As a consequence, the result table Q_{-s_1} contains two rows each representing a distinct partial matching of the query Q (partial, because nodes q_2 and q_3 are ignored at this stage). The second step of the query plan, s_2^Q , matches the query node q_2 through another join with the element table and decides the $Child(q_1, q_2)$ edge. Since there is a corresponding name **child** below the **person** element in both a_2 and a_3 , each of the two partial matchings can be expanded with a match to q_2 (the p_2 values 21 and 30 in Figure 8.4 *c.*, respectively). The last step, s_3^Q , joins q_3 and decides the $Child_1^*(q_1, q_3)$ edge. While the **person** node in a_2 has a matching **edu** descendant that is added to the result table (element 25 in the last column in Figure 8.4 *d.*), there is no such descendant in a_3 . Hence the second partial matching from the previous result Q_{-s_2} (last row in Figure 8.4 *c.*) is discarded. Thus at the end of phase 2, the result table Q_{-s_3} in Figure 8.4 *d.* contains all possible matchings to the entire query Q (in this case, a single document-level match for the only schema hit χ^Q retrieved in phase 1).

```

CREATE
  TABLE Q_s1 AS -- create new result table
SELECT
  sid, p1, p2, p3, p4, w1, w2, w3, w4, -- copy schema hits
  ET4.eid AS e4, -- add matches to q4
  ET4.eid - (ET4.eid % w1) AS e1 -- reconstruct Parent*(q4,q1)
FROM
  Q_s0, ElementTable ET4 -- join previous result table with element table
WHERE
  ET4.pid = p4 AND -- match unary constraints on q4
  ET4.key = 'female'

```

a. step s_1 of document-level matching (phase 2)

```

CREATE
  TABLE Q_s2 AS -- create new result table
SELECT
  sid, p1, p2, p3, p4, w1, w2, w3, w4, -- copy schema hits
  e1, e4, -- copy matches from previous steps
  ET2.eid AS e2 -- add matches to q2
FROM
  Q_s1, ElementTable ET2 -- join previous result table with element table
WHERE
  ET2.pid = p2 AND -- match unary constraints on q2
  ET2.key = ' ' AND
  ET2.eid > e1 AND ET2.eid < e1 + w1 -- decide Child*(q1,q2)

```

b. step s_2 of document-level matching (phase 2)

```

CREATE
  TABLE Q_s3 AS -- create new result table
SELECT
  sid, p1, p2, p3, p4, w1, w2, w3, w4, -- copy schema hits
  e1, e2, e4, -- copy matches from previous steps
  ET3.eid AS e3 -- add matches to q3
FROM
  Q_s2, ElementTable ET3 -- join previous result table with element table
WHERE
  ET3.pid = p3 AND -- match unary constraints on q3
  ET3.key = ' ' AND
  ET3.eid > e1 AND ET3.eid < e1 + w1 -- decide Child*(q1,q3)

```

c. step s_3 of document-level matching (phase 2)

Figure 8.12: SQL code for evaluating the query Q in Figure 2.2*b*. on page 10 on the document level (phase 2). The document-level matching of Q is divided into three steps, s_1 – s_3 , according to the query plan in Figure 8.11*b*. on page 109. As before, blue, red and green colour highlights code related to element-table joins, reconstruction and decision, respectively. *a.* Step s_1 : The query node q_4 is matched through a join of the element table (see Figure 6.1 on page 82) with the schema-matching result (see Figure 8.4*a*. on page 102). Matches to q_1 are obtained by reconstructing the $Parent^*(q_4, q_1)$ edge. This produces the result table in Figure 8.4*b*. on page 102. *b.* Step s_2 : The query node q_2 is matched through a join of the element table with the intermediate result from step s_1 . The $Child(q_1, q_2)$ constraint is adapted to $Child^*(q_1, q_2)$ (see Figure 8.14) and then decided. The result table is shown in Figure 8.4*c*. on page 102. *c.* Step s_3 : The query node q_3 is matched through a join of the element table with the intermediate result from step s_2 , and the $Child^*(q_1, q_3)$ constraint is decided. This produces the result table in Figure 8.4*d*. on page 102.

$$\frac{R(q_0) \quad R \in \{\text{Contains}_k, \text{Governs}_k\}}{R(v_0)} \text{ (DA}_0\text{)}$$

Figure 8.13: RCADG document adaptation rules for unary query constraints ($q_0 \in Q_v$; $v_0 \in V$; v_l denotes a document node matching q_l).

$$\begin{array}{cc} \frac{R_i^j(q_0, q_1) \quad R \in \{\text{Parent}, \text{Child}\}}{R_0^*(v_0, v_1)} \text{ (DA}_1\text{)} & \frac{R_i^j(q_0, q_1) \quad R \in \mathcal{R}_2 \setminus \{\text{Parent}, \text{Child}\}}{R_i^j(v_0, v_1)} \text{ (DA}_3\text{)} \\ \frac{R_i^*(q_0, q_1) \quad R \in \{\text{Parent}, \text{Child}\}}{R_0^*(v_0, v_1)} \text{ (DA}_2\text{)} & \frac{R_i^*(q_0, q_1) \quad R \in \mathcal{R}_2 \setminus \{\text{Parent}, \text{Child}\}}{R_i^*(v_0, v_1)} \text{ (DA}_4\text{)} \end{array}$$

Figure 8.14: RCADG document adaptation rules for binary query constraints ($q_0, q_1 \in Q_v$; $v_0, v_1 \in V$; $i < j \in \mathbb{N}$; v_l denotes a document node matching q_l).

Generating SQL code. The SQL code for creating the three intermediate result tables Q_{-s_1} , Q_{-s_2} and Q_{-s_3} in Figures 8.4 *b–d*. is given in Figure 8.12 on the preceding page. Three SQL statements are generated and executed by the procedure *matchDocLevel* which is called once for each evaluation step (see line 11 in Algorithm 8.1 on page 101). The rest of this subsection explains how *matchDocLevel* expresses query constraints as SQL statements that are then handed over to the RDBS.

For instance, consider the first join with the element table in step s_1 , expressed by the SQL statement in Figure 8.12 *a*. on the preceding page. Some parts of the code are either fixed or like a template to be filled in with the query node in $Join_1$, whereas others are inferred from the query constraints in s_1 using a set of document-level rules (see below). A fixed code block is the projection of schema-level information (first line of the SELECT clause): it simply copies the schema hits retrieved in phase 1, along with their BIRD weights which may be needed for reconstruction and decision (see Section 8.3). Since $Join_1 = \{q_4\}$, the schema matching result Q_{-s_0} is joined with an instance ET_4 of the element table (FROM clause), and the matches to q_4 in the *eid* column of ET_4 are added to the new result table Q_{-s_1} (SELECT clause). The result table names to be used in the CREATE and FROM parts follow directly from the current evaluation step (in this case, s_1). Furthermore, a first join condition selects those tuples in the element table with a tag path matching q_4 (first row in the WHERE clause). These code templates apply analogously to $Join = \{q_2\}$ in Figure 8.12 *b*. and $Join = \{q_3\}$ in Figure 8.12 *c*.

The remaining code fragments in Figure 8.12 are derived for each step s_i from unary keyword constraints on the join node in $Join_i$ and from the binary constraints in Rec_i and Dec_i . As during phase 1, two distinct sets of rules specify how to translate these query constraints to SQL: *document adaptation rules* select the constraints to be matched on the document level whereas *document matching rules* generate appropriate selection and projection expressions for these constraints.

Adapting query constraints to the document level. The document adaptation rules for unary and binary constraints are given in Figures 8.13 and 8.14, respectively. The rule DA_0 in Figure 8.13 states that keyword constraints are the only unary constraints to be matched on the document level (recall from Section 8.4.2 that tag, type, root and level constraints are handled during phase 1). The adaptation rules DA_1 and DA_2 on the left-hand side in Figure 8.14 suppress proximity bounds of *Parent* and *Child* constraints, because they have already been translated to level predicates during schema matching (see rules SM_{Child^i} and SM_{Child^j} in Figure 8.10 on page 106). Binary constraints other than *Parent* and *Child* are matched unmodified on the document level, as specified by rules DA_3 and DA_4 on the right-hand side of Figure 8.14.

Matching binary query constraints on the document level. After applying the document adaptation rules, the resulting unary and binary constraints on elements are translated to join conditions and projection clauses using a set of document matching rules. These rules handle exactly the *D*-constraints listed in Definition 2.8 on page 12. We first discuss the matching of binary *D*-constraints with BIRD. Keyword matching is explained below.

$$\frac{\text{Contains}_{k_0}(v_0) \quad \theta \quad \cdots \quad \theta \quad \text{Contains}_{k_m}(v_0) \quad k_0, \dots, k_m \in K}{v_0.\text{key} = k_0 \quad \theta \quad \cdots \quad \theta \quad v_0.\text{key} = k_m} \quad (\text{DM}_{\text{Contains}_{k_i}})$$

$$\frac{\text{Governs}_{k_0}(v_0) \quad \vee \quad \cdots \quad \vee \quad \text{Governs}_{k_m}(v_0) \quad k_0, \dots, k_m \in K}{\exists w \in V : \text{Child}_0^*(v_0, w) \quad \wedge \quad (\text{Contains}_{k_0}(w) \vee \cdots \vee \text{Contains}_{k_m}(w))} \quad (\text{DM}_{\text{Governs}_{k_i}})$$

Figure 8.15: RCADG document matching rules for unary query constraints ($q_0 \in Q_v$; $v_0 \in V$; v_l denotes a document node matching q_l). In the first rule for keyword containment, $\theta = \wedge$ or $\theta = \vee$ depending on whether the keyword constraints on q_0 are marked as conjunctive or disjunctive, respectively. The second rule translates a disjunction of government constraints into a single condition. By contrast, given a conjunction of government constraints the rule applies to each constraint separately (treating it as a singleton disjunction).

The binary constraints to be matched on the document level include *Child*, *NextSib*, *NextElt*, *Following* and their inverses as well as *Sibling* and *Self*. All of them can be decided using the BIRD document matching rules in Figure 8.2 on page 100. Alternatively, *Parent*, *PrevSib*, *NextSib* and *Self* constraints may be reconstructed with the rules in Figure 8.3 on page 100. However, no ambiguities arise since the query plan for a given query specifies which constraints to decide and which to reconstruct, as described above.

For instance, reconsider the query plan P^Q in Figure 8.11 b. on page 109 and the corresponding SQL statements in Figure 8.12 on page 113. In the first step, s_1 , the constraint $\text{Parent}_1^*(q_4, q_1)$ shall be matched through reconstruction. The adaptation rule DA_2 in Figure 8.14 on the facing page replaces query nodes with elements and modifies the lower proximity bound in order to prepare the application of a suitable document matching rule. This yields the adapted constraint $\text{Parent}_0^*(v_4, v_1)$ where v_4 stands for any match to the query node q_4 , and likewise for v_1 . The unique reconstruction rule that is relevant to this constraint is $\text{DM}_{\text{Parent}_0^*}^{\text{Rec}}$ in Figure 8.3 on page 100. Applied to the pair $\langle v_4, v_1 \rangle$ (which is called $\langle v_0, v_1 \rangle$ in Figure 8.3), the lower part of the rule states that for any match v_4 to q_4 , the element label of the corresponding ancestor matching q_1 can be computed as $v_1 = v_4 - (v_4 \bmod \pi(v_1).\text{weight})$ where $\pi(v_1).\text{weight}$ is the BIRD weight of the tag path of v_1 . Now compare this to the SQL code for s_1 in Figure 8.12 a. on page 113. Here the reconstruction formula for the $\text{Parent}_1^*(q_4, q_1)$ edge in Q is expressed as the projection clause that is highlighted red. The BIRD weight $\pi(v_1).\text{weight}$ of v_1 is available in the column $w1$ of the result table $Q_{\rightarrow s0}$ from phase 1 (see Section 8.4.2). The matches to q_4 are taken from the *eid* column of the instance $ET4$ of the element table, hence v_4 becomes $ET4.\text{eid}$. Finally, the matches to q_1 to be reconstructed are given the alias $e1$. Thus $v_1 = v_4 - (v_4 \bmod \pi(v_1).\text{weight})$ translates to $ET4.\text{eid} - (ET4.\text{eid} \% w1)$ AS $e1$ in SQL.

As in the example above, the lower part of any reconstruction rule is added to the SELECT part of the SQL statement to be created. More complex rules like $\text{DM}_{\text{PrevSib}_i^*}^{\text{Rec}}$ or $\text{DM}_{\text{NextSib}_i^*}^{\text{Rec}}$ in Figure 8.3 also have preconditions concerning the parent v_2 of the element v_0 whose sibling v_1 shall be reconstructed. Note that when applying such a rule to a sibling constraint on v_0 and v_1 , matches to v_2 are guaranteed to be already known because (1) the schema-level rewriting of the query ensures that v_0 and v_2 are connected with a *Parent* edge and (2) this parent edge is reconstructed no later than the sibling constraint on v_0 , according to the query planning algorithm in Section 8.4.4. The precondition on v_0 and v_2 in the upper part of rules $\text{DM}_{\text{PrevSib}_i^*}^{\text{Rec}}$ and $\text{DM}_{\text{NextSib}_i^*}^{\text{Rec}}$ is added to the WHERE clause of the SQL statement to be created. This way the node label of v_1 is computed for any tuple containing elements v_0 and v_2 that satisfy the precondition. Other tuples are silently dropped.

The following steps in the query plan P^Q in Figure 8.11 b. on page 109 involve the decision of binary constraints. Here the document matching rules in Figure 8.2 on page 100 are applied to create suitable join predicates for the SQL statement. For instance, in step s_2 the constraint $\text{Child}_1^*(q_1, q_2)$ is decided. As described before, it is adapted to $\text{Child}_0^*(v_1, v_2)$ by rule DA_2 in Figure 8.8 on page 106. The first decision rule in Figure 8.2, $\text{DM}_{\text{Child}_0^*}^{\text{Dec}}$, produces the join predicate $v_2.\text{eid} \geq v_1.\text{eid} \wedge v_2.\text{eid} < v_1.\text{eid} + \pi(v_1).\text{weight}$ which is translated to the SQL expression $ET2.\text{eid} > e1$ AND $ET2.\text{eid} < e1 + w1$, as shown in the WHERE part of Figure 8.12 b. on page 113 (highlighted green). The $\text{Child}_1^*(q_1, q_3)$ edge in step s_3 is treated analogously. As in the reconstruction case, some decision rules like $\text{DM}_{\text{NextSib}_1^*}^{\text{Dec}}$ and $\text{DM}_{\text{PrevSib}_1^*}^{\text{Dec}}$ in Figure 8.2 assume parent matches to be known, which is safe with the query rewriting and planning introduced above.

Matching keyword constraints on the document level. According to Definition 2.8 on page 12, the only unary constraints to be matched on the document level are keyword constraints. Like binary D -constraints, they are first adapted to the document level (using rule DA_0 in Figure 8.13 on page 114) and then translated by applying document-matching rules ($DM_{Contains_k}$ and $DM_{Governs_k}$ in Figure 8.15 on the previous page). For instance, consider again the query plan P^Q for the query Q in Figure 8.11 *a.* on page 109. The query node q_4 to be matched through an element-table join in the first step s_1^Q has a keyword constraint $Contains^{“female”}(q_4)$. According to the adaptation rule DA_0 this constraint must be enforced on the elements matching q_4 , which are contained in the instance ET_4 of the element table in Figure 8.12 *a.* on page 113. The matching rule $DM_{Contains_k}$ in Figure 8.15 specifies the appropriate join predicate for a conjunction or disjunction of keywords $k_0 \theta \dots \theta k_m$ ($\theta \in \{\wedge, \vee\}$). The resulting predicate for the singleton keyword “female” in the keyword column of ET_4 is shown in the last line of code in Figure 8.12 *a.*: $ET_4.key = \text{‘female’}$. The same mechanism applies to the join nodes of the subsequent steps s_2 and s_3 (q_2 and q_3 , respectively). Since these do not have query keywords, a containment constraint for the empty keyword is assumed by default. Recall from Section 6.2 that an extra row is added to the element table for each element and the empty keyword. This way matches to query nodes without keyword constraints can be looked up efficiently with a precise equality predicate like $ET_2.key = \text{‘ ’}$ or $ET_3.key = \text{‘ ’}$ in Figures 8.12 *b.* and 8.12 *c.*, respectively.

The join predicate for government constraints is a little more involved, due to the fact that the element table only indexes contained keywords explicitly. The data model in Section 2.1 defines the keywords governed by an element v as those $k \in K$ that are contained either in v or in any of its descendants. This definition also captures conjunctions and disjunctions of government constraints, as follows. An element v governs a disjunction $k_0 \vee \dots \vee k_m$ of keywords if either v or any of its descendants contains at least one of these keywords. The matching rule $DA_{Governs_k}$ in Figure 8.15 on the preceding page reflects exactly this definition, replacing the government constraints on a given element v_0 with containment constraints on another element w that is either v_0 itself or one of its descendants (i.e., $Child_0^*(v_0, w)$). Similarly, v governs a conjunction $k_0 \wedge \dots \wedge k_m$ of keywords if each of these keywords is contained in any node in the subtree rooted in v . Note, however, that not all keywords are necessarily contained in the same node in the subtree. Simply substituting “ \wedge ” to “ \vee ” in $DA_{Governs_k}$ would therefore be too restrictive. Instead the rule $DA_{Governs_k}$ is applied separately to each government constraint $Governs_{k_i}(v)$ ($0 \leq i \leq m$), so that the keyword disjunction in the upper part of the rule consists only of k_i . This way distinct descendants of v are accepted for distinct keywords k_i .

As an example, consider a variant of the query Q in Figure 8.11 *a.* where q_4 has two government constraints, $Governs^{“female”}(q_4)$ and $Governs^{“PhD”}(q_4)$, instead of the containment constraint. If the two government constraints are marked as disjunctive, then the join predicate $ET_4.key = \text{‘female’}$ in Figure 8.12 *a.* is replaced with the code listed in Figure 8.16 *a.* on the next page. Here a single subquery checks whether any descendant of a match to q_4 contains either “female” or “PhD”. By contrast, a conjunction of independent subqueries for all keywords is needed when the two government constraints are marked as conjunctive (see Figure 8.16 *b.*).

8.4.6 Computing the Final Query Result

The last intermediate result table created during phase 2 contains all *query matchings* as defined before (see Definition 2.3 on page 10). Recall from Section 2.2 that the final *query answer* $ans(Q)$ is obtained by restricting these matchings to the set Q_r of result nodes given as part of the query specification. Also, the result tables contain the schema hits and weights corresponding to each matching, which are not part of the query answer. Therefore a final query is needed to extract all relevant data from the last intermediate result table and return it as the final result $ans(Q)$ to the user. This is done by the procedure *createResult* called in line 14 of Algorithm 8.1 on page 101.

The SQL code for creating the final result of the query Q in Figure 8.11 *a.* on page 109 is given in Figure 8.17 on the next page. It consists of a single statement that simply projects the last intermediate result table, Q_{s3} in Figure 8.4 *d.* on page 102, onto those columns e_i which contain the matches to result query nodes q_i . In the example we assume that all nodes in Q are results nodes, i.e., $Q_r = Q_v$. Note the use of the keyword `DISTINCT` to remove duplicate results from the final output. In fact, explicit duplicate


```

... -- CREATE, SELECT, FROM as before
WHERE
  ET4.pid = p4 AND -- select matches v to q4
  ET4.key = ' ' AND
  EXISTS ( -- match Governs disjunction using a single descendant w of v
    SELECT eid
    FROM ElementTable ET4desc
    WHERE
      ET4desc.eid >= ET4.eid AND ET4desc.eid < ET4.eid + w4 AND -- match Child0*(v,w)
      (ET4desc.key = 'female' OR ET4desc.key = 'PhD') -- match Contains disjunction on w
  )

```

a. disjunction of government constraints on node q_4 in query Q

```

... -- CREATE, SELECT, FROM as before
WHERE
  ET4.pid = p4 AND -- select matches v to q4
  ET4.key = ' ' AND
  EXISTS ( -- match Governs"female"(q4) using a descendant w of v
    SELECT eid
    FROM ElementTable ET4desc
    WHERE
      ET4desc.eid >= ET4.eid AND ET4desc.eid < ET4.eid + w4 AND -- match Child0*(v,w)
      ET4desc.key = 'female' -- match Contains"female"(w)
  ) AND
  EXISTS ( -- match Governs"PhD"(q4) using another descendant w of v
    SELECT eid
    FROM ElementTable ET4desc
    WHERE
      ET4desc.eid >= ET4.eid AND ET4desc.eid < ET4.eid + w4 AND -- match Child0*(v,w)
      ET4desc.key = 'PhD' -- match Contains"PhD"(w)
  )

```

b. conjunction of government constraints on node q_4 in query Q

Figure 8.16: SQL code for matching keyword government constraints. The code is generated for a variant of query Q in Figure 8.11 *a.* on page 109 where the containment constraint $Contains^{“female”}(q_4)$ on node q_4 has been replaced with two government constraints $Governs^{“female”}(q_4)$ and $Governs^{“PhD”}(q_4)$. The two government constraints are either disjunctive (*a.*) or conjunctive (*b.*). Each of the two statements is meant to replace the code for the first evaluation step s_1 in the query plan P^Q in Figure 8.11 *b.* on page 109. The CREATE, SELECT and FROM clauses remain unchanged (see Figure 8.12 *a.* on page 113). Only the WHERE is modified according to the document matching rule $DA_{Governs_k}$ in Figure 8.15 on page 115, as follows. Let v be a match to the query node q_4 . *a.* A disjunction of the two government constraints is translated into a single subquery selecting any descendant w of v that contains either keyword. *b.* A conjunction of the two government constraints translates to a conjunction of two separate subqueries. Each subquery independently selects a descendant w of v that contains a specific keyword.

```

SELECT
  DISTINCT e1, e2, e3, e4 -- copy matches to result nodes
FROM
  Q_s3 -- retrieve answer from the last intermediate result
ORDER BY
  e1, e2, e3, e4 -- order result as needed

```

Figure 8.17: SQL code for computing the final result of the query Q in Figure 8.11 *a.* on page 109. The last intermediate result from phase 2 (see Figure 8.4 *d.* on page 102) is projected onto matches to the result nodes (in this case, all query nodes). This produces the query answer shown in Figure 8.4 *e.* on page 102.

elimination is only needed when some match columns are dropped, i.e., when $Q_r \subsetneq Q_v$. The ORDER BY clause serves to return the query answer in some specific order. In this case, it is sorted so that all matches to the query node q_1 appear in document order. (Tatarinov et al. [2002] mention different output modes to be applied analogously.)

The output of the SQL query in Figure 8.17 is shown in Figure 8.4 *e.* on page 102. Here $ans(Q)$ consists of the tuple $\langle 18, 21, 25, 26 \rangle$ of node labels that denotes exactly the document subtree depicted in Figure 2.1 *e.* on page 8 (the corresponding node labels are given in Figure 4.3 *a.* on page 46). Of course a different result presentation may be chosen for the user. For instance, given the original XML representation of the documents and a mapping from node labels to the corresponding byte offsets in the XML code, the query answer could be presented as XML fragments (possibly rendered using stylesheets). Alternatively, XML code might be generated on the fly. However, these presentation details are beyond the scope of this work.

8.5 Experimental Evaluation

To evaluate the practical use of XML indexing with the RCADG, we created path and element tables for different document collections in an RDBS and implemented the evaluation procedure *evaluateQuery* (see Algorithm 8.1 on page 101) in a retrieval engine called *Document eXplorer (DoX)*. *DoX* evaluates XML queries like those used throughout this work by translating them into SQL statements against the path and element tables, as described above. The system is compared to (1) the native XML engine X^2 that was already used for the experiments with the CADG in Section 6.4; (2) the relational node indexing scheme XPath Accelerator by Grust et al. [2002; 2004] (see Section 7.3.2); and (3) the relational path indexing scheme XRel by Yoshikawa et al. [2001] (see Section 7.4.1). All query engines have been implemented (or reimplemented, in the case of XPath Accelerator and XRel) in Java. Details of the hardware and software set-up are given in the appendix (see Test Environment A in Section 13.1).

We ran a number of queries against the four document collections *IMDb*, *XMark 1100*, *INEX* and *DBLP* listed in Section 13.2 of the appendix. The *Internet Movie Database (IMDb)* comprises more than 8 GB of XML documents describing movies and actors from a commercial web site [IMDB], whereas *XMark 1100* consists of 1 GB recursive XML synthetically generated by a benchmarking tool [XMark]. The highly heterogeneous *INEX* benchmark [INEX] contains scientific articles in full-text. *DBLP* [DBLP] is an on-line collection of bibliographic data from computer science. The key results of the evaluation are the following:

1. The RCADG outperforms both the native and the relational baseline systems by two orders of magnitude and more in terms of retrieval speed. Complex queries with large results causing the baseline systems to break down are answered within seconds by the RCADG. Querying XML in a relational database system benefits greatly from native XML indexing techniques (see Section 8.5.2 below). To a certain extent this also confirms previous findings reported by Chen et al. [2004] for the BLAS storage scheme (see Section 7.4.2).
2. The RCADG easily scales up to collections of multiple gigabytes both in terms of retrieval speed and storage demands. The path table is typically several orders of magnitude smaller than the original data (see Section 8.5.4).
3. Query planning has a significant impact on the performance of the RCADG. While very encouraging results were obtained with the planning strategies described above, in some cases inappropriate planning may prevent a performance gain. Also, enhancing the relational optimizer with tree statistics seems promising (see Section 8.5.3).
4. Keyword-driven schema matching using signatures in the path table does not entail a significant performance gain in our experiments. The overhead for signature comparison lies between 100 ms and 300 ms, whereas the time needed for creating signatures is negligible.

Table 8.5 on the next page summarizes the performance results for the RCADG (averaged after removing the best and worst of five runs). Sample queries are given as their closest XPath equivalents. The

Corpus	QID	result size	closest XPath query	processing time (s)
<i>IMDb</i>	I3	6507	<code>//*[title="love"]/production_year</code>	1.27
	I4	118,150	<code>//movie[//genre="documentary"]//actor</code>	8.77
<i>XMark 1100</i>	X4	2	<code>/site/open_auctions/open_auction[bidder[personref/@person="person20"]/following-sibling:: bidder[personref/@person="person17290"]]/reserve</code>	0.44
	X15	1890	<code>/site/closed_auctions/closed_auction/annotation/description/parlist/listitem/parlist/listitem/text/emph/keyword</code>	0.52
	X14	9461	<code>/site//item[contains(description, "gold")]/name</code>	3.34
	X13	22,000	<code>/site/regions/australia/item[name and description]</code>	0.88
	X2	597,777	<code>/site/open_auctions/open_auction/bidder/increase</code>	17.54

Table 8.5: RCADG query performance, in seconds. The original queries are given here as their closest XPath equivalent. *XMark 1100* queries are adapted from the XQuery benchmark [XMark]. Only matches to XPath result nodes were computed (unlike Table 8.6 on the following page).

XMark 1100 queries X2, X4, X13, X14 and X15 as well as X1 (in Table 8.6 on the following page) capture the XPath portion in the corresponding queries from the XQuery benchmark [XMark]. As can be seen in Table 8.5, the RCADG scales well with both the size of the document collection and the number of query results. The rest of this section discusses more results (see Tables 8.6, 8.7 and 8.8) in greater detail.

8.5.1 Test Systems

The *DoX* system consists of (1) an indexer for creating the RCADG tables and BIRD labels, (2) a system kernel including modules for query rewriting, planning and SQL code generation, and (3) a runtime for creating XML queries, triggering the kernel operations and sending the resulting SQL statements to the RDBS. The two RCADG tables are indexed using B^+ -Trees, as follows. The path table has a cluster index on the $\langle pid, maxid, tag \rangle$ fields and an additional index on the tag field. The element table has a cluster index on the $\langle pid, key, eid \rangle$ fields and an additional index on the $\langle key, eid \rangle$ fields. In the RDBS, only standard relational operators are used; in particular, no structural join for sets of XML elements is available.

Our native XML baseline database is X^2 [Meuss et al. 2005], which combines the original CADG index (see Chapter 6) and the BIRD labelling scheme (see Chapter 4). At system start-up, the CADG schema tree is loaded into main memory. During query evaluation, X^2 fetches sets of elements from the relational back-end and combines them into query matchings in memory. Element sets are joined using a variant of the *TwigStack* structural join by Bruno et al. [2002]. The query planning algorithm is similar to the one used by *DoX* (see Section 8.4.4 above). In particular, X^2 benefits from BIRD reconstruction, too. Note that X^2 always computes matches to all query nodes, i.e., $Q_r = Q_v$ by default.

As a baseline for relational query evaluation without a schema index we implemented the XPath Accelerator storage scheme by Grust et al. [2002; 2004] (see Section 7.3.2). Each query is translated into an m -fold join of the node table where m is the number of nodes in the query. Any element is labelled by its pre- and postorder ranks in the document tree, which also serve to decide binary query constraints. Reconstruction is not supported, and no schema-level index is available. The node table is indexed using separate B^+ -Trees on the preorder, postorder, parent label, tag and node type fields, as described by Grust [2002]. Textual contents are kept in a separate table indexed both on the preorder and keyword fields. We also applied the *shrink-wrapping* optimization proposed by Grust [2002]. By contrast, non-standard relational operators like the *Staircase Join* [Grust et al. 2003] are not available. Query planning for the node-table join happens entirely in the realm of the RDBS.

A second relational baseline system is our implementation of XRel [Yoshikawa et al. 2001] (see Section 7.4.1). XRel indexes tag paths as strings in a path table and performs schema-level matching through string search in the path table. A foreign key connects each tag path to its occurrence in a node table. Each XML query is translated into a single SQL statement joining the path and node tables. Again, query planning is done by the RDBS kernel. Like XPath Accelerator, XRel uses a subtree labelling scheme (see Section 3.3) to decide binary query constraints on the document level. Reconstruction is not supported.

Corpus	QID	result size	closest XPath query	processing time (s)		
				RCADG	CADG	XPAcc
<i>IMDb</i>	I1a	12	//person[name="mastroianni" and born/@place]/biography/movie	0.10	12.72	0.04
	I1b	3	//person[name="felix" and born/@place]/biography/movie	0.11	12.60	0.50
	I1c	24	//person[name="cooper" and born/@place]/biography/movie	0.15	12.84	215.83
	I1d	72	//person[name="steve" and born/@place]/biography/movie	0.52	12.85	> 600
	I2	6507	//title[.="love"]	0.37	0.30	> 600
	I3	6507	//*[title="love"]/production_year	1.52	26.07	> 600
	I4	118,150	//movie[//genre="documentary"]//actor	34.68	⚡	> 600
<i>XMark 1100</i>	X1	1	/site/people/person[@id="person0"]/name	0.09	6.85	0.02
	X21	13	//site/europe//item[//description//keyword[.="abandon" and //bold] and //name and (//category or //*[@category]) and //mail[//date and //from and //to]]	0.32	21.80	> 600
	X13	22,000	/site/regions/australia/item[name and description]	2.35	2.79	61.46
	X2	597,777	/site/open_auctions/open_auction/bidder/increase	122.43	⚡	292.14

Table 8.6: Query performance comparison for RCADG, CADG (CADG) and XPath Accelerator (XPAcc), in seconds. The original queries are given here as their closest XPath equivalent. *XMark 1100* queries are adapted from the XQuery benchmark [XMark]. Unlike Tables 8.5 and 8.8, matches to *all* query nodes were computed. The symbol “⚡” indicates that a specific query was not answered properly.

8.5.2 Runtime Performance

RCADG versus CADG. A first set of experiments measures the performance gain the RCADG achieves over native XML retrieval with X^2 (see the RCADG and CADG columns in Table 8.6). To avoid a handicap for the X^2 system, which always matches the entire query graph, the systems treated all query nodes as result nodes. For the RCADG, the runtime performance therefore differs from the results in Table 8.5 on the preceding page.

Queries I1a to I1d retrieve the place of birth and the movies of different people mentioned in the movie database *IMDb*. Note how the performance of both the RCADG and the CADG remains stable as the selectivity of the query keyword decreases: while the keyword “*mastroianni*” is contained only in 406 elements, the frequency of “*felix*” is almost ten times higher; “*cooper*” occurs in 10,398 elements and “*steve*” in 38,983 elements. The RCADG’s performance gain is two orders of magnitude for the most selective keyword (I1a) and still more than a factor 20 for the most frequent keyword (I1d). As queries I2 and I3 illustrate, the overhead incurred by the CADG is mainly due to “output” nodes like `place` and `movie` which are not subject to keyword constraints. While the CADG is highly competitive for queries without such unselective nodes, such as I2, the `production_year` node in I3 slows down the native system by two orders of magnitude. Unlike the RCADG, the CADG retrieves matches to the `title` and `production_year` nodes in the element table and transfers them into main memory for deciding their binary query constraint (the `child` step).⁴ By contrast, the RCADG translates binary constraints into join conditions supported by relational indices on the element table, and therefore faces no such overhead for loading large element sets.

Query I4 illustrates another potential weak-spot in native retrieval systems which compose matches to tree queries in main memory: processing large intermediate result sets containing tens or hundreds of thousands of tuples easily exceeds the hardware capacities. During the evaluation of I4, X^2 quickly ran out

⁴The huge overhead for I3 compared to I2 might not be faced by native systems which do not compose path occurrences in this way but retrieve entire tree fragments instead, like the *NatiX* system [May et al. 2004; Fiebig et al. 2002].

Corpus	QID	closest XPath query	result size		processing time (s)	
			RCADG	XRel	RCADG	XRel
INEX	N1a	//p	609	609	< 0.01	0.09
	N1b	//p[sub]/b	27	3,485,916	0.01	515.22

Table 8.7: Query performance comparison for RCADG and XRel on the schema level, in seconds. Processing times and intermediate result sizes are measured at the end of phase 1. The original queries are given here as their closest XPath equivalent.

of memory; allocating more than 800 MB on our 1-GB machine avoided a crash but resulted in swapping. The RCADG, however, copes well with large result sets.

Query X21 against the *XMark 1100* collection examines how the systems cope with tasks whose complexity is in the query structure, not the result size. The RCADG invests 85% of a total of 321 ms in generating extremely efficient SQL code that involves the reconstruction of four *Parent* constraints. By contrast, X^2 is again trapped in too many decision operations. The results for X1 and X2 in Table 8.6 confirm the earlier observations for I3 and I4, respectively. Note that when returning only matches to the XPath result nodes, the RCADG answers the same queries again up to 7 times faster (see Table 8.5), retrieving more than half a million matches in less than 20 seconds.

RCADG versus XPath Accelerator. It has been mentioned before that XPath Accelerator decides all binary constraints via selfjoins on the node table, lacking both reconstruction capabilities and schema-level information. Consequently, in our test with different keyword selectivities (queries I1a to I1d in Table 8.6 on the preceding page), the evaluation time rapidly grows with the size of intermediate results, reaching 820 seconds for I1d compared to only 0.52 seconds with the RCADG. Less selective queries like I2 to I4 also take longer than ten minutes to evaluate. Only for highly selective queries like I1a or X1, XPath Accelerator is slightly faster than the RCADG, possibly because the latter issues multiple SQL queries rather than only a single one. The impact of a complex query graph like X21 is much higher for XPath Accelerator than for the native or relational CADGs. Since XPath Accelerator selects tuples in the element table based only on singleton tags rather than tag paths, it has to join large intermediate results.

The most unselective query in our test suite, X2, has a much simpler structure (no branches and no descendant steps). Here XPath Accelerator is faster than for X21, but still takes more than twice as long as the RCADG. Query X2 is reported as critical by Grust et al. [2004], too. Note that when retrieving only matches to the leaf of the path, in XPath style, the RCADG outperforms XPath Accelerator by one order of magnitude (22 seconds versus 220 seconds). As query I4 shows, XPath Accelerator does not scale well to unselective queries with many descendant steps, which involve range conditions in the selfjoin of the node table. Here the RCADG is two orders of magnitude faster. Note that even the special relational index structures and join operators employed by Grust [2002], which are reported to recover up to one order of magnitude of processing time, are unlikely to remedy this handicap completely. Obviously the RCADG takes considerable advantage from BIRD reconstruction when answering query I4, using the keyword-restricted `genre` node as a starting point in the query plan.

Summing up, the experiments prove that the native XML indexing techniques underlying the RCADG entail a decisive performance gain in the relational domain.

RCADG versus XRel. As explained in Section 7.4.1, XRel’s atomic representation of tag paths as strings has a number of disadvantages, compared to the compositional path representation of the RCADG. First, string matching tends to be slower than the comparison of numeric node labels, especially for query paths starting with a descendant step. The following experiment quantifies this overhead using queries against the *INEX* collection. Table 8.7 compares how fast RCADG and XRel match a query graph on the schema level (phase 1) and how many matches they retain for document-level matching (phase 2). For N1a both systems retrieve 609 matches, but the RCADG is slightly faster. Second, XRel produces many partial matches to be discarded later in phase 2: for N1b its intermediate result is five orders of magnitude larger than that of the RCADG. As explained in Section 7.4.1, XRel’s atomic path representation is not precise enough to discard combinations of `sub` and `b` elements that do not belong to the same `p` parent.

Corpus	QID	closest XPath query	result size		processing time (s)	
			RCADG	XRel	RCADG	XRel
<i>DBLP</i>	D1a	//article[author="codd"]/title	34	34	0.12	9.18
	D1b	/dblp/article[author="codd"]/title	34	34	0.12	9.14
<i>XMark 1100</i>	X1	/site/people/person[@id="person0"]/name	1	1	0.09	3.96
	X22	//parlist[./text[.="zenelophon"]]/listitem/text	133	⚡ 183	0.14	27.95
	X14	/site//item[contains(description,"gold")]/name	9461	9461	3.34	> 600
	X13	/site/regions/australia/item[name and description]	22,000	22,000	0.88	> 600
	X23	//regions[contains(., "zyda@ask")]/keyword	416,175	416,175	32.21	310.03
	X2	/site/open_auctions/open_auction/bidder/increase	597,777	597,777	17.54	6.12

Table 8.8: Query performance comparison for RCADG and XRel, in seconds (phases 1 and 2). The original queries are given here as their closest XPath equivalent. Only matches to XPath result nodes were computed (unlike Table 8.6 on page 120). The symbol “⚡” indicates that a specific query was not answered properly.

This also slows down the subsequent document-level matching, as shown in Table 8.8. Here the processing time subsumes the entire query evaluation process (phases 1 and 2), and the result size only counts only elements that are part of the final answer to the query. On the *DBLP* collection the RCADG is almost two orders of magnitude faster than XRel (D1a), even for an absolute query path (D1b). On *XMark 1100*, the difference is between one and three orders of magnitude. XRel outperforms the RCADG only for a single unselective query without branching nodes and descendant steps (X2). For such queries matching exactly one path in the schema, the RCADG’s compositional path representation has no extra benefit, but rather entails a small overhead compared to exact string matching without wildcards.

By contrast, for proper tree queries with descendant steps, XRel not only takes more processing time but may also produce wrong final results on recursive collections like *XMark 1100*. For instance, in the case of query X22, XRel is two orders of magnitude slower than the RCADG and retrieves 50 false hits. By contrast, the query evaluation with the RCADG is fast and correct, owing to its compositional path representation and BIRD reconstruction. This phenomenon is explained as follows. For illustration, reconsider the query in Figure 8.5 *a*. on page 104. The RCADG answers this query with only two element-table joins, as specified by the corresponding query plan in Figure 8.11 *d*. on page 109. The SQL code generated to answer the same query with XRel is given in Figure 8.18 on the next page. Here we ignore the query node q_6 and the *NextSib* edge because XRel does not support sibling constraints. For the resulting query graph comprising the five query nodes q_1 to q_5 , XRel combines a five-fold join of the path table with another five-fold join of the node and content tables (see the FROM clause in Figure 8.18). As described in Section 7.4.1, tag path patterns are created from the query and matched against the *pathexp* column in the path table (black part of the WHERE clause in Figure 8.18). The path IDs retrieved this way act as foreign keys to the node and content tables (blue part of the WHERE clause in Figure 8.18). Finally, all binary query constraints are decided on the document level, using region encoding (green part of the WHERE clause in Figure 8.18). Note how matches to distinct tag paths are first retrieved independently and then combined through the join predicates on the node and content tables. This causes the large intermediate result after phase 1 for N1b in Table 8.7.

Compared to XRel, the RCADG (1) replaces suffix and infix string matching involving numerous wildcards with efficient numeric equality predicates in the selfjoin of the path table, (2) saves three out of five expensive joins with the element table through BIRD reconstruction, (3) looks up fewer schema hits in the element table in cases where the individual query paths have disparate partial matches in the documents (as in query N1b above), and (4) correctly discards partial matches from the final result in presence of a recursive schema. For instance, assume that the sample query from Figure 8.5 *a*. is run against a document collection containing nested *person* elements. Then the code in Figure 8.18 on the facing page wrongly accepts those *person* elements which lack a suitable *watches* child, but instead have a *person* descendant with such a *watches* child. The reason is that XRel loses track of the common *person* ancestors of matches to node q_2 (*watches*) and q_4 (*profile*), which are treated simply as matches to two dis-

```

SELECT
  NT3.start, NT3.end, NT4.start, NT4.end           -- add matches to q3 and q4
FROM
  PathTable PT1, PathTable PT2, PathTable PT3,    -- join path, node and content tables
  PathTable PT4, PathTable PT5,
  NodeTable NT1, NodeTable NT2, NodeTable NT3, NodeTable NT4,
  ContentTable CT5
WHERE
  PT1.pathexp LIKE '#%/person' AND                -- match tag paths
  PT2.pathexp LIKE '#%/person#/watches' AND
  PT3.pathexp LIKE '#%/person#/watches#/open_auction' AND
  PT4.pathexp LIKE '#%/person#/profile' AND
  PT5.pathexp LIKE '#%/person#/profile#/gender' AND
  NT1.pathid = PT1.pathid AND                    -- match unary constraints
  NT2.pathid = PT2.pathid AND
  NT3.pathid = PT3.pathid AND
  NT4.pathid = PT4.pathid AND
  CT5.pathid = PT5.pathid AND
  CT5.value = 'XML' AND
  NT1.start < NT2.start AND NT1.end > NT2.end AND -- decide Child(q1,q2)
  NT2.start < NT3.start AND NT2.end > NT3.end AND -- decide Parent*(q3,q2)
  NT1.start < NT4.start AND NT1.end > NT4.end AND -- decide Parent*(q4,q1)
  NT4.start < CT5.start AND NT4.end > CT5.end    -- decide Parent(q5,q4)
ORDER BY
  NT3.start, NT3.end, NT4.start, NT4.end         -- order result as needed

```

Figure 8.18: SQL code for query evaluation with XRel (see Section 7.4.1). Blue colour highlights code related to joins with the node or content table, whereas green colour is used for the decision of binary query constraints. The query being evaluated is a variant of the query in Figure 8.5 *a*. on page 104 where the node q_6 and the binary constraint $NextSib(q_6, q_5)$ have been removed (since XRel does not support sibling constraints).

tinct path patterns (`#%/person#/watches` and `#%/person#/profile` in the WHERE part of the SQL statement). By contrast, the RCADG keeps tuples of matches to all nodes in the query graph as intermediate results and hence never mixes up distinct `person` ancestors. Faced with two nested partial `person` matches as just described (one satisfying only the constraints related to q_2 and the other to q_4), the RCADG rejects both during phase 2 at the latest, but possibly even earlier during schema matching. In the same way, it discards nested `parlist` elements that only partially match the root of query X22 in Table 8.8.

8.5.3 Impact of Query Planning and Optimization

The RCADG offers a considerable potential for query optimization. First, the path table neatly accommodates certain statistical information about the document tree and its textual content, as sketched in Section 8.2. The need to enhance relational query optimizers with such tree-specific data was pointed out by Krishnamurthy et al. [2003]. Second, query evaluation may benefit greatly from logical query planning and rewriting. For instance, query X4 from the *XMark* benchmark (see Table 8.6 on page 120) originally enforces only a *NextElt* constraint between the two `personref` nodes. Replacing this with a more restrictive *NextSib* constraint between the `bidder` nodes reduces the processing time with the RCADG from 5287 ms to 508 ms, if matches to all query nodes are to be computed. If only the `reserve` node is regarded as a result node, the rewriting techniques described in Section 8.4.3 remove the `site`, `open_auctions` and `personref` nodes after schema matching, which again saves 68 ms, resulting in a total processing time of 440 ms as shown in Table 8.5 on page 119. Processing the second keyword constraint earlier in the query plan would probably further accelerate the evaluation.

8.5.4 Storage Requirements

Maintaining schema information besides the actual element data, as with the RCADG or CADG, comes at only little extra cost in terms of storage. In our experiments, the path table occupies merely between 48 kB (*DBLP*) and 120 kB (*XMark 1100*) on disk, including the various B⁺-Trees and optional fields mentioned before. The CADG schema tree in memory occupies 2 MB in both cases (with keyword signatures and statistical information attached to the schema nodes, as described before). Only for the heterogeneous *INEX* corpus the path table is nearly 2 MB on disk and the schema tree 25 MB in memory.

By contrast, the element table of the RCADG grows to 17 GB for *XMark 1100* and 34 GB for *IMDb* (again including all relational index structures). In particular, the materialized join of elements and the keywords they contain introduces considerable redundancy, which on the other hand speeds up query evaluation. For instance, to support efficient keyword search, different keywords occurring in the same element are stored in distinct rows of the element table, rather than in a single tuple containing the full textual element content as a string (this has been proposed, e.g., by Grust et al. [2004] for XPath Accelerator). While our approach is certainly less compact (because there may be multiple tuples for the same element/tag path pair), it permits to solve keyword constraints with efficient equality conditions instead of substring matching. We applied the same technique to the XPath Accelerator storage scheme to make both systems comparable.

Note that in the experiments, the XPath Accelerator node and content tables together are only a little smaller than the RCADG's element table (*XMark 1100*: 3 GB + 11 GB; *IMDb*: 12 GB + 19 GB). By contrast, the CADG originally stores elements with the same tag path and keyword together as a list in the same tuple (non-first normal form), rather than in separate rows as with the RCADG. Therefore its element table is considerably smaller (4.5 GB on *XMark 1100*, 5.2 GB on *IMDb*). For the RCADG, storing elements in non-first normal form is infeasible because it prevents index conditions on individual elements, as used by the BIRD rules in Figures 8.2 and 8.3.

To sum up, our experiments show that the RCADG scales up to multiple gigabytes and is far from the quadratic space needed by highly redundant techniques like a fully materialized *Parent*₁^{*} relation, as proposed by Jiang et al. [2002]. The native CADG is more storage-efficient than the two relational schemes. Yet we believe that from a user's perspective, and given the steady advances of storage technology, retrieval speed should be given a higher priority than space consumption.

8.6 Summary and Discussion

The Relational CADG (RCADG) presented in this chapter exploits native XML indexing techniques for the efficient evaluation of XML queries a relational database system. In particular, it has been shown how a centralized structural summary (the CADG) and a decentralized structural summary (the BIRD labelling scheme) can be migrated to the relational data model, and how suitable query planning and rewriting exploit these summaries to reduce the number and size of element sets to be joined in the RDBS.

The benefit of path indexing in RDBSs has been discussed in the previous chapter for storage schemes like XRel, which stores tag paths as strings in a path table, and BLAS, which represents tag path suffixes as numeric intervals. Like the RCADG, these schemes match simple path expressions with fewer joins than those without a path index [Krishnamurthy et al. 2003], like the XPath Accelerator or Edge schemes. As described in Section 7.4, path-based approaches retrieve elements based on more restrictive selection predicates, which simplifies index scans and reduces the size of intermediate results to be joined. Besides, path-specific information like the node type is no longer stored redundantly for all elements with a given tag path, but only once in the corresponding path table entry.

Contributions of the RCADG. The RCADG further enhances relational path indexing in several respects, addressing several open problems mentioned in the literature. The key contribution of our approach compared to previous work is the *precise compositional representation of tag paths*. To the best of our knowledge the RCADG is the only relational storage scheme to represent every tag path prefix as a sequence of nodes (i.e., tuples in the path table). This has a number of advantages, also compared to the abovementioned string-based or suffix-based approaches:

- The entire query graph structure is matched against the path table before accessing the large element table. Unsatisfiable schema constraints are detected extremely fast during retrieval phase 1. Access to the element table are restricted to paths satisfying all *S*-constraints. In particular, branching path expressions are matched already on the schema level. This discards partial schema hits during phase 1 that cannot be detected by less precise path indices like XRel and BLAS.
- Schema-level matching receives excellent indexing support through B^+ -Trees on the numerical interval labels for schema nodes. This is more efficient than XRel's substring matching, especially for paths with leading descendant steps. Matching tag path constraints through a selfjoin of the small path table is cheap and happens entirely in the realm of the relational query optimizer.
- Representing the schema as a tree in the RDBS allows the RCADG to take advantage of BIRD reconstruction, which avoids expensive joins with the element table.
- The RCADG efficiently evaluates queries involving any XPath axis and `//*` steps, which string-based approaches as proposed by Yoshikawa et al. [2001] and Jiang et al. [2002] support only with more complex regular expressions. Furthermore, `/*` steps do not entail extra selfjoins of the element table as with XRel.
- Existential XPath predicates are handled correctly even for recursive collections without a massive join overhead, which is considered an open problem by Krishnamurthy et al. [2003]. False positives in the final query result as with XRel are avoided.
- Prefixes shared by multiple tag paths are not stored redundantly as with XRel. No string operations are needed to concatenate query path fragments, as with XRel or BLAS.
- Unlike the P-labels used by BLAS, the tag-path references in the RCADG's element table are robust against changes to the schema tree.

Fast tree matching in large recursive document collections. XRel's incorrect handling of certain queries against recursive collections was hinted at by Krishnamurthy et al. [2003], who concluded that "the general problem of translation of path expressions with predicates for the path-based schema-oblivious schemes is still open". Node-indexing approaches like the XPath Accelerator or Edge schemes answer such queries by triggering a selfjoin of the node table for each step in a query path, which is costly. With the exception of BLAS, the RCADG is the only path-indexing approach we know of that correctly handles these queries. BLAS achieves this by checking additional level constraints on elements, a technique which might also fix XRel's defective evaluation of such queries. The RCADG does the same already on the schema level and therefore needs much fewer comparison operations. Besides, the RCADG benefits from the reconstruction capabilities of the BIRD tree encoding to avoid expensive joins with the element table. As shown in Chapter 4, this feature is paramount to efficient large-scale processing of XML queries. While the experimental results reported by Chen et al. for BLAS are in line with our findings concerning path- versus node-indexing schemes, they are insufficient to judge the scalability of their approach up to tens of gigabytes.

Tree-aware query planning. The query planning and rewriting techniques presented in this chapter strive to optimize the generation of SQL code for evaluating tree queries in an RDBS. However, the way a relational query kernel processes this code could also benefit from techniques specific to the tree data model. Tatarinov et al. [2002] point out that "relational optimizers need to understand the hierarchical structure of XML". With the RCADG, statistical information kept in the path table enables more accurate query planning based on properties of the document tree (not merely the set of tuples stored in the element table), such as the number of elements with a given tag path or the number of distinct keywords contained or governed by these elements. While in our experiments encouraging results were obtained without such a "tree-aware" RDBS kernel, we expect that physical plans estimating access costs based on XML statistics will further speed up the query evaluation.

Indexing textual element contents. It has been mentioned before that XPath Accelerator (and many other relational storage schemes) keep elements and their textual contents in separate node and content tables, unlike the RCADG which combines both in a single element table. As a matter of fact, the RCADG's element table is a materialized join of the node and content tables used by node-indexing storage schemes, augmented with schema-level information in the form of references to the path table. Grust et al. [2004] argue that separate node and content tables allow to match structural constraints on the document level without accessing textual contents in the first place. This is an advantage when given unselective keyword constraints. In particular, queries without any keyword constraint may run faster against a node table that is clustered, say, in document order [Grust et al. 2004].

Of course, keyword constraints are often selective and thus help to reduce the size of intermediate result to be joined. With separate node and content tables, such queries are also easily processed through a simple join on the document node labels. Here the RDBS query kernel automatically figures out whether to start the join with the structure or keyword constraints. However, potentially selective schema constraints (such as a specific combination of tag paths in the query) are not taken into account.

Therefore the RCADG combines tag paths, keywords and elements in a single element table which is clustered by tag paths, keywords and elements, in that order. For queries with selective schema and keyword constraints, this reduces the I/O during the element look-up. However, unselective constraints may cause more disk pages to be accessed than with the separate node and content tables, due to the clustering. Thus schemes like XPath Accelerator and the RCADG are optimized towards distinct kinds of query. However, while the difference between separate versus combined indexing of content and structure is crucial in native XML retrieval (see Chapter 6), the impact in the relational scenario is probably lower.

The way element contents are indexed also has an impact on how to obtain an XML serialization of the query results. Grust et al. [2004] sketch a method to create XML fragments on the fly by sequentially scanning the node table in document order. Since the RCADG's element table is clustered by tag paths and keywords rather than elements, this could result in much random disk I/O. Therefore we keep the original documents and the byte offsets for all elements instead. This allows to retrieve the original serialization of any result element from the documents in time linear in the size of the XML fragment, not the overall size of all documents.

8.7 Optimizations and Open Problems

The most obvious way to enhance query evaluation with the RCADG further is more sophisticated query planning based on selectivity estimates of keywords, tag paths, and combinations of both. An interesting question in this context is how much of the planning can be realized outside the RDBS and at which point the relational optimizer must be modified. As shown before, query planning is also tightly related to query rewriting, where more advanced rules might be developed. In fact, much work has been done in the field of XML query optimization so far, which is largely complementary to our approach. For references to related work, see Section 8.4.3 above.

Another obvious enhancement of the RCADG is the use of a structural join algorithm in the RDBS. However, at the time of this writing such XML-specific functionality is not available in most off-the-shelf database systems. Realizing structural joins as user-defined functions might be considered in the future.

We have also outlined how the RCADG seamlessly integrates keyword signatures, a heuristic technique from Information Retrieval, in order to detect keyword mismatches early during schema matching. However, in a small-scale experiment this technique did not expedite the query evaluation. A more thorough analysis is needed to understand whether this observation also holds for other queries on different document collections.

Finally, it should be mentioned that the RCADG could in principle be combined with labelling schemes other than BIRD, although this would require parts of the query rewriting, planning and matching to be revised. Chapter 3 has presented a small number of alternative labelling schemes with similar expressivity. In particular, an approach with better update support, such as ORDPATH [O'Neil et al. 2004], could be attractive at least for certain applications with highly dynamic document collections. However, since BIRD offers high query performance combined with reasonable space consumption and integrates well with relational query evaluation, we believe that it is a good choice for possible future work on the RCADG.

Part V

Caching Techniques for XML

Caching Techniques for Incremental XML Retrieval

9.1 Overview

The preceding chapters have introduced different contributions to making the evaluation of XML queries more efficient. So far it has been assumed that each query is evaluated *from scratch*, i.e., regardless of any previously computed search results for the same or other queries. However, in a typical query workload there may be numerous queries whose results overlap at least partially. This applies in particular to an iterative retrieval process as discussed in Section 1.3, where the user is encouraged to modify and run a given query repeatedly in order to improve the retrieval results. This chapter reviews different ways to store available query results in a *query cache* after evaluation so that new queries might be answered based on these cached results. Such reuse of query results is called *incremental query evaluation* in the sequel because parts of the answers to future queries emerge gradually during the retrieval process. In the literature the term *semantic caching* is also very common. The main challenge in incremental query processing is to detect and exploit containment or overlap of query results with only a small overhead for the cache look-up. Later a new query cache will be presented that allows to do this more efficiently than with the approaches discussed in this chapter. Experiments will also show that the incremental evaluation of a given query is often much more efficient than its evaluation from scratch.

Since cached query results can be regarded as views on (part of) the original document data, incremental query processing is an instance of the problem of query answering in the presence of views. Calvanese et al. [2003] distinguish the following two variants of the problem: in *view-based query containment*, queries and view definitions are compared on the intensional level only, i.e., without accessing the actual data. By contrast, in *view-based query answering* the results of a given query are computed from both the view definitions and extensions. Notice that this requires the views to be materialized. Query answering with views is notorious for being inherently complex in the relational data model, and the same is true for semistructured data (see below). Despite the high theoretical complexity, however, many different approaches have been proposed that strive to push the practical efficiency of incremental XML query processing to its limits. They build on a variety of different computational and data models and retrieval techniques such as, e.g., native XQuery engines [Chen and Rundensteiner 2005; Shah and Chirkova 2003] or XQL engines [Quan et al. 2000], two-way finite state automata [Calvanese et al. 2002], tree automata [Chen et al. 2002] or *Lightweight Directory Access Protocol (LDAP)* servers [Marrón and Lausen 2002].

This chapter reviews a selection of representative approaches to the incremental evaluation of XML queries using some sort of query cache. A thorough comparison and survey of the topic seems to be missing as of the time of this writing. Besides the underlying data model and the expressiveness of the query language, there are other potential criteria for comparison and classification. For instance, one could start out by distinguishing *schema-aware* and *schema-oblivious* approaches, as for the relational storage schemes in Chapter 7. However, most systems reviewed below either ignore the document schema or consider it only for query evaluation from scratch, but not for caching purposes. As a tentative guideline to this overview – but also for future work in the field –, the following questions highlight the most salient issues that can serve as marks of distinction when comparing XML caching techniques.

1. *query representation*: Which query language and features are supported? If queries may be partially answered using uncached data, which query engine is used for that purpose?
2. *cache representation*: How are cache contents represented? Is the representation intensional or extensional? Is the cache held in main memory, or secondary storage, or both?
3. *cache usage*: Does the cache exploit or require a DTD or other schema specification? Can results of distinct cached queries be combined to answer a given new query? How to choose the best among alternative reusable queries in the cache? Does the system support a combined evaluation from the cache and from scratch?
4. *query comparison*: How are cached and new queries compared? Does the system take advantage of result overlap, result containment, or only the repeated evaluation of the same query? What is the time complexity for detecting query containment or overlap?
5. *scalability*: How does the system avoid cache overflows? Is there a cache replacement strategy? Does the overhead for cache look-ups grow with the cache size?
6. *practical benefit*: Has the system been evaluated experimentally to quantify the practical benefit of incremental query evaluation compared to the evaluation from scratch? How effective and efficient is the cache look-up?

9.2 XML Query Containment and Overlap

Before reviewing a couple of methods for processing XML queries incrementally, a formal description of the underlying decision problems is in order. The fundamental notion of *query containment* has already been mentioned. It is often understood in a fairly abstract sense with no regard to the actual representation of queries and their results, and sometimes even used without specifying its formal semantics. Intuitively, a query Q^c is said to contain another query Q^n iff the answer to Q^c subsumes all matches to Q^n . Typically, this intuition implies that if Q^c contains Q^n and the result of Q^c is available in the cache, then Q^n can be answered from the cache only, without accessing the original documents or any representation thereof.

However, after taking a closer look it turns out that whether access to data outside the cache is needed to evaluate Q^n depends on the exact representation of Q^c 's answer in the cache. In fact, the conclusion just mentioned silently assumes that each node in the document tree D that is part of some match to Q^c is cached together with its entire subtree in D . For instance, suppose $Q^c = //person/name$ and $Q^n = //person/name[. = "Lee"]$, where Q^n restricts Q^c with an additional keyword constraint. Obviously the set of matches to Q^n is a subset of the set of matches to Q^c , hence Q^c contains Q^n in the above sense. However, to process Q^n incrementally, we must have access to the full textual content of the `name` elements in Q^c 's answer. If matches to Q^c are only represented as sets of unique element labels in the cache (as in the result tables produced by the RCADG evaluation that was presented in the previous chapter), then answering Q^n requires access to data outside the cache. Therefore, discussing the question of access to data inside or outside the cache generally makes sense only with the concrete data representation of a given caching approach in mind.

A second notion that is fundamental to incremental XML retrieval is *query overlap* or *partial query containment*. The overlapping of two XML queries that do not fully contain each other is ignored by almost all caching techniques we know of, and therefore rarely defined in the literature. In fact, there are multiple ways how queries can overlap or partially contain each other, some of which are easier to exploit than others. Figure 9.1 on the next page contrasts different cases of overlap/partial containment (*a.–c.*) with full query containment (*d.*). The following definitions capture these differences.¹ In this context, recall from Definition 2.3 on page 10 that each match to a query Q is essentially a set of document nodes that together match the query nodes in Q . Given such a match a , let $v(a)$ be the set of document nodes in a .

¹Here we assume a fixed document collection D against which the queries are executed, as before. Note that the definitions in this chapter are easily generalized to capture query containment and overlap without a fixed document collection.

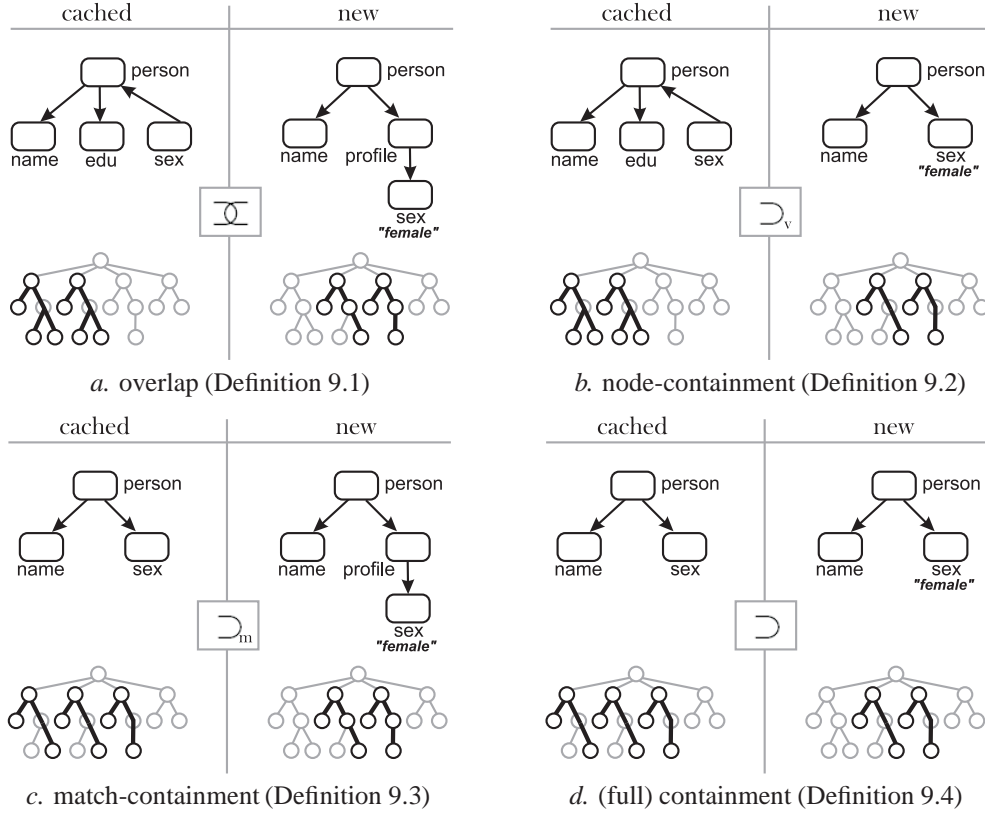


Figure 9.1: Query containment and overlap. Each of the four subfigures depicts a cached query and a new query to be evaluated incrementally, together with their extensions (matches) in the document tree D from Figure 2.1 b. on page 8. Note that subfigures a.–d. show different pairs of queries and results with a varying degree of similarity. Four decision problems are presented, roughly speaking, in order of decreasing hardness for most caching systems: a. Overlapping queries only share individual elements in their results. Missing elements or entire matches must be retrieved from scratch. b. If the new query is node-contained in a cached query, some of its matches are entirely present in the cache while others must be computed from scratch. c. A cached query that match-contains the new query provides at least some elements for each match of the new query. d. Full query containment guarantees that the cached query is no more restrictive than the new query, and that all elements in the query result are present in the cache. However, note that the cached query result may need to be purged of false matches with respect to the new query.

Definition 9.1 (Query overlap) Let Q^c and Q^n be two queries. We say that Q^c overlaps with Q^n , $Q^c \sqcap Q^n$, iff for at least one match $a^n \in \text{ans}(Q^n)$ there exists a match $a \in \text{ans}(Q^c)$ such that $v(a^n) \cap v(a) \neq \emptyset$. \square

Definition 9.2 (Node-containment) Let Q^c and Q^n be two queries. Besides, let A^n be the subset of matches to Q^n that share nodes with matches to Q^c , i.e., $A^n = \{a^n \in \text{ans}(Q^n) \mid \exists a \in \text{ans}(Q^c) : v(a^n) \cap v(a) \neq \emptyset\}$. Finally, let $V_{Q^n} = \bigcup_{a \in A^n} v(a)$ and $V_{Q^c} = \bigcup_{a \in \text{ans}(Q^c)} v(a)$. We say that Q^c node-contains Q^n , $Q^c \supset_v Q^n$, iff $Q^c \sqcap Q^n$ and $V_{Q^c} \supset V_{Q^n}$. \square

Definition 9.3 (Match-containment) Let Q^c and Q^n be two queries. We say that Q^c match-contains Q^n , $Q^c \supset_m Q^n$, iff for each match $a^n \in \text{ans}(Q^n)$ there exists a match $a \in \text{ans}(Q^c)$ such that $v(a^n) \cap v(a) \neq \emptyset$. \square

Definition 9.4 (Query containment) Let Q^c and Q^n be two queries. We say that Q^c (fully) contains Q^n , $Q^c \supset Q^n$, iff Q^c match-contains Q^n and Q^c node-contains Q^n . \square

schema constraints	query constraints	complexity	proved by
with DTD	/, [], //, *	EXPTIME-complete	Neven and Schwentick 2003, Wood 2003
	/, []	CONP-complete	Neven and Schwentick 2003, Wood 2001
	/, //, *	P ^T IME	Neven and Schwentick 2003
without DTD	/, [], //, *	CONP-complete	Miklau and Suciu 2002
	/, [], //	P ^T IME	Amer-Yahia et al. 2001
	/, [], *	P ^T IME	Wood 2001

Table 9.1: Complexity of XPath query containment with selected query and schema constraints.

9.3 Complexity of XML Query Containment

As mentioned before, the incremental evaluation of XML queries based on cached results depends on methods to detect containment or overlap between queries in the cache and a new query to be answered. Many papers have studied the theoretical complexity of query containment on semistructured data for different query languages which are mostly based on regular path expressions. There are a number of parameters to the query containment problem that have a considerable impact on its theoretical complexity. These parameters include: (1) the query language used to express the cached queries or views and the new query; (2) the input alphabet (i.e., set of symbols in the XML documents) which may be either finite or infinite; (3) whether we are interested in containment on either a fixed set of documents or any document collection or all documents that are valid with respect to a given schema specification (e.g., a DTD); and, most prominently, (4) particular features and restrictions of the query language, schema (if any) and the concrete queries to be compared.

Table 9.1 summarizes some important complexity results concerning the containment of XPath queries with various constraints on the queries and the document schema. The complexity of XPath containment depends mainly on the allowed features of the language (e.g., which axes are used) and on the availability of schema constraints (e.g., in the form of a DTD). Most authors have studied different combinations of XPath features such as predicates (“[]”), wildcard node-test (“*”) and the child and descendant axes (“/”, “//”). They found that unless “*” and “//” are both allowed, XPath containment can be decided in polynomial time [Amer-Yahia et al. 2001; Wood 2001; Miklau and Suciu 2002]. Miklau and Suciu also describe an algorithm for deciding containment in polynomial time that is sound, but not complete. The situation is different in the presence of schema constraints, however. As Wood [2003] points out, child or sibling constraints inferable from a DTD permit to detect some cases of containment that are not visible from the query intension alone. Yet such constraints also increase the complexity of the query comparison. Neven and Schwentick [2003] show that as soon as predicates are allowed, deciding XPath containment in the presence of a DTD requires exponential time.

Various other query and schema constraints have been considered in the literature to obtain a more precise picture of the complexity of the problem. For instance, Neven and Schwentick extended the XPath language with existential variable bindings and disjunction. Wood inquired into child, sibling and functional constraints in DTDs as well as certain restrictions of tag repetitions in DTDs and queries. Before, Deutsch and Tannen [2001] observed the impact of integrity constraints on XPath queries in the presence of DTDs. Others have dealt with different query languages. For instance, Calvanese et al. [2000; 2002; 2003] considered both conjunctive queries over relational views and regular path queries over semistructured views of the documents, stressing the impact of whether the queries are expressed in terms of the original or the view alphabet. They proved that deciding containment of regular path queries (with or without inverse axes) needs exponential time in any case, although the tight complexity bounds vary.

9.4 XML Query and Result Caching

9.4.1 Incomplete Trees

In general a query cache can be expected to contain only part of the information needed to answer a given query. Abiteboul et al. [2001b] therefore represent cached query results in a data structure that also speci-

fies which information is missing in the cache. To this end, earlier work by Imielinski and Lipski [1984] on incompleteness in relational databases is applied to a simple semistructured data and query model. Unlike most other authors, Abiteboul et al. regard the XML data being cached as unordered. They describe the document structure using a structural summary that is effectively a simplified variant of a DTD. The structural summary is mainly used for formulating queries. Any query is a prefix of the structural summary, i.e., a subtree of the summary tree that includes its root, together with optional keyword constraints and tag negation predicates. Queries may contain branchings of multiple root-to-leaf path patterns. More advanced features such as XPath's tag wildcard ("*") and the descendant axis ("//") are not supported.

Abiteboul et al. propose the *Incomplete Tree* as a main-memory cache data structure that comprises both extensional and intensional parts. The extensional part of the Incomplete Tree is a prefix of the document tree D , i.e., a copy of the upper part of D (including the root of D) that gradually becomes larger as more and more query results are being added to the cache during retrieval.² To indicate which data are missing from the cache, the logical complements of all cached queries are intensionally represented by extra nodes in the Incomplete Tree that have keyword constraints or DTD-style multiplicity predicates attached. For instance, an intensional node below a path `/people/person/name` in the Incomplete Tree might specify that the occurrences of `name` elements in the cache exclude elements which do *not* contain the keyword "Lee". This information is inferred by negating queries whose results are to be cached (in the example, a query involving the path `/people/person/name` combined with a containment constraint for "Lee"). Using the incompleteness information, non-redundant *remainder queries* can be created in polynomial time, i.e., queries extracting precisely those parts of the desired data that is missing in the cache. Abiteboul et al. even claim that so-called *local queries*, which are evaluated with cached elements as context nodes, retrieve exactly the missing elements from the documents. However, it remains unclear how document subtrees which have never been cached (due to some mismatch with all prior queries) can be reached when using cached elements as context nodes. Moreover, the authors concede that generating non-redundant remainder queries does not guarantee practical efficiency. No experimental evaluation is provided in the literature.

A second issue concerns the size of the cache. Note that specifying missing information through query negation may result in an exponential growth of the cache. Abiteboul et al. point out that this is a general lower bound for representing the complement of a sequence of queries in the cache. They also describe several workarounds which guarantee a maximum cache size that is polynomial in the total size of all cached queries and their results. However, these techniques either make the cache look-up more complex (in some cases, NP-hard) or further restrict the query language.

An alternative way to bound the cache size has been developed by Hristidis and Petropoulos in 2002. Their *XCacher* system builds on the work by Abiteboul et al. and comes with support for a subset of XQuery. Simplified XQuery expressions (no query nesting, document order, or LET clauses) are first translated into expressions of the same prefix-selection query language that has been used by Abiteboul et al. (see above). The main difference of *XCacher* compared to previous work lies in its central data structure, the *Modified Incomplete Tree (MIT)*. The extensional part of the MIT is similar to the original Incomplete Tree. Unlike the latter, however, the MIT intensionally describes the data currently cached, rather than the data missing from the cache. Thus there is no need to compute query negations when adding results to the cache. Combined with a partitioning of the possible element content into a limited number of predefined domain ranges (e.g., fixed intervals for numeric data in the documents), this avoids the exponential growth of the incomplete tree. Notice that although both systems store all elements on root-to-leaf paths to query matches in the cache, they still assume that the extensional part of the cache tree can be held in main memory. To cope with obviously resulting space limits and avoid cache overflows, Hristidis and Petropoulos opt for expelling selected cache contents, and also sketch a simple replacement strategy (*least-recently used*).

9.4.2 HLCaches

HLCaches by Marrón and Lausen [2002] is a cache for XPath queries. The *HLCaches* system uses an *LDAP* server as its storage back-end. *LDAP* (short for *Lightweight Directory Access Protocol*) is a net-

²Since queries do not contain descendant steps, query results comprise all nodes on a root-to-leaf path in the document tree. Adding such a result to the Incomplete Tree is therefore guaranteed to preserve its prefix property.

working protocol for accessing TCP/IP-based directory services [LDAP]. The *LDAP* server provides a query interface for simple navigation in the directory tree. HLCaches uses the *LDAP* infrastructure and query engine for storing and retrieving the contents of XML documents, as follows. Each XML element or text node is stored as a node in the *LDAP* directory tree. The *LDAP* server also maintains metadata including the distinct kinds of directory nodes and their nesting, which resembles the schema tree for XML data introduced before. This schema information is used for evaluating queries from scratch, but not for query comparison. An XPath query to be evaluated is first split into subqueries each of which entails a separate query against the *LDAP* directory. During the creation of *LDAP* queries, XPath navigation patterns in the XML document tree translate to navigation patterns in the *LDAP* directory tree. Note, however, that the *LDAP* query language described by Marrón and Lausen does not support all XPath axes. In fact, HLCaches seems to be restricted to XPath queries involving only the *child* and *descendant* axes as well as their inverse relations.

Special *LDAP* nodes are reserved for caching processed queries and their results. The query cache contains for each evaluated subquery the corresponding XPath expression (represented both as a string and a hash code) as well as the set of elements matching that subquery. Each result element is stored together with its context node. This allows for the following combined intensional and extensional query comparison. A new XPath query is decomposed into subqueries whose hash codes are looked up in the cache part of the *LDAP* hierarchy. Subqueries are normalized before the look-up to capture XPath-specific syntactic variants such as inverse axes, etc. For each cached subquery whose intension (i.e., XPath expression) is equivalent to a new subquery, the sets of their context nodes are compared to determine whether the two subqueries are equivalent or overlapping. In particular, if the cached set of context nodes includes the set of context nodes of the new subquery, then the latter can be evaluated entirely from cache contents.

Notice that query containment detection is limited to those subqueries that have been processed in the same form before (modulo syntactic variation). By contrast, cached subqueries that are strictly more general than a given new subquery are not recognized as reusable. This might seriously limit the effectiveness of the cache (no experimental results for HLCaches are given in the paper). Moreover, Marrón and Lausen do not explain how to combine cached queries that only partially contain the new query with other overlapping cache contents or with fresh results retrieved directly from the documents. In particular, the related issues of duplicate elimination and integration with the evaluation from scratch are not covered. Finally, no strategy is given for decomposing new or cached queries to be looked up or stored in the cache.

9.4.3 Prefix-Based Containment

Another XPath cache with a string-based look-up procedure was proposed by Mandhani and Suciu in 2005. Their approach covers a subset of tree-shaped XPath queries (in particular, only *child* and *descendant* steps are allowed, and value joins are prohibited). The system requires a hybrid storage back-end to combine relational data and XML fragments representing the cache contents. The cache consists of a number of tables containing both the query intensions (as strings) and query extensions (as XML fragments). Similar to HLCaches, any new query Q^n is split into subqueries which are then normalized and represented as slightly modified XPath strings. These strings are looked up in the cache in order to find a cached query Q^c that contains Q^n . The main difference to HLCaches lies in the way queries are decomposed and compared. In particular, the technique put forward by Mandhani and Suciu does not only retrieve identical subqueries in the cache, but also benefits from certain cached queries that are strictly more general than the query to be evaluated. Note, however, that while the system offers limited support for checking the containment of numeric value predicates in queries, keyword constraints cannot be compared during the cache look-up. Besides, partial containment in cached queries is not exploited: any given query can reuse results from at most one query in the cache, and there is no way to complete such cached results with other results computed from scratch.

Mandhani and Suciu focus on a special case of full query containment that we refer to as *Prefix-Based Containment* in the sequel. Given any tree-shaped XPath query Q , let the *query axis* of Q be the path from the root of the query tree down to the unique XPath result node in Q . A *query prefix* of Q is obtained by choosing any node on the query axis of Q as a *split node* and removing all nodes below it. Obviously there are as many distinct prefixes of Q as there are query axis nodes. The unique prefix of Q that is obtained by choosing Q 's result node as split node is said to be *maximal* because it includes the whole query axis of Q .

The following sufficient condition for the containment of a new query Q^n in a cached query Q^c is proved by Mandhani and Suciu: Q^c contains Q^n if a split node q^n on the query axis of Q^n can be chosen such that (1) the resulting prefix of Q^n is equivalent to the maximal prefix of Q^c , and (2) each predicate below the result node of Q^c is mirrored in Q^n , by a predicate below q^n that is either equivalent or more selective. Intuitively, this means that Q^c and Q^n are equivalent down to the level of the split node q^n in Q^n , and Q^c is no more restrictive than Q^n in the remaining query parts. The main problems are to choose a suitable split node in Q^n such that there are cached queries satisfying the first condition, and then to check efficiently whether they also fulfill the second condition.

The tables used for storing any cached query Q^c include columns for the maximal prefix of Q^c and for the set of predicates below the result node in Q^c . These subqueries of Q^c are stored as strings, after some normalization intended to unify XPath-specific syntactic variations. An index on the prefix column enables the fast selection of cached queries with a particular maximal prefix. When a new query Q^n arrives, first its maximal prefix is looked up in the cache (i.e., the result node of Q^n is chosen as split node q^n in the beginning). Every cached query with the same maximal prefix (after normalization) is then examined to determine whether each of the predicates below its result node has a counterpart in Q^n that is either equivalent or more selective. To avoid the expensive computation of tree pattern embeddings between predicates in Q^c and Q^n , Mandhani and Suciu suggest creating certain generalizations of the predicates below q^n in Q^n as soon as the split node is chosen. With this sort of query expansion, the above condition on Q^c 's predicates is easy to check: each of the result-node predicates in Q^c must appear in the expanded set of predicates below q^n in Q^n . Note that for efficiency reasons only a limited number of generalized predicates can be created, which might cause reusable queries in the cache to be overlooked. Also there is no index on the cache table for supporting the predicate check.

The first cached query that is proven to contain Q^n in this way is used for answering Q^n incrementally. If the predicates in Q^n are strictly more selective than those in Q^c , then the cached result of Q^c is restricted accordingly. By contrast, if no reusable query could be found in the cache, the next higher node on the query axis of Q^n is chosen as split node, and the query expansion and look-up recommence. This bottom-up iteration through Q^n 's axis nodes stops when either a containing query is found in the cache or the query comparison eventually fails for the root of Q^n (in which case Q^n must be evaluated from scratch). The search for a good split node in Q^n is performed bottom-up because a greater speedup is expected when Q^c and Q^n share a longer prefix, since fewer predicates in Q^n need to be evaluated on a smaller cached result. In the experiments an average speedup factor of 2.6 was obtained compared to the evaluation from scratch, for a large query workload including many queries with locality (which is favourable to incremental processing). As a small caveat, Mandhani and Suciu mention that the results also reflect the locality of disk pages fetched before the actual experiments, when the cache is created. This could mean that for systems with a persistent cache, where no disk pages are fetched during start-up, the absolute response times are longer and hence the speedup factor is smaller.

9.4.4 ACE-XQ

The ACE-XQ system by Chen et al. [2002; 2003; 2004; 2005] (formerly XCache [Chen et al. 2002]) answers XQuery expressions using materialized views. A *containment mapping* is established between the variables in a new XQuery expression and a cached one. To this end, queries to be cached are normalized and then described in terms of the variables occurring in the RETURN clause or elsewhere in the query, the path expression connecting them and conditions such as keyword constraints. To benefit from cached results, variables in the new query may only involve stricter conditions on structure or content than their counterparts in the cached query. In other words, only full query containment is exploited (although Chen and Rundensteiner [2005; 2002] report on experimental results for overlapping queries, which are not explained). Recent work by Chen and Rundensteiner [2005] elaborates on XQuery containment in the presence of *hierarchical multi-valued dependencies* among variables, which can define different groupings of the same data. Cache replacement strategies have been studied in the 2004 paper by Chen et al. However, the problem of how to choose the best cached query for containment mapping remains open.

9.4.5 Caching Based on Access Frequencies

Shah and Chirkova [2003] address the materialization of XML views on relational data accessed through an XQuery interface. Unlike all other approaches mentioned so far, they assume a constant *query workload*, i.e., a fixed set of queries repeatedly evaluated by the system. The results of the most frequent queries are stored as XML text fragments in a cache relation. To this end, *access counters* record which tuples are used most often in query processing. From time to time, data in other frequently used tuples is added to the materialized XML fragments, provided they are related to the cache contents in some way (e.g., because they contain the same keywords). The authors claim that the data to be cached is chosen by a learning algorithm, although the choice is made primarily based on the value of the access counters, and there are no adaptive parameters that change over time like, say, weights in neural networks or other machine learning paradigms. Also there is no training phase in the ordinary sense, where some sort of feedback loop leads to a stepwise self-adjustment of the adaptive system parameters. The only adaptive parameter is the threshold for selecting data that is access sufficiently often to be cached. However, the value of this threshold is determined once empirically and then stays fixed.

The caching scheme proposed by Shah and Chirkova has a number of disadvantages. First, the cache creation and maintenance requires much manual intervention by the database administrator. In particular, the choices to be made by the administrator include a value for the access count threshold, queries for testing it, the relations to be monitored, and a suitable schema for the cached data. Furthermore, since query results are cached as strings representing XML fragments, they cannot benefit from XML indexing techniques nor can they serve as partial results to new queries, their structure being invisible to the relational query processor. This makes it hard to exploit query containment and overlap in many cases. In fact, the experiments reported by the authors mostly show that retrieving XML results materialized after a previous run of the same query takes less time than computing the answer again from scratch, which is trivial.

9.4.6 Argos

The *Argos* system by Quan et al. [2000] addresses incremental query evaluation from a different point of view. Targeting view maintenance for dynamic resources, it assumes a fixed query workload known in advance that is evaluated repeatedly against documents which change over time. Contrast this to the approaches described before, which are designed to handle previously unseen queries against a static document collection. Covering a fragment of tree-shaped XQL queries, *Argos* retrieves cached results to queries that have been processed before. During an initialization phase, all queries are evaluated once with their keywords removed in order to produce materialized views on the current structural matches in the documents. Those matches that also satisfy the keyword conditions are flagged using truth values. The flags guide the both the cache look-up and the insertion of new data into the collection. Whenever the textual document content changes, the flags for all affected structural matches are updated accordingly. However, to cope with changes to the structure of the documents queries must be reevaluated. Note that query overlap and partial query evaluation using materialized views are not examined. Besides, Quan et al. only evaluate the proposed update algorithm, while the look-up efficiency is ignored in their experiments.

9.5 Summary and Discussion

The goal of this chapter was to provide an informative, albeit non-exhaustive, overview of different caching techniques for the incremental evaluation of XML queries. The various contributions reviewed above differ in their way of representing queries, documents (possibly including the document schema) and cached results; looking up reusable query results in the cache through query comparison; handling previously unseen queries and partially relevant cached results; combining cache contents with each other and with data retrieved directly from the documents; maintaining and cleaning up the cache over time; and evaluating the practical benefit of the system on a real-world scale. The rest of this section highlights problems and potential optimizations that have been largely ignored so far. Some of these issues will be reconsidered and addressed in the next chapter where we present a novel approach to the efficient incremental processing of XML queries, based on the contributions introduced in previous parts of this work.

Extensional cache look-up and query comparison. Most of the caching approaches described above either completely ignore the document schema or consider it only for query evaluation from scratch, but not for retrieving relevant cache contents. A given pair of a cached and a new query is typically compared on a purely intensional basis. However, the question to what extent the results of two queries in a given document collection actually overlap or even contain one another often cannot be answered from their intensions alone. In such cases purely intensional approaches, failing to recognize valuable cache contents, needlessly repeat the (possibly expensive) evaluation from scratch. In terms of the Three-Level Model of XML Retrieval (see Figure 2.3 on page 13), it may be beneficial to compare queries not only on the query level, but also on the schema level as an approximate view of the query extension that can be accessed efficiently.

As mentioned before, some authors have investigated the use of DTDs for inferring structural constraints underlying the documents, which allow to detect otherwise invisible query overlap or containment. However, query comparison in the presence of DTDs has been mostly addressed from a theoretical point of view in order to derive complexity bounds, while practical issues regarding efficient data structures and algorithms are often ignored. The Incomplete Trees used by Abiteboul et al. as well as Hristidis and Petropoulos (see Section 9.4.1) go in this direction, but lack indexing support for instantaneous access to relevant parts of the DTD tree. Besides, prescriptive schema specifications such as DTDs are usually designed to capture a larger class of documents. Therefore they tend to be more general than descriptive schemata like the CADG, which mirror the current structure of the documents more closely and thus may reveal additional constraints to be exploited in the comparison. In the next chapter we show how to make use of CADG-based schema information and a suitable index structure for quick access to potentially reusable query results in the cache.

Reuse of overlapping query results. The systems reviewed above exploit query containment to a varying extent. While some only benefit from the cache when exactly the same query is evaluated repeatedly, others take advantage of cached queries that are strictly more general than the current query to be answered. For instance, HLCaches makes use of cached queries containing only a subquery of the new query to be evaluated, by looking up subqueries independently and then restricting their results to satisfy the remaining constraints. However, while full containment can be handled this way, cached queries that deliver only part of the final answer cannot be exploited. In fact only few systems take advantage from partial query containment or overlap. The Incomplete Tree approach seems to generate suitable remainder queries for completing partial query results retrieved from match-containing queries in the cache, at the expense of a potential cache blow-up. By contrast, it remains unclear whether the proposed solution with so-called local queries can really add previously unseen matches to the query result, which is mandatory for exploiting cached queries that indeed overlap with the new query, but do not match-contain it.

One problem that most systems would need to solve before handling such queries is the need for an integrated query evaluation from cache and from scratch. (Abiteboul et al. [2001b] regard this as a kind of mediation between the domain of the cache and the domain of the original documents.) The next chapter presents a way to reuse cached query results that (perhaps partially) cover some matches to a given new query, and to compute all missing matches (and missing parts of incomplete matches) from scratch in an integrated retrieval process.

Reuse of intermediate query results. All caching techniques discussed so far assume that only final query results are stored in the cache. However, XML queries (notably those with branching path patterns) are typically evaluated not in a single operation, but rather stepwise by composing multiple intermediate results that have been obtained for smaller subqueries. For instance, to answer the XPath query $Q_1 = //person[name = "Lee"]//edu$, some systems would first retrieve two node sets, namely, the set of `person` elements whose `name` child contains an occurrence of the keyword “Lee” and the set of all `edu` nodes with a `person` ancestor. In a second step a structural join of the two sets would produce the final XPath result, i.e., those `edu` nodes that satisfy all query constraints. Let us assume that the final query result is cached by any of the aforementioned systems, while the two intermediate result sets are discarded. Now suppose that the system is given two new queries for incremental evaluation: $Q_2 = //person[name = "Lee"]$ and $Q_3 = //person//edu$. Obviously neither of these queries is con-

tained in the cached query. In fact, the results of Q_1 and Q_2 are disjoint (assuming XPath result node semantics), while there is overlap between Q_1 and Q_3 . In any case, a system that only exploits query containment fails to answer the new queries from the cache, although their results were readily available to the system during the evaluation of the cached query.

Two conclusions can be drawn from the phenomenon just described. First, by examining the intermediate results produced during the evaluation of a cached query one may discover query containment or overlap with new queries, even though the final result in the cache is too restrictive to answer these queries. Consequently, caching intermediate results can increase the effectiveness of the cache, allowing to exploit query containment or overlap that exists only up to a certain step during the evaluation of the cached query.

Second, the benefit of caching intermediate query results depends not only on the query workload, but also on the planning strategies that were in effect when evaluating the queries that are now in the cache. For instance, an alternative evaluation plan for the sample query Q_1 above would be to retrieve three node sets in the first place, namely, `person` nodes, `name` nodes containing the keyword “*Lee*”, and `edu` nodes. Two structural joins would then be needed to produce the final answer to be cached. If the three node sets were cached as intermediate results, both Q_2 and Q_3 above could be answered from the cache, although some extra effort would be needed to match the `person` constraint. By contrast, the result of joining the `person` and `name` node sets during the evaluation of Q_1 could also be kept in the cache, which would immediately answer Q_2 .

We are not aware of any systems that store both final and intermediate results in the cache. Consequently, the impact of query planning on the cache contents has been completely ignored so far. The next chapter presents a new approach to caching both intermediate and final results, as well as an experimental quantification of the resulting impact on the effectiveness of the cache. There we will also discuss related issues such as the question which intermediate results to cache.

Choice of cache contents to be reused. There are only few approaches where all query results to be cached are merged into a single data structure, such as the Incomplete Tree used by Abiteboul et al. [2001b] (see Section 9.4.1). Most other systems store the results of distinct queries separately in the cache. If the look-up for a new query Q^n retrieves multiple cached queries that are not equivalent to Q^n but can be reused, these systems face the question which cached query to choose in order to minimize the computation and I/O required to answer Q^n incrementally. This problem of choosing the best among several reusable queries in the cache is frequently ignored in the literature. The simple strategy proposed by Mandhani and Suciu [2005] for Prefix-Based Containment (see Section 9.4.3) is based on a purely intensional comparison of the queries, ignoring extensional aspects such as the selectivity of query constraints in Q^n that remain to be processed. However, just like the extensional comparison of queries based on schema information can help to detect query containment or overlap (see above), the choice of cached results to be reused can benefit from access to query extensions, too. The next chapter explains a way to combine intensional and extensional information in order to make a good choice.

Choice of query results to be cached. A general problem related to incremental query processing is the question which query results should go into the cache or be removed from it at a given point in time. In fact this question splits up into several subproblems that we only mention here briefly. The first question is how to decide whether a given query is worth caching. For instance, a very unselective query with a huge result might be a bad candidate because it occupies much space in the cache while hardly facilitating the incremental evaluation of more specific queries to come. Second, if we assume a fixed size limit of the cache, the problem of a cache overflow arises. Here the question is which queries to keep in the cache and which to discard (if any). An imminent cache overflow may also affect the selection criteria for new queries to be cached, thus relating back to the first problem. Finally, for some applications it might be useful to set up a functional cache at system start-up, rather than to begin with an empty cache. Here the problem is to generate appropriate cache contents before the actual queries come in.

The RCADG Cache for XML Queries and Results

10.1 Overview

With the work on BIRD, CADG and RCADG that has been presented before, we have developed and combined different contributions to making XML retrieval more efficient. An underlying assumption was that each query to be answered would be evaluated “from scratch”, i.e., regardless of the answers to other queries processed earlier, although such previous results might contain some or even all matches to the new query being processed. In this chapter we present the *RCADG Cache*, an XML query cache that allows for efficient and scalable incremental query processing with the RCADG.

The benefit of caching query results for future use was recognized long before the advent of XML. Experience with view-based query answering on relational data [Halevy 2001] shows that the incremental evaluation based on cached query results can substantially improve the performance of RDBSs compared to the evaluation from scratch. In fact, the main problems related to caching are similar both for relational and XML data: (1) to determine which cache entries contain (part of) the desired data, and (2) to choose those cache entries from which the final result can be obtained with the smallest computational and I/O effort. Yet for accelerating XML search it is not enough to apply techniques developed for view-based query answering in RDBSs to XML data stored as tuples. An explicit representation of the hierarchical structure of the data is needed to decide if and how some cached query results can contribute to answering a given new query (except in the trivial case where the same query is asked repeatedly).

The preceding chapter has reviewed a number of caching techniques designed specifically for the incremental retrieval in XML documents. Among these approaches, there are few RDBS-based systems. Prior work on XML query caching has focussed mostly on native or hybrid retrieval engines. Therefore the idea of extending the RCADG – which owes much of its efficiency and scalability to its entirely relational nature – to incremental query processing was a particularly interesting challenge in its own right. But besides that, the RCADG Cache also addresses some of the other issues mentioned before that earlier approaches have left open.

Most notably, we present a way to take advantage of schema information provided by structural summaries for finding reusable queries in the cache that would be overlooked by purely intensional approaches, and for retrieving cache contents that overlap with the desired answer. More precisely, the *schema hits* that we compute when evaluating queries with the RCADG also help to detect query containment and overlap efficiently in a combined intensional and extensional comparison procedure. Here we exploit the fact that *even when two queries cannot be compared directly, their schema hits can*. The schema hits of a cached query Q are held in a main-memory index structure for fast cache look-ups without access to the actual query results on disk. Comparing the schema hits of Q' and Q may reveal that while the matches to some schema hits to Q' must be computed from scratch, others are (perhaps partially) contained in the match set of a schema hit to Q ; in other words, Q overlaps with Q' . Similarly, if the matches to all schema hits of Q' are fully contained in Q 's match sets, then Q contains Q' . After the query look-up and comparison, an integrated evaluation process retrieves part of the final result of Q' from one or more cached queries, if possible, and the rest from scratch.

The different notions of query containment and overlap have been formally defined in the preceding chapter (see Section 9.2). Recall that while results of a cached query Q containing a new query Q^n may only need to be purged of false positives with respect to Q^n , exploiting query overlap or partial containment is more challenging because it allows only for an incomplete evaluation of Q^n to be finished in following steps, perhaps by accessing the full data set. As shown in Figure 9.1 *a.–c.* on page 131, the result of an overlapping query Q may be incomplete in two ways: not necessarily all parts of Q^n are matched in Q , and also entire hits can be missing (which might be obtained from other cached queries, though). Unlike prior work addressing only query containment, where all desired data is subsumed by the result of a single cached query, we consider the more general overlap problem because (1) completing partial and retrieving missing matches is usually still faster than answering Q^n from scratch; (2) the cached part of the result is quickly available while the evaluation of the missing part is going on in the background; (3) when performing top- k search, those matches retrieved in the cache may even suffice to fulfill the request. These advantages can be particularly rewarding in the interactive retrieval scenario that motivated this work (see Chapter 1).

The RCADG Cache also addresses the open question of how to benefit from intermediate results computed during the evaluation of queries to be cached. We have observed that all incremental approaches we know of restrict themselves to caching merely final query results, despite the fact that even when the final result to a cached query Q is too restricted and hence useless for answering Q^n , a partial evaluation of Q may yield full or partial matches to Q^n . As a matter of fact, intermediate results are sometimes much more likely to overlap with subsequent queries (for an example see Section 10.3 below). Therefore the RCADG Cache also stores intermediate results obtained during the evaluation of cached queries. The intermediate results tables produced by the RCADG (see Chapter 8) conveniently provide the query processor with multiple “snapshots” of a query result as it evolved during the stepwise evaluation process. The information which snapshots (i.e., intermediate or final results) are available for a particular query in the cache is derived from the query plan that was used to compute them. Details about the underlying query plan are kept as annotations to the schema hits in the main-memory part of the cache. One problem besides efficient cache look-ups is to avoid a cache blow-up in space, due to the rich information about query intensions, extensions and plans in the cache.

Finally, we consider a preliminary strategy for choosing the best among several reusable queries in the cache. Based on the query plans for the candidate queries, alternative plans are deduced for incrementally answering the current query from the respective results in the cache. These plans are then compared in terms of their execution cost, in order to exploit possibly the most useful cache contents.

Before explaining the nuts and bolts of incremental XML query processing with the RCADG Cache, the next two sections present a couple of examples that illustrate the general ideas behind our approach. Section 10.4 then explains to what extent the RCADG Cache takes advantage of query containment and overlap. In Section 10.5 all essential data structures and algorithms of the cache are presented in detail. Section 10.6 reports on our experimental evaluation of the RCADG Cache. The rest of the chapter highlights differences to other approaches as well as open issues and possible optimizations.

10.2 Schema Information in the RCADG Cache

Schema information is useful for incremental query processing because it helps to detect query overlap or partial containment for queries that are hard to compare on a purely intensional basis. For instance, consider the three queries in Figure 10.1 *a.–c.* on page 143 which represent the intensional viewpoint, depicting exactly the information that is visible on the query level. For the sake of the example, assume that the final results of the queries Q' and Q (*a.*, *b.*) have already been retrieved and stored in the query cache (ignoring its exact structure for the moment). Notice that the third query Q^n (*c.*) cannot be proved to be contained in any of the cached queries from the intensions alone: the keyword constraints “*Lee*” and “*female*” make Q^n more restrictive than Q and Q' , but at the same time the tag disjunction $\text{gender} \vee \text{sex}$ is less restrictive. Thus we cannot decide whether the three result sets overlap, nor retrieve exactly the intersection of Q^n with Q or Q' , unless we compare the actual results in the documents. Since this would require Q^n to be evaluated from scratch, the cache contents seem useless for answering Q^n . However, below we show how to translate the intensions of all three queries to extensional constraints on the schema level, which are then compared in order to obtain part of the answer to Q^n from the cache at low computational

and I/O cost.

In many situations the schema information is indispensable for exploiting cache contents. For a cached query Q^c and a new query Q^n whose intensions tell nothing about containment or overlap, schema hits may show whether Q^c nevertheless contains Q^n , or else which parts of $ans(Q^n)$ are missing in $ans(Q^c)$ and would need to be retrieved from other cached results or from the documents. Typical cases include the following:

- Q^c has a *Parent* constraint where Q^n allows *Parent*^{*} (similar for *Child* and/or different proximity)
- Q^c has a specific tag, type or level constraint that is missing in Q^n
- Q^c has specific tag constraints whereas Q^n accepts the disjunction of a superset of these tags
- any combination of the above

As mentioned above, descriptive schemata, as up-to-date summaries of the current document structure, often allow to detect more of the reusable cache contents than prescriptive schemata, which tend to be too general. A key concept in comparing queries on the schema level are *S*-constraints and *D*-constraints (see Definitions 2.7 and 2.8 on page 12). Recall from Chapter 8 that a given query Q^n is evaluated (from scratch) with the RCADG in two phases: during schema matching, we match the *S*-constraints in Q^n against the schema tree, which produces a set of schema hits. Then during document matching we successively retrieve the occurrences of these schema hits while matching Q^n 's *D*-constraints in an interleaved process. To rephrase the caching problem, we would like to reuse (maybe partial) matches to (at least some) schema hits from cached queries with constraints similar to Q^n , and match only the missing constraints in Q^n against them. *S*-constraints play an important role in efficiently finding reusable queries in the cache. In the sequel we assume that every query has at least one binary *S*-constraint (caching queries without *Parent* and *Child* edges with the RCADG Cache is discussed in Section 10.8).

10.2.1 A Simple Example

First consider a cached query Q^c that has exactly the same structure as Q^n in Figure 10.1 c. on page 143, but lacks the keyword constraints. Clearly Q^c and Q^n have identical *S*-constraints and hence the same two schema hits $\chi_1^{Q^c} = \chi_1^{Q^n}$ and $\chi_2^{Q^c} = \chi_2^{Q^n}$ (see Figures 2.1 f., g. on page 8), but possibly different result sets. To decide whether Q^c overlaps with Q^n , we obviously need to compare those *D*-constraints in both queries that correspond to each other. In this simple example the correspondence is easy to spot because Q^c and Q^n are isomorphic. More involved cases are discussed below. The general idea is to compare the (extensional) schema hits matching both queries along with their (intensional) query constraints. In what we call *schematization*, all unary and binary *D*-constraints in a query are applied to those nodes of a particular schema hit which match the query nodes involved in these constraints. Intuitively, each query node is “replaced” with the corresponding node in the schema hit. In the sequel, let $Q \downarrow \chi^Q$ denote the schematization of a given query, Q , with one of its schema hits, χ^Q . For instance, Figures 10.1 f., g. depict $Q^n \downarrow \chi_1^{Q^n}$ and $Q^n \downarrow \chi_2^{Q^n}$, i.e., the schematizations of Q^n with $\chi_1^{Q^n}$ and $\chi_2^{Q^n}$, respectively. For $\chi_1^{Q^n}$, e.g., the schematized *D*-constraints are *Contains*_{“Lee”}(#2), *Contains*_{“female”}(#5), *Parent*(#2,#1) and *Parent*^{*}(#5,#1) (the inversion of the binary constraints is explained later). Schematizing Q^c with $\chi_1^{Q^c}$ yields the same result, except that the keyword constraints are missing.

The schematization of D-constraints tells us which parts of Q^n and Q^c must be reconciled: Q^c and Q^n overlap with respect to $\chi_1^{Q^n}$ and $\chi_1^{Q^c}$ if the schematized *D*-constraints that we get for Q^c are no more restrictive than those obtained for Q^n on the same schema nodes. For the binary constraints in Q^c and Q^n , this is trivial since they are equal (*Parent*(#2,#1) and *Parent*^{*}(#5,#1) in either query). However, the condition would also be satisfied, say, if we had a binary *D*-constraint *NextElt*₁⁺(#2,#5) in $Q^c \downarrow \chi_1^{Q^c}$ and a corresponding *D*-constraint *NextSib*₁⁺(#2,#5) or *PrevSib*₂⁺(#5,#2) in $Q^n \downarrow \chi_1^{Q^n}$ (now shown in Figure 10.1). In our example, the overlap test also succeeds for the unary constraints in both queries because the empty keyword constraint attached to node #2 in $Q^c \downarrow \chi_1^{Q^c}$ is obviously less restrictive than the keyword constraint *Contains*_{“Lee”}(#2) in $Q^n \downarrow \chi_1^{Q^n}$, and likewise for node #5. The test would fail, e.g., if $Q^c \downarrow \chi_1^{Q^c}$ specified a single keyword other than “Lee” for node #2 or an additional binary constraint not mirrored in $Q^n \downarrow \chi_1^{Q^n}$.

Note that even if the test fails, the answers to Q^c and Q^n might happen to overlap or even be exactly equal for the particular document collection in question. However, such coincidental overlap cannot be detected without access to the document level. Hence our approach is necessarily incomplete, just like the purely intensional techniques developed earlier.

In our simple example, the D -constraints in Q^c are *necessary conditions* for any match to Q^n because $Q^c \supset Q^n$, i.e., Q^c contains Q^n as defined in the preceding chapter (see Definition 9.4 on page 131). In fact, we can even make stronger statements with respect to the individual schema hits of Q^c and Q^n , namely, $ans(\chi_1^{Q^c}) \supset ans(\chi_1^{Q^n})$ and $ans(\chi_2^{Q^c}) \supset ans(\chi_2^{Q^n})$. Note that such containment between the matches to particular schema hits may hold even when the two queries do not fully contain one another. For instance, imagine that Q^c is modified so as to accept only elements with tag `sex` as matches to q_3^c , but not those with tag `gender`. Clearly there is no full containment $Q^c \supset Q^n$ under these circumstances: while Q^c indeed node-contains Q^n , as required in Definition 9.4 on page 131, the matches in $ans(\chi_2^{Q^n})$ are not mirrored in $ans(Q^c)$, i.e., Q^c does not match-contain Q^n . But still we have $ans(\chi_1^{Q^c}) \supset ans(\chi_1^{Q^n})$, so that at least a part of Q^n 's result can be retrieved in the cache. Thus the schematization sometimes permits to reuse cache contents (in this case, the cached matches to $\chi_1^{Q^c}$) that would otherwise be ignored.

Given that $ans(\chi_1^{Q^c}) \supset ans(\chi_1^{Q^n})$, the *sufficient conditions* needed to retrieve exactly $ans(\chi_1^{Q^n})$ follow from the comparison of the schematized D -constraints in $Q^c \downarrow \chi_1^{Q^c}$ and $Q^n \downarrow \chi_1^{Q^n}$. The goal is to create a *remainder query* [Hristidis and Petropoulos 2002] that returns $ans(\chi_1^{Q^n})$ based on $ans(\chi_1^{Q^c})$ that is stored in the cache, without repeating work that was done for before when evaluating Q^c . Here the remainder query for computing $ans(\chi_1^{Q^n})$ from $ans(\chi_1^{Q^c})$ consists simply of the keyword constraints *Contains_{“Lee”}* (#2) and *Contains_{“female”}* (#5). In other words, the matches to $\chi_1^{Q^n}$ are obtained by selecting those matches to $\chi_1^{Q^c}$ in the cache where “Lee” occurs in the `name` element and “female” occurs in the `sex` element. If the cached elements are stored with their textual contents, we save at least two accesses to the documents compared to evaluating Q^n from scratch. But even when the keyword constraints are checked against the documents (as it is the case for the RCADG Cache, see below), starting from a limited set of cached matches that already satisfy a certain number of query constraints (e.g., all binary constraints in Q^c) typically substantially reduces the evaluation cost in terms of CPU time and I/O operations. This is where the benefit of incremental query processing comes from.

In fact we do not require Q^c to node-contain Q^n in order to take advantage of cached matches to $\chi_1^{Q^c}$, as in the example above. Instead Q^n may well have some extra nodes not mirrored in Q^c whose matches can be fetched from the documents during the evaluation of the remainder query. While this does cause joins and possibly I/O, at least the cached matches to $\chi_1^{Q^c}$ tell us exactly where to find the missing data in the documents. For instance, suppose Q^n also included a *Parent* constraint to a fourth query node with tag `profile`, similar to Q' in Figure 10.1a. on the next page. Retrieving the missing `profile` element for every match to Q^c in the cache could be done very efficiently with the RCADG, given the set of these matches as a starting point. Supporting overlapping queries in this way makes the cache much more effective than other approaches that are restricted to full query containment or even equivalence (see Chapter 9). The definition of *schema-hit containment* below formally describes the degree of overlap supported by the RCADG Cache.

10.2.2 The General Case

The simple example above illustrates how the schematization of D -constraints reveals which constraints in Q^n and Q^c correspond and must be compared in their restrictiveness. While for isomorphic queries this is trivial, the real benefit of schematization shows when Q^n and Q^c are structurally different. Two problems must be solved here. First, we would like to be able to identify overlapping queries for Q^n in a (possibly large) number of queries in the cache, and second, we need a way to compare Q^n 's schema hits and cached schema hits that are not isomorphic.

To tackle the first problem, we also schematize the S -constraints in all queries to be cached or evaluated incrementally. *The schematization of S -constraints helps in locating cached queries that are potentially useful for evaluating Q^n .* Note that while this look-up technique turns out to be very effective and efficient, it cannot guarantee to produce only relevant candidates because even finding cached queries that overlap

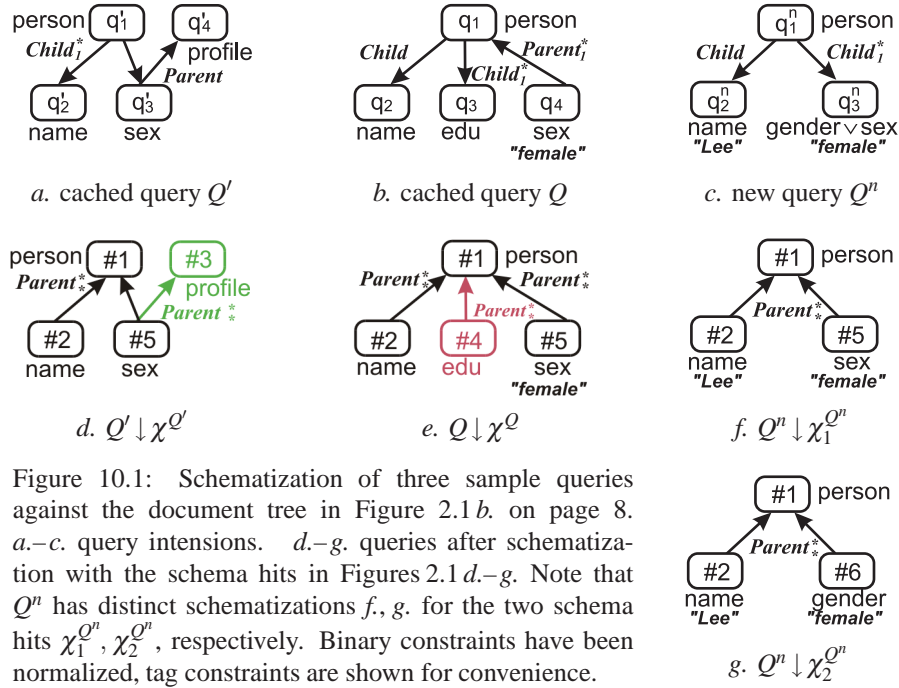


Figure 10.1: Schematization of three sample queries against the document tree in Figure 2.1 b. on page 8. a.–c. query intensions. d.–g. queries after schematization with the schema hits in Figures 2.1 d.–g. Note that Q^n has distinct schematizations f., g. for the two schema hits $\chi_1^{Q^n}$, $\chi_2^{Q^n}$, respectively. Binary constraints have been normalized, tag constraints are shown for convenience.

with Q^n would require access to the document level, let alone those that contain Q^n . For efficiency reasons, we simply look up all cached queries that share an edge with the new query after the schematization of S -constraints. For instance, consider the query Q^n in Figure 10.1 c. again. Schematizing the two binary S -constraints in Q^n , $Child'(q_1^n, q_2^n)$ and $Child'_1(q_1^n, q_3^n)$, as before yields $Child'(\#1, \#2)$, $Child'_1(\#1, \#5)$ for the schema hit $\chi_1^{Q^n}$ and $Child'(\#1, \#2)$, $Child'_1(\#1, \#6)$ for $\chi_2^{Q^n}$. Any schematized query with one of these edges in the cache is regarded as a candidate for the comparison of D -constraints, as described above.

However, there may be even more equally relevant queries in the cache that also include one of the S -constraints above, albeit in a syntactically different way. For instance, $Child'_1(\#1, \#5)$ is of course equivalent to $Parent'_1(\#5, \#1)$. Moreover, after schematization we can even treat $Parent'_i(\#5, \#1)$ and $Parent'^j(\#5, \#1)$ as interchangeable for any i, j because the vertical distance between the schema nodes $\#5$ and $\#1$ is fixed, so that it does not matter which proximity bounds were specified in the original query.¹ Therefore the schematized S -constraints in any query being added to or looked up in the cache are *normalized*, as follows: (1) every $Child'$ constraint is replaced with its unique equivalent $Parent'$ constraint; (2) all proximity bounds for $Parent'$ edges are replaced with the “*” symbol; (3) $Parent'$ becomes $Parent$ to prepare the subsequent comparison of D -constraints; (4) tag, type and level constraints are discarded, being unambiguous for schema nodes. In the case of $Q^n \downarrow \chi_1^{Q^n}$, this yields $Parent^*(\#2, \#1)$, $Parent^*(\#5, \#1)$ (see Figure 10.1 f.), whereas for $Q^n \downarrow \chi_2^{Q^n}$ we have $Parent^*(\#2, \#1)$, $Parent^*(\#6, \#1)$ (see Figure 10.1 g.).

Now assume that these constraints for Q^n are looked up in a cache that contains the results of the two queries Q' and Q shown in Figures 10.1 a. and b., respectively. As mentioned earlier (see Figures 2.1 d., e. on page 8), Q' and Q each have one schema hit ($\chi^{Q'}$ and χ^Q , respectively). The outcome of schematizing Q' with $\chi^{Q'}$, $Q' \downarrow \chi^{Q'}$, and Q with χ^Q , $Q \downarrow \chi^Q$, is shown in Figures 10.1 d. and e., respectively. The six binary S -constraints depicted there make up the schema-level contents of the cache (we ignore the cached query answers on the document level for the moment). Looking up $Parent^*(\#2, \#1)$, $Parent^*(\#5, \#1)$ and $Parent^*(\#6, \#1)$ for Q^n in the cache, we retrieve both $Q' \downarrow \chi^{Q'}$ and $Q \downarrow \chi^Q$ (each sharing two binary constraints with $Q^n \downarrow \chi_1^{Q^n}$ and one with $Q^n \downarrow \chi_2^{Q^n}$, see Figure 10.1). Now that we have found candidates for the incremental evaluation of Q^n , we need to check whether there is actually query containment or overlap of Q^n and Q' or Q . This is done by comparing the schematized D -constraints. The following definition

¹Note that proximity bounds may only be ignored when schematizing the vertical tree relations $Child$ and $Parent$.

captures a sufficient condition for query overlap that is exploited by the RCADG Cache:

Definition 10.1 (Schema-hit containment) *Let D be a document collection and let S be the schema tree for D . Besides, let χ^{Q^c} and χ^{Q^n} respectively be schema hits in S for a cached query Q^c and a new query Q^n against D . We say that χ^{Q^c} contains χ^{Q^n} , $\chi^{Q^c} \supset_s \chi^{Q^n}$, iff all of the following conditions are satisfied:*

1. $Q^c \downarrow \chi^{Q^c}$ either is a subgraph of $Q^n \downarrow \chi^{Q^n}$, or else contains no additional binary constraints that introduce a proper restriction of $\text{ans}(\chi^{Q^c})$ in D .
2. Given any D -constraint in $Q^c \downarrow \chi^{Q^c}$ that has a corresponding D -constraint in $Q^n \downarrow \chi^{Q^n}$, the former is at most as restrictive as the latter. \square

The first condition in Definition 10.1 explicitly states that certain additional constraints in Q^c which are not mirrored in Q^n may be ignored. For instance, the schematization of Q' with $\chi^{Q'}$ produces the binary constraint $\text{Parent}_*^*(\#5, \#3)$ (highlighted green in Figure 10.1 *d.* on the previous page) which is missing in $Q^n \downarrow \chi_1^{Q^n}$ (see Figure 10.1 *f.*). However, removing this constraint from Q' would not alter $\text{ans}(Q')$ because the ancestors of the schema node #5 are unambiguously fixed. Hence this additional constraint in Q' can be ignored. Since the second condition in Definition 10.1 is also satisfied, we have $\chi^{Q'} \supset_s \chi_1^{Q^n}$.

By contrast, the other cached query, Q , contains a binary constraint that must not be ignored. The $\text{Parent}_*^*(\#4, \#1)$ edge highlighted red in Figure 10.1 *e.*, which is not mirrored in Figure 10.1 *f.*, indeed makes Q more restrictive: compare Figures 2.1 *e., f.* on page 8 to verify that the match a_3 in $\text{ans}(\chi_1^{Q^n})$ is not part of $\text{ans}(\chi^Q)$, because of this edge. Hence χ^Q does not contain $\chi_1^{Q^n}$ in the sense of Definition 10.1. Likewise, since the $\text{Parent}_*^*(\#5, \#1)$ constraint in $Q' \downarrow \chi^{Q'}$ and $Q \downarrow \chi^Q$ (see Figures 10.1 *d., e.*) is missing in $Q^n \downarrow \chi_2^{Q^n}$ (see Figure 10.1 *g.*), the second schema hit $\chi_2^{Q^n}$ for Q^n is not contained in any cached schema hit. Therefore $\text{ans}(\chi_2^{Q^n})$ cannot be computed incrementally with the RCADG Cache. In this way we examine all schematized constraints in a cached query that are not mirrored in Q^n to decide whether the query can still contribute matches to Q^n . By contrast, extra constraints in the new query Q^n are simply added to the remainder query (see above). The second condition in Definition 10.1 is checked by comparing D -constraints as described before.²

Through schematization we learn that part of the answer to Q^n – namely, the matches to the first schema hit $\chi_1^{Q^n}$ – can be obtained incrementally from Q' by matching the keyword constraints for “*Lee*” and “*female*” against $\text{ans}(\chi^{Q'})$ in the cache. By contrast, the rest of the answer to Q^n – namely, the matches to the second schema hit $\chi_2^{Q^n}$ – must be retrieved from scratch. Again, this distinction would be impossible on the intensional level and even if a DTD were given.

10.3 Intermediate Query Results in the RCADG Cache

The examples above assumed that only the final results of the two queries Q and Q' are stored in the cache. However, the RCADG evaluation algorithm matches D -constraints step-wise, not all at once. Recall from Chapter 8 that the result of every step in a query plan is stored in a separate table in the RDBS. Caching these intermediate results can further increase the effectiveness of the cache when partial matches to a cached query happen to coincide with results for the new query Q^n .

For instance, assume that the D -constraints in the query Q from Figure 10.1 *b.* on the previous page have been matched according to the query plan P^Q shown in Figures 8.11 *a., b.* on page 109. Recall that P^Q comprises three steps: in the first two steps, s_1^Q and s_2^Q , the D -constraints $\text{Child}(q_2, q_1)$, $\text{Parent}_1^*(q_4, q_1)$ and $\text{Contains}_{\text{“female”}}(q_4)$ are matched, producing as an intermediate result the two matches a_2, a_3 . This intermediate result after step s_2^Q is symbolized by the blue ellipse in Figure 10.2 on the facing page. Only in the third step, s_3^Q , the *edu* node q_3 is matched, causing a_3 to be discarded from the final answer to Q (grey ellipse in Figure 10.2). Thus before s_3^Q , all matches to $\chi_1^{Q^n}$ (namely, a_2 and a_3) can be obtained from

²Note that when looking up a new schematized query $Q^n \downarrow \chi^{Q^n}$ in the cache, every schematized query $Q^c \downarrow \chi^{Q^c}$ that is retrieved shares some binary S -constraints (i.e., Parent' or Child' edges) with $Q^n \downarrow \chi^{Q^n}$. For the corresponding binary D -constraints (i.e., Parent and Child edges) in $Q^c \downarrow \chi^{Q^c}$, the second condition in Definition 10.1 is trivially fulfilled. This means that in fact this condition need only be checked for additional constraints in $Q^c \downarrow \chi^{Q^c}$ that are not mirrored in $Q^n \downarrow \chi^{Q^n}$.

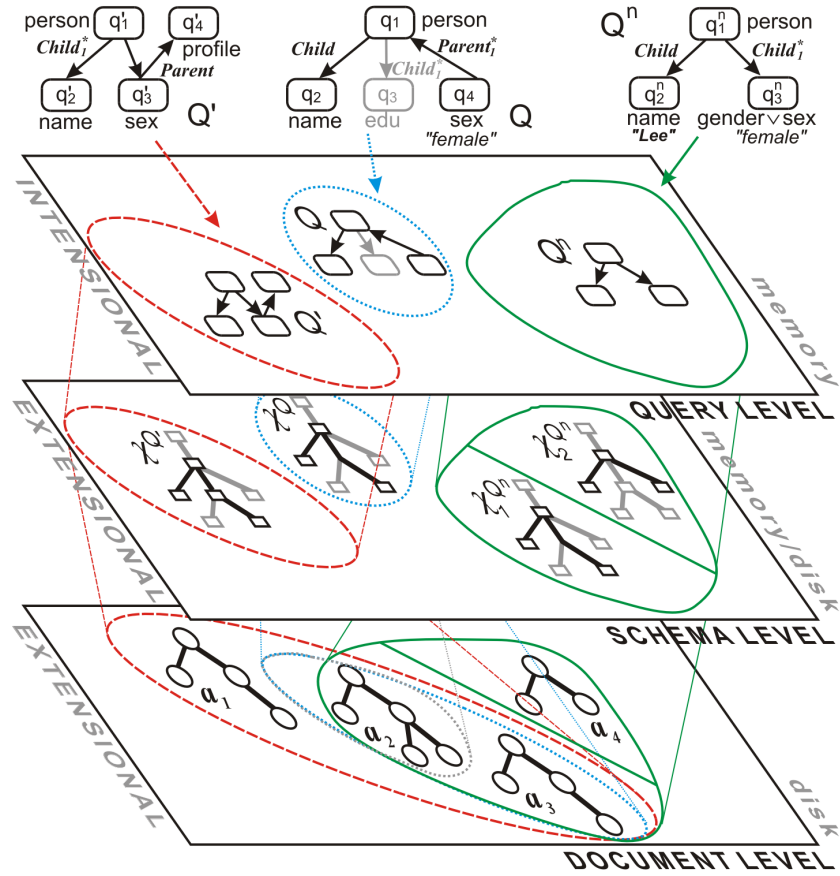


Figure 10.2: Containment and overlap of intermediate and final query results. An intermediate result for the query Q shown on top (center) contains two document matches, a_2 and a_3 (blue ellipse on the bottom level). In the last step of the evaluation of Q , the match a_3 is discarded from the final answer to Q (grey ellipse). Thus only the intermediate result for Q contains all matches to the first schema hit $\chi_1^{Q^n}$ of Q^n (left half of the green area on the bottom level), whereas Q 's final answer is too restrictive for Q^n .

the intermediate result of Q in the cache. (Observe that in Figure 10.2 the blue ellipse denoting $ans(\chi^Q)$ and the left part of the green area denoting $ans(\chi_1^{Q^n})$ contain the same set of matches, namely, a_2 and a_3). This makes Q a competitor of Q' in the contribution of cached query results for evaluating Q^n . Moreover, Q 's matches already satisfy the keyword constraint $Contains_{\text{female}}(\#5)$ that also appears in Q^n , but not Q' . Thus the intermediate result for Q in the cache even permits to answer Q^n more efficiently than when using Q' . The query planner described in Section 10.5 below therefore prefers Q to Q' , thus saving an access to the document level.

The example illustrates how the caching of intermediate results can improve both the effectiveness of the cache and the efficiency of the evaluation of remainder queries. Of course, this benefit comes at the expense of higher storage demands (see Section 10.6 for experimental results). In order to keep track of the intermediate results available for the query Q in the cache, we annotate each schematized D -constraint in Q with the unique step in the underlying query plan P^Q in which that constraint was matched during the evaluation of Q . This allows to determine the latest evaluation step in P^Q after which the cached schema hit χ^Q can be reused for answering the new query Q^n . Let $\llbracket \chi^Q \rrbracket_{s_2^Q}$ denote the part of Q after schematization with χ^Q that has been matched before or in step s_2^Q , i.e., everything but the highlighted portion of Figure 10.1 *e*. on page 143. In our example, we have $ans(\chi_1^{Q^n}) = ans(\llbracket \chi^Q \rrbracket_{s_2^Q})$, hence the intermediate result for Q obtained in the step s_2^Q can be used for answering part of Q^n .

10.4 Exploiting Containment and Overlap with the RCADG Cache

The preceding sections have sketched how to take advantage of queries in the RCADG Cache that overlap with or even contain a query Q^n to be evaluated incrementally. However, not all cached matches to such queries can be exploited in that way. In fact, our technique reuses the sets of matches to cached schema hits that contain all matches to a given schema hit χ^{Q^n} of Q^n , as defined above. In other words, $\text{ans}(\chi^{Q^n})$ cannot be obtained by combining sets of matches to multiple schema hits in the cache. Note, however, that matches to distinct schema hits of Q^n may well be obtained from different schema hits or even different queries in the cache. The following definition formally specifies which part of the answer to Q^n can be taken from the cache:

Definition 10.2 (RCADG Cache overlap) *Let D be a document collection, let S be the schema tree for D , and let C be an RCADG Cache built from queries against D . Besides, let Q^n be a query against D to be evaluated incrementally using the contents of C . Furthermore, for any query Q against D let X^Q be the set of schema hits of Q in S , and let $X^C = \bigcup_{Q^c \in C} X^{Q^c}$ be the set of all schema hits stored in the cache C . Finally, let $X = \{\chi^{Q^n} \in X^{Q^n} \mid \exists \chi^C \in X^C : \chi^C \supset_s \chi^{Q^n}\}$ be the set of schema hits of Q^n that are contained in any cached schema hit (see Definition 10.1 on page 144).*

The RCADG Cache overlap $\text{ans}_C(Q^n)$ for Q^n in C is defined as $\text{ans}_C(Q^n) = \bigcup_{\chi^{Q^n} \in X} \text{ans}(\chi^{Q^n})$. \square

The RCADG Cache overlap $\text{ans}_C(Q^n)$, $\text{ans}_C(Q^n) \subset \text{ans}(Q^n)$, denotes exactly the subset of document matches to Q^n that is taken from the cache (and possibly completed with data from the RCADG element table through remainder queries, as explained in the next section). Note that we can compute the union in the definition of $\text{ans}_C(Q^n)$ without checking for duplicate matches since the sets of matches to distinct schema hits of the same query are always disjoint, as observed in Section 2.3.

As can be seen from Definition 10.2, $\text{ans}_C(Q^n)$ subsumes all matches to those schema hits for Q^n that are contained in any cached schema hit. In other words, schema-hit containment is necessary for detecting and exploiting query overlap with the RCADG Cache. On the other hand, remember that schema-hit containment is only a sufficient condition for query overlap, i.e., there may be partial or even full containment between queries whose schema hits violate either condition in Definition 10.1 on page 144. This is because query overlap and containment are defined in terms of the document matches to the queries, but the schema-level view provided by the schema hits is only an approximation of the actual query extension on the document level. Hence the method to detect query overlap that we propose is necessarily incomplete with respect to the definitions on page 131. However, it is complete in the sense that a cache look-up for a given schematization of Q^n retrieves all schematizations of cached queries whose S - and D -constraints are equivalent or more general.

10.5 Incremental Query Evaluation with the RCADG Cache

This section presents the data structures and algorithms for incremental query evaluation with the RCADG Cache. The cache stores the queries, query plans and query results (both intermediate and final) obtained in the RCADG evaluation procedure that is described in Chapter 8. It consists of (1) a main-memory index structure C containing the intensions, schema-level extensions and evaluation plans of the cached queries, and (2) the document-level matches to all cached queries, which reside in result tables in the RDBS.

Each query to be cached is normalized and schematized as described above (see Section 10.2). The resulting graph is decomposed into its *schema edges* (the binary constraints between schema nodes in Figures 10.1 *d.–g.* on page 143), which are then stored in C . The same decomposition, applied to a schematized new query Q^n , produces the schema edges to be looked up in C . The look-up result is a mapping L^{Q^n} between schema edges created for Q^n and schema edges belonging to some cached queries, together with information about the query plans that were used to match the latter.

The schema edges retrieved in the cache tell us which cached queries and schema hits are candidates for (partially) answering Q^n . Every pair of schematizations of a cached query and Q^n that have a schema edge in common (like $Q \downarrow \chi^Q$ and $Q^n \downarrow \chi_1^{Q^n}$ in Figures 10.1 *e., f.*) must be tested for schema-hit containment, as sketched before (see Section 10.2.2). This way we compute a set H^{Q^n} of *cache hits* specifying (1) all

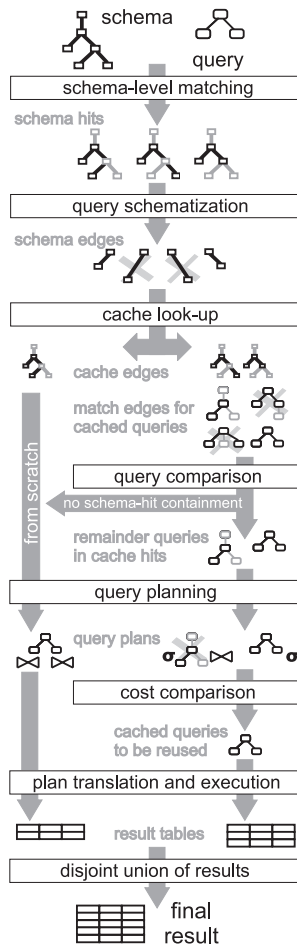


Figure 10.3: Integrated query evaluation with the RCADG and RCADG Cache.

cached schema hits that contain a schema hit for Q^n (see Section 10.2), (2) the evaluation steps providing the right “snapshots” of their sets of matches in the cache (see Section 10.3), and (3) the remainder queries for restricting and/or completing these cached results, depending on the additional constraints in Q^n . Once the cache hits are available, a query plan is created for each remainder query, telling us how to obtain the desired matches to Q^n based on the data specified by the corresponding cache hit and perhaps the full data set in the element table. Since the same subset of Q^n 's answer may be obtained from distinct cache hits, we propose a cost measure that indicates which of several alternative plans to execute in order to exploit the best-fitting cache hit. Owing to schema information, no duplicates need to be eliminated when merging results from distinct query plans.

Figure 10.3 illustrates the integrated query evaluation with the RCADG and RCADG Cache. Every query to be evaluated incrementally is first matched on the schema level. The resulting schematizations are then decomposed into schema edges, which are looked up in the main-memory part of the cache. Those schematizations for which no relevant cache contents could be retrieved are evaluated from scratch, as explained in Chapter 8. The others enter the query comparison phase, where query constraints are examined in order to decide for which schema hits matches are available in the cache. Again some schema hits may be scheduled for the evaluation from scratch. Query planning is essentially the same for both evaluation threads, except that for the incremental evaluation only remainder query plans are devised, not full evaluation plans. The aforementioned cost estimation selects the most promising among multiple alternative plans for matching a given schema hit from the cache. The routines for translating and executing query plans are identical. The disjoint union of all results for distinct schema hits yields the final result.

schema edge	cache edges
$Parent_*(\#5, \#1)$	$\mapsto \{ \langle Parent_*(q_4, q_1), s_1^Q, \{\chi^Q\} \rangle, \langle Parent_*(q'_3, q'_1), s_1^{Q'}, \{\chi^{Q'}\} \rangle \}$
$Parent_*(\#2, \#1)$	$\mapsto \{ \langle Parent_*(q_2, q_1), s_2^Q, \{\chi^Q\} \rangle, \langle Parent_*(q'_2, q'_1), s_2^{Q'}, \{\chi^{Q'}\} \rangle \}$
$Parent_*(\#5, \#3)$	$\mapsto \{ \langle Parent_*(q'_3, q'_4), s_1^{Q'}, \{\chi^{Q'}\} \rangle \}$
$Parent_*(\#4, \#1)$	$\mapsto \{ \langle Parent_*(q_3, q_1), s_3^Q, \{\chi^Q\} \rangle \}$

Figure 10.4: The RCADG Cache $C_{\{Q', Q\}}$ containing the schematized queries $Q' \downarrow \chi^{Q'}$ and $Q \downarrow \chi^Q$ from Figures 10.1 *d*, *e*. on page 143. Every distinct *schema edge*, i.e., a binary constraint from any of the schematized queries (left-hand side), is mapped to a set of *cache edges* for different queries in the cache (right-hand side). Each such cache edge specifies (1) the corresponding query edge before schematization, (2) the evaluation step in which the constraint was matched, and (3) the set of schema hits that produced the constraint during schematization. In our example, there are only singleton sets of schema hits because both Q' and Q each have only one schema hit.

10.5.1 Storing Queries in the RCADG Cache

The main-memory part C of the RCADG Cache is a mapping from schema edges to sets of so-called *cache edges* that indicate which queries and schema hits in the cache produced a particular schema edge during schematization. Figure 10.4 depicts C after adding the queries Q' and Q from Figure 10.1 on page 143, assuming the schema hits and query plans discussed above. We refer to this particular cache as $C_{\{Q', Q\}}$.

For instance, consider the first entry in $C_{\{Q', Q\}}$, which maps the schema edge $c_s = Parent_*(\#5, \#1)$ to two distinct cache edges. The first cache edge, $\langle Parent_*(q_4, q_1), s_1^Q, \{\chi^Q\} \rangle$, indicates that schematizing the binary constraint $Parent_*(q_4, q_1)$ in Q with the schema hit χ^Q produced the schema edge c_s , and that this constraint was matched on the document level in step s_1^Q during the evaluation of Q . Likewise, the second cache edge associated with c_s in $C_{\{Q', Q\}}$ states that the same schema edge is also part of $Q' \downarrow \chi^{Q'}$. Note that the two cached queries Q and Q' bind the same schema node $\#5$ to query nodes with different D -constraints: as shown in Figures 10.1 *d*, *e*. on page 143, the query node q_4 in Q has a keyword constraint for “female” whereas the query node q'_3 has no keyword constraint. When retrieving the two cache edges during the look-up for a new query Q'' , whose schematization also contains c_s , the different D -constraints attached to q_4 and q'_3 will need to be compared to the D -constraints in Q'' (see Section 10.5.3 below).

In Figure 10.4 each cache edge covers exactly one schematization of a query in the cache. For instance, the cache edge in the first row represents a binary constraint in $Q \downarrow \chi^Q$ and the cache edge in the second row represents one in $Q' \downarrow \chi^{Q'}$. Note, however, that in general multiple schema hits for the same cached query may produce the same schema edge. An example is given in Figures 10.1 *f*, *g*. on page 143 where the schema edge $Parent_*(\#2, \#1)$ is part of both $Q'' \downarrow \chi_1^{Q''}$ and $Q'' \downarrow \chi_2^{Q''}$. Therefore each cache edge stands for a set of schema hits, as indicated by the curly braces around χ^Q and $\chi^{Q'}$ in Figure 10.4. It is easy to see that all schema hits in a given cache edge coincide on the corresponding query edge, i.e., they map its source and target nodes to the same pair of schema nodes. Also note that a cache edge for a particular evaluation step has no other (but maybe fewer) schema hits than any cache edge an earlier step in the same query plan because schema hits may be discarded, but not added during the evaluation (see Chapter 8).

As more queries are added to the cache, new cache edges with different binary constraints, evaluation steps and schema hits are associated with new or existing schema edges in C . Hence C is a one-to-many mapping from schema edges to cache edges.

10.5.2 Retrieving Cache Contents

Every new query to be evaluated incrementally first undergoes the same schema-level rewriting and matching procedures that were described for the evaluation from scratch (see Sections 8.4.1 and 8.4.2). In the

schema hit	evaluation step	match edges	
		new edge	cache edge
$\chi_1^{Q^n}$	$\mapsto \{$ s_1^Q s_2^Q	$\mapsto \{ \langle \text{Parent}_*^*(q_3^n, q_1^n), \langle \text{Parent}_*^*(q_4, q_1), s_1^Q, \{\chi^Q\} \rangle \rangle \}$	
		$\mapsto \{ \langle \text{Parent}_*^*(q_2^n, q_1^n), \langle \text{Parent}_*^*(q_2, q_1), s_2^Q, \{\chi^Q\} \rangle \rangle \}$	
	$s_1^{Q'}$ $s_2^{Q'}$	$\mapsto \{ \langle \text{Parent}_*^*(q_3^n, q_1^n), \langle \text{Parent}_*^*(q'_3, q'_1), s_1^{Q'}, \{\chi^{Q'}\} \rangle \rangle \}$	
		$\mapsto \{ \langle \text{Parent}_*^*(q_2^n, q_1^n), \langle \text{Parent}_*^*(q'_2, q'_1), s_2^{Q'}, \{\chi^{Q'}\} \rangle \rangle \}$	
$\chi_2^{Q^n}$	$\mapsto \{$ s_2^Q $s_2^{Q'}$	$\mapsto \{ \langle \text{Parent}_*^*(q_2^n, q_1^n), \langle \text{Parent}_*^*(q_2, q_1), s_2^Q, \{\chi^Q\} \rangle \rangle \}$	
		$\mapsto \{ \langle \text{Parent}_*^*(q_2^n, q_1^n), \langle \text{Parent}_*^*(q'_2, q'_1), s_2^{Q'}, \{\chi^{Q'}\} \rangle \rangle \}$	

Figure 10.5: The result $L_{\{Q', Q\}}^{Q^n}$ of looking up cache edges in $C_{\{Q', Q\}}$ (see Figure 10.4 on the preceding page) for the schematized binary constraints in $Q^n \downarrow \chi_1^{Q^n}$ and $Q^n \downarrow \chi_2^{Q^n}$ (see Figures 10.1f, g. on page 143). The look-up result is a nested mapping with the following structure. Each of the two schema hits for Q^n (left column) is mapped to a nested mapping that groups the retrieved cache edges by evaluation steps. Every distinct evaluation step from any of the retrieved cache edges (middle column) is mapped to a set of *match edges* each representing the matching of a relevant binary constraint which happened in that step. A match edge simply binds a cache edge to the corresponding query edge in Q^n , and thus indicates which D -constraints in a cached query and in Q^n must be compared.

case of our sample query Q^n , this yields the two schema hits $\chi_1^{Q^n}$ and $\chi_2^{Q^n}$. Next, the query is normalized and schematized with each of these schema hits, as shown in Figures 10.1f, g. on page 143. The resulting schema edges are then looked up in the main-memory part of the RCADG Cache. In our example, three distinct schema edges are looked up, namely, $\text{Parent}_*^*(\#5, \#1)$, $\text{Parent}_*^*(\#2, \#1)$ and $\text{Parent}_*^*(\#6, \#1)$. In the cache $C_{\{Q', Q\}}$, four cache edges are retrieved for the first two schema edges (top four rows in Figure 10.4 on the facing page) whereas there is no hit for the third one.

The look-up result for Q^n is rearranged in a nested map L^{Q^n} (see Figure 10.5), as follows. Each cache edge c_c retrieved for a schema edge c_s is bound to the binary constraint c in Q^n that created c_s . For instance, looking up the schema edge $c_s = \text{Parent}_*^*(\#5, \#1)$ that was created from the binary constraint $c = \text{Parent}_*^*(q_3^n, q_1^n)$ in Q^n , we retrieve the cache edge $c_c = \langle \text{Parent}_*^*(q_4, q_1), s_1^Q, \{\chi^Q\} \rangle$ in $C_{\{Q', Q\}}$. Therefore c and c_c are associated in the first entry of L^{Q^n} in Figure 10.5. Henceforth we refer to such a pair $\langle c, c_c \rangle$ of a query edge c in Q^n and a cache edge c_c retrieved for c as a *match edge*. Match edges specify which D -constraints in a cached and a new query must be reconciled for schema-hit containment to hold true. In this case, the first match edge in Figure 10.5 specifies that the second condition in Definition 10.1 on page 144 must be checked for the D -constraints attached to two pairs of query nodes, namely, q_3^n, q_4 and q_1^n, q_1 . The differences and relations between schema edges, cache edges and match edges is summarized in Table 10.1 on the following page.

As can be seen in Figure 10.5, the look-up for Q^n in $C_{\{Q', Q\}}$ produces six match edges (right-hand side, one match edge in each row). The nested structure of L^{Q^n} emerges when grouping these match edges by (1) by the schema hit for Q^n for which the cache edges were retrieved (left column) and (2) by the evaluation steps in the cache edges (middle column), in that order. For instance, the first four match edges in L^{Q^n} were retrieved for $Q^n \downarrow \chi_1^{Q^n}$ and the last two for $Q^n \downarrow \chi_2^{Q^n}$. Note that since $Q^n \downarrow \chi_1^{Q^n}$ and $Q^n \downarrow \chi_2^{Q^n}$ share the same schema edge $\text{Parent}_*^*(\#2, \#1)$ (see Figures 10.1f, g. on page 143), the match edges for $\chi_2^{Q^n}$ in the last two rows of Figure 10.5 are duplicates of the match edges for $\chi_1^{Q^n}$ in rows two and four. This redundancy will allow us to obtain matches to distinct schema hits for Q^n independently, which is a characteristic of the notion of RCADG Cache overlap introduced before (see Definition 10.2 on page 146). In fact, L^{Q^n} is usually not materialized in its entirety at any given point in time. Instead we successively and separately create, then process and finally discard each of the distinct top-level entries for all schema hits of Q^n (see below).

As indicated by the curly braces in Figure 10.5, each nesting level in L^{Q^n} is a one-to-many mapping. On the lower level (right-hand side), there may be multiple cache edges representing binary constraints in

edge type	description
<i>query edge</i> c (Fig. 10.1 a.-c.)	Specifies a binary query constraint on the intensional level. These are the edges in the query graph. There are query edges for expressing all XPath axes.
<i>schema edge</i> c_s (Fig. 10.1 d.-g.)	Represents a schema-level match to a query edge for a specific schema hit. Schema edges are created by schematizing queries to be cached or to be looked up in the cache. They serve as keys in the main-memory part of the cache, allowing to retrieve cached candidate queries for a new query to be evaluated incrementally.
<i>cache edge</i> c_c (Fig. 10.4)	Indicates which query edge in a cached query corresponds to a particular schema edge, and which schema hits produced that schema edge during the schematization of that query. Cache edges serve to collect all schema hits to a cached query that are relevant to a specific schema edge being looked up in the cache. Each cache edge also specifies in which evaluation step the query edge in question was matched on document level.
<i>match edge</i> c_m (Fig. 10.5)	Binds a cache edge to a query edge that belongs a new query being looked up in the cache. Match edges specify which D -constraints in a cached query correspond to which D -constraints in the new query. This is essential for deciding schema-hit containment and creating remainder queries that return the RCADG Cache overlap.

Table 10.1: Different representations of binary query constraints (“edges”) during the incremental evaluation process. Only query edges (first row) are part of the query model (see Section 2.2). All other types of edge are needed for retrieving and comparing queries that are stored in the RCADG Cache.

a specific cached query that were matched in the same evaluation step (although this is not the case for our sample queries Q' and Q in the cache). The upper level of $L^{Q'}$ (left-hand side of Figure 10.5) is a one-to-many mapping, too, since for the same schema hit of a new query, cache edges for different queries and evaluation steps may be retrieved in the cache, as shown in the figure.

Finally, note that the look-up result $L^{Q'}$ for Q' only covers the first two steps in the evaluation of the cached queries Q' and Q . In particular, the cache entries in $C_{\{Q',Q\}}$ for the schema edges $Parent^*(\#5,\#3)$ and $Parent^*(\#4,\#1)$ (last two rows in Figure 10.4 on page 148) are not retrieved because these are not part of any schematization of Q' (see Figure 10.1 on page 143). Provided that the mapping underlying $C_{\{Q',Q\}}$ is implemented so as to avoid a sequential scan of the memory-resident cache part (e.g., using suitable hash functions), such irrelevant cache contents are typically never touched during the look-up. This means that even as the cache grows, the promising candidate queries are retrieved very efficiently. In Section 10.6 we experimentally confirm the scalability of the RCADG Cache.

10.5.3 Deciding Schema-Hit Containment

This subsection presents an algorithm for computing the RCADG Cache overlap (see Definition 10.2 on page 146) for a new query Q' to be evaluated incrementally, given the cache look-up result $L^{Q'}$. At the heart of the algorithm is the decision procedure for schema-hit containment. For each cached schema hit χ in $L^{Q'}$ that was retrieved for a schema hit $\chi^{Q'}$ of Q' , we check whether $\chi \supset_s \chi^{Q'}$ as defined on page 144. If the test succeeds, we create a cache hit saying that $\chi \supset_s \chi^{Q'}$ to the set $H^{Q'}$ of cache hits for Q' . Before explaining the containment test and the creation of cache hits, let us take a brief look at the set of cache hits that are eventually produced for the query Q' in Figure 10.1 on page 143, assuming the cache $C_{\{Q',Q\}}$ that contains Q' and Q , as before.

Figure 10.6 on the next page depicts $H_{\{Q',Q\}}^{Q'}$, i.e., the set of cache hits obtained for Q' in the example above. Two cache hits have been created from the look-up result $L_{\{Q',Q\}}^{Q'}$ in Figure 10.5 on the preceding page. Each cache hit specifies in the three leftmost columns how to obtain the matches to a specific schema hit for Q' (in the example, $\chi_1^{Q'}$) from the matches to a particular schema hit in the cache (χ^Q or $\chi^{Q'}$) using a fixed snapshot (steps s_2^Q and $s_2^{Q'}$, respectively). For instance, the cache hit κ in the first row in Figure 10.6 tells us that $ans(\chi_1^{Q'})$ is a subset of $ans([\chi^Q]_{s_2^Q})$. Furthermore, from the pairs of corresponding edges in the queries Q' and Q (middle), we see that the matches to the query node q_1' in Q' are taken from the set of

cache hit	final step	schema hits	corresponding edges in new and cached queries	constraints in remainder query
κ	s_2^Q	$\{\llbracket \chi^Q \rrbracket_{s_2^Q} \supset_s \chi_1^{Q^n}\}$	$\left\{ \left\langle \begin{array}{l} \text{Parent}^*(q_2^n, q_1^n) \\ \text{Parent}^*(q_2, q_1) \end{array} \right\rangle, \left\langle \begin{array}{l} \text{Parent}^*(q_3^n, q_1^n) \\ \text{Parent}^*(q_4, q_1) \end{array} \right\rangle \right\}$	$\{ \text{Contains}^{\text{“Lee”}}(q_2^n) \}$
κ'	$s_2^{Q'}$	$\{\llbracket \chi^{Q'} \rrbracket_{s_2^{Q'}} \supset_s \chi_1^{Q^n}\}$	$\left\{ \left\langle \begin{array}{l} \text{Parent}^*(q_2^n, q_1^n) \\ \text{Parent}^*(q_2', q_1') \end{array} \right\rangle, \left\langle \begin{array}{l} \text{Parent}^*(q_3^n, q_1^n) \\ \text{Parent}^*(q_3', q_1') \end{array} \right\rangle \right\}$	$\left\{ \begin{array}{l} \text{Contains}^{\text{“Lee”}}(q_2^n) \\ \text{Contains}^{\text{“female”}}(q_3^n) \end{array} \right\}$

Figure 10.6: The set $H_{\{Q', Q\}}^{Q^n}$ of cache hits for Q^n , constructed from $L_{\{Q', Q\}}^{Q^n}$ in Figure 10.5 on page 149. The two cache hits κ and κ' both obtain $\text{ans}(\chi_1^{Q^n})$ from the cache, whereas $\text{ans}(\chi_2^{Q^n})$ must be computed from scratch. The cache hit in the first row, κ , reuses the intermediate result that was cached after the second step in the evaluation of the query Q , with one keyword constraint as remainder query. The cache hit in the second row, κ' , needs two keyword constraints against the final answer to the query Q' in the cache.

matches to q_1 in the mentioned subset, and likewise for q_2^n, q_2 as well as q_3^n, q_4 . Finally, the remainder query in the rightmost column indicates which subset of the cached results is relevant to Q^n . In the case of κ , a single keyword constraint narrows $\text{ans}(\llbracket \chi^Q \rrbracket_{s_2^Q})$ down to those tuples where the elements matching q_2^n (i.e., q_2) contain the keyword “Lee”. Alternatively, the second cache hit κ' shows how to compute the same set $\text{ans}(\chi_1^{Q^n})$ from $\text{ans}(\llbracket \chi^{Q'} \rrbracket_{s_2^{Q'}})$. Note that in this case, the remainder query has two keyword restrictions instead of one as with κ , because q_3' in Q' does not enforce the constraint $\text{Contains}^{\text{“female”}}$ that is required by q_3^n (see Figure 10.1 c. on page 143), unlike the node q_4 in Q that is used by κ .

Creating cache hits. Algorithm 10.1 on the following page lists pseudocode for processing a schema hit χ^{Q^n} of Q^n , given the cache look-up result L^{Q^n} and an initially empty set H^{Q^n} of cache hits to be created for χ^{Q^n} . The procedure *createCacheHits* successively visits all sets of match edges for χ^{Q^n} and distinct evaluation steps in L^{Q^n} . Evaluation steps belonging to the same query plan are processed one after the other, in the order defined by the plan. Remember that the match edges for a specific evaluation step indicate which pairs of query nodes and edges in Q^n and a cached query might correspond. The outer **for** loop in Algorithm 10.1 (lines 8–41) finds all consistent combinations of match edges in each step s_i (lines 27–30), and tests for which of these combinations there is a cached schema hit χ such that $\llbracket \chi \rrbracket_{s_i} \supset_s \chi^{Q^n}$. In line with Definition 10.1 on page 144, the containment test involves the comparison of keyword constraints attached to corresponding query nodes (lines 15–25) as well as of the binary D -constraints that have been matched up to step s_i (lines 32–40). These two issues are elaborated below.

Each combination of match edges is represented as a cache hit containing the corresponding pairs of new and cached query edges as well as the remaining constraints in Q^n . Cache hits that were successful in step s_i are added to the set H_{cur} of currently active cache hits. If there is another iteration for step s_{i+1} , these cache hits are extended with additional match edges from that step to find out whether $\llbracket \chi \rrbracket_{s_{i+1}} \supset_s \chi^{Q^n}$ holds true, too. Successful cache hits for step s_i that fail in step s_{i+1} are removed from H_{cur} and are collected in H_{old} instead. They remember s_i as the last reusable snapshot of the results they represent, but do not participate in any further iterations. The other cache hits enter yet another round of containment tests until there are either no more steps in the current plan, or one step is missing in L^{Q^n} (lines 10–12). A missing step indicates that none of the constraints matched in this step is mirrored in $Q^n \downarrow \chi^{Q^n}$. As a consequence, all subsequent snapshots of the cached query result after the missing step cannot be reused for χ^{Q^n} .

In the end, all cache hits that were successful for any step in any plan are added to the result set H^{Q^n} (lines 43–48). H^{Q^n} collects the cache hits for all schema hits of Q^n , which are computed in successive calls to *createCacheHits*. Cache hits that represent the same combination of corresponding query edges for the same evaluation step are merged. Thus a single cache hit in H^{Q^n} may specify multiple schema-hit containment pairs for different schema hits of Q^n (hence the curly braces in the third column in Figure 10.6). This way each cache hit for a step s_i can be translated into a single remainder query plan operating on the matches to multiple schema hits at once, which are all stored in the result table for s_i (and maybe those of its successors). Query planning for Q^n is explained in Section 10.5.4.

```

1 // createCacheHits: creation of cache hits for a new schema hit
2 // →  $\chi^{Q^n}$ : a schema hit for a new query  $Q^n$ 
3 // →  $L^{Q^n}$ : the cache look-up result for  $Q^n$ 
4 // ⇔  $H^{Q^n}$ : the set of cache hits to be created
5 procedure createCacheHits( $\chi^{Q^n}$ : schema hit,  $L^{Q^n}$ : map,  $H^{Q^n}$ : set of cache hits)
6   group the steps with key  $\chi^{Q^n}$  in  $L^{Q^n}$  by the plan they belong to
7    $H_{cur} := \emptyset$ ;  $H_{old} := \emptyset$  for each new plan being processed
8   for all steps  $s_i$  in a given plan, in the order of their execution do
9     // only results obtained in successive evaluation steps can be used
10    if  $i > 1$  and the step before  $s_i$  was skipped then
11      break loop
12    end if
13    // find cached and new query edges whose D-constraints can be reconciled
14     $M := \emptyset$ 
15    for all match edges  $c_m$  associated with  $s_i$  in  $L^{Q^n}$  do
16       $c^n :=$  the query edge from  $Q^n$  in  $c_m$ 
17       $c :=$  the query edge from the cache edge in  $c_m$ 
18       $q_s^n, q_t^n :=$  the source and target nodes of  $c^n$ 
19       $q_s, q_t :=$  the source and target nodes of  $c$ 
20       $K_s :=$  call checkKeywords( $q_s^n, q_s$ )
21       $K_t :=$  call checkKeywords( $q_t^n, q_t$ )
22      if  $K_s \neq \text{nil}$  and  $K_t \neq \text{nil}$  then
23         $M := M \cup \{ \langle c^n, c, K_s \cup K_t \rangle \}$ 
24      end if
25    end for
26    // update the set of cache hits with new pairs of corresponding query edges
27     $H :=$  the cache hits in  $H_{cur}$  that are inconsistent with any subset of edge pairs in  $M$ 
28     $H_{cur} := H_{cur} \setminus H$ ;  $H_{old} := H_{old} \cup H$ 
29     $H :=$  all consistent cache hits created from  $H_{cur}$  using any subset of edge pairs in  $M$ 
30     $H_{cur} := H_{cur} \cup H$ 
31    // keep only cache hits contributing a schema hit that contains  $\chi^{Q^n}$ 
32    for all cache hits  $\kappa \in H_{cur}$  do
33       $X :=$  call checkSnapshot( $\kappa, s_i, L^{Q^n}$ )
34      if  $X = \emptyset$  then
35         $H_{cur} := H_{cur} \setminus \{ \kappa \}$ ;  $H_{old} := H_{old} \cup \{ \kappa \}$ 
36      else
37        for an arbitrary  $\chi \in X$ , add  $\llbracket \chi \rrbracket_{s_i} \supset_s \chi^{Q^n}$  to  $\kappa$  (replacing any existing statement for  $\chi^{Q^n}$ )
38        replace the step in  $\kappa$  with  $s_i$ 
39      end if
40    end for
41  end for
42  // collect and possibly merge successful cache hits for all steps and plans
43  for all cache hits  $\kappa \in H_{cur} \cup H_{old}$  with a schema-hit containment for  $\chi^{Q^n}$  do
44    if  $\exists \kappa' \in H^{Q^n}$ :  $\kappa, \kappa'$  have the same corresponding query edges and step then
45      add  $\kappa$ 's schema-hit containment for  $\chi^{Q^n}$  to  $\kappa'$ 
46    else
47       $H^{Q^n} := H^{Q^n} \cup \{ \kappa \}$ 
48    end if
49  end for
50 end procedure

```

Algorithm 10.1: Creation of cache hits with the RCADG Cache. The input is a schema hit χ^{Q^n} for the new query Q^n to be evaluated, the result L^{Q^n} of looking up Q^n in the RCADG Cache, and a set H^{Q^n} for collecting the cache hits to be created. A sample output is shown in Figure 10.6 on the previous page.

Checking unary D -constraints. The only unary D -constraints to be compared in the containment test are keyword constraints.³ The procedure *createCacheHits* in Algorithm 10.1 on the facing page compares the keyword constraints of every pair of query nodes that are the source or target nodes of two query edges in the same match edge (lines 15–25). Only edges whose source and target node constraints can be reconciled pairwise are added to the set M (line 23) that is used to create new cache hits (lines 27–30).

The actual comparison of keyword constraints is triggered by calls to *checkKeywords* in lines 20 and 21 of Algorithm 10.1. The pseudocode for *checkKeywords* is given in Algorithm 10.2. The procedure compares the keyword constraints of two query nodes q^n and q belonging to the new query Q^n and a cached query Q , respectively. It returns the subset of q^n 's keyword constraints that remain to be checked against the cached matches to q , or **nil** if q 's keyword constraints are too strict for q^n . The empty set is returned (line 60) if q^n and q specify the same keywords with essentially the same Boolean junctor (conjunction or disjunction) and scope (containment or government). If only q has keyword constraints, **nil** is returned (line 63). If on the contrary only q^n has keyword constraints, all these constraints must be matched (line 66).

In all remaining cases the keyword constraints of q^n and q must be compared more thoroughly, as shown in Figure 10.7 on page 155. The right-hand side of the figure (coloured) comprises sixteen areas of eight squares each, most of them containing a relational symbol, which are arranged in pairs (a grey square on the left and a coloured or white square on the right). Each of the sixteen areas corresponds to a particular combination of the following four parameters: *junctor*(q), *scope*(q) (horizontal) and *junctor*(q^n), *scope*(q^n) (vertical). The upper left area, e.g., applies if both nodes specify a disjunction of containment constraints.

The four pairs of relation symbols in each area are to be read as follows: “=”, “ \subset ”, “ \supset ” and “ \cap ” denote the equality, containment (in either direction) and non-empty intersection (overlap) of sets, respectively. Any pair $\langle \theta, \theta' \rangle$ of a grey and a coloured symbol indicates that if the two sets of keywords used in the constraints of q and q^n are in relation θ (grey square), then the two sets of elements that satisfy these constraints are in relation θ' (coloured square). For instance, consider the upper left pair $\langle =, = \rangle$ in Figure 10.7. It says that if q and q^n both specify a disjunction of containment constraints for the same set of keywords, then they will be matched by the same set of elements (as far as keyword constraints are concerned, i.e., ignoring all other query constraints that q and q^n may be involved in). This obvious fact is captured by the first conditional branch of the procedure *checkKeywords* in Algorithm 10.2 on the following page, along with the other four $\langle =, = \rangle$ pairs (highlighted grey and red).

The other pairs in Figure 10.7 deal with less obvious cases. All pairs with a “ \supset ” symbol on the right-hand side (highlighted yellow) indicate that q 's keyword constraints are no more restrictive than those of q^n , which is exploited in lines 74 and 77 of Algorithm 10.2. If q and q^n both specify a conjunction of such constraints with the same scope (the two yellow “ \supset ” symbols directly below the two lower-right red “=” symbols in Figure 10.7), then only the constraints in q^n that are missing in q need to be part of the remainder query (line 74). For instance, given two sets of constraints $\text{Contains}_{k_0}(q) \wedge \text{Contains}_{k_1}(q)$ and $\text{Contains}_{k_0}(q^n) \wedge \text{Contains}_{k_1}(q^n) \wedge \text{Contains}_{k_2}(q^n)$ for q and q^n , respectively, only $\text{Contains}_{k_2}(q^n)$ must be checked against the matches to q in the cache. In all other cases where the keyword constraints can be reconciled (remaining pairs with yellow “ \supset ” symbols in Figure 10.7), the remainder query includes the entire set of keyword constraints of q^n .

For all but the yellow and red pairs in Figure 10.7 (symbols “=” and “ \supset ”, respectively), either the set of elements matching q 's keyword constraints is known to be a subset of q^n 's set of matches (blue “ \subset ” symbols), or no specific relation between the match sets can be inferred (white squares with no symbol). For these junctor/scope/keyword combinations, the procedure *checkKeywords* returns **nil** (line 69 in Algorithm 10.2 on the next page), which causes the corresponding match edge to be discarded from cache-hit creation (line 23 in Algorithm 10.1 on the facing page).

Checking binary D -constraints. The notion of schema-hit containment in Definition 10.1 on page 144 implies that the schematized cached query does not contain any D -constraints which make its extension too restrictive with respect to the schematized new query Q^n . For every binary D -constraints in the cached query, this means that if the constraint has a counterpart in Q^n , they must be reconciled, and if not, the

³Recall from Definition 2.7 on page 11 that the other unary query constraints specifying tag, type and level conditions are S -constraints. Being fully captured by schema nodes, they need not be matched on the document level.

```
51 // checkKeywords: comparison of keyword constraints
52 // →  $q^n$ : a query node in the new query  $Q^n$ 
53 // →  $q$ : a query node in a cached query  $Q$ 
54 // ← a set of keyword constraints for the remainder query, or nil
55 procedure checkKeywords ( $q^n$ : query node,  $q$ : query node)
56 //  $q^n$  and  $q$  have similar constraints for the same keywords
57 if keywords( $q^n$ ) = keywords( $q$ ) and
58   ( $\text{junctor}(q^n) = \text{junctor}(q)$  or |keywords( $q^n$ )| < 2) and
59   ( $\text{scope}(q^n) = \text{scope}(q)$  or |keywords( $q^n$ )| = 0) then
→ 60   return  $\emptyset$ 
61 // only  $q$  has keyword constraints
62 else if keywords( $q^n$ ) =  $\emptyset$  then
→ 63   return nil
64 // only  $q^n$  has keyword constraints
65 else if keywords( $q$ ) =  $\emptyset$  then
→ 66   return the constraints for keywords( $q^n$ )
67 //  $q$ 's keyword constraints are too restrictive
68 else if  $\langle q, q^n \rangle$  does not have a yellow “ $\supset$ ” in the “matches” column in Figure 10.7 then
→ 69   return nil
70 // some keyword constraints in  $Q^n$  are already subsumed by  $q$ 
71 else if  $\text{junctor}(q^n) = \wedge$  and
72    $\text{junctor}(q^n) = \text{junctor}(q)$  and
73    $\text{scope}(q^n) = \text{scope}(q)$  then
→ 74   return the constraints for  $\text{keywords}(q^n) \setminus \text{keywords}(q)$ 
75 // all keyword constraints in  $Q^n$  must be matched
76 else
→ 77   return the constraints for keywords( $q^n$ )
78 end if
79 end procedure
```

Algorithm 10.2: Comparison of keyword constraints with the RCADG Cache. This procedure is needed for verifying the second condition in Definition 10.1 on page 144. The input is a query node in the new query Q^n to be evaluated and a query node from a query Q in the RCADG Cache. The output is the (possibly empty) subset of the keyword constraints of q^n that need to be matched as part of the remainder query for Q^n . A return value **nil** indicates that the keyword constraints of q^n and q cannot be reconciled. For a given query node q , $\text{junctor}(q)$ is the Boolean operator (“ \wedge ” or “ \vee ”), and $\text{scope}(q)$ is either containment or government.

			new query node q^n							
			disjunction		conjunction					
			contain	govern	contain	govern				
cached query node q	disjunction	contain	=	=	=	⊂	=	⊃	=	
			⊂	⊂	⊂	⊂	⊂	⊃	⊂	
			⊃	⊃	⊃		⊃	⊃	⊃	
			⊄		⊄		⊄	⊃	⊄	
		govern	=	⊃	=	=	=	⊃	=	⊃
			⊂		⊂	⊂	⊂	⊃	⊂	⊃
			⊃	⊃	⊃	⊃	⊃	⊃	⊃	⊃
			⊄		⊄		⊄	⊃	⊄	
	conjunction	contain	=	⊂	=	⊂	=	=	=	⊂
			⊂	⊂	⊂	⊂	⊂	⊃	⊂	
			⊃	⊂	⊃	⊂	⊃	⊂	⊃	
			⊄	⊂	⊄	⊂	⊄		⊄	
		govern	=		=	⊂	=	⊃	=	=
			⊂		⊂	⊂	⊂	⊃	⊂	⊃
			⊃		⊃	⊂	⊃		⊃	⊂
			⊄		⊄	⊂	⊄	⊃	⊄	

Figure 10.7: Comparison of the keyword constraints of a node q in a cached query and a node q^n from a new query to be evaluated incrementally. Each cell in the table represents a specific relation between the two sets of keywords used in the constraints (left half of the cell, highlighted grey) and the resulting relation between the two sets of elements that satisfy these constraints (right half of the cell, white or coloured). The pairs of relations vary with the nature of the keyword constraints in q and q^n . For instance, if both nodes specify a disjunction of containment constraints (four upper-left pairs) and q has more keywords in the disjunction than q^n (third pair, symbol “ \supset ” highlighted grey), then it may also have a superset of the matches to q^n (symbol “ \supset ” highlighted yellow). By contrast, if both query nodes feature a conjunction of government constraints (four lower-right pairs) and q has again more keywords than q^n , then it may only have a subset of the matches to q^n (third pair, symbol “ \subset ” highlighted blue).

constraint must not introduce a proper restriction. This is verified by the procedure *checkSnapshot* listed in Algorithm 10.3 on the following page. The procedure is called repeatedly by *createCacheHits* in line 33 of Algorithm 10.1 on page 152 for a (preliminary) cache hit κ and an evaluation step s_i of a particular query Q in the cache. At this point in time, κ contains a set of corresponding query edges from Q and Q^n as well as a set of remainder query constraints for Q^n , as illustrated in Figure 10.6 on page 151. The pairs of query edges in κ indicate which binary constraints in Q have which counterparts in Q^n after schematization. Q^n is schematized with a specific schema hit χ^{Q^n} given as a parameter to *createCacheHits* (see above). The schema hits for Q that produced the pairs of query edges in κ are available from the corresponding cache edges for χ^{Q^n} and s_i in the look-up result L^{Q^n} (see Figure 10.5 on page 149). Let X_{s_i} be the set of these schema hits. Now the task is to check whether there is at least one $\chi \in X_{s_i}$ such that the binary D -constraints in $Q \downarrow \chi$ and $Q^n \downarrow \chi^{Q^n}$ comply with Definition 10.1 (the unary D -constraints were already compared before κ was created, see above).

Note that to confirm $[[\chi]_{s_i}] \supset_s \chi^{Q^n}$ we only need to examine those binary D -constraints of Q that were matched in step s_i , because constraints in earlier steps of the same plan have been checked in previous iterations of the outermost **for** loop in *createCacheHits* (line 8 in Algorithm 10.1). As observed in Section 10.5.1, this is true for all schema hits in the set X_{s_i} , which is contained in the set of schema hits retrieved

```

80 // checkSnapshot: decision of schema-hit containment
81 // → κ: a preliminary cache hit for a schema hit  $\chi^{Q^n}$  of the new query  $Q^n$ 
82 // →  $s_i$ : a step in the evaluation plan of a cached query  $Q$ 
83 // →  $L^{Q^n}$ : the cache look-up result for  $Q^n$ 
84 // ← a set of schema hits for  $Q$  that contain  $\chi^{Q^n}$  in step  $s_i$ 
85 procedure checkSnapshot ( $\kappa$ : cache hit,  $s_i$ : evaluation step,  $L^{Q^n}$ : map)

86    $X := \text{nil}$ 

87   // all edges of  $Q$  decided or reconstructed in  $s_i$  must be mirrored in  $Q^n \downarrow \chi^{Q^n}$ 
88   for all query edges  $c \in Dec_i \cup Rec_i$  do
89     if  $c$  is not among the edges from cached queries in  $\kappa$  then
90       return  $\emptyset$ 
91     end if
92      $X_c :=$  the schema hits from the cache edge for  $\chi^{Q^n}$ ,  $s_i$  and  $c$  in  $L^{Q^n}$ 
93     if  $X = \text{nil}$  then
94        $X := X_c$ 
95     else
96        $X := X \cap X_c$ 
97     end if
98   end for

99   // all query nodes of  $Q$  joined in  $s_i$  must be mirrored in  $Q^n \downarrow \chi^{Q^n}$ 
100  for all query nodes  $q \in Join_i$  do
101    if  $q$  is not among the nodes from cached queries in  $\kappa$  then
102      return  $\emptyset$ 
103    end if
104  end for

105  return  $X$ 

106 end procedure
    
```

Algorithm 10.3: Decision of schema-hit containment with the RCADG Cache (slightly simplified). This procedure is needed for verifying the first condition in Definition 10.1 on page 144. The input is a cache hit κ created for a schema hit χ^{Q^n} of the new query Q^n to be evaluated, an evaluation step s_i for a cached query Q that supplies the cached edges in κ , and the cache look-up result for Q^n . The output is the set of schema hits for Q that contain χ^{Q^n} in step s_i . A return value \emptyset indicates that κ associates edges from both queries in such a way that the containment test fails, because some binary constraints in the schematized query Q that were decided or reconstructed in step s_i are not mirrored in $Q^n \downarrow \chi^{Q^n}$.

for any predecessor of s_i . The procedure *checkSnapshot* returns the subset $X \subset X_{s_i}$ of all schema hits for which the containment test succeeded. If X is non-empty, κ 's evaluation step and schema-hit containment for χ^{Q^n} are updated accordingly (lines 37, 38 in Algorithm 10.1 on page 152). Note that since all schema hits in X coincide on the query edges represented by κ (see Section 10.5.1), only one statement of the form $[\chi]_{s_i} \supset_s \chi^{Q^n}$ is added to κ in line 37, for any schema hit $\chi \in X$.

The procedure *checkSnapshot* in Algorithm 10.3 tests whether any binary constraint was decided or reconstructed in s_i lacks a counterpart in $Q^n \downarrow \chi^{Q^n}$ (line 89). If so, the containment test fails for κ and s_i , and the empty set is returned. Otherwise we fetch for each query edge $c \in Dec_i \cup Rec_i$ the corresponding set of schema hits in L^{Q^n} , which is contained in the cache for χ^{Q^n} , s_i and c (see Figure 10.5 on page 149). The set X eventually contains the intersection of all these sets of schema hits (lines 92–97).

To understand why query edges that were decided or reconstructed in s_i must have a counterpart in Q^n for schema-hit containment to hold, consider a query edge $c = R(q_s, q_t)$ in $Dec_i \cup Rec_i$ that is not mirrored in $Q^n \downarrow \chi^{Q^n}$. If both q_s and q_t have counterparts in $Q^n \downarrow \chi^{Q^n}$, then matching c may have caused tuples in the intermediate result to Q to be discarded in step s_i . For instance, if $c = PrevSib(q_s, q_t)$, then all tuples where q_s is matched by a leftmost sibling are dropped in s_i . However, these tuples might well be part of the answer to Q^n , which accepts matches to q_s and q_t for which the relation R does not hold.

Now assume that q_s has no counterpart in $Q^n \downarrow \chi^{Q^n}$. From the query planning algorithm presented in

Section 8.4.4, it is obvious that matches to q_s must have been obtained in step s_i or earlier: either by a join with the element table which may have caused matches to Q^n to be discarded from the intermediate result of Q in step s_{i-1} , or by reconstructing another query constraint whose target node is q_s . However, since there are no cycles in the set Rec_i of reconstructed edges (see Chapter 8.4.4), the matching of q_s must have involved an element-table join at some point of the evaluation, either directly or indirectly, which violates the first condition in Definition 10.1 on page 144.

If $q_s \in Dec_i$, then the same argument applies in cases where q_t has no counterpart in $Q^n \downarrow \chi^{Q^n}$. Now assume that $q_s \in Rec_i$. In general the reconstruction of c may have caused tuples to be discarded that would have been matches to Q^n (again, consider the case where $c = PrevSib(q_s, q_t)$ and q_s is matched by a leftmost sibling). Therefore query constraints in Dec_i and Rec_i are treated alike in line 88 of *checkSnapshot*. However, in fact we can show that under certain circumstances, reconstructed query edges whose target node is not mirrored in Q^n are admissible. The argument behind this is sketched in Section 10.8.

As mentioned before, every join with the element table in step s_i may eliminate tuples from the intermediate result of Q produced in that step. This might prevent the incremental evaluation of Q^n based on this snapshot of Q 's answer, unless the join conditions are also implied by Q^n . Therefore we need to check whether all query nodes in Q that are matched through an element-table join in step s_i have a counterpart in $Q^n \downarrow \chi^{Q^n}$. Nodes in $Join_i$ typically have at least one adjacent edge in Dec_i or Rec_i . If such a node is not mirrored in $Q^n \downarrow \chi^{Q^n}$, this edge does not satisfy the condition in line 89 of Algorithm 10.3 on the facing page, so that the containment test fails, as intended. However, the query planning algorithm in Section 8.4.4 may also produce evaluation steps that contain query nodes from Q to be joined, but no query edges to be decided or reconstructed. As a simple solution, we explicitly verify that each node in $Join_i$ has a counterpart in $Q^n \downarrow \chi^{Q^n}$ (lines 100–104 in Algorithm 10.3), which ensures that the unary constraints of the two nodes were compared before. Again, possible optimizations are discussed in Section 10.8.

To sum up, throughout this subsection we have seen five reasons why the containment test for two schema hits χ^{Q^n} of Q^n and χ of Q may fail in a particular query step s_i :

1. No match edges were retrieved for another evaluation step preceding s_i in the same query plan (see *createCacheHits* in Algorithm 10.1 on page 152, line 10). For instance, this is the case for the schema hit $\chi_2^{Q^n}$ of Q^n (see Figure 10.5).
2. The schematization with χ^{Q^n} and χ produces pairs of query nodes in $Q^n \downarrow \chi^{Q^n}$ and $Q \downarrow \chi^Q$ whose keyword constraints cannot be reconciled (see *checkKeywords* in Algorithm 10.2 on page 154, lines 63 and 69).
3. At least one binary constraint from Q that is not mirrored in $Q^n \downarrow \chi^{Q^n}$ was decided in step s_i (see lines 88–98 in Algorithm 10.3 on the preceding page). An example is the constraint $Parent^*(\#4, \#1)$ in Figure 10.1 *e.* on page 143 that was decided in step s_3^Q of the evaluation of Q .
4. At least one binary constraint from Q that is not mirrored in $Q^n \downarrow \chi^{Q^n}$ was reconstructed in step s_i (see lines 88–98 in Algorithm 10.3 on the preceding page). Constraints that do not introduce a proper restriction may be ignored. An example is the query edge $Parent^*(\#5, \#3)$ in the query Q' , see Figure 10.1 *d.* on page 143.
5. At least one query node from Q that is not mirrored in $Q^n \downarrow \chi^{Q^n}$ was matched through an element-table join in step s_i (see lines 100–104 in Algorithm 10.3 on the preceding page).

10.5.4 Remainder Query Planning

As mentioned before, a separate query plan is created for each member of the set H^{Q^n} of cache hits for Q^n . Such a plan specifies how to obtain the matches to one or more schema hits of Q^n from a specific intermediate result table in the cache that is determined by the unique evaluation step represented by the cache hit. Recall from Figure 10.6 on page 151 that every cache hit κ lists all constraints in a particular remainder query for Q^n whose answer subsumes exactly these matches. From the remainder query constraints in κ , a query plan for κ is created as follows.

$$\begin{array}{ll}
 \text{plan } P_{\kappa}^{Q^n} = \langle s_{\kappa,1}^{Q^n} \rangle & \text{plan } P_{\kappa'}^{Q^n} = \langle s_{\kappa',1}^{Q^n}, s_{\kappa',2}^{Q^n} \rangle \\
 \text{step } s_{\kappa,1}^{Q^n} = \langle \text{Join}_1, \text{Rec}_1, \text{Dec}_1 \rangle & \text{step } s_{\kappa',1}^{Q^n} = \langle \text{Join}_1, \text{Rec}_1, \text{Dec}_1 \rangle \\
 \text{Join}_1 = \{q_2^n\} & \text{Join}_1 = \{q_2^n\} \\
 \text{Rec}_1 = \{\} & \text{Rec}_1 = \{\} \\
 \text{Dec}_1 = \{\} & \text{Dec}_1 = \{\} \\
 & \text{step } s_{\kappa',2}^{Q^n} = \langle \text{Join}_2, \text{Rec}_2, \text{Dec}_2 \rangle \\
 & \text{Join}_2 = \{q_3^n\} \\
 & \text{Rec}_2 = \{\} \\
 & \text{Dec}_2 = \{\}
 \end{array}$$

a. plan based on cache hit κ for query Q b. plan based on cache hit κ' for query Q'

Figure 10.8: RCADG Cache query plans for the incremental evaluation of the query Q^n in Figure 10.1 c. on page 143, based on the different cache hits in Figure 10.6 on page 151. *a.* The query plan created for the cache hit κ (first row in Figure 10.6). It computes $\text{ans}(\chi_1^{Q^n})$ from $\text{ans}(\llbracket \chi^Q \rrbracket_{s_2^Q})$. *b.* The query plan created for the cache hit κ' (second row in Figure 10.6). It computes $\text{ans}(\chi_1^{Q^n})$ from $\text{ans}(\llbracket \chi^{Q'} \rrbracket_{s_2^{Q'}})$.

The core of the planning algorithm sketched in Section 8.4.4 is common to both the evaluation from scratch and the evaluation with the RCADG Cache. The only difference is that when reusing cache contents, some D -constraints in the new query Q^n need not be matched any more. Therefore the planning procedure *createPlan* in Algorithm 8.2 on page 110 is called with restricted sets M_v and M_c of query nodes and edges, rather than all query nodes and edges in Q^n as for the evaluation from scratch: M_v comprises all query nodes of Q^n that are involved in any unary or binary remainder query constraint in κ , and M_c contains all binary remainder query constraints in κ . Consequently, in the resulting incremental query plan for Q^n the element table is joined only for query nodes and keyword constraints in Q^n that are missing in the cached query. Additional binary constraints in Q^n are decided if they involve matches in the cached result, otherwise reconstructed if possible.

Figure 10.8 depicts alternative query plans for computing the matches to the first schema hit of Q^n , $\chi_1^{Q^n}$ (see Figure 10.1 e.), based on results of either of the cached queries Q and Q' . The plan $P_{\kappa}^{Q^n}$ in Figure 10.8 a. for the cache hit κ from Figure 10.6 specifies how the matches to $\chi_1^{Q^n}$ are computed based on matches to χ^Q , using the result snapshot cached after the second step in the evaluation of Q . The plan $P_{\kappa}^{Q^n}$ has only one step created for the remainder query constraint *Contains_{Lee}*(q_2^n) in κ . The step involves a single join with the element table, needed to retrieve those matches to q_2 in $\text{ans}(\llbracket \chi^Q \rrbracket_{s_2^Q})$ which contain an occurrence of the keyword “Lee”.

An alternative plan $P_{\kappa'}^{Q^n}$ in Figure 10.8 b., created for the cache hit κ' in Figure 10.6, obtains the matches to $\chi_1^{Q^n}$ from $\text{ans}(\llbracket \chi^{Q'} \rrbracket_{s_2^{Q'}})$, i.e., a part of the snapshot of the answer to query Q' cached after step $s_2^{Q'}$. This plan requires two element-table joins because the remainder query in κ' comprises two keyword constraints, *Contains_{Lee}*(q_2^n) and *Contains_{female}*(q_3^n). Note that both $P_{\kappa}^{Q^n}$ and $P_{\kappa'}^{Q^n}$ compute the same result – namely, $\text{ans}(\chi_1^{Q^n})$ – from distinct query results in the cache.

In general, to avoid the repeated matching of the same schema hit for Q^n , we need to decide for each schema hit which cache hit to use. This is done based on a cost measure for the query plans created for the different cache hits, which at the moment simply counts the number of element-table joins needed to execute a given plan. Thus in the example above, $P_{\kappa}^{Q^n}$ has a lower cost than $P_{\kappa'}^{Q^n}$, hence κ is used for answering Q^n while κ' is discarded. More sophisticated methods could also take into account selectivity estimates for keyword and tag constraints. Essentially the same optimizations that were evoked for planning the evaluation from scratch also apply to the incremental query evaluation.

Figure 10.9 on the next page shows the SQL code generated for executing the query plan $P_{\kappa}^{Q^n}$. It joins

```

SELECT
  DISTINCT e1 AS e1, e2 AS e2, e4 AS e3           -- project result nodes as in cache hit  $\kappa$ 
FROM
  Q_s2 RT,                                       -- retrieve answer from the penultimate intermediate result of  $Q$ 
  ElementTable ET2                             -- join intermediate result table with element table
WHERE
  RT.sid = ' $\chi^Q$ ' AND                         -- select cached schema hit specified by cache hit  $\kappa$ 
  ET2.pid = RT.p2 AND                          -- match unary constraint on  $q_2$  in remainder query
  ET2.eid = RT.e2 AND
  ET2.key = 'Lee'
ORDER BY
  e1, e2, e4                                   -- order result as needed

```

Figure 10.9: SQL code for retrieving the matches to the schema hit $\chi_1^{Q^n}$ of query Q^n in Figure 10.1 c. using a cached intermediate result of the query Q in Figure 10.1 b. on page 143. The query statement computes $ans(\chi_1^{Q^n})$ from $ans(\llbracket \chi^Q \rrbracket_{s_2, Q})$, as specified by the cache hit κ in Figure 10.6 on page 151. The table Q_s2 containing this particular snapshot of the matches to χ^Q is shown in Figure 8.4 c. on page 102. The keyword constraint in the remainder query in κ entails a join with the element table in Figure 6.1 on page 82.

the snapshot of Q 's result in Figure 8.4 c. on page 102 with the element table shown in Figure 6.1 on page 82. From the result table of Q , all tuples representing matches to χ^Q are selected. The matches to the query node q_2 in these tuples are then looked up in the element table to verify that they indeed contain the keyword “Lee”, as demanded by the remainder query constraint in κ . The statement returns both tuples in Q 's result table in Figure 8.4 c., projected onto the fields $e1$, $e2$ and $e4$. The projection clause in Figure 10.9 reflects the pairs of corresponding query nodes in κ . The resulting tuples $\langle 18, 21, 26 \rangle$ and $\langle 27, 30, 34 \rangle$ are illustrated as matches a_2 and a_3 on the document level in Figure 10.2 on page 145. Note that the second tuple, a_3 , is not available in the last result table of Q (see Figure 8.4 d. on page 102), which again underpins the benefit of caching intermediate query results.

In our running example, only matches to $\chi_1^{Q^n}$ are retrieved in the cache while $ans(\chi_2^{Q^n})$ must be obtained without cache support. Therefore two distinct query plans must be executed to obtain the complete result of Q^n . In general, there may be multiple cache hits for distinct schema hits of Q^n , each with its own query plan, plus one additional plan covering all remaining schema hits of Q^n that must be matched from scratch. In our test system, these query plans are executed sequentially in the order of increasing estimated execution cost. However, since the results of the different query plans are guaranteed to be disjoint (see above), the RCADG Cache is particularly amenable to the parallel processing of multiple cache hits. This is likely to improve the user experience especially in a highly interactive, browsing-oriented retrieval scenario like the one sketched in the introduction.

10.6 Experimental Evaluation

To evaluate the incremental query processing described in the previous section, we have conducted two different experiments. A small-scale experiment studies how the performance for the incremental evaluation of a few hand-picked queries varies when cached queries with different degrees of similarity are available. The second experiment relates the cost and benefit of caching on a larger scale, using a randomly generated cache content and query workload. The two document collections used in the experiments are *IMDb* and *XMark 1100*, respectively (see Section 13.2 in the appendix). Both experiments observe a number of different performance measures explained below. Salient properties of all test queries are shown in Figure 10.10 a. on page 161 for the small-scale experiment and in 10.11 a. on page 162 for the large-scale experiment.⁴ All query processing times presented in the sequel represent the average time needed to compute all matches to all nodes in a given query with the RCADG Cache, as explained above. The average is computed over three out of five consecutive runs after discarding the best and worst result, in order to

⁴Here the terms *small-scale* and *large-scale* refer to the size of the query workload submitted to evaluation, not to the size of the test document collections. In fact, the larger of our two collections is used in the small-scale experiment.

minimize artefacts. The correctness and completeness of the results returned by the RCADG Cache have been verified against the results computed from scratch. The cache contents always include intermediate and final results. All result tables on disk are indexed with a B⁺-Tree on the *sid* column (see Figure 8.4 on page 102).

The test system is a Java implementation (JDK 1.5.0) of the data structures and algorithms presented above. The mapping from schema edges to cached tuples in the main-memory part *C* of the RCADG Cache (see Figure 10.4 on page 148) is a hash table providing access in amortized constant time. At system start-up *C* is loaded into memory, and JDBC connections to the RDBS back-end are established once for the whole test session. This takes 1-2 seconds. During the experiments, the test system and the RDBS are both running on the same machine. Apart from these two tasks, the computer is idle during the experiments. All queries are processed sequentially in Test Environment C (see Section 13.1).

10.6.1 Cost and Benefit of Evaluating Queries with the RCADG Cache

We quantify the benefit of incremental query evaluation by measuring the *processing time* and the *number of joins* needed to compute the result (although counting the number of tuples being joined would be more accurate). Since schema matching is the same for the evaluation from cache and from scratch, we do not count the *n*-way selfjoin of the path table but only the number of joins with the larger element table (the processing time includes both phases). On the cost side, retrieving and matching overlapping queries and their schema hits in the main-memory part of the cache takes some extra computation time not needed when evaluating a query from scratch. We refer to this overhead as (*cache*) *search time*. Besides, the persistent cache data structures consume extra storage both in main memory and on disk, which we denote as *cache size* (*in memory* and *on disk*, respectively).

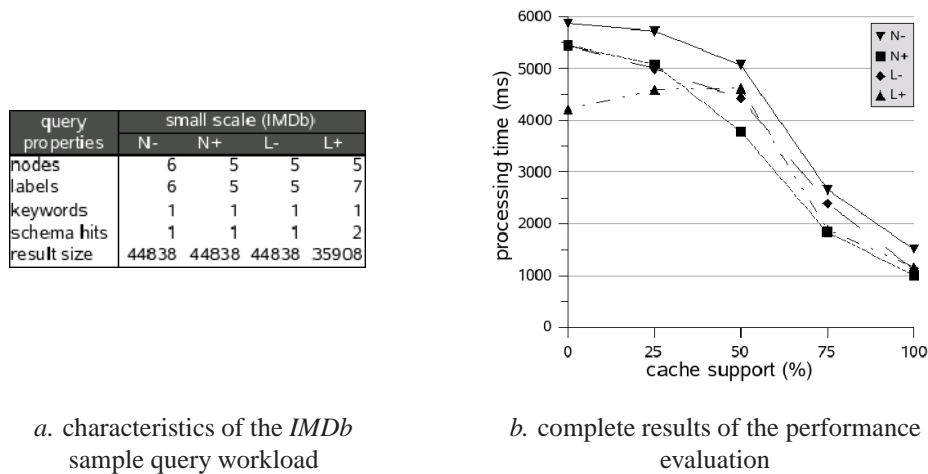
We now define the notion of *cache support* to measure how “useful” the cache contents in *C* are for evaluating a given query Q^n incrementally. Let X be the set of schema hits for Q^n , and let P a query plan for processing Q^n from scratch with minimal estimated execution cost $cost(P) > 0$ (see Section 10.5.4). Besides, let $\bigcup_j X_{\kappa_j}$ be a partition of X such that each X_{κ_j} contains exactly the schema hits represented by the cache hit κ_j computed for Q^n and *C*. Finally, let P_{κ_j} denote the query plan devised for κ_j . Then the cache support for Q^n and *C* is defined as $\left(1 - \frac{\sum_j |X_{\kappa_j}| \cdot cost(P_{\kappa_j})}{|X| \cdot cost(P)}\right) \cdot 100\%$. In this formula, the numerator denotes the estimated cost of processing the selected cache hits, accumulated over all the schema hits of Q^n that they represent. The denominator subsumes the estimated cost of computing the matches to all schema hits from scratch. Thus the entire formula quantifies the execution cost saved in comparison to the evaluation from scratch.

For simplicity, we henceforth assume that $cost(P)$ is again the number of element-table joins needed to execute P . Note that with this coarse cost estimation function, a cache support of 100% does *not* necessarily mean that Q^n itself is found in the cache, only that no joins with the element table are needed to evaluate Q^n incrementally using *C*. In the experiments described next, the cache support indicates to what extent the evaluation of Q^n can possibly benefit from the cache. The following guiding questions summarize three major optimization goals:

1. *effectiveness*: Are useful cached queries exploited if available?
2. *efficiency*: Does the benefit of caching outweigh the overhead?
3. *scalability*: How does the overhead vary with growing cache size?

10.6.2 Small-Scale Experiment

To answer the first of the above questions, we consecutively evaluate a fixed test query against five different cached queries, in the order of increasing cache support. The experiment illustrates on a small scale the effectiveness of our approach, by showing how the incremental processing time is correlated with the cache support, which the RCADG Cache strives to optimize. Let Q_i^n be the query to be evaluated incrementally, and let Q_{ij} , $1 \leq j \leq 5$, denote the five queries serving as cache contents in the consecutive runs.



a. characteristics of the *IMDb* sample query workload

b. complete results of the performance evaluation

Figure 10.10: Results of the small-scale experiment on the *IMDb* collection.

For this experiment we use the *IMDb* collection containing nearly 9 GB of XML documents about movies and actors. The incremental query evaluation against a sequence of different cached queries is conducted four times with distinct queries Q_i^n and corresponding cache contents ($1 \leq i \leq 4$). For each query Q_i^n , the cached queries Q_{ij} are derived from Q_i^n by applying a specific class of editing operations, as they typically occur in user sessions with relevance feedback: N denotes modifications of the query structure and L of the tag constraints; -/+ means making the query more or less restrictive, respectively. Combining the two degrees of freedom yields the four classes N-, N+, L- and L+. For instance, adding a query node is in class N-, whereas adding an alternative tag constraint to a node that already has a tag constraint (or removing the existing constraint) would be in L+. Figure 10.10 a. lists some properties of the query Q_i^n for each class of editing operations. For instance, the column N- lists characteristics of the query Q_1^n that is incrementally evaluated against a sequence of queries created through node restriction. Note the large number of document matches in the last row.

Figure 10.10 b. plots the processing time in milliseconds for each of the five queries in each of the four sequences. As can be seen on the abscissa, the cache support grows from 0% ($j = 1$) to 100% ($j = 5$). For $j = 1$, the cache contains only the query Q_{i1} that allows no joins with the element table to be saved, compared to evaluating Q_i^n from scratch. By contrast, for $j = 2$ the cache contains the query Q_{i2} instead which provides a cache support of 25%, and so on. This pattern applies to all four sequences tested (see the key in Figure 10.10 b.). The results show that for all classes of editing operations, the processing time decreases significantly with growing cache support, down to 20% of the time needed without the cache.

10.6.3 Large-Scale Experiment

The second experiment targets all three optimization goals in a large-scale setting. The goal is to monitor the actual benefit experienced by users of a system that makes intermediate and final query result available for reuse in the RCADG Cache. To this end, we simulate a cache growing from 0 to 199 distinct queries in five stages, as it could evolve during a longer retrieval period with continuous incremental query evaluation. Figure 10.11 d. on the following page lists some statistics of the cache in the four stages C1 to C4. In the initial stage C0, the cache is empty (omitted in Figure 10.11 d. on the next page). As more and more queries (i.e., cache edges from schematized queries) are added, the size of the cache grows from 1 MB in memory and 77 MB on disk (C0, leftmost column) to 5 MB in memory and nearly 1 GB on disk (C4, rightmost column).

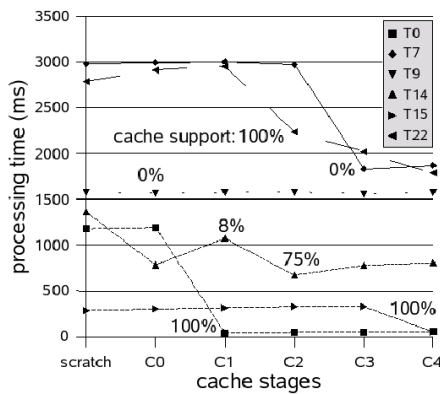
In the absence of a real-world query workload which could only be extracted from the log of a system in productive use, we model the workload as a sequence of random queries, including some popular or “hotspot” queries which are more likely to be asked repeatedly (possibly with modifications as in the small-scale experiment above). The test queries are obtained as follows. From a seed of 150 distinct randomly

query properties	large scale (XMark)																	
	T0	T4	T14	T20	T22	T21	T7	T18	T19	T23	T2	T3	T5	T9	T13	T10	T16	T17
nodes	4	4	11	13	5	5	8	11	2	10	4	5	6	2	5	2	2	2
labels	4	1	4	4	5	1	8	10	1	8	5	4	5	2	4	2	2	2
keywords	1	1	1	1	0	1	0	1	1	0	1	0	0	1	1	1	1	1
schema hits	1	6	12	48	1	6	1	1	514	4	84	6	1	1	57	1	8	1
result size	18	96	32	1859	60588	147123	16859	11000	157239	9576	6343	6	1	489	4941	1	1	1

a. characteristics of selected queries in the XMark 1100 sample query workload

query	scratch	time 0	time 1	time 2	time 3	time 4	sup 0	sup 1	sup 2	sup 3	sup 4	ovh 0	ovh 1	ovh 2	ovh 3	ovh 4
T0	1181	1193	39	50	54	58	0	100	100	100	100	0	37	42	45	47
T4	7629	7672	79	94	113	113	0	100	100	100	100	0	33	45	32	52
T14	1363	783	1073	671	780	807	0	8	75	75	75	0	4	12	13	32
T15	289	305	318	330	332	54	0	0	0	0	100	0	3	6	7	45
T20	29765	30372	30143	6139	4097	4721	0	0	98	98	98	0	0	4	4	46
T22	2788	2914	2951	2237	2020	1788	0	0	100	100	100	0	0	1	1	1
T21	16005	15403	15492	14049	14875	14953	0	17	17	17	17	0	0	0	1	1
T7	2980	2996	3005	2970	1833	1872	0	0	0	0	0	0	1	1	2	3
T18	1154	1177	1102	1122	979	954	0	0	0	0	50	0	1	2	3	3
T19	3243	3267	3380	3320	3263	3013	0	0	0	0	0	0	1	1	2	3
T23	16842	16919	16965	16894	6379	6489	0	0	0	0	0	0	0	0	1	1
T2	11015	11159	11220	11335	12767	13572	0	0	0	0	0	0	1	2	4	6
T3	40	50	85	101	117	157	0	0	0	0	17	0	35	41	50	52
T5	39	54	61	78	69	84	0	0	0	0	0	0	29	42	39	39
T9	1576	1567	1574	1580	1560	1576	0	0	0	0	0	0	0	0	0	0
T13	6896	6974	7079	7191	8977	9050	0	0	0	2	2	0	1	3	4	7
T10	26	27	27	29	27	26	0	100	100	100	100	0	0	0	8	9
T16	41	58	48	51	50	52	0	100	100	100	100	0	2	3	3	4
T17	48	47	54	66	67	80	0	100	100	100	100	0	15	26	41	31

b. complete results (time: processing time (ms); sup: cache support (%); ovh: search time (%))



cache contents	cache stages			
	C1	C2	C3	C4
queries	38	73	134	199
cache edges	525	1097	3169	5932
mem. size (MB)	1	1	3	5
disk size (MB)	77	378	646	872

c. selected results

d. contents of the evolving cache

Figure 10.11: Results of the large-scale experiment on the XMark 1100 collection.

generated tree queries against the *XMark 1100* collection, we randomly remove 15 hotspot queries. Then we create five exact copies and five variants of each hotspot query. Query variants are obtained by applying different editing operations such as, e.g., adding a query node or removing a tag or keyword constraint. The complete set of test queries is the union of the resulting 150 hotspot queries and the remaining 135 seed queries.

Now 19 distinct queries are randomly removed from this set. These so-called *new* queries are to be evaluated incrementally in the experiment. Figure 10.11 *a.* on the facing page shows some properties of selected new queries. The remaining *cached* queries are added to the RCADG Cache (see below). Obviously hotspot queries among the new queries are likely to enjoy a higher cache support, owing to their duplicates and variants in the cache. Note that the probability of a hotspot query being selected for incremental evaluation is equal to the probability that a hotspot query occurs in the entire test set, which is reasonable.

After removing the 19 new queries from the workload and eliminating duplicates among the other queries, 199 queries remain to be added to the cache, as follows. The set of 199 queries is randomly sorted and partitioned into four subsets of 38, 35, 61 and 65 distinct queries, respectively. These are evaluated from scratch, and the answers for each query set are successively added to the initially empty cache. This yields the stages C1 to C4 in Figure 10.11 *d.* on the preceding page. Note that the main-memory footprint of the RCADG Cache is modest even when nearly 1 GB of results are cached on disk (including the B⁺-Trees on intermediate result tables). Since the growth of the cache contents on disk is linear in the number of cache edges, we expect that the system easily scales up further by three orders of magnitude.

The results of evaluating all 19 new queries from scratch and against the five cache stages are listed in Figure 10.11 *b.* on the facing page. For each query (rows) and cache stage (groups of columns), the time, sup and ovh columns respectively list the processing time in milliseconds, the cache support in percent, and the search time (as a percentage of the processing time). Runtime measurements subsume all retrieval phases including rewriting, planning and translation, for all schema hits (those retrieved in the cache and those matched from scratch). For the purpose of analyzing the outcome of the experiment, the new queries are divided into four groups of three to seven members (groups of rows in Figure 10.11 *b.*)⁵

All seven queries in the first group benefit from specific cache contents available in different stages of the cache evolution. For instance, consider the fields time0, time1 and sup0, sup1 in the first two rows in Figure 10.11 *b.* At some point in time during the transition from stage C0 to C1, cache contents have been added that overlap with the queries T0 and T4, which avoids additional joins (cache support 100%) and decreases the processing time by a factor 30 for T0 and a factor 97 for T4. In subsequent stages (C2–C4), the search time increases a little, but clearly does not depend on the overall size of the cache. The other five queries in the upper part of Figure 10.11 *b.* benefit only at later stages (T14 in C2; T15 in C4; T20 and T22 in C2 and later; and T21 in C2). Up to this point where the cache becomes useful for a given query, the cache look-up causes only a negligible overhead.

Of the 130,000 distinct matches retrieved when answering the query T21, 10% are retrieved from the cache after only 0.6 seconds (not shown in Figure 10.11 *b.*). This illustrates how incremental evaluation may increase the reactivity of the system even when only part of the results can be obtained from the cache. Note that for T22 which already has 100% cache support in C2, the performance further improves in C3 and C4 where newly cached queries permit more efficient query plans. This effect, which we also observe for the second group of queries in Figure 10.11 *b.*, is not reflected in the cache values because our primitive cost estimation is too coarse. Figure 10.11 *c.* plots selected results from the first six columns in Figure 10.11 *b.* (scratch and time0 – time4) for all cache stages. Note the negligible overhead introduced by look-ups in the empty cache, compared to the evaluation from scratch (left-hand side of Figure 10.11 *c.*).

The third group of queries in Figure 10.11 *b.* lists queries that do not benefit from cache contents, mostly by lack of overlapping queries in the cache. Again we observe a small search overhead (inevitable for deciding whether or not to use the cache) which grows much slower than the cache. The results of T3 and T13 are computed from 2-3 overlapping queries with small cache support, hence the evaluation from scratch is faster. Query planning with selectivity estimates, as mentioned above, is likely to eliminate such cases. The same applies to the queries in the fourth group (last three rows in Figure 10.11 *b.*), where the

⁵This grouping of the new queries must not be confused with the partitioning of the cached queries into the four cache stages that was described earlier.

cache look-up does not pay off compared to the extremely fast evaluation from scratch.

The query T2 in the third group benefits largely from the fact that every cache edge c_c collectively represents all schema hits that share a particular schema edge. Recall from Section 10.5.2 that this allows to compare the corresponding query constraints only once for the whole set of schema hits in c_c . Due to the structure of the query T2, there are thousands of containing schema hits in the cache which only differ with respect to a single schema edge. We found that matching the many shared schema edges repeatedly would cost a needless extra 14 seconds, which is completely avoided with our data structures. Nevertheless, this observation indicates that in extreme cases where a query entails a huge look-up result, the runtime overhead might be considerable. However, such cases are detected immediately after schema matching and before the schematization, as soon as the number of schema hits becomes available. If a certain threshold is exceeded, one may still decide at this point of the evaluation to look up only some schema hits in the cache, or evaluate the whole query from scratch. Besides, queries with very large result sets should probably not be cached (also in view of the storage consumption, see Section 10.8).

10.7 Summary and Discussion

As a last step in this work on increasing the efficiency of XML retrieval, this chapter has presented the RCADG Cache as a practical example of how to use schema information from a structural summary for incrementally answering XML queries, based on a cache containing both intermediate and final results of prior queries. The benefit of incremental XML retrieval in general has been discussed in the previous chapter, along with some problems and possible solutions that have been ignored so far. The RCADG Cache addresses several of these issues:

Use of query extensions. The incremental retrieval algorithm proposed here is largely tailored to the two-phase retrieval performed by the RCADG, which first matches queries on the schema level so as to obtain document-level matches more efficiently. For the RCADG, the schema tree serves as a path index locating parts of documents with specific properties, such as tag paths and textual content. The RCADG Cache uses the same schema information to retrieve and compare cached queries that resemble a new query to some extent. Here the schema hits provide an approximate view of the query extensions on the document level. Inspecting these result views, one may be able to reuse certain query results in the cache that are ignored by purely intensional approaches. In doing so, the RCADG Cache uses only schema information that is supplied by the RCADG anyway, and therefore does not introduce an extra overhead compared to the evaluation from scratch. Unlike the few DTD-aware systems reviewed in Chapter 9, our approach relies on a descriptive schema and is therefore closer to the current state of the documents.

Reuse of overlapping query results. Most approaches to incremental XML retrieval are quite limited in their effectiveness, taking advantage only of cached queries that are either equivalent or strictly more general than the new query to be evaluated. Moreover, combined query processing with and without cache support has mostly been neglected. The RCADG Cache exploits query overlap to a large extent by partitioning the query extension into sets of matches to distinct schema hits. This way part of the query result may be obtained from the cache while another part is retrieved from scratch. Since the two partial answers are computed independently, the approach is inherently amenable to parallelization. In the end all results are simply put together in a disjoint union. We have outlined an integrated evaluation procedure that efficiently detects and exploits any query overlap that can be handled by the RCADG Cache, and retrieves all missing results from scratch.

Reuse of intermediate query results. It has been mentioned before that intermediate results computed during the evaluation of cached queries have so far been disregarded. In this work, we have shown how the techniques developed for caching final query results can be extended to apply also to intermediate results, and how this allows to answer new queries incrementally for which no final result in the cache could have been reused, thus again increasing the effectiveness of the cache. It turns out that if intermediate results are available, they may be treated in just the same way as final query results, with only a modest amount

of book-keeping required. The main challenge here is to find a suitable representation of the available “snapshots” of the cached query results as they evolved over time. Again, we find that the necessary information is readily available, namely, in the form of query plans that capture all steps the evaluation has gone through to answer the cached query. Thus we efficiently determine which “snapshot” of a given query result is preferable, using no additional data structures.

Of course, keeping both intermediate and final results in the cache entails a higher cost in terms of storage. However, our experiments show that the RCADG Cache scales well up to the gigabyte level in term of both storage consumption and runtime performance. In particular, its main-memory footprint is very low because the bulk of the cache contents is kept on disk.

Runtime performance. In an extensive performance evaluation, we have demonstrated the practical benefit of using the RCADG Cache, assuming an unbiased synthetical query workload. Compared to the RCADG system, the RCADG Cache achieves a speedup of up to two orders of magnitude in our experiments. There are only few cases where the overhead for the cache look-up is not compensated for by faster retrieval, so that the evaluation from scratch is faster. Although the loss in performance is not large in these cases, there is some potential for optimization here. Another issue is the possibly larger overhead caused by extremely unselective queries in the cache (see below).

A comparative study of the performance of different caching approaches is missing at the time of this writing. One reason is that many authors have addressed mainly the theoretical side of the problem. We therefore merely highlight some performance-related differences between earlier approaches and the RCADG Cache. First, the RCADG Cache designed to leverage and take advantage of the efficiency and scalability of the RCADG. In particular, it takes over the relational storage scheme of the RCADG. Recall from Chapter 8 that elements in RCADG result tables are not represented as XML fragments including their entire document subtree, but rather by their unique element label only. As a consequence, the matching of *D*-constraints in the remainder query (such as the keyword constraint in the example above) requires access to data “outside” the RCADG Cache. However, the missing data is simply obtained through a join with the element table, which resides in the same RDBS as the cache contents. Thus remainder queries can be processed as efficiently as any other query, as shown in the experiments.

Marrón and Lausen [2002] argue that the hierarchical *LDAP* data model they use for their *HLCaches* system (see Section 9.4.2) fits XML data better than the relational model. On the other hand, their query interface is quite restricted, and not performance results are given that could support their claim. Kang et al. [2005] survey different storage schemes for XML caches, without commitment to any specific query language. Their experiments suggest that query results cached in binary or plain text format can be retrieved and updated faster than cache contents stored in an RDBS. However, this is mostly due to an extra overhead for serializing relational data to XML text fragments, which are used to transfer results from the database to the cache and further on to the user. Our database-resident cache deliberately departs from such a strict three-tier architecture, hence the results reported by Kang et al. do not apply.

In contrast to work based on Incomplete Trees [Abiteboul et al. 2001b; Hristidis and Petropoulos 2002], the RCADG Cache comes with a main-memory index structure for quick access to cached queries with specific extensional properties. Finally, while Hristidis and Petropoulos store the root path of every cached element in their Modified Incomplete Tree, our approach exploits the BIRD labelling scheme to reconstruct root paths, which therefore need not be cached. This not only expedites the query evaluation, but also reduces the storage requirements of the cache.

10.8 Optimizations and Open Problems

There are a number of ways in which the RCADG Cache may be enhanced over what has been described above. Most optimizations center around cache look-up and cache maintenance issues. The rest of this chapter outlines the most salient issues; a more thorough investigation is left for the future.

Binary query constraints other than *Parent* and *Child*. So far we have assumed that every query to be cached has at least one *Parent* or *Child* edge, and have restricted the cache look-up to these constraints.

As a matter of fact, *NextSib*, *PrevSib* and *Self* edges can be handled similarly. For instance, an additional constraint $NextSib_1^2(q_2^n, q_3^n)$ in the query Q^n in Figure 10.1 c. on page 143 can be handled as follows: during the schematization of Q^n , the *NextSib* edge translates to the S -constraint *Sibling'*. The schema node #2 that matches q_2^n has only one sibling satisfying the tag constraint $gender \vee sex$, namely, node #6. Therefore there is only one schema hit for the modified query Q^n : it looks exactly like $\chi_2^{Q^n}$ and produces a schematization similar to the one in Figure 10.1 g. on page 143, only with an additional schema edge $NextSib_1^2(\#2, \#6)$.⁶ This schema edge is used as a look-up key in C just like any other edge that represents a *Parent* or *Child* constraint. If the query node q_3^n had no tag constraint, then there would be three distinct schema hits for Q^n which would match q_3^n with the schema nodes #2, #3 and #6, respectively.

In contrast to the sibling and *Self* constraints, the D -constraints *Following* and *NextElt* and their reverse variants do not have corresponding S -constraints. In fact, if Q^n contained a query edge $Following(q_2^n, q_3^n)$ and q_3^n had no tag constraint, then q_3^n could be matched by any schema node in S , not only #2, #3 and #6 as with *NextSib*. These binary tree relations are therefore likely to produce too many schema hits and hence too many schema edges to be looked up in the cache. Hence we restrict the schematization to S -constraints as described before. Note that as a consequence, the RCADG Cache cannot handle queries like `//person/following::name` that do not contain any S -constraint.

Transitive query constraints. Another look-up issue concerns the chaining of transitive query constraints such as *Parent* or *Child*. For instance, consider the two queries $Q_1 = //person/profile/edu$ and $Q_2 = //person//edu$ against the same document collection D as in the running example (see Figure 2.1 b. on page 8). From the schema tree in Figure 2.1 c., one can see that both Q_1 and Q_2 have the same schema nodes matching their `person` and `edu` nodes (namely, #1 and #4, respectively). Clearly both $ans(Q_1)$ and $ans(Q_2)$ are part of the RCADG Cache overlap for D , so Q_2 should be evaluated incrementally when Q_1 is in the cache and vice versa. However, if only the two schema edges of Q_1 , $Parent_*(\#4, \#3)$ and $Parent_*(\#3, \#1)$, are cached, a cache look-up for the schema edge in Q_2 , $Parent_*(\#4, \#1)$, will fail. Analogously, looking up either of the schema edges of Q_1 would ignore the schema edge of Q_2 . Note that this does not cause wrong result to be produced, but we needlessly miss a chance for incremental query evaluation, which decreases the efficiency of the cache. The most straightforward solution to this problem is to extend the schematization such that transitive constraints like $Parent_*(\#4, \#1)$ are silently added to the cache as well as to the set of schema edges being looked up in C .

D -constraints in the schema-hit containment test. The notion of schema-hit containment (see Definition 10.1 on page 144) implies that binary D -constraints in a cached query Q^c which do not introduce a proper restriction should be ignored in the containment test, even if they are missing in the new query Q^n being looked up in the cache. For instance, an unmirrored *Parent* edge that was reconstructed during the evaluation of Q^c is admissible because ancestor reconstruction cannot cause partial matches to be discarded. This case is illustrated in Figure 10.1 a. on page 143 for the query edge $Parent(q_3^c, q_4^c)$ in query Q^c (the corresponding schema edge $Parent_*(\#5, \#3)$ in Figure 10.1 d. is highlighted green). Similarly, statistics in the RCADG path table could reveal that certain constraints that are restrictive at first sight (e.g., an existential child constraint expressed in an XPath predicate) are actually safe to ignore in the given document collection. Furthermore, binary constraints in Q^c need not have an exact counterpart in Q^n for schema-hit containment to hold. An example has been given in Section 10.2.1 where the schema edge $NextElt_1^+(\#2, \#5)$ in Q^c corresponds to a more restrictive schema edge $NextSib_1^+(\#2, \#5)$ in Q^n . The procedure *checkSnapshot* as outlined in Algorithm 10.3 on page 156 does not recognize such cases of schema-hit containment. However, the necessary modifications are straightforward.

Cache maintenance. In this work we have not addressed the problems related to maintaining the cache contents over time that were sketched at the end of the previous chapter. In practice these issues are fundamental to any caching technique, not only in XML retrieval. While some efforts have been devoted to the maintenance of XML query caches, the research in this field stills seems very much in flux. Recall

⁶Note that for horizontal relations such as sibling constraints, the normalization must not remove the proximity bounds because these are not fixed as for *Child* and *Parent*.

from Section 9.5 that major questions here are (1) which results to put into the cache; (2) which cache contents to expel to avoid an overflow; and (3) how to initialize the cache so that the system can jump-start with an appropriate sample of the expected query workload. As a possible criterion for deciding the first question, we have mentioned above that highly unselective queries with many schema edges are likely to suffer from a considerable look-up overhead, and therefore should be shunned in caching. Second, to retain the most useful data in a cache of limited size, a replacement strategy is needed. Besides standard strategies for the maintenance of priority queues, like *least-recently/least-frequently used*, possible hints for assigning appropriate priorities could come from explicit user feedback or silent monitoring of user interaction. Alternatively, one might choose to retain those results that were most expensive to compute. Mandhani and Suciu [2005] sketch a simple solution based on a fixed size limit, but only determined an empirical workload-specific value for the threshold. They also propose a warm-up technique for cache initialization.

A fourth problem related to cache maintenance occurs when the underlying document collection is updated. In this situation some or all cache contents may become stale. Since the tag paths to updated elements are available in the element table, we might use the existing main-memory index C to retrieve stale cache contents efficiently, exploiting schema information in the same way as for detecting query overlap. To some extent, the robustness of our cache also depends on the underlying tree encoding. Some more hints at database update techniques are given by Quan et al. [2000], although in a different retrieval scenario.

Part VI

Conclusion

Summary and Discussion

This work is about how structural summaries for XML data can contribute to making XML retrieval systems more efficient. For studying this question the following preliminaries have been introduced. First of all, the notions of XML documents, the structure or *schema* of such documents, their textual contents, and queries specifying structural and textual properties of desired portions of the documents have been defined and summarized in the *Three-Level Model of XML Retrieval* (see Section 2.4 in Part I). Second, we have given an informal definition of the term *structural summary*, which is deliberately general enough to include both *centralized* approximate representations of the document schema and *decentralized* exact representations of relations between individual XML elements, in the form of labelling schemes. Third, we consider distinct kinds of XML retrieval system, namely, *native*, *relational* and *hybrid* ones.

The main argument of the thesis is that certain kinds of structural summary, when applied appropriately, speed up the query evaluation significantly even in very large collections of XML documents, while causing only a modest storage overhead. This claim is underpinned by extensive experiments that evaluate the proposed new techniques and also compare them to prior approaches known from the literature. The preceding parts of this work have considered different types and various aspects of structural summaries that all contribute to the efficiency improvement, which is achieved in native, hybrid and purely relational retrieval systems. In the sequel we briefly recapitulate our findings, highlighting both problems that have been solved and questions that remain open.

Part II: Labelling Schemes for XML. Labelling schemes are decentralized structural summaries that serve to match binary query constraints on the document level without accessing the documents themselves. Query constraints can be either *decided* or *reconstructed*. Labelling schemes differ greatly in their expressivity (i.e., if and how they match specific tree relations), time and space efficiency, and robustness against modifications to the document tree. These conflicting optimization goals span a *trade-off space* where different labelling schemes occupy different positions. In Chapter 3 we have seen three classes of labelling schemes. First, *subtree encodings* (including as subclasses *interval*, *pre-/postorder* and *region encodings*) use node labels that represent the size of the subtree of a given document node. This gives them rich decision capabilities and limited robustness, but prevents support for reconstruction. Second, *path encodings* (which subsume *total* and *partial path encodings*) concatenate node labels along the root path of a given document node. The resulting labels are possibly large and therefore compressed using different binary encodings. For reconstruction and decision the labels can mostly be manipulated in their binary form. Path encodings are typically fairly robust against document updates. Third, a small number of *multiplicative encodings* label the document tree as if it had a highly regular structure, using different non-materialized homomorphisms. The resulting labelling schemes are typically rather sparse but offer fast decision and reconstruction of many tree relations.

Chapter 4 has presented the *BIRD* labelling scheme [Weigel et al. 2005c; Weigel et al. 2005d], a multiplicative encoding whose labels are created using certain numerical components, or *weights*, that reflect properties common to multiple document nodes. BIRD uses the schema tree as a centralized structural summary to ensure fast access to the weights for reconstruction and decision. BIRD is among the most

expressive labelling schemes in the literature. Experiments show that it outperforms almost all other approaches in terms of retrieval speed and maximal label size. The efficiency of reconstruction and comparison operations is shown to be paramount for good retrieval performance. Only one competitor of BIRD is much more space-efficient, but less expressive. A major drawback of BIRD in its current form is its poor updatability when faced with node insertions in certain positions of the document tree. Several potential optimizations have been sketched that should make BIRD labels and weights more stable and at the same time smaller than in our experiments.

Part III: Index Structures for XML. Different native index structures for XML documents are surveyed in Chapter 5, including traditional inverted lists from flat-text Information Retrieval, adaptations thereof to elements with their tag paths, and finally tree data structures like the schema tree that can be used to index tag paths and keywords simultaneously. The latter are mostly variants of centralized structural summaries like the schema tree, and as such can be combined with labellings schemes for better retrieval performance. The look-up latency of such path indices depends much on how tag paths, textual contents, and their combinations are physically represented, especially when processing queries with branching paths.

Chapter 6 briefly reviews the *CADG* index [Weigel et al. 2004a; Weigel 2003], which achieves significant performance gains by materializing the join of tag path and keyword information that prior approaches have computed at runtime. Of course this also has an impact on the size of the index structure, but the space overhead is modest and practically restricted to secondary storage. Note, however, that rather than trying to assess the retrieval performance of a path index in isolation, it makes more sense to take into account also which labelling schemes and structural join algorithms are used with it. In Part II we have given the results of combining the *CADG* with different labelling schemes in a hybrid retrieval system, with positive outcome in terms of time- and space efficiency as well as scalability. In the remainder of the thesis, the combination of *CADG* and *BIRD* is further evaluated in a relational setting.

Part IV: Relational Storage of XML. For various reason mentioned in the introduction to this work (see Chapter 1), the storage and retrieval of XML data in relational database systems has aroused much interest in recent years. In Chapter 7 we have reviewed different ways to store the inherently hierarchical XML documents in specific schemata of the flat and rigid relational data model. Most earlier approaches simply represent either singleton elements or pairs of parent and child elements as tuples in a table, thereby losing the originally explicit information about the nesting of elements and tags. Expensive structural joins are needed to restore this information when matching query constraints on the document level or the schema level at runtime. Only few relational storage schemes have been described in the literature that attempt to represent schema-level information (most notably, tag paths) in the RDBS. They mainly suffer from a lossy representation of the hierarchical nesting in the documents, which can cause many partial matches to be retrieved in vain during the evaluation of tree queries.

Our experiments in Chapter 8 reproduce such cases where the sets of intermediate result retrieved by these systems needlessly blow up to millions of elements, compared to several hundred with our approach. We basically use the same combination of *CADG* and *BIRD* as in Chapter 4, after migrating both to the relational data model. The storage scheme of the resulting *Relational CADG (RCADG)* [Weigel et al. 2005b] is straightforward since we only need to fix a suitable relational schema for the structural summary part of the *CADG*. However, we carefully avoid the lossy representation of tag paths mentioned above. The *RCADG* also comes with some basic query rewriting techniques that could probably be extended. However, the core of the relational query evaluation with the *RCADG* is the query planning algorithm which is described in great detail. The algorithm is designed to benefit as much as possible from *BIRD*'s reconstruction capabilities, which have proved crucial to good performance in our experiments with the hybrid system (see Chapter 4).

As a matter of fact, there are at least two conflicting optimization goals that a good planning strategy should try to reconcile somehow. On the one hand, the query should be evaluated with the least possible number of joins with the element table, where the bulk of the document-level information resides. Every binary query constraint that is reconstructed (in our case, using *BIRD*) saves one join with the element table. On the other hand, to minimize the size of intermediate results to be joined when matching branching queries, the element table should be initially probed with the most restrictive selection predicates. The

problem here is twofold. First, those query nodes with the most selective unary query constraints are not necessarily those which allow to reconstruct a large number of binary constraints, hence the conflict in query planning. Moreover, while the selectivity of tag path constraints is easily kept in the structural summary, estimating the cardinality of the set of matches to a combined path/keyword constraint is non-trivial given a limited amount of space for storing selectivity statistics. Therefore we currently apply a simple heuristic that simply prefers query nodes with any keyword constraint over those without keyword constraints, regardless of the frequency with which the actual keyword in the query occurs in the documents.

While this already yields very good results in our experiments, where the RCADG outperforms other relational and hybrid systems by up to three orders of magnitude, we expect even better results from more sophisticated query optimization and planning techniques. Another possible enhancement is the implementation of a structural join operator in the RDBS. Currently the RCADG uses standard nested-loop and indexed-loop joins since no tree-aware operator is available in our RDBS.

Part V: Caching Techniques for XML. Chapter 9 surveys a number of approaches to the incremental evaluation of XML queries using cached results of earlier queries. The different caching techniques are compared in terms of various criteria of theoretical or practical interest. These include, among others, the underlying data and query model, the way cached queries and results are represented, the extent to which only partially relevant query results in the cache can be reused, and the scalability of the approach (determined by the cache size and the look-up latency). A major issue here are the notions of *query containment* and *query overlap*. Even the apparently unambiguous idea of query containment between a cached and a new query can have different meanings for semistructured data, depending on whether the entire result of the new query must be physically present in the cache or whether only each match to the new query must have a (possibly partial) counterpart in the result of the cache query. In the latter case, additional joins with the element table may be needed to obtain the complete result of the new query, however this might still be done much faster than evaluating the new query from scratch. In a third variant, only some of the matches to the new query are retrieved in the cache whereas the remaining matches must be computed from scratch. This requires the integration of distinct query evaluation procedures that may or may not use the cache.

To facilitate the comparison of the various cache proposals in the literature, we therefore formally define different degrees of query containment and query overlap. It turns out that almost all known approaches are restricted to full query containment. Notice that this does not mean that all cases of strict query containment are detected. Since this problem has exponential complexity, all of the reviewed algorithms are incomplete. This also applies to the *RCADG Cache*, an XML query cache that we introduce in Chapter 10. The RCADG Cache enhances the RCADG with efficient and scalable incremental query processing. Unlike almost all other approaches mentioned before, the RCADG Cache compares queries not only based on their *intensions*, but also their *extensions* on the schema level, which provide an approximate view on the actual query results on the document level. In a process called *schematization*, queries are first matched against the structural summary to find representatives, or *schema hits*, of different disjoint parts of the query result, which is unknown at that time. The schema hits are then looked up in a main-memory index to the cached queries and results in the RCADG Cache. A sophisticated comparison of query intensions and extensions (i.e., query constraints and schema hits) allows to detect certain cases of containment or overlap which cannot be exploited without the structural summary, even if a DTD is given. Again, the use of structural summaries brings a decisive advantage over schema-oblivious approaches, a phenomenon that we already observed in previous parts of this work.

Furthermore, the RCADG Cache is the only XML cache we know of that exploits not only final but also intermediate query results which usually emerge naturally during query evaluation. In the case of the RCADG, intermediate results are conveniently stored as temporary tables in the RDBS, which need only be made persistent to be included in the cache. Obviously, caching more results generally increases not only the effectiveness, but also the size of the cache. Therefore one major challenge to be overcome for exploiting query overlap and intermediate results was the design of a suitable cache index and look-up procedure that enable fast access to potentially relevant queries in the cache, even when the overall number of cached queries is huge. A second precondition for successful exploitation of overlapping queries in the cache is that those matches to a new query that cannot be obtained from the cache must be computed

from scratch, and later be combined with the remainder of the query results that was retrieved in the cache. Since the schema hits we use for selecting relevant cache contents represent disjoint sets of matches on the document level, we can simply evaluate the same query with and without cache in parallel and finally union the partial result sets without duplicate elimination. In Chapter 10 we give a detailed description of all necessary data structures and algorithms, along with several examples that illustrate the benefit of the salient contributions of the RCADG Cache, namely, query overlap detection and use of intermediate results.

To evaluate the RCADG Cache, we have conducted two different experiments that simulate potential user behaviour in an interactive retrieval system such as the one sketched in the introduction to this work (see Section 1.3). We find that with the RCADG Cache, the query evaluation is accelerated by up to two orders of magnitude, depending on the query workload assumed to fill the cache. A careful set-up monitors a growing cache as it may evolve during the continuous use of a cache-enabled retrieval system. An obvious issue here is the maintenance of the cache over time, most notably, the choice of query results to be cached and others to be expelled from the cache when it grows too large. These questions are not fully addressed here, however preliminary solutions have been outlined.

Perspectives and Outlook

At this point, where all issues that are covered by this thesis have been mentioned and all contributions made in about three years of work have been developed, documented and evaluated, a final word is in order on how the results can guide or entail future work in the field. We will briefly recapitulate what can be gathered from this work as far as efficient XML retrieval is concerned, and then sketch two other applications of structural summaries that can also benefit directly or indirectly from our results.

12.1 Lessons Learnt

The key conclusion that should be drawn from what has been presented here is that *structural summaries are indeed at the core of making XML retrieval efficient and scalable enough to face today's challenges and tomorrow's expectations*. We have seen how structural summaries can solve some of the most fundamental problems that arise during XML query evaluation, with whatever system or technique:

1. provide fast access to occurrences of specific tags or tag paths in the documents;
2. identify certain unsatisfiable queries immediately, without accessing the documents;
3. decide the question whether a certain tree relation holds between two elements, in constant time without any I/O;
4. reconstruct a part of the neighbourhood of a given element, in constant time without any I/O;
5. hold data that is specific to a certain class of elements, so that it is readily available for any of these elements without redundant storage.

These features have been exploited at various places throughout this work: the native retrieval system X^2 uses the CADG as path index and BIRD for deciding and reconstructing tree relations without access to the document level; the relational retrieval system *DoX* does the same with the RCADG; BIRD uses the CADG or RCADG to store its weights; the μ PID scheme does the same with the DataGuide; and so on (another example will be given below). From a bird's eye view, the reason why structural summaries are the method of choice in the various cases is always the same: simply because they provide the right amount of information about the underlying XML data in the right way, and ignore the rest. This is exactly what users expect from a retrieval system, and also what the query kernel expects from its index structures and access paths. In other words, a "good" structural summary for a given purpose provides just the right abstraction of the data that is needed to avoid the expensive manipulation of the data itself. The *Three-Level Model of XML Retrieval* depicted in Figure 2.3 on page 13 is meant to visualize just that intuition, in a sufficiently generic way to be applied also to other retrieval scenarios that are different, but related. For instance, the picture might be adapted to a streamed data source, or a set of distributed data sources, or the combination of distinct summaries (abstractions) on multiple intermediate levels.

From a more down-to-earth perspective, the different data structures and algorithms introduced in this work are of course the predominant contribution, which we hope will be broadly applicable in other situations where similar problems arise. While the RCADG Cache with its rather specialized data structures is likely more interesting from a system-centric point of view, BIRD and the RCADG storage scheme are generic enough to be adopted without much need for modification. The abridged survey of labelling schemes presented in Chapter 3 illustrates such transfer of solutions across quite disparate domains of research: thus some of the most frequently cited labelling schemes in the XML literature were actually designed for routing in communication networks. In fact, some of the work that was done before the advent of XML in the Discrete Mathematics community seems to have been reconsidered (and sometimes rediscovered) later, with new applications in mind.

It is equally true, however, that much of the more theoretical achievements in labelling tree and graph data never made it into the Related Work sections of papers on XML retrieval. The history of science probably abounds with examples where “new” solutions (more precisely, solutions yet unconsidered) to a specific problem emerged just because someone realized the link to work that had been done by someone else before. This is said to emphasize the value of surveys and analytical or empirical comparisons of alternative approaches to similar, if not identical questions. Thus if a prominent place in this work has been reserved for classification, systematic comparison, visualization, and terminology, this was done with such methodological considerations in mind.

A practical application of the classification criteria that have been proposed for labelling schemes could be a recommender tool that suggests a suitable scheme to be applied to a given document collection, based on characteristics of the documents (e.g., structural heterogeneity, maximum path length, maximum fan-out, markup-to-text ratio), of the data source (static versus dynamic, continuous versus bulk updates), of the query workload to be expected (most common tree relations queried, frequency of complex branching patterns, proportion of structural to textual query constraints), of the runtime environment (primary and secondary storage available, access speed to secondary storage) and of the user’s skills and expectations (expert versus novice, real-time information need versus off-line analysis). It is easy to see that these parameters reflect quite closely the optimization goals of different labelling schemes that are plotted in Figure 4.10 on page 65.

Such a recommender tool is also conceivable for choosing a suitable path index, or for indexing frequently queried parts of the document tree in some privileged fashion. Similar indexing assistance is already offered by some commercial RDBSs. Analogous techniques might also apply to cache maintenance, where the system must decide which query results to put into the query cache and which to expel once the available resources (storage or look-up time, in the case of very unselective queries) are exhausted. In the end, monitoring the aforementioned user, data and system parameters (which may change over time) could lead to a largely autonomous system administration agent running in the background. Going through continuous maintenance cycles, it would adjust the system set-up to the current real usage, rather than the fictitious usage assumed once before the system start-up.

12.2 Further Applications of Structural Summaries

Finally, we would like to hint at two other aspects of XML retrieval besides efficiency where structural summaries are useful, namely, relevance ranking and user interaction in XML retrieval systems. These fields being beyond the scope of this work, the following description is necessarily cursory. A more balanced discussion of the various benefits of structural summaries is found elsewhere [Weigel 2006].

12.2.1 Relevance Ranking

In Chapter 1 it was pointed out that Information Retrieval (IR) systems for XML documents face the problem of relevance ranking with respect to both the textual contents and the markup structure of the documents. Most ranking models for structured documents are adaptations of flat-text models such as *tf·idf* [Salton and McGill 1983], which computes relevance scores based on (1) the *term frequency*, i.e., the number of occurrences of a given term (keyword) in a specific document, and (2) the *document frequency*, i.e., the number of documents in the collection that contain at least one occurrence of that term. When

applying such models to XML documents, other or perhaps additional frequencies are needed that reflect the distribution of terms with respect to distinct elements or tags or tag paths. For instance, some XML ranking models redefine the document frequency as the number of elements *with a specific tag path* that contain least one occurrence of a given term. Note that with this definition the document frequency is a function of a term and a tag path, while the former definition above treated it as a function of a term only.

Redefining frequencies in this way has immediate consequences for the storage structures used to implement a given ranking model. For instance, while the merely term-specific document frequency (first definition above) easily fits an inverted text file (see Figure 5.1 *a.* on page 72), the term/tag path-specific document frequency (second definition above) has no place in the inverted text file because tag path information is not covered by this data structure. Suitable index structures for this sort of document frequency values include, e.g., the inverted text/path file (see Figure 5.1 *d.*), the two- or three-dimensional path bitmaps (see Figure 5.2 on page 73 and Figure 5.3 on page 74, respectively) and the element table of the CADG (see Figure 6.1 on page 82).

In earlier work [Weigel et al. 2004b] we have developed a method to find out which index structure is capable of storing all sorts of frequency that are used by a particular ranking model for structured documents. A generic classification scheme, the *Path/Term/Node Hierarchy*, is introduced which describes XML ranking models in terms of their dependency on three building blocks of an XML document (namely, *Path*, *Term*, and *Node*). Since the same vocabulary is used to specify which document properties a given XML index can store, the *Path/Term/Node Hierarchy* makes it easy to relate the needs of a ranking model to the capabilities of an index. It turns out that the more basic ranking models can benefit from most of the centralized structural summaries reviewed above. However, among the index structures presented in Part III of this work, only the CADG (see Chapter 6), the IndexFabric (see Section 5.4.2) and the inverted text/path file (see Section 5.2) are capable of storing so-called *PTN frequencies*, i.e., frequency values that are at the same time term-, path- and node-specific. *PTN* frequencies are used by some of the more sophisticated ranking models such as *XPRES* [Wolff et al. 2000].

Following the *PTN* analysis of the CADG and other index structures, we have created a modified CADG, the *Integrated-Ranking CADG (IR-CADG)*, which can be configured so as to meet the demands of a variety of different ranking models for XML [Weigel et al. 2005a; Weigel et al. 2004b]. The IR-CADG is an example of how the ranking of structured documents can take advantage of centralized structural summaries. In terms of the Three-Level Model of XML Retrieval, the frequency values stored in the summary make certain properties regarding keyword and path distributions on the document level visible on the schema level. This permits systems to use advanced ranking models with complex frequency parameters that mirror more closely the textual and structural properties of the documents, which may eventually lead to better precision and recall. As a by-product, efficiency benefits discussed for unranked XML retrieval carry over to the ranked case.

12.2.2 User Interaction

Section 1.3 has sketched a new way for users to interact with the retrieval system, which makes heavy use of a graphical representation of the schema tree as structural summary. In earlier work [Meuss et al. 2005] we have described a preliminary version of such a graphical user interface (GUI). The system proposed there provides separate views on the schema, queries and results. Once a query has been formulated and evaluated, the retrieved hits are explored in a graphical representation reflecting the query structure. While browsing the result view, users often wish to modify the query, realizing mismatches with their information need. Currently this requires re-editing and re-running the query outside the result view (perhaps after consulting the schema again). This not only causes needless computations to retrieve data which is already known, but also makes it hard for the user to keep track of updates to the query result.

The most salient feature of the new GUI to be developed is the tight integration of the schema, query and result views. Ideally the user would silently issue new queries or modify previous ones while browsing the document schema or query result through a point-and-click interface, as follows. First the user activates interesting tag paths in the schema tree and perhaps annotates them with keyword constraints.¹

¹Note the similarity between these user-specified query patterns in the schema tree and the schematized queries introduced for caching purposes in Chapter 10.

The occurrences of these tag path patterns (actually, manually collected schema hits) span a set of subtrees in the documents which in turn induce a partial schema tree that is specific to the current activation. For instance, consider the sample document tree D in Figure 2.1 *b.* on page 8. Initially the schema tree S for D looks like the one shown in Figure 2.1 *c.* on page 8. If the user activates the `person` and `gender` nodes in S , then the subtrees a_1 to a_3 in D are temporarily ignored, being irrelevant to the current user interest (since neither of the three subtrees contains a `gender` node). Consequently, the schema tree S is reduced to reflect exactly the schema of the remaining subtree a_4 of D . In this case, this means that the schema nodes #3, #4 and #5 disappear from the schema view. Whenever the user changes the activation pattern in the schema view, the structure of the schema tree shown there is immediately updated, e.g., by hiding paths outside the reduced schema as above, or by making hidden paths reappear. Note that finding the currently relevant subtrees of D can benefit from our efficient tree matching techniques, just like the evaluation of explicit user queries.

Note that the user can at any point in time either narrow down or expand the schema tree S by changing the path activation. Moreover, distinct paths can be *merged*, i.e., treated as equivalent both in query evaluation and in the GUI. Conversely, occurrences of the same tag path can be distinguished in S , based on their textual content or statistics such as subtree size, by *splitting* the corresponding node in the schema view. In terms of the Three-Level Model of XML Retrieval, this blurs to some extent the distinction between the schema and document levels. However, since users are in full control over the shape of the schema tree, we believe that this feature will actually help them locate relevant information in the document and schema trees more naturally than when a rigid separation of the two levels is enforced at all times. Again, the benefit of structural summaries results from their providing the “right” abstraction of the document contents. Users should ideally decide themselves what level of abstraction is currently appropriate, given the information need they have in mind.

Part VII

Appendix

Experimental Set-up

13.1 Hardware and Software

Test Environment A

CPU: *AMD Athlon XP 2600+*, 2.1 GHz, 256 kB cache
RAM: 1 GB
OS: *Slackware Linux*, version 9.1, kernel version 2.4.26
RDBS: *PostgreSQL*, version 7.3.2 (database cache disabled)
JAVA: *Sun JDK*, version 1.4.2

Test Environment B

CPU: *AMD Athlon XP 1800+*, 1.5 GHz
RAM: 1 GB
OS: *SuSE Linux*, version 8.2, kernel version 2.4.20
RDBS: *PostgreSQL*, version 7.3.2 (database cache disabled)
JAVA: *Sun JDK*, version 1.4.1

Test Environment C

CPU: *AMD Athlon XP 2600+*, 2.1 GHz, 256 kB cache
RAM: 1 GB
OS: *Slackware Linux*, version 9.1, kernel version 2.4.26
RDBS: *PostgreSQL*, version 7.3.2 (database cache disabled)
JAVA: *Sun JDK*, version 1.5.0

13.2 Document Collections

name	XML size	nodes	keywords	tag paths	depth
<i>Cities</i>	1.3 MB	16,000	19,000	253	7
<i>XMark 29</i>	30 MB	417,000	84,000	515	13
<i>DBLP</i>	157 MB	5,390,160	757,451	129	7
<i>NP</i>	510 MB	4,585,000	130,000	2,349	40
<i>INEX</i>	536 MB	12,049,113	496,169	10,203	17
<i>XMark 1100</i>	1,145 MB	20,532,979	84,000	549	13
<i>IMDb</i>	8,633 MB	83,404,825	2,340,060	276	5

Table 13.1: Test document collections.

Comparative Performance Evaluation of Five Labelling Schemes

This section provides some detailed observations from the comparative performance evaluation of different labelling schemes that is described in Section 4.6.3. As mentioned there, the BIRD, ORDPATH, μ PID, and Virtual Nodes schemes are compared against each other and against the preorder labelling as baseline. The experiment is carried out on two document collections, *DBLP* and *XMark 1100* (see Section 13.2 above). Each set of queries against either collection is evaluated repeatedly with three different path join strategies, namely, *ALWAYS*, *FIRST* and *NEVER* [Weigel et al. 2005c].

Tables 14.1 *a.* and *b.* list the eight queries that are run against the *DBLP* and *XMark 1100* collections, four against each. Queries with equal number resemble each other to a certain extent: both *XMark 1100*'s and *DBLP*'s Q0 queries are small trees with a single branching node, a textual constraint and a moderate number of results (where matches for all query nodes are counted as mentioned above). The Q1 queries are structurally similar but lack the textual constraint, which makes them less selective than their Q0 counterparts. The Q2 queries stress the path join capabilities of the system, whereas each of the Q3 queries consists of only one path.

The detailed performance results for all queries against the *DBLP* and *XMark 1100* collections are given in Tables 14.2 *a.* and *b.*, respectively. For each of the three path join strategies, there are five columns listing the average time in milliseconds spent by a given labelling scheme in different evaluation stages for a given query. Each of the five stages accumulates all instances of one of the following problems that occur during evaluation of a single query:

1. REC: reconstruction of the $parent^i$ relation
2. DEC: decision of the $Child^i$ relation¹
3. JOIN: path join (subsumes part of REC, DEC and COMP)
4. FETCH: retrieval of document nodes from the RDBS²
5. COMP: node label comparison

Running Q0 against both collections produces largely similar results. When applying the *ALWAYS* strategy, BIRD outperforms ORDPATH and μ PID and is 2-3 times faster than Virtual Nodes thanks to faster reconstruction, whereas preorder is prohibitively slow. This changes when the *FIRST* strategy introduces decision. On *DBLP*, preorder evaluation of Q0 is even slightly faster than BIRD (2.2%) and outperforms Virtual Nodes by far. The latter is especially handicapped during the join. On *XMark 1100*, preorder is clearly inferior to any other scheme for *FIRST*. μ PID and BIRD are more than twice as fast as ORDPATH

¹This subsumes part of COMP. Note that the Virtual Nodes scheme decides $Child^i(u, v)$ for two document nodes u, v by reconstructing $parent^i(v)$ and then testing whether the reconstructed ancestor label equals u . This extra reconstruction is subsumed by DEC and not included in REC values.

²Note that since preorder labels support neither decision nor reconstruction, REC, DEC and JOIN may subsume considerable portions of fetching time in the baseline tests.

QID	HITS	QUERY
Q0	136	//article[./author[contains(., "codd")] and ./title]
Q1	4805	//incollection[./author and ./title]
Q2	1269	//article[./author[contains(., "i")] and ./title/i and ./year]
Q3	5419	//book/cite/*/@attribute::label

a. *DBLP*

QID	HITS	QUERY
Q0	128	//europe/item[./parlist[contains(., "bedford")] and ./emph/keyword]
Q1	14699	//europe/item[./parlist and ./emph/keyword]
Q2	225	//site/namerica/item[./description/keyword[contains(., "abandon") and ./bold] and ./name and ./*/@attribute::category]
Q3	1777	//people/person/address/city[contains(., "munich")]

b. *XMark 1100*

Table 14.1: Sample queries against the *DBLP* and *XMark 1100* collections (see Section 4.6.3).

and beat Virtual Nodes by one order of magnitude. Applying *NEVER* slows down evaluation roughly by a factor 2 on *DBLP* and much more on *XMark 1100*. Due to faster decision, BIRD remains on the top.

Evaluating Q1 on *XMark 1100* takes somewhat longer than evaluating Q0 (typically one order of magnitude) because due to the missing textual query constraints, far bigger node sets must be joined. The size of the query results differs by two orders of magnitude. BIRD and μ PID retrieve more than 14,000 nodes in less than 3 seconds, followed by ORDPATH (6 seconds). As before, performance breaks down when reconstruction is disabled. Thus the performance ranking is similar to Q0 except that for *FIRST* and *NEVER*, Virtual Nodes is far slower even than the baseline since its join handicap weighs particularly heavy for this query. On *DBLP*, Q1 reveals a pattern similar to Q0 but is evaluated much faster. The reason is that the number of matches to all three query nodes in Q0, ignoring the textual constraint, exceeds that for Q1 by two orders of magnitude (e.g., 157,382 titles in Q0 versus 1,195 titles in Q1). Therefore joining is much easier for Q1 even though the final result is bigger than that of Q0. As a consequence, nearly 5000 nodes are retrieved in only a few hundred milliseconds by most schemes and strategies.

The evaluation of Q2 on *XMark 1100* is lengthy despite the small number of final matches. After all, joining sets of some 100,000 name nodes, 100,000 bold nodes and 380,000 category nodes with the 102 keyword nodes containing the query keyword puts the system to a hard test. Without decision, BIRD and μ PID do the job in 14 seconds, saving 20 seconds compared to ORDPATH and Virtual Nodes. As for Q0 and Q1, the preorder scheme is not competitive. With the *FIRST* strategy, where decision comes into play, the former three schemes are not affected whereas the response time of Virtual Nodes grows by a factor 1.8 due to the join overhead. Interestingly, preorder benefits largely from decision for joining, increasing its performance by a factor 40 compared to *ALWAYS*, and evaluates Q2 slightly faster than BIRD. The top-down join algorithm applied by *FIRST* lets preorder save much time that is otherwise needed for reconstruction (and hence, fetching). Disabling reconstruction decreases the performance by roughly a factor 3, but the scheme ranking remains the same.

On *DBLP*, the task is somewhat easier (as long as reconstruction is allowed) because the `//title/i` branch has only 664 matches, which quickly narrows down the 3,747 candidates of the leftmost branch in the Q2 tree. Consequently, performance figures for *ALWAYS* and *FIRST* hardly change compared to Q0 (BIRD before ORDPATH, μ PID, as well as Virtual Nodes and preorder). With reconstruction disabled, however, fetching 157,382 `article` matches slows down the evaluation and increases the differences between individual labelling schemes. As observed for *XMark 1100*'s Q2 query, BIRD outperforms ORDPATH and μ PID by 1 second, preorder by 4.5 seconds, and Virtual Nodes by 5 minutes. The latter again suffers from the join overhead.

Finally, the queries Q3 are degenerated trees each consisting of a single path, such that there are no decision and join costs for *ALWAYS* and *FIRST*. As could be expected, differences between these two strategies in the performance of any given labelling scheme are negligible on either collection. BIRD

QID	SCHEME	PATH JOIN STRATEGY (USE OF RECONSTRUCTION)														
		ALWAYS					FIRST					NEVER				
		REC.	DEC.	JOIN	FETCH	COMP.	REC.	DEC.	JOIN	FETCH	COMP.	REC.	DEC.	JOIN	FETCH	COMP.
Q0	BIRD	95	0	555	1344	3163	0	0	479	1372	3145	0	0	454	3288	5777
	ORDPATH	309	0	754	1458	3305	0	1	463	1442	3337	0	1	482	3435	6071
	μPID	105	0	633	1321	3210	0	0	489	1302	3235	0	1	515	3293	5862
	Virtual Nodes	279	0	871	1789	8196	0	11155	13600	1593	8229	0	19207	22846	3189	14217
	Preorder	105985	52	115670	104931	3298	112	96	560	1423	3126	177	191	657	2728	5762
Q1	BIRD	3	1	46	44	87	3	2	171	53	83	0	1	142	51	94
	ORDPATH	6	2	82	43	90	5	5	169	46	96	0	9	146	44	102
	μPID	3	1	60	36	86	3	2	99	42	88	0	7	178	45	93
	Virtual Nodes	7	4	83	37	224	8	3951	4621	51	337	0	6415	7826	39	232
	Preorder	3991	1671	2614	3953	108	3950	2493	2617	3896	109	5558	5963	6333	5484	136
Q2	BIRD	121	0	276	1465	2842	1	0	137	1359	2847	0	2	265	4695	7910
	ORDPATH	275	0	436	1498	3045	4	0	136	1402	2983	0	8	304	4844	8461
	μPID	116	0	290	1377	2934	1	1	148	1316	3025	0	5	301	3956	8117
	Virtual Nodes	279	0	480	1653	7937	5	1683	1915	1943	7485	0	313732	372942	5635	20749
	Preorder	101840	2	109615	101066	3040	1577	237	385	2799	2818	4259	4560	4886	7950	7960
Q3	BIRD	4	0	0	33	113	3	0	27	119	0	3	238	63	149	
	ORDPATH	25	0	0	35	114	10	0	0	40	124	0	14	270	66	168
	μPID	2	0	0	32	113	5	0	0	41	121	0	10	253	50	156
	Virtual Nodes	22	0	0	40	460	9	0	0	38	373	0	9821	12011	55	402
	Preorder	3677	0	0	3654	153	3645	0	0	3592	136	7189	7767	8284	7088	192

a. DBLP

QID	SCHEME	PATH JOIN STRATEGY (USE OF RECONSTRUCTION)														
		ALWAYS					FIRST					NEVER				
		REC.	DEC.	JOIN	FETCH	COMP.	REC.	DEC.	JOIN	FETCH	COMP.	REC.	DEC.	JOIN	FETCH	COMP.
Q0	BIRD	23	0	34	214	322	4	0	18	119	321	0	6	913	1710	2427
	ORDPATH	43	0	54	92	1416	22	1	39	87	1390	1	60	765	1856	11229
	μPID	10	0	31	234	303	5	1	26	101	322	0	14	431	2190	3206
	Virtual Nodes	89	1	91	161	1562	20	4138	4385	123	1516	1	288881	325576	3663	11933
	Preorder	22489	57	17651	22199	355	6402	1218	1244	6354	354	14764	15668	16427	16071	3191
Q1	BIRD	22	1	224	449	1856	24	4	226	431	1879	0	28	2442	2181	5560
	ORDPATH	135	13	396	422	6276	127	33	451	408	6236	0	299	4691	2543	18612
	μPID	45	4	239	406	1882	28	9	268	391	1895	0	71	2856	2142	5475
	Virtual Nodes	246	43	485	520	6840	201	630458	672159	736	6561	0	4789521	5255989	3306	19672
	Preorder	85921	14391	31301	84439	2256	88464	34430	34903	86842	2346	161888	174078	179089	161040	6225
Q2	BIRD	277	0	1119	4823	10647	0	0	739	4401	10614	0	0	991	6559	14670
	ORDPATH	1208	0	2483	5772	34048	0	0	1296	5058	34620	0	1	1117	8291	34493
	μPID	297	0	1346	4664	10621	0	0	856	3962	10690	0	0	782	7214	14297
	Virtual Nodes	1381	0	2681	6153	33399	0	28756	32258	5915	34196	0	35127	39637	10018	43217
	Preorder	523485	272	553241	516648	11439	357	304	1209	4617	10590	558	555	1345	6670	13854
Q3	BIRD	1	0	0	9	32	1	0	8	29	0	1	64	3676	8707	
	ORDPATH	3	0	0	9	68	2	0	8	80	0	46	159	4980	23936	
	μPID	2	0	0	8	26	0	0	7	30	0	2	71	3200	8543	
	Virtual Nodes	2	0	0	9	72	3	0	9	81	0	200549	242698	5103	22680	
	Preorder	874	0	0	867	34	879	0	0	867	37	4941	5004	5117	8305	8243

b. XMark 1100

Table 14.2: Efficiency profiling of query evaluation with different labelling schemes (see Section 4.6.3).

retrieves 1,777 matches from *XMark 1100* in 30 milliseconds on average, more than three times as fast as *ORDPATH*. *μPID* comes close behind. Disabling reconstruction, the *NEVER* strategy entails fetching for all inner nodes on the query path. While on *DBLP* this causes 3,748 nodes to be fetched, which affects only the performance of *Virtual Nodes* and *preorder* whose decision is less efficient, on *XMark 1100* 382,316 nodes undergo fetching and joining. Again *BIRD* and *μPID* cope best with the decision problem (10 and 11 seconds, respectively), followed by *preorder* (14 seconds), *ORDPATH* (25 seconds, due to label comparison), and *Virtual Nodes* (3.8 minutes, due to the join overhead).

INDEX

- access control, 6
- alignment, 28, 33
- ALWAYS, 61–63, 183, 184
- arithmetic, 17, 34, 42, 59, 99
- arity, 32, **32**, 33, *see also* fan-out
- artefact, 61, 84, 160
- asymptotic behaviour, 27, 37, 38, 60
- attribute, 6, 7, 7fn, 8, 9, 9fn, 11, 22, 23, 52fn, 77, 81, 90, 103

- B⁺-Tree, 23, 73, 84, 119, 124, 125, 160, 163
- back-end, 57, 84, 119, 133, 134, 160
- backtracking, 75–79, 81, 82
- balancing, 43, 44, 46, 51, 54, 57, 64, 66
- benchmark, 57, 104, 118–120, 123
- bibliographic data, 3, 54, 118
- binarization, 36
- binary encoding, *see* encoding
- bit-string, 27, 29, 33, 37, 59, 76, 107
 - operator, 106, 107fn
 - separator, 28, 29fn, 33
- bitmap, 73, 74, 79
 - path, 73, 74, 79, 177
- Boolean
 - function, 19
 - keyword constraint, 74
 - operator, 99, 153, 154
 - retrieval, 73
- breadth-first, 7fn, 35, 35fn, 36, 38, 59, 63, 109
- browsing, 5, 6, 159, 177
- bulk update, 40, 176

- cache
 - content, 130, 133, 134, 136, 138, 141, 146, 148, 159–161, 163, 165–167
 - irrelevant, 140, 150, 163
 - reusable, V, 134, 136–142, 147, 158, 160, 174
 - database, 57, 181
 - edge, 148, **148**, 149, 150, 152, 155, 156, 161, 163, 164
 - file system, 61, 84
 - growth, 133, 150, 160, 161, 163, 174
 - look-up, 129, 130, 133, 134, 136, 137, 146, 150–152, 156, 163–166
 - maintenance, 165–167, 176, *see also* replacement strategy
 - overflow, 130, 133, 138, 167
 - semantic, 129
 - size, 130, 133, 160, 173
 - stage, 163, 163fn
 - support, 159–161, 163, 164
- caretting-in, 29, **29**, 30, 30fn, 58, 64
- classification, 18, 37, 84, 89, 129, 176, 177
- closure, 8, 9
 - reflexive-transitive, 19
 - transitive, 9, 19
- cluster, 38
 - index, 119
 - table, 126
- collapse, 104, **104**, 108
- communication, 21, 176
- comparison
 - lexicographical, 28
 - numeric, 28, 32, 121, 122
- complexity, 57, 65, 111, 121, 129, 130, 132, 137, 173
 - exponential, 35, 65, 75fn, 78, 132, 133, 173
- compression, 32–34, 37, 65, 76, 79, 171
- concurrency, 5, 6, 89
- conjunction, 71, 74, 79, 106, 107, 107fn, 115–117, 153, 155
- content/structure join, 81, 82, 85, 97, *see also* materialized join
- corpus, 10, 26, 40, 57, 84, 85, 119–122, 124, *see also* document collection
- cost estimation, 112, 125, 126, 147, 158–160, 163, 173
- cross-link, 77, 78, *see also* IDREF

- data mining, 91
- data type, 9fn, 62, 90fn
- DataGuide, 11, 32–34, 37, 38, 75, 75fn, **75**, 77–79, 81, 82, 84, 85, 175

- DBLP*, *see* *Digital Bibliography and Library Project*
- decision, 19, 19fn, **19**, 20, 23, 27, 30, 33, 34, 37, 38, 42, 43, 48, 50, 52, 52fn, 53, 56, 57, 59–64, 66, 91, 97, 99, 101, 102, 108, 109, 111, 113–115, 121, 123, 130, 131, 150, 156, 171, 183, 183fn, 184, 185
- depth-first, 7fn, 26
- descriptive schema, *see also* document schema
- Dewey, 27, 27fn, 28–30, 32, 37–40, 54, 56, 64, 66
- Dewey, Melvil, 27fn
- Digital Bibliography and Library Project (DBLP)*, 54, 57–62, 118, 122, 124, 181, 183–185
- Discrete Mathematics, 37, 176
- disjunction, 9, 10, 12, 74, 76, 83, 106, 107, 107fn, 115–117, 132, 140, 141, 153, 155
- distance, 19, 20, 26, 30, 32, 36, 39, 40, 50, 60, 62, 71, 101, 143, *see also* proximity
- distributed data source, 6, 175
- document
- collection, V, 4, 6, 18, 30fn, 37, 40, 42, 52, 54, 57, 58, 61, 66, 78, 82–84, 89, 93, 95, 118, 119, 122, 125, 126, 130fn, 132, 136, 137, 142, 144, 146, 159, 159fn, 166, 167, 176, 181, 183, *see also* corpus
 - height, 7, 23, 35, 36, 44, 46, 48, 51, 64, 65, 94
 - hierarchy, V, 3, 27fn, 89
 - level, *see* level
 - order, 4, 7–9, 11, 17, 20, 22, 26–28, 30, 32, 38, 41, 43, 46, 48, 49, 51, 54, 56, 62, 63, 65, 66, 107, 118, 126, 133
 - schema, V, 4–6, 12, 74, 83, 85, 90, 94, 124, 129, 132, 134, 136–141, 147, 164, 167, 171, 177, *see also* Document Type Definition
 - descriptive, **4**, 137, 141, 164
 - heterogeneous, 18, 57, 64, 67, 75, 75fn, 78, 84, 118, 124, *see also* irregular
 - homogeneous, 54, 64, 84, *see also* regular
 - irregular, V, 3, 34, *see also* heterogeneous
 - prescriptive, **4**, 89–91, 94, 137, 141
 - regular, 34, 36, 38, 41, 91, 171, *see also* homogeneous
 - structured, V, 5, 21, 26, 34, 71, 74, 81, 176, 177
 - text-centric, 3, 26
 - tree, 6, 7, 7fn, **7**, 9–13, 17–24, 26, 27, 29, 30, 30fn, 31–36, 38, 40–48, 50–52, 54–56, 59, 63–67, 71–74, 75fn, 76, 77, 81, 82, 90, 91, 94, 98, 99, 112, 119, 123, 125, 130, 131, 133, 133fn, 134, 143, 171, 172, 176, 178
- Document Object Model (DOM), 4, 26fn
- Document Type Definition (DTD), 4, 30, 38, 89, 94, 132, 133, 137, 144, 173, *see also* document schema
- document/term matrix, **73**, 74
- DOM, *see* Document Object Model
- DTD, *see* Document Type Definition
- duplicate, 11, 32, 73, 78, 79, 93, 116, 134, 146, 147, 149, 163, 174
- durability, 21, 22, *see also* robustness
- dynamic, 17, 30fn, 40, 52, 66, 126, 136, 176
- effectiveness, V, 5, 34, 85, 108, 111, 112, 130, 134, 138, 142, 144, 145, 160, 164, 173
- encoding, *see also* labelling scheme
 - binary, 27, 29, 32, 37, 56, 171
 - prefix-free, 27, **27fn**, 28, 33
- equijoin, *see* join
- evaluation plan, *see* query plan
- existential quantification, 125, 132, 166
- expressivity, 18, **18**, 20, 27, 30, 32, 34, 37–39, 41, 42, 43fn, 57, 65, 66, 126, 171
- false positive, 19, 41, 76, 77, 83, 95, 140
- fan-out, 29, 29fn, 35, 36, 40, 48, 65, 176, *see also* arity
- field, 98, 105, 107, 119, 159
 - mandatory, 98
 - optional, 98, 111, 124
- filter, 11, 85, 92, 94
- Finite-State Transducer (FST), 30, 33, 37, 38, 66, *see also* transducer
- FIRST*, 61–63, 183, 184
- flat text, V, 3–5, 71, 73, 172, 176
- foreign key, 81, 83, 92, 98, 119, 122
- FST, *see* Finite-State Transducer
- gap positions, 22, 40
- generalization, 4, 26, 52, 54, 130fn, 135
- global data, 4, 5, 17, 30, 33, 34, 37, 39, 66
- government, 9, 75, 83, 90, 98, 107, 115–117, 153–155
- grammar, 4
- heuristic, 79, 126, 173
- holistic, **79**
- homomorphism, 36, 171
- horizontal proximity, 38, 166fn
- hybrid retrieval system, *see* retrieval system
- hypertext, 27
- I/O, V, 13, 19, 40, 41, 52, 82, 83, 95, 97, 109, 126, 138, 139, 141, 142, 175
- IDREF, *see also* cross-link
- IDREF, 6, 7, 77
- IMDb*, *see* *Internet Movie Database*
- incomplete, 133, 137, 140, 142, 146, 173

- index, 4, 4fn, 6, 11, 18, 58, 69, 71, 72, 74–78, 84, 85, 89, 121, 124, 137, 139, 146, 165, 172, 175, 177
- path, 51, 72, 92, 124, 164, 172, 175, 176
 - atomic, 73
 - compositional, 75
 - weight, 51, 58, 67
- INEX*, *see Initiative for the Evaluation of XML Retrieval*
- information need, 176–178
- Information Retrieval (IR), 3–5, 71, 73, 76, 126, 172, 176
- InfoSet, 26fn
- Initiative for the Evaluation of XML Retrieval (INEX)*, 57, 58, 64, 75fn, 82, 118, 121, 124, 181
- inlining, 90, **90**
- insertion, 21–23, 27, 29, 30, 30fn, 37, 40, 54, 56, 63, 64, 66, 67, 136, 172
- integrated query evaluation, *see* query evaluation
- Integrated-Ranking CADG, 177, **177**
- Internet Movie Database (IMDb)*, 54, 56, 63, 64, 77, 118–120, 124, 159, 161, 181
- inverted file, 71, **71**, 72, 73, 79, 81, *see also* inverted list
 - path, 73–77, 81–83
 - tag, 72
- inverted list, **71**, 172, *see also* inverted file
- IR, *see* Information Retrieval
- IR-CADG, *see* Integrated-Ranking CADG
- join
 - equi, 90, 91
 - self, 90–92, 99, 101, 105, 121, 122, 125, 160
 - structural, 4, 4fn, 19, 21, 28, 36, 38, 40, 62, 66, 72, **72**, 75, 79, 94, 119, 126, 137, 138, 172, 173
- keyword signature, 76, **76**, 77, 79, 83, 98, 106, 107, 124, 126
- label
 - invariant, 43, **43**, 46
 - size
 - fixed, 33, 37, 40, 57, 77
 - maximum, 18, 29, 37, 54fn, 57, 58, 64, 172
 - variable, 28, 37, 40, 56, 57
- labelling scheme
 - multiplicative, 34, **34**, 35, 38, 40–42, 54, 64, 171
 - path, 27, **27**, 31, 32, 34, 37, 38, 40, 41, 52, 54, 56, 64, 171
 - subtree, 21, **21**, 24, 26, 27, 30, 38, 40, 57, 66, 119, 171
- Layered BIRD, 54–56, 64, 66
- layering, 38, 54fn, **54**, 56, 63
- level, *see also* Three-Level Model of XML Retrieval
 - document, **12**, 14, 41, 50–54, 56, 74–77, 95, 97–99, 102, 104, 107, 108, 112–116, 119, 122, 126, 142, 143, 145, 146, 148, 150, 153fn, 159, 164, 171–175, 177, 178
 - query, 12, **12**, 137, 140
 - schema, 13, **13**, 14, 41, 74, 76, 82, 83, 92–95, 98, 101, 102, 104, 105, 107, 108, 114, 115, 119, 121, 125, 126, 137, 140, 141, 143, 146–148, 150, 164, 172, 173, 177
- limitations, 6, 65
- local data, 4, 11, 18, 40, 78, 79
- locality, 135
- lossiness, 38, 66, 172
- markup, V, 3, 26, 71, 176
- match edge, 149, **149**, 150–153, 157
- matching
 - candidate, 13, **13**, 19, 94, 184
 - document-level, 99, 101, 103, 107, 108, 112, 113, 121, 122
 - partial, 99, 112, 121, 122, 140, 144, 166, 172
 - rule, *see* rule
 - schema-level, 99, 101, 103, 105, 119, 125
- materialization, 74, 79, 81, 90–92, 124, 129, 135, 136, 149, 171, 172
 - join, 81, 85, 124, 126
- mediation, 91, 137
- metric, 65
- modulo, **42fn**, 99
- multiplicative encoding, *see* labelling scheme
- namespace, 7fn, 8, 9, 9fn, 52fn
- native retrieval system, *see* retrieval system
- nesting, V, 4, 11, 22, 57, 64, 122, 123
 - interval, 21fn
 - region, 21
- NEVER*, 61–63, 183–185
- on-the-fly, 79, 94
- optimization goal, 57, 64, 65, 78, 160, 161, 171, 176
- order
 - combined pre-/post, **7fn**, 21, 21fn, 26
 - document, *see* document order
 - inverse postorder, 51, 52, **52**
 - sibling, 7, **7**, 8, 32fn, 98
 - partial, 32
- overflow
 - cache, *see* cache
 - graph, 91
 - label, 40, 54, 56, 63
 - weight, 63, 64, 66, 67
- overhead

- runtime, 59, 62, 63, 79, 82, 118, 120, 120fn, 121, 122, 125, 129, 130, 160, 163–165, 167, 184, 185
- space, V, 30, 57, 64, 79, 85, 171, 172
- padding, 28, 34
- paging, 10, 23, 76
- parallelization, 90, 159, 164, 174
- parameter, 5, 6, 20, 43, 46, 64, 78, 109fn, 132, 136, 153, 155, 176, 177
- partition, 54, 76, 78, 90, 133, 160, 163, 163fn, 164
- path encoding, *see* labelling scheme
- path occurrence, 77, 78, 120fn
- Path/Term/Node* Hierarchy (*PTN*), 177
- PCDATA, 30
- performance measure, 59, 61, 84, 85, 120, 121, 159, 160, 163
- persistent storage, 17, 102, 135, 160, 173
- plane
 - pre/max*, 25
 - pre/post*, 22, 23, 25
 - pre/size*, 25, 27
 - start/end*, 25, 27
- posting, 32, 71, **71**, 73, 74, 82
- pre-weight, 44, **44**, 45, 47, 51, *see also* weight
- precision, 97, 177
- predecessor, 23, 35, 36, 49, 51, 52, 101, 156
- prefix-free encoding, *see* encoding
- prescriptive schema, *see also* document schema
- priorities, 37, 112, 124, 167
- privacy, 6
- profiling, 61, 62, 185
- projection, 99, 102, 104, 107, 114, 115, 159
- proximity, 8, 9, 21, 26, 29, 30, 36, 40, 52, 52fn, 59, 60, 103, 104, 106, 107, 141, *see also* distance
 - bounds, 12, 99, 103, 104, 106–108, 114, 115, 143, 143fn, 166fn
- pruning, 79, 84
- PTN*, *see Path/Term/Node* Hierarchy
- query
 - answer, *see* query result
 - branching, 61, 62, 79, 92, 94, 95, 121, 122, 125, 133, 137, 172, 176, 183, 184
 - candidate, 138, 140, 142, 143, 146, 150
 - comparison
 - extensional, 134, 137–139
 - intensional, 134, 137–140, 142, 164
 - containment, V, 5, 6, 14, 129, 130, 130fn, 131, **131**, 132, 134–143, 145, 146, 173
 - match, 131, **131**, 137, 142
 - node, 131, **131**, 142
 - editing, 161, 163, 177
 - evaluation, *see also* query processing
 - from scratch, V, 5, 101, 109, 111, 112, 129, **129**, 130, 131, 134–137, 139–142, 144, 147, 148, 151, 158–161, 163–165, 173, 174
 - incremental, V, 41, 81, 112, 129, **129**, 130, 132, 135–140, 142, 143, 146, 147, 150, 157–161, 163, 166, 173
 - integrated, 137, 139, 147, 164
 - extension, 12, **12fn**, 13, 129–131, 133, 134, 137, 138, 140, 141, 146, 153, 164, 165, 173
 - graph, 10, 101, 103, 107, 111, 120–123, 125, 150
 - hotspot, **161**, 163
 - intension, 12, **12fn**, 14, 129, 130, 132–134, 137, 138, 140, 141, 144, 146, 150, 173
 - language, 4, 6, 9, 18, 19, 26, 61, 129, 130, 132–134, 165
 - level, *see* level
 - optimization, 4, 5, 23, 37, 89, 97, 102fn, 108, 112, 119, 123, 126, 158, 172, 173
 - overlap, V, 5, 6, 14, 129, 130, 130fn, 131, **131**, 132, 134–146, 160, 163, 164, 167, 173, 174
 - path, 75, 76, 78, 84, 92, 93, 132
 - performance, 27, 119–122, 126
 - plan, 102, 108, **108**, 109–113, 115–117, 121–123, 138, 140, 144–148, 151, 156–160, 163, 165
 - alternative, 111, 130, 138, 140, 147, 158
 - planning, 5, 89, 92, 97, 98, 103, 105, 108–111, 115, 118, 119, 123–126, 138, 147, 151, 156, 157, 163, 172, 173
 - preprocessing, 95, 102
 - processing, 13, 81, 129, 136, 138–140, 142, 159, 164, 173, *see also* query evaluation
 - remainder, 142, **142**, 144, 147, 151, 153–155, 157–159, 165
 - result, V, 5, 6, 10, **10**, 12fn, 14, 61, 71, 72, 75–77, 79, 82, 83, 90, 93, 94, 99, 102, 104, 105, 112, 116–119, 121, 122, 126, 129–133, 133fn, 134–147, 151, 156–161, 163–165, 167, 173, 174, 176, 177, 184, *see also* result table
 - false, 93, 122, 125
 - final, 99, 102, 116, 117, 122, 125, 137–140, 144, 145, 147, 151, 160, 164, 165, 184
 - intermediate, V, 79, 92, 95, 97, 99, 101, 102, 105, 108, 111–114, 117, 120–124, 126, 137, 138, 140, 144, 145, 151, 156, 157, 159, 163, 164, 172–174
 - partial, 136, 137, 164, 173
 - snapshot, 140, **140**, 147, 150, 151, 157–159,

- 165
- rewriting, 99, 102, 102fn, 103, 104, 107–109, 115, 119, 126, 172
- document-level, 107
 - schema-level, 102
- selectivity, 84, 97, 112, 120, 126, 138, 158, 163, 173
- semantics, 8, 9, 66, 103, 104, 130, 138
- translation, 91, 97, 99, 101, 105, 107, 112, 114–120, 125, 133, 134, 140, 147, 151, 163, 166
- tree, 61, 62, 94, 134
- twig, 4, 79
- queue, 109, 111, 167
- R-Tree, 22
- ranking model, 5, 176, 177
- RDBS, *see* relational database system
- recall, 177
- reconstruction, 19, 19fn, **19**, 20, 21, 27, 28, 30, 33–38, 41–43, 48–51, 56, 57, 59–66, 97, 99, 101, 102, 108, 109, 111–115, 119, 121, 122, 125, 157, 166, 171, 172, 183, 183fn, 184, 185
- recovery, 6, 89
- recursive
- definition, 44, 46
 - query, 90, 91
 - schema, 38, 75, 84, 93–95, 118, 122, 125
- redundancy, 32, 33, 37, 51, 73, 75, 75fn, 76–79, 92, 95, 102, 124, 125, 133, 149, 175
- regular expression, 4, 66, 79, 93, 93fn, 125
- path, 91, 132
- relabelling, 30, 30fn, 54, 63
- relational algebra, 5, 89, 99
- relational database system, V, 3, 5, 6, 41, 57, 61, 66, 84, 85, 89, 91, 92, 94, 95, 97–99, 108, 112, 114, 118, 119, 124–126, 139, 144, 146, 160, 165, 172, 173, 176, 183
- relational retrieval system, *see* retrieval system
- relevance feedback, 161
- relevance ranking, 5, 6, 92, 176
- removal, 40
- replacement strategy, 130, 133, 167, *see also* cache maintenance
- response time, 85, 135, 184
- result table, 99, 102, 105, 108, 112–116, 130, 146, 151, 159, 160, 165, *see also* query result
- retrieval phase, 99, 103, 107, 125, 163
- retrieval system
- hybrid, V, **4**, 5, 18, 85, 89, 92, 94, 134, 139, 171–173
 - native, V, 4, 4fn, **4**, 5, 18, 57, 63, 89, 91, 94, 118–120, 120fn, 121, 124, 126, 175
 - relational, V, **4**, 5, 18, 63, 90, 91, 95, 171, 175
- robustness, 18, **18**, 22, 27, 30, 37, 40–42, 64, 66, 78, 83, 167, 171, *see also* updatability
- rule
- adaptation, 105–107, 114–116
 - matching, **99**, 100, 103, 105–107, 114–117
- safety, 5, 89
- satisfiable, 84, 85, 103, 125, 175
- scalability, V, 3, 5, 41, 64, 89, 94, 125, 130, 139, 150, 160, 165, 172, 173, 175
- schema, *see* document schema
- aware, **129**
 - based, 89fn, **89**, 90, 91, 94
 - edge, 146–148, **148**, 149, 150, 160, 164, 166, 167
 - hit containment, 142, **144**, 146, 149–153, 156, 166
 - level, *see* level
 - level matching, *see* matching
 - level query rewriting, *see* query rewriting
 - oblivious, 90, 90fn, **90**, 91, 125, 129, 173
- schematization, 141, **141**, 142–146, 148, 150, 155, 157, 164, 166, 173
- scratch, evaluation from, *see* query evaluation
- search time, 160, 163
- selection, 93, 103, 104, 107, 114, 124, 172
- selfjoin, *see* join
- semistructured data, V, 3, 71, 75, 91, 129, 132, 133, 173
- separation level, **20**, 29, 39, 50
- serialization, 7, 7fn, 21, 22, 126
- set-at-a-time, 38
- SGML, *see* Standard Generalized Markup Language
- shredding, 89, 90, 98
- shrink-wrap, 23, **23**, 25, 91, 119
- skew, 29fn, 37
- snapshot, *see* query result
- space consumption, 27, 37, 57, 63, 64, 112, 124, 126
- sparseness, **22**, 30, 34, 35, 37, 38, 41, 52, 54, 63, 64, 74, 79, 171
- Staircase Join*, 23, 91, 94, 119
- Standard Generalized Markup Language (SGML), 3, 26, 32
- static, 40, 52, 136, 176
- statistics, 78, 84, 92, 94, 97, 98, 105, 111, 118, 123–125, 161, 166, 173, 178
- stemming, 71, 85
- stop word, **71**, 85
- storage scheme, 89–94, 97, 118, 119, 124, 126, 129, 165, 172, 176
- streamed data source, 6, 175
- string matching, 4, 92–94, 121, 122

- structural join, *see* join
- structural summary, 8, 10, 11, **11**, 13, 17, 33, 42, 51, 58, 64, 67, 85, 90, 133, 164, 171–173, 175, 177
 - centralized, 4–6, 11, **11**, 17, 18, 32, 34, 36, 37, 41–43, 48, 71, 78, 81, 89, 94, 95, 97, 124, 171, 172, 177
 - decentralized, 4, 11, **11**, 17, 41, 71, 78, 94, 97, 124, 171
- structured document, *see* document
- subtree encoding, *see* labelling scheme
- successor, 19, 151
- tag path, 11, **11**, 32, 33, 37, 42, 43, 54, 63, 66, 67, 72–75, 75fn, 76–79, 81–84, 90, 92–95, 98, 99, 101, 108, 114, 115, 119, 122, 124, 125, 172, 173, 177, 178
- tag-specific sibling codes, 27, 32, 32fn, 38
- template, 77, **77**, 114
- terminology, 41, 49, 89, 176
- Three-Level Model of XML Retrieval, 6, 12–14, 74–76, 137, 171, 175, 177, 178
- threshold, 136, 164, 167
- topology, 40, 65
- trade-off, 34, 37, 57, 58, 65
 - space, 65, 67, **171**
- transaction, 5, 89
- transducer, 33, *see also* Finite-State Transducer
- transitive, 104, 108, 166
- tree database, 17, 18
- tree neighbourhood, 21, 34, 42, 175
- tree-aware, 125, 173
- Treebank*, 75fn, 82
- Trie, 73, 74fn, 76, 77, 79
- tuning, 77
- Unicode, 28fn
- union, 8, 90, 146, 147, 163, 164, 174
- unique, 7, 7fn, 11, 19, 27, 33, 34, 34fn, 42fn, 43, 71, 78, 98, 115, 134, 143, 145, 157
 - node label, V, 4, 7, 11, 17, 28, 32, 41, 46, 71, 90, 92, 105, 107, 130, 165
- updatability, **37**, 38, 40, 42, 57, 63–65, 172, *see also* robustness
- user, 5, 6, 12, 14, 85, 116, 118, 124, 129, 159, 161, 165, 167, 174–177, 177fn, 178
 - expert, 176
 - interaction, 6, 14, 167, 176, 177
 - novice, 176
- user-defined function, 66, 99, 126
- UTF-8, 28, 28fn, 29, 37
- versioning, 6
- vertical proximity, 7, 30, 143, 143fn
- visualization, 65, 102, 175, 176
- web, V, 3, 118
 - search, 5, 85
- weight, 41–44, **44**, 45, 47, 49–51, 53–56, 58, 63, 64, 66, 67, 98, 99, 101, 115, *see also* pre-weight
 - invariant, 43, **43**, 44
- well-formed, 21fn, 22, 26
- wildcard, 73, 75, 78, 92–94, 122, 132, 133
- XLink, *see* XML Linking Language
- XMark, *see* XML Benchmark
- XML Benchmark (XMark), 104, 119, 120, 122, 123
- XML Linking Language (XLink), 6, 78
- XML Pointer Language (XPointer), 6, 78
- XPointer, *see* XML Pointer Language

BIBLIOGRAPHY

- ABITEBOUL, S., KAPLAN, H., AND MILO, T. 2001a. Compact Labeling Schemes for Ancestor Queries. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 547–556.
- ABITEBOUL, S., SEGOUFIN, L., AND VIANU, V. 2001b. Representing and Querying XML with Incomplete Information. In *Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*.
- AL-KHALIFA, S., JAGADISH, H. V., KOUDAS, N., PATEL, J. M., SRIVASTAVA, D., AND WU, Y. 2002. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE)*. 141–152.
- ALSTRUP, S., GAVOILLE, C., KAPLAN, H., AND RAUHE, T. 2002. Nearest Common Ancestors: A Survey and a New Distributed Algorithm. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 258–264.
- AMER-YAHIA, S., CHO, S., LAKSHMANAN, L. V. S., AND SRIVASTAVA, D. 2001. Minimization of Tree Pattern Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 497–508.
- BALMIN, A. AND PAPAKONSTANTINOY, Y. 2005. Storing and Querying XML Data using Denormalized Relational Databases. *The VLDB Journal* 14, 1, 30–49.
- BARBOSA, D., BARTA, A., MENDELZON, A. O., MIHAILA, G. A., RIZZOLO, F., AND RODRIGUEZ-GUIANOLLI, P. 2001. ToX – the Toronto XML Engine. In *Proceedings of the International Workshop on Information Integration on the Web (IIW)*. 66–73.
- BERTINO, E. AND KIM, W. 1989. Indexing Techniques for Queries on Nested Objects. 1, 2.
- BOHANNON, P., FREIRE, J., ROY, P., AND SIMÉON, J. 2002. From XML Schema to Relations: A Cost-based Approach to XML Storage. In *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE)*. 64–75.
- BÖHM, C., BERCHTOLD, S., KRIEGEL, H.-P., AND MICHEL, U. 2000. Multidimensional Index Structures in Relational Databases. *Journal of Intelligent Information Systems (JIIS)* 15, 1, 51–70.
- BONCZ, P., GRUST, T., VAN KEULEN, M., MANEGOLD, S., RITTINGER, J., AND TEUBNER, J. 2005a. Pathfinder: XQuery—The Relational Way. In *Proceedings of the 31st Conference on Very Large Data Bases (VLDB)*. 1322–1325.
- BONCZ, P., MANEGOLD, S., AND RITTINGER, J. 2005b. Updating the Pre/Post Plane in MonetDB/XQuery. In *Proceedings of the 2nd International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*.
- BOSAK, J. 1998. Religion 2.00: Four Religious Works, annotated in XML. Available on-line at <http://metalab.unc.edu/bosak/xml/eg/rel200.zip>.
- BOSAK, J. 1999. Shakespeare 2.00: The Plays of Shakespeare, annotated in XML. Available on-line at <http://metalab.unc.edu/bosak/xml/eg/shaks200.zip>.

- BREMER, J.-M. AND GERTZ, M. 2006. Integrating XML Document and Data Retrieval Based on XML. *The VLDB Journal* 15, 1, 53–83.
- BRUNO, N., KOUDAS, N., AND SRIVASTAVA, D. 2002. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 310–311.
- CALVANESE, D., GIACOMO, G. D., LENZERINI, M., AND VARDI, M. Y. 2000. Query Processing using Views for Regular Path Queries with Inverse. In *Proceedings of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 58–66.
- CALVANESE, D., GIACOMO, G. D., LENZERINI, M., AND VARDI, M. Y. 2002. View-based Query Answering and Query Containment over Semistructured Data. In *Proceedings of the 8th International Workshop on Database Programming Languages (DBPL)*. 40–61.
- CALVANESE, D., GIACOMO, G. D., LENZERINI, M., AND VARDI, M. Y. 2003. Reasoning on Regular Path Queries. *SIGMOD Record* 32, 4, 83–92.
- CHEBOTKO, A., LIU, D., ATAY, M., LU, S., AND FOTOUHI, F. 2005. Reconstructing XML Subtrees from Relational Storage of XML Documents. In *Proceedings of the 2nd International Workshop on XML Schema and Data Management (XSDM)*. 87–96.
- CHEN, L. 2003. A Semantic Caching System for XML Queries. Ph.D. thesis, Worcester Polytechnic Institute.
- CHEN, L. AND RUNDENSTEINER, E. 2002. ACE-XQ: A Cache-aware XQuery Answering System. In *Proceedings of the 5th International Workshop on the Web and Databases (WebDB)*.
- CHEN, L. AND RUNDENSTEINER, E. A. 2005. XQuery Containment in Presence of Variable Binding Dependencies. In *Proceedings of the 14th International Conference on the World Wide Web (WWW)*. 288–297.
- CHEN, L., RUNDENSTEINER, E. A., AND WANG, S. 2002. XCache: a Semantic Caching System for XML Queries. In *Proceedings of the 21st ACM SIGMOD International Conference on Management of Data*. 618–618. Demo.
- CHEN, L., WANG, S., AND RUNDENSTEINER, E. A. 2004. Evaluation of Replacement Strategies for XML Query Cache. *Data and Knowledge Engineering Journal* 49, 2, 145–175.
- CHEN, T., LU, J., AND LING, T. W. 2005b. On Boosting Holism in XML Twig Pattern Matching using Structural Indexing Techniques. In *Proceedings of the 24th ACM SIGMOD International Conference on Management of Data*. 455–466.
- CHEN, Y. AND ABERER, K. 1998. Layered Index Structures in Document Database Systems. In *Proceedings of the Seventh International ACM Conference on Information and Knowledge Management (CIKM)*. 406–413.
- CHEN, Y. AND ABERER, K. 1999. Combining Pat-Trees and Signature Files for Query Evaluation in Document Databases. In *Proceedings of the 10th International Conference on Database and Expert Systems Applications (DEXA)*. 473–484.
- CHEN, Y., DAVIDSON, S., HARA, C., AND ZHENG, Y. 2003. RRXS: Redundancy Reducing XML Storage in Relations. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*.
- CHEN, Y., DAVIDSON, S. B., AND ZHENG, Y. 2004. BLAS : An Efficient XPath Processing System. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. 47–58.
- CHEN, Z., GEHRKE, J., KORN, F., KOUDAS, N., SHANMUGASUNDARAM, J., AND SRIVASTAVA, D. 2005a. Index Structures for Matching XML Twigs using Relational Query Processors. In *Proceedings of the 2nd International Workshop on XML Schema and Data Management (XSDM)*. 1273.
- CHIEN, S.-Y., TSOTRAS, V. J., ZANIOLO, C., AND ZHANG, D. 2001. Storing and Querying Multiversion XML Documents using Durable Node Numbers. In *Proceedings of the 2nd International Conference on Web Information Systems Engineering (WISE)*. 232–244.

- CHIEN, S.-Y., TSOTRAS, V. J., ZANIOLO, C., AND ZHANG, D. 2002. Efficient Complex Query Support for Multiversion XML Documents. In *Proceedings of the 8th International Conference on Extending Database Technology (EDBT)*. 161–178.
- CHIEN, S.-Y., TSOTRAS, V. J., ZANIOLO, C., AND ZHANG, D. 2006. Supporting Complex Queries on Multiversion XML Documents. *ACM Transactions on Internet Technology (TOIT)* 6, 1, 53–84.
- CHIEN, S.-Y., VAGENA, Z., ZHANG, D., AND TSOTRAS, V. J. 2002. Efficient Structural Joins on Indexed XML Documents. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*. 263–274.
- CHRISTOPHIDES, V., SCHOLL, M., AND TOURTOUNIS, S. 2003. On Labeling Schemes for the Semantic Web. In *Proceedings of the 14th International Conference on the World Wide Web (WWW)*. 544–555.
- CHUNG, C.-W., MIN, J.-K., AND SHIM, K. 2002. APEX: An Adaptive Path Index for XML Data. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 121–132.
- CLARKE, C. L. A., CORMACK, G. V., AND BURKOWSKI, F. J. 1995. An Algebra for Structured Text Search and a Framework for its Implementation. *The Computer Journal* 38, 1, 43–56.
- COHEN, E., KAPLAN, H., AND MILO, T. 2002. Labeling Dynamic XML Trees. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 271–281.
- CONSENS, M. P. AND MILO, T. 1994. Optimizing Queries on Files. In *Proceedings of the 13th ACM SIGMOD International Conference on Management of Data*. 301–312.
- CONSENS, M. P. AND MILO, T. 1995. Algebras for Querying Text Regions. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 11–22.
- CONSENS, M. P. AND MILO, T. 1998. Algebras for Querying Text Regions: Expressive Power and Optimization. *Journal of Computer and System Sciences* 57, 3, 272–288.
- COOPER, B., SAMPLE, N., FRANKLIN, M. J., HJALTASON, G. R., AND SHADMON, M. 2001. A Fast Index for Semistructured Data. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*. 341–350.
- DBLP. The Digital Bibliography and Library Project (DBLP). Available on-line at <http://dblp.uni-trier.de>.
- DDC. The Dewey Decimal Classification. Available on-line at <http://www.oclc.org/dewey>.
- DEHAAN, D., TOMAN, D., CONSENS, M. P., AND ÖZSU, M. T. 2003. A Comprehensive XQuery to SQL Translation using Dynamic Interval Coding. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 623–634.
- DEUTSCH, A., FERNÁNDEZ, M., AND SUCIU, D. 1999. Storing Semistructured Data with STORED. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 431–442.
- DEUTSCH, A. AND TANNEN, V. 2001. Containment and Integrity Constraints for XPath. In *Proceedings of the 8th International Workshop on Knowledge Representation meets Databases (KRDB)*.
- DIETZ, P. F. 1982. Maintaining Order in a Linked List. In *Proceedings of the 14th ACM Symposium on Theory of Computing (STOC)*. 122–127.
- DOM. Document Object Model (DOM) Level 1 Specification. W3C Recommendation. 1998. Available on-line at <http://www.w3.org/TR/REC-DOM-Level-1>.
- FIEBIG, T., HELMER, S., KANNE, C.-C., MOERKOTTE, G., NEUMANN, J., SCHIELE, R., AND WESTMANN, T. 2002. Anatomy of a Native XML Base Management System. *VLDB Journal* 11, 4, 292–314.
- FLESCA, S. AND FURFARO, F. 2003. On the Minimization of XPath Queries. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*.
- FLORESCU, D. AND KOSSMANN, D. 1999. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin* 22, 3, 27–34.

- FREDKIN, E. 1960. Trie Memory. *Communications of the ACM* 3, 9 (September), 490–499.
- GAVOILLE, C. AND PELEG, D. 2003. Compact and Localized Distributed Data Structures. *Journal of Distributed Computing* 16, 2-3, 111–120.
- GOLDMAN, R. AND WIDOM, J. 1997. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*. 436–445.
- GOTTLOB, G., KOCH, C., AND SCHULZ, K. U. 2006. Conjunctive Queries over Trees. *Journal of the ACM* 53, 2, 238–272.
- GRUST, T. 2002. Accelerating XPath Location Steps. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 109–120.
- GRUST, T., SAKR, S., AND TEUBNER, J. 2004. XQuery on SQL Hosts. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*. 252–263.
- GRUST, T. AND VAN KEULEN, M. 2003. Tree Awareness for Relational DBMS Kernels: Staircase Join. In *Intelligent Search on XML Data*. Lecture Notes in Computer Science, vol. 2818. Springer, 231–245.
- GRUST, T., VAN KEULEN, M., AND TEUBNER, J. 2003. Staircase Join: Teach a Relational DBMS to Watch its Axis Steps. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*. 524–535.
- GRUST, T., VAN KEULEN, M., AND TEUBNER, J. 2004. Accelerating XPath Evaluation in Any RDBMS. *ACM Transactions on Database Systems* 29, 1, 91–131.
- GUTTMAN, A. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 3rd ACM SIGMOD International Conference on Management of Data*. 47–57.
- HALEVY, A. Y. 2001. Answering Queries using Views: A Survey. *VLDB Journal* 10, 4, 270–294.
- HALVERSON, A., BURGER, J., GALANIS, L., KINI, A., KRISHNAMURTHY, R., RAO, A. N., TIAN, F., VIGLAS, S. D., WANG, Y., NAUGHTON, J. F., AND DEWITT, D. J. 2003. Mixed Mode XML Query Processing. In *Proceedings of the 29th International Conference on Very Large Data Bases*. 225–236.
- HARDING, P. J., LI, Q., AND MOON, B. 2000. X-Ray—Towards Integrating XML and Relational Database Systems. In *Proceedings of the 19th International Conference on Conceptual Modeling (ER)*. 339–353.
- HARDING, P. J., LI, Q., AND MOON, B. 2003. XISS/R: XML Indexing and Storage System using RDBMS. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*. Demo.
- HRISTIDIS, V. AND PETROPOULOS, M. 2002. Semantic Caching of XML Databases. In *Proceedings of the 5th International Workshop on the Web and Databases (WebDB)*. 25–30.
- HUFFMAN, D. A. 1952. A Method for the Construction of Minimum Redundancy Codes. In *Proceedings of the Institute of Radio Engineers*. Vol. 40. 1098–1101.
- IMDB. The Internet Movie Database (IMDb). Available on-line at www.imdb.org.
- IMIELINSKI, T. AND LIPSKI, W. 1984. Incomplete Information in Relational Databases. *Journal of the ACM* 31, 4, 761–791.
- INEX. Initiative for the Evaluation of XML Retrieval (INEX). Available on-line at <http://inex.is.informatik.uni-duisburg.de>.
- INFOSET. XML Information Set. W3C Recommendation. 2001. Available on-line at <http://www.w3.org/TR/xml-infoset>.
- JAGADISH, H. V., AL-KHALIFA, S., CHAPMAN, A., LAKSHMANAN, L. V., NIERMAN, A., PAPARIZOS, S., PATEL, J., SRIVASTAVA, D., WIWATWATTANA, N., WU, Y., AND YU, C. 2002. TIMBER: A Native XML Database. *VLDB Journal* 11, 4, 274–291.

- JAGADISH, H. V., LAKSHMANAN, L. V. S., SRIVASTAVA, D., AND THOMPSON, K. 2004. TAX: A Tree Algebra for XML. In *Proceedings of the 8th International Workshop on Database Programming Languages (DBPL)*. 149–164.
- JIANG, H., LU, H., WANG, W., AND OOI, B. C. 2003. XR-Tree: Indexing XML Data for Efficient Structural Join. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*. 253–263.
- JIANG, H., LU, H., WANG, W., AND YU, J. X. 2002. Path Materialization Revisited: An Efficient Storage Model for XML Data. In *Australasian Computer Science Communications*. Vol. 24. 85–94.
- JIANG, H., WANG, W., LU, H., AND YU, J. X. 2003. Holistic Twig Joins on Indexed XML Documents. In *Proceedings of the 29th Conference on Very Large Data Bases (VLDB)*. 273–284.
- KANG, H., HAN, S., AND KIM, Y. 2005. Schemes of Storing XML Query Cache. In *Proceedings of the 16th Australasian Database Conference (ADC)*. 55–64.
- KANNAN, S., NAOR, M., AND RUDICH, S. 1992. Implicit Representation of Graphs. *SIAM Journal on Discrete Mathematics* 5, 4, 596–603.
- KANNE, C.-C. AND MOERKOTTE, G. 2000. Efficient Storage of XML Data. In *Proceedings of the 16th IEEE International Conference on Data Engineering (ICDE)*. 198.
- KAPLAN, H. AND MILO, T. 2000. Short and Simple Labels for Small Distances and Other Functions. In *Proceedings of the 7th International Workshop on Algorithms and Data Structures*. 246–257.
- KAPLAN, H., MILO, T., AND SHABO, R. 2002. A Comparison of Labeling Schemes for Ancestor Queries. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 271–281.
- KAUSHIK, R., BOHANNON, P., NAUGHTON, J. F., AND KORTH, H. F. 2002a. Covering Indexes for Branching Path Queries. In *Proceedings of ACM SIGMOD International Conference on Management of Data*. 133–144.
- KAUSHIK, R., SHENOY, P., BOHANNON, P., AND GUDES, E. 2002b. Exploiting Local Similarity for Efficient Indexing of Paths in Graph Structured Data. In *Proceedings of the 18th International Conference on Database Engineering (ICDE)*. 129–140.
- KEMPER, A. AND MOERKOTTE, G. 1992. Access Support Relations: An Indexing Method for Object Bases. *Information Systems* 17, 2, 117–145.
- KHA, D. D., YOSHIKAWA, M., AND UEMURA, S. 2001. An XML Indexing Structure with Relative Region Coordinate. In *Proceedings of the 17th IEEE International Conference on Data Engineering (ICDE)*. 313–320.
- KIMBER, W. E. 1993. HyTime and SGML: Understanding the HyTime HYQ Query Language. Tech. Rep. Version 1.1, IBM Corporation. August.
- KORMAN, A., PELEG, D., AND RODEH, Y. 2004. Labeling Schemes for Dynamic Tree Networks. *Theory of Computing Systems* 37, 1, 49–75.
- KRISHNAMURTHY, R., KAUSHIK, R., AND NAUGHTON, J. F. 2003. XML-to-SQL Query Translation Literature: The State of the Art and Open Problems. In *Proceedings of the 1st International XML Database Symposium (XSym)*. 1–18.
- LAM, F., SHUI, W. M., FISHER, D. K., AND WONG, R. K. 2003. Skipping Strategies for Efficient Structural Joins. Tech. Rep. UNSW-CSE-TR-0320, University of New South Wales.
- LDAP. Lightweight Directory Access Protocol (LDAP). IETF RFC 4511. 2006. Available on-line at <http://www.ietf.org/rfc/rfc4511.txt>.
- LEE, Y. K., YOO, S.-J., YOON, K., AND BERRA, P. B. 1996. Index Structures for Structured Documents. In *Proceedings of the 1st ACM International Conference on Digital Libraries*. 91–99.
- LI, H., LEE, M. L., AND HSU, W. 2005. A Path-Based Labeling Scheme for Efficient Structural Join. In *Proceedings of the 3rd International XML Database Symposium (XSym)*. 34–48.

- LI, Q. AND MOON, B. 2001. Indexing and Querying XML Data for Regular Path Expressions. In *Proceedings of the 27th Conference on Very Large Data Bases (VLDB)*. 361–370.
- LU, J., LING, T. W., CHAN, C.-Y., AND CHEN, T. 2005. From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*.
- MANDHANI, B. AND SUCIU, D. 2005. Query Caching and View Selection for XML Databases. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*. 469–480.
- MARRÓN, P. J. AND LAUSEN, G. 2002. Efficient Cache Answerability for XPath Queries. In *Proceedings of the International Workshop on Data Integration over the Web (DIWeb)*. 35–45.
- MAY, N., HELMER, S., KANNE, C.-C., AND MOERKOTTE, G. 2004. XQuery Processing in Natix with an Emphasis on Join Ordering. *Proceedings of the 1st International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, 49–54.
- MCHUGH, J., ABITEBOUL, S., GOLDMAN, R., QUASS, D., AND WIDOM, J. 1997. Lore: A Database Management System for Semistructured Data. *SIGMOD Record* 26, 3, 54–66.
- MCHUGH, J. AND WIDOM, J. 1997. Query Optimization for Semistructured Data. Tech. rep., Stanford University, Computer Science Department, Database Group. November.
- MCHUGH, J., WIDOM, J., ABITEBOUL, S., LUO, Q., AND RAJAMARAN, A. 1998. Indexing semistructured data. Tech. rep., Stanford University, Computer Science Department, Database Group. January.
- MEUSS, H. 2000. Logical Tree Matching with Complete Answer Aggregates for Retrieving Structured Documents. Ph.D. thesis, Dept. of Computer Science, University of Munich.
- MEUSS, H., SCHULZ, K., AND BRY, F. 2003. Visual Querying and Exploration of Large Answers in XML Databases with X²: A Demonstration. In *Proceedings of the 19th International Conference on Database Engineering (ICDE)*. 777–779.
- MEUSS, H., SCHULZ, K. U., WEIGEL, F., LEONARDI, S., AND BRY, F. 2005. Visual Exploration and Retrieval of XML Document Collections with the Generic System X². *Journal of Digital Libraries* 5, 1 (March), 3–17.
- MIKLAU, G. AND SUCIU, D. 2002. Containment and Equivalence for an XPath Fragment. In *Proceedings of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. 65–76.
- MILO, T. AND SUCIU, D. 1999. Index Structures for Path Expressions. In *Proceedings of the 7th International Conference on Database Theory (ICDT)*. 277–295.
- NAUGHTON, J. F., DEWITT, D. J., MAIER, D., ABOULNAGA, A., CHEN, J., GALANIS, L., KANG, J., KRISHNAMURTHY, R., LUO, Q., PRAKASH, N., RAMAMURTHY, R., SHANMUGASUNDARAM, J., TIAN, F., TUFTE, K., VIGLAS, S., WANG, Y., ZHANG, C., JACKSON, B., GUPTA, A., AND CHEN, R. 2001. The Niagara Internet Query System. *IEEE Data Engineering Bulletin* 24, 2, 27–33.
- NESTOROV, S., ULLMAN, J. D., WIENER, J. L., AND CHAWATHE, S. S. 1997. Representative Objects: Concise Representations of Semistructured, Hierarchical Data. In *Proceedings of the 13th IEEE International Conference on Data Engineering (ICDE)*. 79–90.
- NEVEN, F. AND SCHWENTICK, T. 2003. XPath Containment in the Presence of Disjunction, DTDs, and Variables. In *Proceedings of the 9th International Conference on Database Theory (ICDT)*. 315–329.
- OESTERLE, J. AND MAIER-MEYER, P. 1998. The GNoP (German Noun Phrase) Treebank. In *Proceedings of the 1st International Conference on Language Resources and Evaluation*. 699–703.
- OLTEANU, D., MEUSS, H., FURCHE, T., AND BRY, F. 2002. XPath: Looking Forward. In *Proceedings of the International Workshop on XML-Based Data Management (XMLDM)*. 109–127.
- O’NEIL, P., O’NEIL, E., PAL, S., CSERI, I., SCHALLER, G., AND WESTBURY, N. 2004. ORD-PATHS: Insert-Friendly XML Node Labels. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. 903–908.

- PAL, S., CSERI, I., SCHALLER, G., SEELIGER, O., GIAKOUMAKIS, L., AND ZOLOTOV, V. V. 2004. Indexing XML Data Stored in a Relational Database. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*. 1134–1145.
- PANKOWSKI, T. 2004. Processing XPath Expressions in Relational Databases. In *Proceedings of the 30th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*. 265–276.
- PAPAKONSTANTINOY, Y., GARCIA-MOLINA, H., AND WIDOM, J. 1995. Object Exchange across Heterogenous Information Sources. In *Proceedings of the 11th IEEE International Conference on Data Engineering (ICDE)*. 251–260.
- PAPARIZOS, S., AL-KHALIFA, S., CHAPMAN, A., JAGADISH, H., LAKSHMANAN, L. V., NIERMAN, A., PATEL, J. M., SRIVASTAVA, D., WIWATWATTANA, N., WU, Y., AND YU, C. 2003. TIMBER: A Native System for Querying XML. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Demo.
- PELEG, D. 1999. Proximity-Preserving Labeling Schemes and their Applications. In *Proceedings of the 25th International Workshop on Graph-Theoretic Concepts in Computer Science*. 30–41.
- PELEG, D. 2000. Informative Labeling Schemes for Graphs. In *Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science*. 579–588.
- PILAR, M. C. 2002. Optimizing XML Path Queries over Relational Databases. M.S. thesis, University of Toronto.
- QUAN, L., CHEN, L., AND RUNDENSTEINER, E. A. 2000. Argos: Efficient Refresh in an XQL-Based Web Caching System. In *Proceedings of the 1st International Workshop on the Web and Databases (WebDB)*. 78–91.
- QUN, C., LIM, A., AND ONG, K. W. 2003. D(k)-Index: An Adaptive Structural Summary for Graph-Structured Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 134–144.
- RAMANAN, P. 2002. Efficient Algorithms for Minimizing Tree Pattern Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.
- RAO, P. AND MOON, B. 2004. PRIX: Indexing And Querying XML using Prufer Sequences. In *Proceedings of the 20th IEEE International Conference on Data Engineering (ICDE)*. 288–300.
- RUNAPONGSA, K. AND PATEL, J. 2002. Storing and Querying XML Data in Object-Relational DBMSs. In *Proceedings of the International Workshop on XML-Based Data Management (XMLDM)*. 266–285.
- SACKS-DAVIS, R., DAO, T., THOM, J. A., AND ZOBEL, J. 1997. Indexing Documents for Queries on Structure, Content, and Attributes. In *Proceedings of the International Conference on Digital Media Information Bases*. Nara, Japan, 236–245.
- SALMINEN, A. AND TOMPA, F. W. 1992. Pat Expressions: An Algebra for Text Search. *Papers in Computational Lexicography (COMPLEX)*, 309–332.
- SALTON, G. AND MCGILL, M. J. 1983. *Introduction to Modern Information Retrieval*. McGraw-Hill.
- SANTORO, N. AND KHATIB, R. 1985. Labelling and Implicit Routing in Networks. *The Computer Journal* 28, 1, 5–8.
- SCHENKEL, R. 2004. FliX: A Flexible Framework for Indexing Complex XML Document Collections. In *Proceedings of the 1st International Workshop on Database Technologies for Handling XML Information on the Web (DataX)*. 240–249.
- SCHENKEL, R., THEOBALD, A., AND WEIKUM, G. 2004. HOPI: An Efficient Connection Index for Complex XML Document Collections. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT)*. 237–255.
- SCHENKEL, R., THEOBALD, A., AND WEIKUM, G. 2005. Efficient Creation and Incremental Maintenance of the HOPI Index for Complex XML Document Collections. In *Proceedings of the 21st IEEE International Conference on Data Engineering (ICDE)*. 360–371.

- SCHMIDT, A., KERSTEN, M., WINDHOUSER, M., AND WAAS, F. 2000. Efficient Relational Storage and Retrieval of XML Documents. *Proceedings of the 3rd International Workshop on the Web and Databases (WebDB)*, 47–52.
- SGML. Standard Generalized Markup Language (SGML). ISO/IEC 8879:1986.
- SHAH, A. AND CHIRKOVA, R. 2003. Improving Query Performance using Materialized XML Views: A Learning-based Approach. In *Proceedings of the 1st International Workshop on XML Schema and Data Management (XSDM)*. 297–310.
- SHANMUGASUNDARAM, J., TUFTE, K., ZHANG, C., HE, G., DEWITT, D. J., AND NAUGHTON, J. F. 1999. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB)*. 302–314.
- SHIN, D., JANG, H., AND JIN, H. 1998. BUS: An Effective Indexing and Retrieval Scheme in Structured Documents. In *Proceedings of the 3rd ACM International Conference on Digital Libraries*. 235–243.
- SQL2. The Structured Query Language (SQL-92). ISO/IEC 9075:1992.
- SQL3. The Structured Query Language (SQL:1999). ISO/IEC 9075:1999.
- TATARINOV, I., VIGLAS, S., BEYER, K. S., SHANMUGASUNDARAM, J., SHEKITA, E. J., AND ZHANG, C. 2002. Storing and Querying Ordered XML using a Relational Database System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 204–215.
- TREEBANK. The Penn Treebank Project. Available on-line at <http://www.cis.upenn.edu/~treebank/home.html>. University of Pennsylvania.
- TSAKALIDIS, A. K. 1984. Maintaining Order in a Generalized Linked List. *Acta Informatica* 21, 1, 101–112.
- WANG, H., PARK, S., FAN, W., AND YU, P. S. 2003b. ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 110–121.
- WANG, K. AND LIU, H. 1998. Discovering Typical Structures of Documents: A Road Map Approach. In *Proceedings of the 21st International ACM SIGIR Conference on Research and Development in Information Retrieval*. 146–154.
- WANG, W., JIANG, H., LU, H., AND YU, J. X. 2003a. PBiTree Coding and Efficient Processing of Containment Joins. In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE)*. 391–404.
- WEIGEL, F. 2002. A Survey of Indexing Techniques for Semistructured Documents. Tech. Rep. CIS 02-131, Centre for Information and Language Processing (CIS), University of Munich (LMU), Available on-line at <http://www.cis.uni-muenchen.de/~weigel/>.
- WEIGEL, F. 2003. Content-Aware DataGuides for Indexing Semi-Structured Data. M.S. thesis, University of Munich, Institute of Computer Science, Available on-line at <http://www.cis.uni-muenchen.de/~weigel/>.
- WEIGEL, F. 2006. Enhancing User Interaction and Efficiency with Structural Summaries for Fast and Intuitive Access to XML Databases. In *Proceedings of the 3rd Ph. D. Workshop at the 10th International Conference on Extending Database Technology (EDBT), Revised Selected Papers*. 54–65.
- WEIGEL, F., MEUSS, H., BRY, F., AND SCHULZ, K. U. 2004a. Content-Aware DataGuides: Interleaving IR and DB Indexing Techniques for Efficient Retrieval of Textual XML Data. In *Proceedings of the 26th European Conference on Information Retrieval (ECIR)*. 378–393.
- WEIGEL, F., MEUSS, H., SCHULZ, K. U., AND BRY, F. 2004b. Content and Structure in Indexing and Ranking XML. In *Proceedings of the 7th International Workshop on the Web and Databases (WebDB)*. 67–72.
- WEIGEL, F., MEUSS, H., SCHULZ, K. U., AND BRY, F. 2005a. Ranked Retrieval of Structured Documents with the S-Term Vector Space Model. In *Proceedings of the 3rd Workshop of the Initiative for the Evaluation of XML (INEX)*. 238–252.

- WEIGEL, F. AND SCHULZ, K. U. 2007. Labelling Schemes for XML and Tree Databases: A Survey. Work in progress.
- WEIGEL, F., SCHULZ, K. U., AND MEUSS, H. 2005b. Exploiting Native XML Indexing Techniques for XML Retrieval in Relational Database Systems. In *Proceedings of the 7th ACM International Workshop on Web Information and Data Management (WIDM)*. 23–30.
- WEIGEL, F., SCHULZ, K. U., AND MEUSS, H. 2005c. Node Identification Schemes for Efficient XML Retrieval. In *Foundations of Semistructured Data*, F. Neven, T. Schwentick, and D. Suciu, Eds. Number 05061 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.
- WEIGEL, F., SCHULZ, K. U., AND MEUSS, H. 2005d. The BIRD Numbering Scheme for XML and Tree Databases – Deciding and Reconstructing Tree Relations using Efficient Arithmetic Operations. In *Proceedings of the 3rd International XML Database Symposium (XSym)*. 49–67.
- WOLFF, J. E., FLÖRKE, H., AND CREMERS, A. B. 2000. Searching and Browsing Collections of Structural Information. In *Proceedings of the IEEE Forum on Research and Technology Advances in Digital Libraries*. 141–150.
- WOOD, P. T. 2001. Minimising Simple XPath Expressions. In *Proceedings of the 4th International Workshop on the Web and Databases (WebDB)*. 13–18.
- WOOD, P. T. 2003. Containment for XPath Fragments under DTD Constraints. In *Proceedings of the 9th International Conference on Database Theory (ICDT)*. 300–314.
- XLINK. XML Linking Language (XLink) 1.0. W3C Recommendation. 2001. Available on-line at <http://www.w3.org/TR/xlink/>.
- XMARK. XML Benchmark Project. On-line resource. Benchmark suite for XML repositories, provided by CWI Amsterdam, INRIA, Microsoft Inc., and BEA Systems.
- XML. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation. 2000. Available on-line at <http://www.w3.org/TR/REC-xml>.
- XPATH. XML Path Language (XPath) 2.0. W3C Working Draft. 2004. Available on-line at <http://www.w3.org/TR/xpath20>.
- XPOINTER. XML Pointer Language (XPointer). W3C Recommendation. 2003. Available on-line at <http://www.w3.org/TR/xptr-framework/>.
- XQUERY. XQuery 1.0: An XML Query Language. W3C Candidate Recommendation. 2006. Available on-line at <http://www.w3.org/TR/xquery>.
- XSD1. XML Schema Part 1: Structures Second Edition. W3C Recommendation. 2004. Available on-line at <http://www.w3.org/TR/xmlschema-1/>.
- XSD2. XML Schema Part 2: Datatypes Second Edition. W3C Recommendation. 2004. Available on-line at <http://www.w3.org/TR/xmlschema-2/>.
- YOON, J. P., RAGHAVAN, V., CHAKILAM, V., AND KERSCHBERG, L. 2001. BitCube: A Three-Dimensional Bitmap Indexing for XML Documents. *Journal of Intelligent Information Systems* 17, 2-3, 241–254.
- YOSHIKAWA, M., AMAGASA, T., SHIMURA, T., AND UEMURA, S. 2001. XRel: A Path-based Approach to Storage and Retrieval of XML Documents using Relational Databases. *ACM Transactions on Internet Technology (TOIT)* 1, 1, 110–141.
- YU, J. X., LUO, D., MENG, X., AND LU, H. 2005. Dynamically Updating XML Data: Numbering Scheme Revisited. *World Wide Web* 8, 1, 5–26.
- ZHANG, C., NAUGHTON, J. F., DEWITT, D. J., LUO, Q., AND LOHMAN, G. M. 2001. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the 20th ACM SIGMOD International Conference on Management of Data*. 425–436.

- ZHANG, X., PIELECH, B., AND RUNDENSTEINER, E. A. 2002. Honey, I Shrunk the XQuery—An XML Algebra Optimization Approach. In *Proceedings of the 4th International Workshop on Web Information and Data Management (WIDM)*. 15–22.

LIST OF FIGURES

2.1	Sample document: XML code, document tree and structural summary	8
2.2	Sample queries against the document tree	10
2.3	Three-Level Model of XML Retrieval: query, schema and document level	13
3.1	Preorder labelling of a sample document tree	18
3.2	Decision and reconstruction of query constraints	20
3.3	Subtree encoding of a sample document tree	24
3.4	Two-dimensional representation of subtree encodings	25
3.5	Binary node representation with path labelling schemes	28
3.6	Path encoding of a sample document tree	31
3.7	Global data structures used by different labelling schemes	33
3.8	Multiplicative encoding of a sample document tree	35
4.1	Unbalanced BIRD encoding of a sample document tree	42
4.2	Child-balanced BIRD labelling of a sample document tree	45
4.3	Child-balanced BIRD labelling for the running example	46
4.4	Totally balanced BIRD labelling of a sample document tree	47
4.5	Reconstruction with the totally balanced BIRD scheme	51
4.6	Layered BIRD labelling of a sample document tree	55
4.7	Efficiency of ancestor reconstruction with different labelling schemes	59
4.8	Efficiency of ancestor decision with different labelling schemes	60
4.9	Effect of updates to the <i>IMDb</i> collection with different labelling schemes	64
4.10	Trade-off between the expressivity, efficiency and updatability of labelling schemes	65
5.1	Different inverted files	72
5.2	Two-dimensional path bitmap	73
5.3	Three-dimensional path bitmap	74
6.1	CADG element table	82
6.2	Runtime performance of the CADG	84
6.3	Storage consumption of the CADG	85
7.1	Node table of the XPath Accelerator scheme	91
8.1	RCADG path table	98
8.2	BIRD document matching rules for deciding binary query constraints	100
8.3	BIRD document matching rules for reconstructing binary query constraints	100
8.4	RCADG result tables	102
8.5	RCADG query rewriting	104
8.6	SQL code for RCADG query evaluation on the schema level	105

8.7	RCADG schema adaptation rules for unary query constraints	106
8.8	RCADG schema adaptation rules for binary query constraints	106
8.9	RCADG schema matching rules for unary query constraints	106
8.10	RCADG schema matching rules for binary query constraints	106
8.11	RCADG query plans	109
8.12	SQL code for RCADG query evaluation on the document level	113
8.13	RCADG document adaptation rules for unary query constraints	114
8.14	RCADG document adaptation rules for binary query constraints	114
8.15	RCADG document matching rules for unary query constraints	115
8.16	SQL code for matching keyword government constraints with the RCADG	117
8.17	SQL code for computing the final query result with the RCADG	117
8.18	SQL code for query evaluation with XRel	123
9.1	Query containment and overlap	131
10.1	Query schematization for the RCADG Cache	143
10.2	Containment and overlap of intermediate and final query results	145
10.3	Integrated query evaluation with the RCADG and RCADG Cache	147
10.4	Sample RCADG Cache contents	148
10.5	Sample RCADG Cache look-up result	149
10.6	Sample RCADG Cache hits	151
10.7	Comparison of keyword constraints with the RCADG Cache	155
10.8	RCADG Cache query plans	158
10.9	SQL code for incremental query evaluation with the RCADG Cache	159
10.10	RCADG Cache query performance evaluation (small scale)	161
10.11	RCADG Cache query performance evaluation (large scale)	162

LIST OF TABLES

2.1	Decidable relations in the document tree	9
3.1	Reconstructible relations in the document tree	20
3.2	Synopsis of various labelling schemes and their expressivity	39
4.1	Reconstruction with the balanced BIRD schemes	50
4.2	Decision with the balanced BIRD schemes	53
4.3	Storage consumption of different labelling schemes	58
4.4	Efficiency of query evaluation with different labelling schemes	61
8.1	Mandatory fields in the RCADG path table	98
8.2	Optional fields in the RCADG path table	98
8.3	Merging of overlapping binary query constraints	103
8.4	Collapsing of transitive binary query constraints	104
8.5	RCADG query performance	119
8.6	Query performance comparison for RCADG, CADG and XPath Accelerator	120
8.7	Query performance comparison for RCADG and XRel (schema level)	121
8.8	Query performance comparison for RCADG and XRel (schema and document level)	122
9.1	Complexity of XPath query containment	132
10.1	Representations of binary query constraints during incremental evaluation	150
13.1	Test document collections	181
14.1	Sample queries against the <i>DBLP</i> and <i>XMark 1100</i> collections	184
14.2	Efficiency profiling of query evaluation with different labelling schemes	185

LIST OF ALGORITHMS

8.1	Query evaluation from scratch with the RCADG	101
8.2	Query planning with the RCADG	110
10.1	Creation of cache hits with the RCADG Cache	152
10.2	Comparison of keyword constraints with the RCADG Cache	154
10.3	Decision of schema-hit containment with the RCADG Cache	156

LIST OF DEFINITIONS

2.1	Document tree	7
2.2	Query	9
2.3	Matching	10
2.4	Query result	10
2.5	Structural summary	11
2.6	Schema tree	11
2.7	<i>S</i> -constraint	11
2.8	<i>D</i> -constraint	12
3.1	Labelling scheme	17
3.2	Subtree encoding	21
3.4	Path encoding	27
3.5	Multiplicative encoding	34
4.1	<i>b</i> -equivalence	43
4.2	Child count	44
4.3	Balanced BIRD weight	44
4.4	Balanced BIRD label	46
9.1	Query overlap	131
9.2	Node-containment	131
9.3	Match-containment	131
9.4	Query containment	131
10.1	Schema-hit containment	144
10.2	RCADG Cache overlap	146

LIST OF LEMMATA

3.3	Pre/Post level/size dependency	23
3.6	Virtual Nodes child reconstruction	35
3.7	Virtual Nodes parent reconstruction	36
4.5	BIRD label order	48
4.6	BIRD labelling function	48
4.7	BIRD label size	48
4.8	BIRD ancestor reconstruction	48
4.9	BIRD child reconstruction	49
4.10	BIRD left-sibling reconstruction	49
4.11	BIRD right-sibling reconstruction	49
4.12	BIRD reconstruction	50
4.13	BIRD totally balanced reconstruction	51
4.14	BIRD decision	52

ABOUT THE AUTHOR



Felix Weigel received his diploma in Computer Science with distinction in 2003 and his Ph. D. in Computer Science with distinction (“magna cum laude”) in 2006 at the University of Munich (LMU). Currently he works as a postdoctoral associate in the database group at Cornell University. His research interests include database technology for XML data, information retrieval in structured documents, user interfaces to document repositories, and knowledge management for the Semantic Web.

