Jiaheng Lu

An Introduction to XML Query Processing and Keyword Search

TSINGHUA UNIVERSITY PRESS

Springer

An Introduction to XML Query Processing and Keyword Search

Jiaheng Lu

An Introduction to XML Query Processing and Keyword Search

With 110 Figures

TSINGHUA UNIVERSITY PRESS

Springer

Jiaheng Lu
School of Information
Renmin University of China
Beijing
People's Republic of China

ISBN 978-3-642-34554-8 ISBN 978-3-642-34555-5 (eBook)
DOI 10.1007/978-3-642-34555-5
Springer Heidelberg New York Dordrecht London

Jointly published with Tsinghua University Press, Beijing
ISBN: 978-7-302-28563-2 Tsinghua University Press, Beijing

Library of Congress Control Number: 2013934532

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

XML is short for eXtensible Markup Language, whose purpose is to aid information systems in sharing structured data, especially via the Internet, to encode documents, and to serialize data. The properties that XML inherited make it the widely popular standard of representing and exchanging data.

When working with those XML data, there are (loosely speaking) three different functions that need to be performed: adding information to the repository, searching and retrieving information from the repository, and updating information from the repository. We focus on all three parts of functions. There are two decades since the beginning of XML, and this field has expanded tremendously and is still expanding. Thus, no book on XML can be comprehensive now—certainly this one is not. We just present which we could present clearly.

What Is the Uniqueness of This Book?

This book aims to provide an understanding of principles and techniques on XML query processing and XML keyword search. For this purpose, progress has been made in the following aspects:

Firstly, we give a brief introduction of XML, including the emergence of XML database, XML data model, and searching and querying XML data. In order to facilitate query process over XML data that conforms to an ordered tree-structure data model efficiently, a number of labeling schemes for XML data have been proposed.

Secondly, we proposed several XML path indexes. Over the past decade, XML has become a commonly used format for storing and exchanging data in a wide variety of systems. Due to this widespread use, the problem of effectively and efficiently managing XML collections has attracted significant attention. Without a structural summary and an efficient index, query processing can be quite inefficient due to an exhaustive traversal on XML data. To overcome the inefficiency, several path indexes have been proposed, such as prefix scheme, extended Dewey ID, and CDBS.

Thirdly, answering twig queries efficiently is important in XML tree pattern processing. In order to perform efficient processing, we introduce two kinds of join algorithms, both of which play significant roles. Also, solutions about how to speed up query processing and how to reduce the intermediate results to save spaces are present.

Fourthly, we show a set of holistic algorithms to efficiently process the extended XML tree patterns. Previous algorithms focus on XML tree pattern queries with only P-C and A-D relationships. Little work has been done on extended XML tree pattern queries which contain wildcards, negation function, and order restriction, all of which are frequently used in XML query languages. The holistic algorithm will make it more completed.

Fifthly, we study XML keyword search semantics algorithms and ranking strategy. We present XML keyword search semantics such as SLCA, VLCA, and MLCEA, which is useful and meaningful for keyword search. Based on some of the semantics, we present XML keyword search algorithms such as DIL Query Processing Algorithm. In addition, we introduce the XML keyword search ranking strategy; we propose an IR-style approach which basically utilizes the statistics of underlying XML data to address these challenges.

Sixthly, we introduce the problem of XML keyword query refinement and offer a novel content-aware XML keyword query refinement framework. We also introduce LCRA, which provides a concise interface where user can explicitly specify their search concern—publications (default) or authors.

Lastly, we present several future works, such as graphical XML data processing, complex XML pattern matching, and MapReduce-based XML query processing.

Jiaheng Lu

Acknowledgement

I would like to express my gratitude to Prof. Tok Wang Ling in National University of Singapore, for his support, advice, patience, and encouragement. He is my PhD advisor and has taught me innumerable lessons and insights on the workings of academic research in general.

My thanks also go to Prof. Mong-Li Lee, Prof. Chee Yong Chan, and Prof. Anthony K H. Tung in National University of Singapore, who provided valuable feedback and suggestions to my idea.

I shall thank my colleagues Prof. Shan Wang, Prof. Xiaoyong Du, Prof. Xiaofeng Meng, Prof. Hong Chen, Prof. Cuiping Li, and Prof. Xuan Zhou in Renmin University of China, who give me tremendous supports and advices on my research on XML data processing.

My thanks also go to my friends Ting Chen, Yabing Chen, Qi He, Changqing Li, Huanzhang Liu, Wei Ni, Cong Sun, Tian Yu, Zhifeng Bao, and Huayu Wu in National University of Singapore. They have contributed to many interesting and good spirited discussions related to this research. They also provided tremendous mental support to me when I got frustrated at times.

My thanks also go to my students Chunbin Lin, Caiyun Yao, Junwei Pan, Haiyong Wang, Si Chen, Siming Yang, and Xiaozhen Huo in Renmin University of China. My thinking on XML was shaped by a long process of exciting and inspiring interactions with my students. I am immensely grateful to all of them.

I would also like to thank Dr. Jirong Wen, Microsoft Research Asia; Prof. Rui Zhang, Melbourne University; Prof. Jianzhong Li, Harbin Institute of Technology; Prof. Bin Cui, Peking University; Prof. Jianhua Feng and Prof. Guoliang Li, Tsinghua University; and Mr. Hanyou Wang and Ms. Hui Xue, Tsinghua University Press, for their recommendation and valuable suggestions.

Last, but not least, I would like to thank my wife Chun Pu for her understanding and love during the past few years. Her support and encouragement was in the end what made this book possible. My parents and parents-in-law receive my deepest gratitude and love for their dedication and the many years of support during my studies.

My research is partially supported by National 863 project (2009AA01Z133); National Science Foundation, China (61170011); Beijing National Science Foundation (109004); and Research Funds of Renmin University of China (No: 11XNJ003).

Jiaheng Lu

Contents

Chapter 1
Introduction

Abstract When working with those XML data, there are three different functions that need to be performed: adding information to the repository, searching and retrieving information from the repository, and updating information from the repository. A good XML database must handle those functions well. In this chapter, we will introduce solutions for XML database, including flat files, relational database, object relational database, and other storage management system.

Keywords Relational database • Object relational database

1.1 XML Data Model

An XML document always starts with a prolog markup. The minimal prolog contains a declaration that identifies the document as an XML document. XML identifies data using tags, which are identifiers enclosed in angle brackets. Collectively, the tags are known as "markup." The most commonly used markup in XML data is element. Element identifies the content it surrounds. For example, Fig. 1.1 shows a simple example XML document. This document starts with a prolog markup that identifies the document as an XML document that conforms to version 1.0 of the XML specification and uses the 8-bit Unicode character encoding scheme (Line 1). The root element (Line 2–14) of the document follows the declaration, which is named as bib element. Generally, each XML document has a single root element. Next, there is an element book (Line 3–13) which describes the information (including author, title, and chapter) of a book. In Line 9, the element text contains both a subelement keyword and character data *XML stands for* . . . .

J. Lu, *An Introduction to XML Query Processing and Keyword Search*,
DOI 10.1007/978-3-642-34555-5_1,

```
<?xml version = "1.0" encoding = “UTF–8”?>
<bib>
 <book>
  <author>Suciu</author>
  <author>Chen</author>
  <title> Advanced Database System </title>
  <chapter><title>XML</title>
   <section><title>XML specification</title>
    <text><keyword>markup</keyword> XML stands for...
    </text>
   </section>
  </chapter>
 </book>
</bib>
```

Fig. 1.1 Example XML document

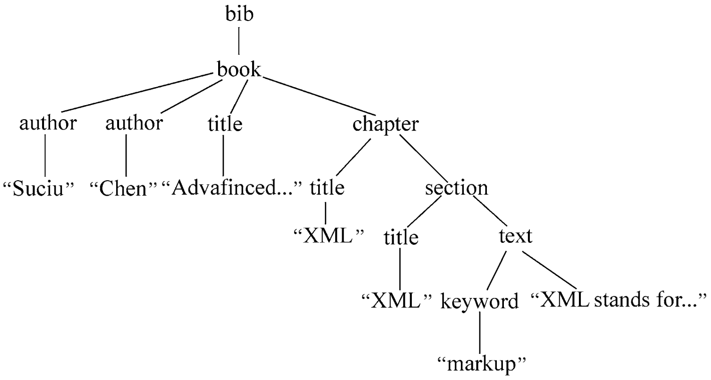


Fig. 1.2 Example XML tree model

Although XML documents can have rather complex internal structures, they can generally be modeled as trees,[1] where tree nodes represent document elements, attributes, and character data and edges represent the element–subelement (or parent–child) relationship. We call such a tree representation of an XML document as an XML tree. Figure 1.2 shows a tree that models the XML document in Fig. 1.1.

XML has grown from a markup language for special purpose documents to a standard for the interchange of heterogenous data over the Web, a common language for distributed computation, and a universal data format to provide users with different views of data. All of these increase the volume of data encoded in XML, consequently increasing the need for database management support for XML documents. An essential concern is how to store and query potentially huge amounts of XML data efficiently [AJP+02, AQM+97, JAC+02, LLHC05, MW99, ZND+01].

[1] For the purpose of this book, when we model XML document as trees, we consider IDREF attributes as not reference links but subelements.

1.2 Emergence of XML Database

XML has penetrated virtually all areas of Internet-related application programming and become the frequently used data exchange framework in the application areas [Abi97, CFI+00, DFS99]. When working with those XML data, there are (loosely speaking) three different functions that need to be performed: adding information to the repository, searching and retrieving information from the repository, and updating information from the repository. A good XML database must handle those functions well. Many solutions for XML database have been proposed, including flat files, relational database [FTS00, Mal99, SSK+01, STZ+99, TVB+02, ZND+01], object relational database [ML02, SYU99], and other storage management system, such as Natix [FM01], TIMBER [JJ06, JLS+04, PJ05, YJR03], and Lore [MAG97]. We briefly discuss these solutions as follows.

1.2.1 Flat File Storage

The simplest type of storage is flat file storage, that is, the main entity is a complete document; internal structure does not play a role. These models may be implemented either on the top of real file systems, such as the file systems available on UNIX, or inside databases where documents are stored as binary large objects (BLOBs). The operation store can be supported very efficiently at low cost, while other operations, such as search, which require access to the internal structure of documents may become prohibitively expensive. Flat file storage is not most appropriate when search is frequent, and the level of granularity required by this storage is the entire document, not the element or character data within the document.

1.2.2 Relational and Object Relational Storage

XML data can be stored in existing relational database. They can benefit from already existing relation database features such as indexing, transaction, and query optimizers. However, due to XML data that is a semistructured data, converting this data model into relation data is necessary. There are mainly two converting methods: generic [FK99] and schema-driven [STZ+99]. Generic method does not make use of schemas but instead defines a generic target schema that captures any XML document.

Schema-driven depends on a given XML schema and defines a set of rules for mapping it to a relational schema. Since the inherent significant difference between rational data model and nested structures of semistructured data, both converting methods need a lot of expensive join operations for query processing.

Mo and Ling [ML02] proposed to use object relational database to store and query XML data. Their method is based on ORA-SS (Object-Relationship-Attribute model for Semistructured Data) data model [DWLL01], which not only reflects the nested structure of semistructured data but also distinguishes between object classes and relationship types and between attributes of objects classes and attributes of relationship types. Compared to the strategies that convert XML to relational database, their methods reduce the redundancy in storage and the costly join operations.

1.2.3 Native Storage of XML Data

Native XML engines are systems that are specially designed for managing XML data [MLLA03]. Compared to the relational database storage of XML data, native XML database does not need the expensive operations to convert XML data to fit in the relational table. The storage and query processing techniques adopted by native XML database are usually more efficient than that based on flat file and relational and object relational storage. In the following, we introduce three native XML storage approaches.

The first approach is to model XML documents using the Document Object Model (DOM) [Abi97]. Internally, each node in a DOM tree has four pointers and two sibling pointers. The filiation pointers include the first child, the last child, the parent, and the root pointers. The sibling pointers point to the previous and the next sibling nodes. The nodes in a DOM tree are serialized into disk pages according to depth-first order (filiation clustering) or breadth-first order (sibling clustering). Lore [MAG97, MW99] and XBase [LWY+02] are two instances of such a storage approach.

The second approach is TIMBER project [JA02], at the University of Michigan, aiming to develop a genuine native XML database engine, designed from scratch. It uses TAX, a bulk algebra for manipulating sets of trees. For the implementation of its Storage Manager module, it uses Shore, a back-end storage system capable for disk storage management, indexing support, buffering, and concurrency control. With TIMBER, it is possible to create indexes on the document's attribute contents or on the element contents. The indexes on attributes are allowed for both text and numeric content. In addition, another kind of index support is the tag index, that, given the name of an element, it returns all the elements of the same name.

Finally, Natix [FM01] is proposed by Kanne and Moerkotte at the University of Mannheim, Germany. It is an efficient and native repository designed from scratch tailored to the requirement of storing and processing XML data. There are three features in Natix system: (1) subtrees of the original XML document are stored together in a single (physical) record; (2) the inner structure of subtrees is retained; and (3) to satisfy special application requirements, the clustering requirements of subtrees are specifiable through a split matrix. Unlike other XML DBMS which provide fully developed functionalities to manage data, Natix is only a repository. It is built from scratch and has no query language, no much work done on indexing and query processing, and no use of DTDs or XML schema.

1.3 XML Query Language and Processing

To retrieve such tree-structured data, a few XML query languages have been proposed in the literature. Examples are Lorel [AQM+97], XML-QL [DFF98], XML-GL [CCD+99], Quilt [CRF00], XPath [BBC04], and XQuery [BCF03]. Of all the existing XML query languages, XQuery is being standardized as the major XML query language. XQuery is derived from the Quilt query language, which in turn borrowed features from several other languages such as XPath. The main building block of XQuery consists of path expressions, which addresses part of XML documents for retrieval, both by value search and structure search in their elements. For example, the following path expression */bib/book[author= 'Suciu']/title* asks for the title of the book written by "Suciu." In Fig. 1.1, this query returns the title *Advanced Database System*.

1.4 XML Keyword Search

The extreme success of web search engines makes keyword Search the most popular search model for ordinary users. As XML is becoming a standard in data representation, it is desirable to support keyword search in XML database. It is a user-friendly way to query XML databases since it allows users to pose queries without the knowledge of complex query languages and the database schema.

Most previous efforts in this area focus on keyword proximity search in XML based on either tree data model or graph (or digraph) data model. Tree data model for XML is generally simple and efficient for keyword proximity search. However, it cannot capture connections such as ID references in XML databases. In contrast, techniques based on graph (or digraph) can capture those connections, but the algorithms based on the graph model are very expensive in many cases. In this book, we will show interconnected object trees model for keyword search to achieve the efficiency of tree model and meanwhile to capture the connections such as ID references in XML by fully exploiting the property and schema information of XML databases. In particular, we will propose ICA (Interested Common Ancestor) semantics to find all predefined interested objects that contain all query keywords. We will also introduce novel IRA (Interested Related Ancestors) semantics to capture the conceptual connections between interested objects and include more objects that only contain some query keywords. Then a novel ranking metric, RelevanceRank, is studied to dynamically assign higher ranks to objects that are more relevant to given keyword query according to the conceptual connections in IRAs. We will design and analyze efficient algorithms for keyword search based on our efficient, which outperforms most existing systems in terms of result quality.

1.5 Book Outline

The content of this book can be divided into five parts.

Part I is an introduction, which contains Chap. 1. Chapter 1 gives a brief introduction of XML, including the emergence of XML database, XML data model, and searching and querying XML data.

Part II discusses query processing, which focuses on tree pattern queries. Part II contains Chaps. 2, 3, 4, and 5. In Chap. 2, in order to facilitate query process over XML data that conforms to an ordered tree-structure data model efficiently, a number of labeling schemes for XML data have been proposed.

The emergence of the Web has increased interests in XML data. Without a structural summary and efficient index, query processing can be quite inefficient due to an exhaustive traversal on XML data. To overcome the inefficiency, several path indexes have been proposed in Chap. 3.

Answering twig queries efficiently is important in XML tree pattern processing. In order to perform efficient processing, Chap. 4 introduces two kinds of join algorithms, both of which play significant roles. Also, solutions about how to speed up query processing and how to reduce the intermediate results to save spaces are present in Chap. 4.

Previous algorithms focus on XML tree pattern queries with only P-C and A-D relationships. Little work has been done on extended XML tree pattern queries which contain wildcards, negation function, and order restriction, all of which are frequently used in XML query languages. Chapter 5 will show a set of holistic algorithms to efficiently process the extended XML tree patterns.

Part III discusses XML keyword search, which contains Chaps. 6, 7, and 8. Chapter 6 presents a survey on the existing XML keyword search semantics algorithms and ranking strategy. In XML keyword search, user queries usually contain irrelevant or mismatched terms, typos, etc., which may easily lead to empty or meaningless results. Chapter 7 introduces the problem of content-aware XML keyword query refinement and offers a novel content-aware XML keyword query refinement framework. Chapter 8 is an introduction to LCRA and LotusX, which provides a concise and graphical interface where users can explicitly specify their search concerns.

Finally, Part IV contains Chap. 9, which summarizes this book and presents several future works.

References

[Abi97] Abiteboul, S.: Querying semi-structured data. In: Proceedings of Database Theory, 6th International Conference, Delphi, Greece, pp. 1–18 (1997)

[AJP+02] Al-khalifa, S., Jagadish, H.V., Patel, J.M., Wu, Y., Koudas, N., Srivastava, D.: Structural joins: a primitive for efficient XML query pattern matching. In: Proceedings of the 20th International Conference on Data Engineering, San Jose, pp. 141–152 (2002)

[AQM+97] Abiteboul, S., Quass, D., Mchugh, J., Widom, J., Wiener, J.L.: The Lorel query language for semistructured data. Int. J. Digit. Libr. **1**(1), 68–88 (1997)

[BBC04] Berglund, A., Boag, S., Chamberlin, D.: XML path language (XPath) 2.0, W3C Working Draft 23 July 2004

[BCF03] Boag, S., Chamberlin, D., Fernandez, M.F.: XQuery 1.0: an XML query language. W3C Working Draft 22 Aug 2003

[CCD+99] Ceri, S., Comai, S., Damiani, E., Fraternali, P. Paraboschi, S., Tanca, L.: XML-GL: a graphical language for querying and restructuring XML documents. In: Proceedings of the 8th International World Wide Web Conference, Toronto, May 1999

[CFI+00] Carey, M.J., Florescu, D., Ives, Z.G., Lu, Y., Shanmugasundaram, J., Shekita, E.J., Subramanian, S.N.: XPERANTO: publishing object-relational data as XML. In: Proceedings of the 3rd International Workshop on the Web and Databases, Dallas, TX, USA (Informal proceedings), pp. 105–110 (2000)

[CRF00] Chamberlin, D.D., Robie, J., Florescu, D.: Quilt: an XML query language for heterogeneous data sources. In: Proceedings of the Third International Workshop on the Web and Databases, Dallas, Texas, USA, pp. 53–62 (2000)

[DFF98] Deutsch, A., Fernandez, M.F., Florescu, D.: A query language for XML, World Wide Web Consortium (1998)

[DFS99] Deutsch, A., Fernandez, M.F., Suciu, D.: Storing semistructured data with STORED. In: Proceedings ACM SIGMOD International Conference on Management of Data, Philadelphia, pp. 431–442 (1999)

[DWLL01] Dobbie, G., Wu, X., Ling, T.W., Lee, M.: ORA-SS: object-relationship-attribute model for semistructured data, Technical Report TR 21/00 National University of Singapore (2001)

[FK99] Florescu, D., Kossmann, D.: Storing and querying XML data using an RDMBS. IEEE Data Eng. Bull. **22**(3), 27–34 (1999)

[FM01] Fiebig, T., Moerkotte, G.: Algebraic XML construction in NATIX. In: Proceedings of WISE, Kyoto, Japan, pp. 212–221 (2001)

[FTS00] Fernandez, M.F., Tan, W.C., Suciu, D.: SilkRoute: trading between relations and XML. Comput. Netw. **33**(1–6), 723–745 (2000)

[JA02] Jagadish, H.V., Al-khalifa, S.: TIMBER: a native XML database, Technical report, University of Michigan (2002)

[JAC+02] Jagadish, H.V., Al-khalifa, S., Chapman, A., Lakshmanan, L.V.S., Nierman, A., Paparizos, S., Patel, J.M., Srivastava, D., Wiwatwattana, N., Wu, Y., Yu, C.: TIMBER: a native XML database. VLDB J. **11**(4), 274–291 (2002)

[JJ06] Jayapandian, M., Jagadish, H.V.: Automating the design and construction of query forms. In: Proceedings of the 22nd International Conference on Data Engineering, Atlanta (2006)

[JLS+04] Jagadish, H.V., Lakshmanan, L.V.S., Scannapieco, M., Srivastava, D., Wiwatwattana, N.: Colorful XML: one hierarchy isn't enough. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, pp. 71–82 (2004)

[LLHC05] Li, H., Lee, M.L., Hsu, W., Cong, G.: An estimation system for XPath expressions. In: Proceedings of the 22nd International Conference on Data Engineering, Tokyo, Japan, pp. 54–65 (2005)

[LWY+02] Lu, H., Wang, G., Yu, G., Bao, Y., Lv, J., Yu, Y.: XBase: making your gigabyte disk files queriable. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Madison, pp. 630–630 (2002)

[MAG97] Mchugh, J., Abiteboul, S., Goldman, R., Quass, D., Widom, J.: Lore: a database management system for semistructured data. SIGMOD Rec. **26**(3), 54–66 (1997)

[Mal99] Malaika, S.: Using XML in relational database applications. In: Proceedings of the 15th International Conference on Data Engineering, Sydney, pp. 167–167 (1999)

[ML02] Mo, Y., Ling, T.W.: Storing and maintaining semistructured data efficiently in an object-relational database. In: Proceedings of the 3rd International Conference on Web Information Systems Engineering, Singapore, pp. 247–256 (2002)

[MLLA03] Meng, X., Luo, D., Lee, M., An, J.: Orientstore: a schema based native XML storage system. In: Proceedings of 29th International Conference on Very Large Data Base, Berlin, pp. 1057–1060 (2003)

[MW99] Mchugh, J., Widom, J.: Query optimization for XML. In: Proceeding of the 25th International Conference on Very Large Data Bases, Edinburgh, pp. 315–326 (1999)

[PJ05] Pararizos, S., Jagadish, H.V.: Pattern tree algebras: sets or sequences. In: Proceedings of 31th International Conference on Very Large Data Bases, Trondheim, Norway, pp. 349–360 (2005)

[SSK+01] Shanmugasundaram, J., Shekita, E.J., Kiernan, J., Krishnamurthy, R., Viglas, S., Naughton, J.F., Tatarinov, I.: A general techniques for querying XML documents using a relational database system. SIGMOD Rec. **30**(3), 20–26 (2001)

[STZ+99] Shanmugasundaram, J., Tufte, K., Zhang, C., He, G., Dewitt, D.J., Naughton, J.F.: Relational database for querying XML documents: limitation and opportunities. In: Proceedings of 25th International Conference on Very Large Data Bases, Edinburgh, pp. 302–314 (1999)

[SYU99] Shimura, T., Yoshikawa, M., Uemura, S.: Storage and retrieval of XML documents using object-relational databases. In: Database and Expert Systems Applications, 10th International Conference, Florence, pp. 206–217 (1999)

[TVB+02] Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J., Zhang. C.: Storing and querying ordered XML using a relational database system. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Madison, pp. 204–215 (2002)

[YJR03] Yu, C., Jagadish, H.V., Radev, D.R.: Querying XML using structures and keywords in TIMBER. In: Proceeding of SIGIR, Toronto, pp. 463–463 (2003)

[ZND+01] Zhang, C., Naughton, J.F., Dewitt, D.J., Luo, Q., Lohman, G.M.: On supporting containment queries in relational database management systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Santa Barbara, pp. 425–436 (2001)

Chapter 2
XML Labeling Scheme

Abstract With the rapid development of the Internet, XML has become the widely popular standard of representing and exchanging data. Documents conforming to the XML standard can be viewed as trees. Elements in XML data can be labeled according to the structure of the document to facilitate query processing. To facilitate query process over XML data that conforms to an ordered tree-structured data model efficiently, this chapter shows a number of labeling schemes for XML data. We classify labeling schemes into two types: static labeling schemes and dynamic labeling scheme.

Keywords Static labeling scheme • Dynamic labeling scheme

2.1 Introducing XML Labeling Scheme

With the rapid development of the Internet, XML has become the widely popular standard of representing and exchanging data. Documents obeying the XML standard can be viewed as trees. Element in XML data can be labeled according to the structure of the document to facilitate query processing. Query language like XPath uses path expressions to traverse XML data. The traditional and most beneficial technique for increasing query performance is the creation of effective indexing. A well-constructed index will allow a query to bypass the need of scanning the entire document for results. Normally, a labeling scheme assigns identifiers to elements such that the hierarchical orders of the elements can be reestablished based on their identifiers. Since hierarchical orders are used extensively in processing XML queries, the reduction of the computing workload for the hierarchy reestablishment is desirable.

To facilitate query process over XML data that conforms to an ordered tree-structured data model efficiently, a number of labeling schemes for XML data have been proposed. We classify labeling scheme into two types: static labeling schemes and dynamic labeling scheme.

J. Lu, *An Introduction to XML Query Processing and Keyword Search*,
DOI 10.1007/978-3-642-34555-5_2,

When XML data are static, the labeling schemes, such as containment scheme (or called region encoding labeling scheme) [BKS02] and prefix scheme (or called Dewey ID labeling scheme) [CKM02, OOP+04, TVB+02], can determine the ancestor–descendant (A-D), parent–child (P-C), etc., relationships efficiently in XML query processing. Some variants have appeared for different purposes. For example, extended Dewey labeling scheme [LLCC05] is developed from Dewey ID labeling scheme [TVB+02]; the unique feature of this scheme is that from the label of an element alone, one can derive the name of all elements in the path from the root to this element. Twig pattern matching also can benefit from it, because TJFast only needs to access labels of leaf nodes to answer queries and significantly reducing I/O cost.

When XML data become dynamic, to efficiently update the labels of labeling scheme, a lot of dynamic XML labeling schemes have been designed for needs, such as region-based dynamic labeling scheme, prefix-based dynamic labeling scheme, and prime labeling scheme. However, most of the techniques have high update cost; they cannot completely avoid relabeling in XML updates. Therefore, we will introduce a compact dynamic binary string (CDBS) encoding [LLH08] and a compact dynamic quaternary string (CDQS) encoding [LLH08] which can be applied broadly to different labeling schemes to efficiently process order-sensitive updates.

2.2 Region Encoding Scheme

Elements in XML data can be labeled according to the structure of the document to facilitate query processing. The region encoding scheme uses textual positions of start and end tags. It can determine the ancestor–descendant (A-D), parent–child (P-C), etc., relationships efficiently in XML query processing if XML data are static.

In the region encoding scheme (or called containment scheme), every node is assigned three values: *start*, *end*, and *level*. *start* and *end* can be generated by counting node numbers from beginning of the document until the *start* and the *end* of the current node. *level* is the depth of the node in the document.

For any two nodes u and v:

1. **[ancestor–descendant]**: If u is an ancestor of v if and only if $u.start < v.start$ and $u.end < v.end$. In other words, the interval of node v is contained in the interval of node u.
2. **[parent–child]**: If u is the parent of v if and only if $u.start < v.start$ and $u.end < v.end$, $u.level = v.level$-1.
3. **[sibling]**: Node u is a sibling of node v if and only if the parent of node u is also a parent of node v.
4. **[preceding–following]**: Node u is a preceding (following) node of node v if and only if $u.start <(>)\ v.start$.

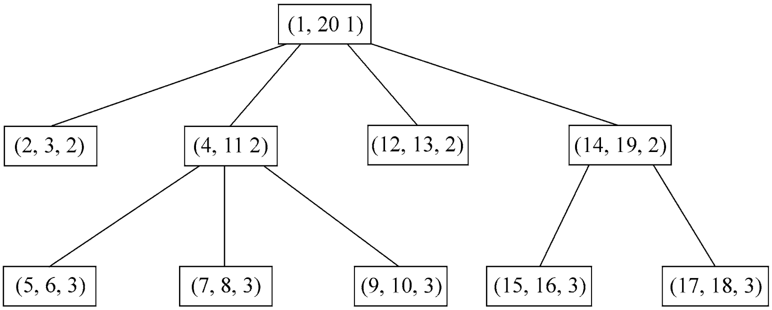


Fig. 2.1 Containment scheme

Ancestor–descendant relationship between two nodes can be determined by comparing intervals. The level of the node is used to distinguish the parent–child relationship from the ancestor–descendant relationship. Therefore, we can use them to evaluate the parent–child and ancestor–descendant relationships between element pairs in a data tree.

Also, this representation of positions of nodes allows for checking order and structural proximity relationships (Fig. 2.1).

2.3 Dewey and Extended Dewey Scheme

2.3.1 Dewey ID Labeling Scheme

In the Dewey ID labeling scheme (or called prefix scheme) [TVB+02], each element is present by a vector:

1. The root is labeled by an empty string ε.
2. The non-root element u, label(u) = label(s).x, where u is the x-th child of s.

Dewey ID labels the x-th child of a node with an integer x, and this x should be concatenated to the prefix (its parent's label) and delimiter (e.g., ".") to form the complete label of this child node (Fig. 2.2).

Also, Dewey ID labeling scheme supports efficient evaluation of structural relationships between nodes.

For any two nodes u and v:

1. Node u is an ancestor of node v if and only if label(u) is a prefix of label(v).
2. Node u is a parent of node v if and only if label(v) has no prefix when removing label(u) from the left side of label(v).
3. Node u is a sibling of node v if they have the same prefix label.
4. Node u is a preceding (following) node of node v if and only if label(u) is smaller (larger) than label(v) lexicographically.

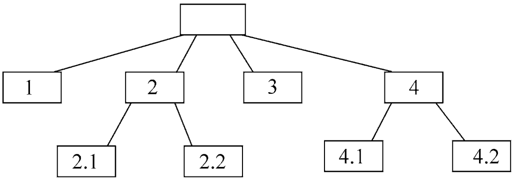


Fig. 2.2 Dewey ID prefix scheme

Dewey ID labeling scheme has a nice property: one can derive the ancestors of an element from its label alone. For example, suppose element *u* is labeled "1.2.3.4", then the parent of *u* is "1.2.3" and the grandparent is "1.2" and so on.

2.3.2 Extended Dewey and FST

With the knowledge of Dewey ID labeling scheme's property, we further consider that if the names of all ancestors of *u* can be derived from label(*u*) alone, then XML path pattern matching can be directly reduced to string matching. For example, if we know that the label "1.2.3.4" presents the path "a/b/c/d", then it is quite straightforward to identify whether the element matches a path pattern (e.g., "//c/d"). Inspired by this observation, in this section, we extend Dewey ID labeling scheme to incorporate the element name information. A straightforward way is to use some bits to present the element name sequence with number presentation, followed by the original Dewey label. The advantage of this approach is simple and easy to implement. However, this method faces the problem of the large label size. In the following, we will propose a more concise scheme to solve this problem. Then we present a "finite state transducer" (FST) to decode element names from this label. For simplicity, we focus the discussion on a single document. The labeling scheme can be easily extended to multiple documents by introducing document ID information (Fig. 2.3).

2.3.2.1 Extended Dewey

The intuition of extended Dewey is to use modulo function to create a mapping from an integer to an element name, such that given a sequence of integers, we can convert it into the sequence of element names.

In the extended Dewey, we need to know a little additional schema information, which we call a child name clue. In particular, given any tag t in a document, the child name clue is all (distinct) names of children of *t*. This clue is easily derived from DTD, XML schema, or other schema constraint. For example, consider the DTD in Fig. 2.4; the tag of all children of bib is only book, and the tags of all

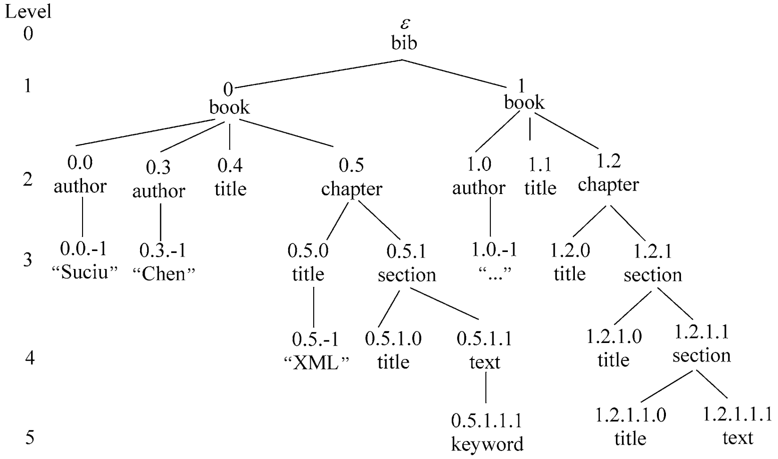


Fig. 2.3 Example of the extended Dewey labeling scheme

```
<!ELEMENT bib (book*)>
<!ELEMENT book (author+, title, chapter*)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT chapter (title, section*)>
<!ELEMENT section (title, (text | section)*)>
<!ELEMENT text (#PCDATA | bold | keyword | emph)*>
<!ELEMENT bold (#PCDATA | bold | keyword | emph)*>
<!ELEMENT keyword (#PCDATA | bold | keyword | emph)*>
<!ELEMENT emph (#PCDATA | bold | keyword | emph)*>
```

Fig. 2.4 DTD for XML document in Fig. 2.3

children of book are author, title, and chapter. Note that even in the case when DTD and XML schema are unavailable, our method is still effective, but we need to scan the document once to get the necessary child name clue before labeling the XML document.

Let us use CT(t) = {t_0, t_1, ..., t_{n-1}} to denote the child name clue of tag t. Suppose there is an ordering for tags in CT(t), where the particular ordering is not important. For example, in Fig. 2.4, CT(book) = {author, title, chapter}. Using child name clues, we may easily create a mapping from an integer to an element name. Suppose CT(t) = {t_0, t_1, ..., t_{n-1}}, for any element e_i with name t_i, we assign an integer x_i to e_i such that xi mod $n = i$. Thus, according to the value of x_i, it is easy to derive its element name. For example, CT(book) = {author, title, chapter}. Suppose e_i is a child element of book and $x_i = 8$, then we see that the name of e_i is chapter, because $x_i \bmod 3 = 2$. In the following, we extend this intuition and describe the construction of extended Dewey labels.

The extended Dewey label of each element can be efficiently generated by a depth-first traversal of the XML tree. Each extended Dewey label is presented as a vector of integers. We use label(u) to denote the extended Dewey label of element u. For each u, label(u) is defined as label(s).x, where s is the parent of u. The computation method of integer x in extended Dewey is a little more involved than that in the original Dewey. In particular, for any element u with parent s in an XML tree:

1. If u is a text value, then $n = -1$.
2. Otherwise, assume that the element name of u is the k-th tag in CT(t_s)($k = 0$, $1, \ldots, n-1$), where t_s denotes the tag of element s.

    (1) If u is the first child of s, then $x = k$.

    (2) Otherwise, assume that the last component of the label of the left sibling of u is y (at this point, the left sibling of u has been labeled), then

$$x = \begin{cases} \left\lfloor \frac{y}{n} \right\rfloor \times n + k & \text{if}(y \mod n) < k \\ \left\lceil \frac{y}{n} \right\rceil \times n + k & \text{otherwise} \end{cases}$$

    where n denotes the size of CT(t_s).

Example 2.1 Figure 2.3 shows an XML document tree that conforms to the DTD in Fig. 2.4. For instance, the labels of chapter under book("0") are computed as follows. Here $k = 2$ (for chapter is the third tag in its child name clue, starting from 0), $y = 4$ (for the last component of "0.4" is 4), and $n = 3$, so $y \bmod 3 = 1 < k$. Then $x = \lfloor 4/3 \rfloor * 3 + 2 = 5$. So "chapter" is assigned the label "0.5".

We show the space complexity of extended Dewey using the following theorem.

Theorem 2.1 *The extended Dewey does not alter the asymptotic space complexity of the original Dewey ID labeling scheme.*

Proof It is not hard to prove that given any element s, the gap between the last components of the labels for every two neighboring elements under s is no more than CT(t_s). Hence, with the binary representation of integers, the length of each component i of extended Dewey label is at most $\log_2|\text{CT}(t_s)|$ more than that of the original Dewey. Therefore, the length difference between an extended Dewey label with m components and an original one is at most $\sum_{i=1}^{m} \log_2 |\text{CT}(t_{s_i})|$. Since m and $|\text{CT}(t_{s_i})|$ are small, it is reasonable to consider this difference is a small constant. As a result, the extended Dewey does not alter asymptotic space complexity of the original Dewey.

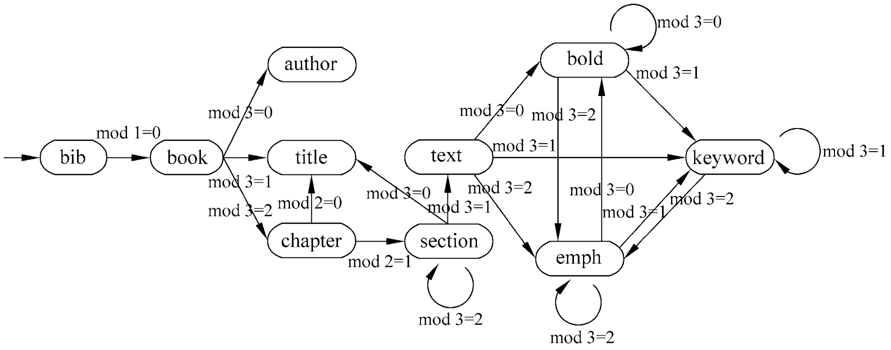


Fig. 2.5 A sample FST for DTD in Fig. 2.4

2.3.2.2 Finite State Transducer

Given the extended Dewey label of any element, we can use a finite state transducer (FST) to convert this label into the sequence of element names which reveals the whole path from the root to this element. We begin this section by presenting a function $F(t, x)$ which will be used to define FST.

Definition 2.1 Let Z denote the nonnegative integer set and Σ denote the alphabet of all distinct tags in an XML document T. Given a tag t in T, suppose $CT(t) = \{t_0, t_1, \ldots, t_{n-1}\}$, a function $F(t, x)$: $\Sigma \times Z \rightarrow \Sigma$ can be defined by $F(t, x) = t_k$, where $k = x \bmod n$.

Definition 2.2 (Finite State Transducer) Given child name clues and an extended Dewey label, we can use a deterministic finite state transducer (*FST*) to translate the label into a sequence of element names. FST is a 5-tuple (I, S, i, δ, o), where:

1. The input set $I = Z \cup \{-1\}$.
2. The set of states $S = \Sigma \cup \{\text{PCDATA}\}$, where PCDATA is a state to denote text value of an element.
3. The initial state i is the tag of the root in the document.
4. The state transition function δ is defined as follows: for $\forall\ t \in \varepsilon$, *if* $x = -1$, $\delta(t, x) = \text{PCDATA}$; otherwise, $\delta(t, x) = F(t, x)$. No other transition is accepted.
5. The output value o is the current state name.

Example 2.2 Figure 2.5 shows the FST for DTD in Fig. 2.4. For clarity, we do not explicitly show the state for PCDATA here. An input –1 from any state will transit to the terminating state PCDATA. This FST can convert any extended Dewey label to an element path. For instance, given an extended Dewey label "0.5.1.1", using the above FST, we derive that its path is "bib/book/chapter/section/text".

As a final remark, it is worth noting three points:

1. In the worst case, the memory size of the above FST is quadratic to the number of distinct element names in XML documents, as the number of transition in FST is quadratic.

2. We allow recursive element names in a document path, which is demonstrated as a loop in FST.
3. The time complexity of FST is linear in the length of an extended Dewey label but independent of the complexity of schema definition.

2.3.2.3 Properties of Extended Dewey

In this section, we summarize the following five properties of extended Dewey labeling scheme:

1. **[Ancestor Name Vision]** Given any extended Dewey label of an element, we can know all its ancestors' names (through FTS).
2. **[Ancestor Label Vision]** Given any extended Dewey label of an element, we can know all its ancestors' label.
3. **[Prefix Relationship]** Two elements have ancestor–descendant relationships if and only if their extended Dewey labels have a prefix relationship.
4. **[Tight Prefix Relationship]** Two elements a and b have parent–child relationships if and only if their extended Dewey labels label(a) and label(b) have a tight prefix relationship. That is, (1) label(a) is a prefix of label(b), and (2) label(b).length – label(a).length $= 1$.
5. **[Order Relationship]** Element a follows (or precedes) element b if and only if label(a) is greater (or smaller) than label(b) with lexicographical order.

The region encoding labeling scheme also can be used for determining ancestor–descendant, parent–child, and order relationships between two elements. But it cannot see the ancestors of an element and therefore has no Property 1 and 2. The original Dewey labeling scheme has Property 2–5, but not Property 1. The first property is unique for extended Dewey. Note that Property 1 and 2 are of paramount importance, since they provide us an extraordinary chance to efficiently process XML path (and twig) queries. For example, given a path query "a/b/c/d", according to the Property 1 and 2, we only need to read the labels of ' to answer this query, which will significantly reduce I/O cost compared to previous algorithms based on the region encoding. One may use extended Dewey labels to design a novel and efficient holistic twig join algorithm, which efficiently utilizes the above five properties (see [LLCC05] for details).

2.4 Dynamic Labeling Scheme

Elements in XML data can be labeled according to the structure of the document to facilitate query processing. Many labeling schemes have been proposed in the literature.

The labeling schemes, such as containment scheme [ABJ89, Die82, LM01, YASU01, ZND+01], prefix scheme [CKM02, OOP+04, TVB+02], and prime scheme [WLH04], can determine the ancestor–descendant (A-D), parent–child (P-C), etc., relationships efficiently in XML query processing if XML data are static.

However, when XML data become dynamic, how to efficiently update the labels of the labeling schemes becomes an important research topic.

[CKM02, TIHW01, TVB+02, XT04] can process updates (inserts or deletes nodes) efficiently if the order of XML is not taken into consideration. However, as we know, the elements in XML are intrinsically ordered, which is referred to as the document order (the element sequence in XML). The relative order of two paragraphs in XML is important because the order may influence the semantics of XML. In addition, the standard XML query languages XPath [BBC+05] and XQuery [BCF+05] include both ordered and unordered queries. Thus, it is very important to maintain the document order when XML is updated.

Some research work [AYU03, CKM02, LL05, OOP+04, SHYY05, TVB+02, WLH04] has been done to maintain the document order in XML updating.

The naive approach to maintain the document order is to leave gaps between adjacent labels in advance [LM01]. Whenever the gaps are filled, that is, the values left in advanced are used up, the labeling schemes have to relabel. This naive approach is suggested in many existing systems, for example, [HBG+03, JAC+02]. But obviously the update cost of this naive approach is expensive, especially when updates frequently happen.

Amagasa et al. [AYU03] use float-point numbers instead of integers to store labels. However, the number of distinct values is limited by the number of bits used in the representation of float-point values in a computer. Thus, due to the float-point precision, the method in [AYU03] still cannot avoid relabeling.

All the existing techniques have high update cost; they cannot completely avoid relabeling in XML updates, and they will increase the label size which will influence query performance. Thus, in this chapter we propose a novel compact dynamic binary string (CDBS) encoding (used to store labels in labeling schemes) and a compact dynamic quaternary string (CDQS) encoding [LLH08] to efficiently process order-sensitive updates. The CDBS is the most compact, and its update cost is the cheapest compared to all other techniques. The CDQS is the only technique which can completely avoid relabeling in XML leaf node updates.

In addition, none of the existing techniques can efficiently process internal node updates; therefore, we also propose techniques to much more efficiently process internal node updates though we cannot completely avoid relabeling in internal node updates.

2.4.1 Region-Based Dynamic Labeling Scheme

It should be noted that relabeling in the region encoding scheme is not only to maintain the document order. If XML trees are not relabeled after a node is inserted, the region encoding scheme cannot work correctly to determine the ancestor–descendant, parent–child, etc., relationships.

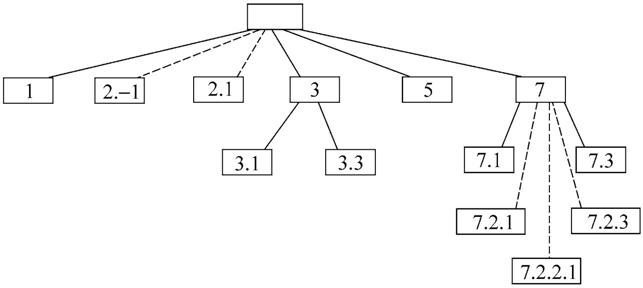


Fig. 2.6 OrdPath prefix scheme

The approach the region encoding scheme used to maintain the document order is to leave gaps between adjacent labels in advance. Whenever the gaps are filled, that is, the values left in advanced are used up, the labeling schemes have to relabel. But obviously the update cost of this approach is expensive, especially when updates frequently happen.

To solve the relabeling problem, we use float-point numbers instead of integers to store labels. Each node in an XML tree is labeled with (start, end), which represents the start and end positions of the element in the document, respectively. It seems that float-point solves the relabeling problem. However, the number of distinct values is limited by the number of bits used in the representation of float-point values in a computer. Even if using values with large gaps, it still cannot avoid relabeling due to the float-point precision. Therefore, using real values instead of integers only provides limited benefits for the label updating.

2.4.2 Prefix-Based Dynamic Labeling Scheme

Compared with region-based labeling scheme, prefix-based labeling scheme is more robust for insertion. However, relabeling still cannot be avoided for insertions between two consecutive siblings.

OrdPath [OOP+04] is a prefix labeling scheme which uses a clever "careting-in" scheme to support insertions. Though OrdPath [OOP+04] is dynamic to some extent to process updates (will encounter the overflow problem, its update cost is not so cheap and it will reduce the query performance).

2.4.2.1 OrdPath Labeling Scheme

OrdPath is similar to Dewey ID [TVB+02], but it only uses odd numbers at the initial labeling (see Fig. 2.6). When the XML tree is updated, it uses the even number between two odd numbers to concatenate another odd number.

Example 2.3 Given three Dewey ID labels “1”, “2”, and “3”, we can easily know that they are siblings. In addition, given two Dewey ID labels “2” and “2.1”, we can easily know that “2” is a parent of “2.1”. But for OrdPath (see Fig. 2.6), its labels are “1”, “3”, “5”, etc.; when inserting a label between “1” and “3”, it uses the even number between “1” and “3”, that is, “2” to concatenate another odd number, that is, “1” as the label of this inserted node, that is, the inserted label is “2.1”. In OrdPath, “2.1” is at the same level as “1”, “3”, etc., that is, “2.1” is a sibling of “1” and “3”. Furthermore, when inserting one more node between “1” and “2.1”, OrdPath uses “2.−1” as the inserted label, and “2.−1” is also the sibling of “1”, “2.1”, and “3”. In this way, OrdPath need not relabel the existing nodes in insertions; however, this makes OrdPath slow in determining the sibling, parent–child, etc., relationships in XML query processing. Therefore, OrdPath gets better update performance by reducing the query performance. This is not desirable.

OrdPath can avoid the relabeling to some extent, but it reduces the query performance and its update cost is expensive.

2.4.2.2 Drawbacks

1. It wastes half of the total numbers compared to Dewey ID (wastes the even numbers; even after insertion, it still wastes the even number, e.g., “2.0” between “2.−1” and “2.1” is still not used after insertion).
2. It can be seen from Example 2.3 that “1”, “2.−1”, “2.1”, and “3” are at the same level, that is, they are siblings. OrdPath needs more time to determine this based on the even and odd numbers (the even number is not a level) which will reduce its query performance.
3. OrdPath needs the addition and division operations to calculate the even number between two odd numbers which is expensive in updating. It is also possible that OrdPath only uses the addition operation to get the even number, but if there are many deletions, the insertion with only addition operation is a bias and the label size will increase fast. Moreover, even if OrdPath only uses the addition operation in processing updates, the addition operation is not so cheap.

2.4.3 Prime Labeling Scheme

In prime labeling scheme [WLH04], we use prime numbers to label XML trees. The root node is labeled with “1” (integer). Based on a top–down approach, each node is given a unique prime number (self-label) and the label of each node is the product of its parent node’s label (parent-label) and its own self-label.

For any two nodes u and v, u is an ancestor of v iff label(v) mod label(u) $=0$. Node u is a parent of node v iff label(v)/self-label(v) $=$ label(u). Node u is a sibling of node v iff label(u)/self-label(u) $=$ label(v)/self-label(v). Prime uses the

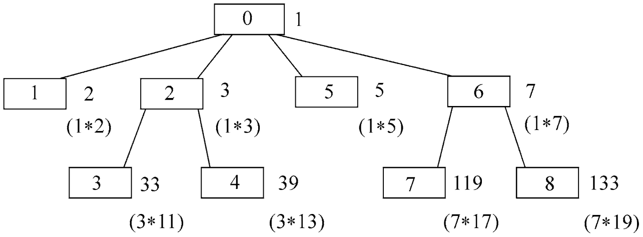


Fig. 2.7 Prime scheme

simultaneous congruence (SC) values to decide the document order, that is, SC mod self-label = document order, then it compares the document orders of two nodes.

Example 2.4 Prime labels the root firstly, then the child of the root, and next the grandchild of the root. We consider one label in Fig. 2.7. The third (document order; the number above the node) node is labeled with "33" (the right number), which is the product of its parent-label "3" and its self-label "11".

Example 2.5 The SC value for the eight nodes (except the root) in Fig. 2.7 is 8,965,025. That is to say, 8,965,025 mod 2 = 1 (here 2 is the self-label and 1 is the document order), 8,965,025 mod 3 = 2, ..., 8,965,025 mod 17 = 7, and 8,065,025 mod 19 = 8. Prime only needs to store this SC value and the self-labels rather than store the document order.

When the document order is changed, prime only needs to recalculate the SC values instead of relabeling.

Example 2.6 When a new sibling node is inserted before the first node (see Fig. 2.7; the inserted node is now the first child of the root), the next available prime number is 23, then the label of the new inserted node is 23 (1 × 23). This new inserted node becomes now the first node (document order), and the orders of the nodes after this inserted node should all be added with 1 (the old orders are calculated based on the old SC value). Prime calculates the new SC value for the new ordering, which is 28,364,406 such that 28,364,406 mod 23 = 1, 28,364,406 mod 2 = 2, 28,364,406 mod 3 = 3, ..., 28,364,406 mod 17 = 8, and 28,364,406 mod 19 = 9.

2.4.4 The Encoding Schemes

The main idea of most labeling schemes is to leave some unused values for future insertions. When the unused values are used up later, they have to relabel the existing nodes, that is, they cannot completely avoid relabeling in XML leaf node updates. Although prime supports order-sensitive updates without relabeling the existing nodes, it needs to recalculate the SC values based on the new ordering of nodes. The recalculation is much more time consuming. OrdPath is dynamic to some extent to process the updates, but it needs to decode its codes and use the addition and division operations to calculate the even number between two odd numbers, which make its update cost not so cheap. In addition, the better update performance of OrdPath does not come without a cost. It wastes a lot of even numbers which makes its label size larger, and it needs more time to determine the prefix levels based on the even and odd numbers in XML query processing.

In this section, we will introduce compact dynamic binary string (CDBS) encoding [LLH08]. CDBS has two important properties: (1) CDBS codes can be inserted between any two consecutive CDBS codes with orders kept and without re-encoding the existing codes; (2) CDBS is orthogonal to specific labeling schemes; thus, it can be applied broadly to different labeling schemes or other applications to efficiently process updates. Moreover, because CDBS will encounter the overflow problem, we also introduce a compact dynamic quaternary string (CDQS) encoding which can completely avoid relabeling in XML leaf node updates no matter what the labeling schemes are.

The most important feature of those approaches is that labels are compared based on the lexicographical order.

2.4.4.1 Lexicographical Order

Definition 2.3 (Lexicographical Order $\prec$) Given two binary strings SL and SR (SL represents the left binary string and SR represents the right binary string), SL is said to be lexicographically equal to SR if and only if they are exactly the same. SL is said to be lexicographically smaller than SR ($SL \prec SR$) if and only if the lexicographical comparison of SL and SR is bit by bit from left to right. If the current bit of SL is 0 and the current bit of SR is 1, then $SL \prec SR$ and stop the comparison, or SL is a prefix of SR.

Based on Algorithm 2.1, we illustrate how to find a binary string SM (SM represents the middle binary string) between two lexicographically ordered binary strings SL and SR such that SL < SM < SR lexicographically.

Algorithm 2.1: AssignMiddleBinaryString(SL, SR)
Input: SL < SM < SR, SL and SR, which are both ended with "1" and SL < SR
Output: SM (ended with 1) such that SL < SM < SR lexicographically

```
if size($S_L$) $\geq$ size($S_R$) then
    $S_M = S_L \odot$ "1";          // case (a), $\odot$ means concatenation
else $S_M = S_R$ with the last bit "1" changed to "01"; // case (b)
end
return $S_M$
```

Note that the last bit of S_L and S_R in Algorithm 2.1 is required to be 1. In Example 2.6, we show why we require the last bit of the binary string to be "1".

Example 2.7 Suppose there are two binary strings "0" and "00", "0" < "00" lexicographically because "0" is a prefix of "00", but we cannot insert a binary string SM between "0" and "00" such that "0" < SM < "00". Accordingly we require the binary strings to be ended with "1".

Theorem 2.2 *Given any two binary strings S_L and S_R which are both ended with* "1" *and $S_L < S_R$, we can always find a binary string S_M based on Algorithm 2.1 such that $S_L < S_M < S_R$ lexicographically.*

Proof

1. If size(S_{L}) ≥ size(S_{R}), we process S_{M} based on Lines 1 and 2 in Algorithm 2.1, that is, $S_{\mathrm{M}} = S_{\mathrm{L}} \odot$ "1".

   (1) S_{M} is that S_{L} concatenates one more "1"; thus, S_{L} is a prefix of S_{M} according to condition 2 in Definition 2.3, $S_{\mathrm{L}} < S_{\mathrm{M}}$ lexicographically.

   (2) Since size(S_{L}) ≥ size(S_{R}) and $S_{\mathrm{L}} < S_{\mathrm{R}}$, condition 1 in Definition 2.3 must be satisfied. That means there is a position: the bit of S_{L} at this position is "0", and the bit of S_{R} at this position is "1". Therefore, when we concatenate one more "1" after S_{L}, S_{M} is still smaller than S_{R} lexicographically (the lexicographical comparison is from left to right), that is, $S_{\mathrm{M}} < S_{\mathrm{R}}$.

   Based on (1) and (2), $S_{\mathrm{L}} < S_{\mathrm{M}} < S_{\mathrm{R}}$ lexicographically when size(S_{L}) ≥ size(S_{R}).

2. Case (b): if size(S_{L}) < size(S_{R}), we process S_{M} based on Line 3 in Algorithm 2.1, that is, $S_{\mathrm{M}} = S_{\mathrm{R}}$ with the last bit "1" changed to "01".

   (1) If the first (size(S_{R}) – 1) bits of S_{R} are larger than S_{L} lexicographically, $S_{\mathrm{L}} < S_{\mathrm{M}}$ because S_{M} is the first (size(S_{R}) – 1) bits of $S_{\mathrm{R}} \odot$ "1". If the first (size(S_{R}) – 1) bits of S_{R} are exactly the same as the S_{L}, $S_{\mathrm{L}} < S_{\mathrm{M}}$ because S_{M} is $S_{\mathrm{L}} \odot$ "1". (S_{L} is the same as the first (size(S_{R}) – 1) bits of S_{R}; S_{L} is a prefix of S_{M}.) Note that the first (size(S_{R}) – 1) bits of S_{R} cannot be smaller than S_{L} lexicographically, otherwise S_{L} will be larger than S_{R} lexicographically (conflict to the condition in Theorem 2.2). Thus, $S_{\mathrm{L}} < S_{\mathrm{M}}$.

   (2) If we do not consider the last two bits "01" of S_{M} and the last bit "1" of S_{R}, S_{M} is exactly the same as S_{R}, and "01" < "1" lexicographically. Thus, $S_{\mathrm{M}} < S_{\mathrm{R}}$.

   Based on (1) and (2), $S_{\mathrm{L}} < S_{\mathrm{M}} < S_{\mathrm{R}}$ lexicographically when size(S_{L}) < size(S_{R}).

Example 2.8 To insert a binary string between "001" and "01", the size of "001" is 3 which is larger than the size 2 of "01"; therefore, we directly concatenate one more "1" after "001". The inserted binary string is "0011" and "001" < "0011" < "01" lexicographically. To insert a binary string between "01" and "011", the size of "01" is 2 which is smaller than the size 3 of "011"; therefore, we change the last "1" of "011" to "01", that is, the inserted binary string is "0101"; obviously "01" < "0101" < "011" lexicographically.

Lemma 2.1 *The S_M in Theorem 2.2 returned by Algorithm 2.1 is ended with* "1".

Proof This is obvious when we check Algorithm 2.1. Lines 1 and 2 indicate that the end bit of S_{M} is "1" when size(S_{L}) $\geq$ size(S_{R}), and Lines 3 indicate that the end bit of S_{M} is "1" when size(S_{L}) < size(S_{R}); therefore, S_{L} is ended with "1".

Corollary 2.1 *Given any two binary strings S_L and S_R which are both ended with* "1" *and $S_L < S_R$, we can always find two binary strings SM1 and SM2 such that $S_L < SM1 < SM2 < S_R$ lexicographically.*

Proof Based on Theorem 2.2, we can insert a binary string S_{M} between S_{L} and S_{R}. Based on Lemma 2.1, we know that S_{M} is also ended with "1". Therefore, based on Theorem 2.2, we can insert another binary string between S_{L} and S_{M} or between S_{M} and S_{R}.

Theorem 2.2 and Corollary 2.1 guarantee the low cost in XML updating. Algorithm 2.1 can be applied to any two ordered binary strings (ended with "1") for insertions. On the other hand, to maintain the high query performance, we should not increase the label size when decreasing the update cost.

2.4.4.2 A Compact Dynamic Binary String Encoding

In this section, we introduce a compact dynamic binary string encoding (CDBS), proposed in [LLH08], and based on Algorithm 2.1, CDBS supports updates efficiently.

We firstly use an example to illustrate how CDBS encodes a set of numbers. Table 2.1 shows the binary number encoding (V-Binary and F-Binary) and CDBS (V-CDBS and F-CDBS) encoding of 18 numbers.

We choose 18 as an example, in fact, CDBS can encode any number. When encoding 18 decimal numbers in binary, they are shown in Column 2 (V-Binary column) of Table 2.1 which have variable lengths, called V-Binary.

The following steps show the details of how to encode the 18 decimal numbers based on CDBS encoding (binary string). Column 3 (V-CDBS column) of Table 2.1 shows CDBS, which is called V-CDBS because it is also encoded with variable lengths.

Step 1: In the encoding of the 18 numbers, we suppose there is one more number before number 1, say number 0, and one more number after number 18, say number 19.

Step 2: We firstly encode the middle number with binary string "1". The middle number is 10 where 10 is calculated in this way, $10 = \mathrm{round}(0 + (19 - 0)/2)$. The V-CDBS code of number 10 is "1".

Step 3: Next we encode the middle number between 0 and 10 and between 10 and 19. The middle number between 0 and 10 is 5 ($5 = \mathrm{round}(0 + (10{-}0)/2)$) and the middle number between 10 and 19 is 15 ($15 = \mathrm{round}(10 + (19 - 10)/2)$).

Table 2.1 Binary and CDBS encoding

Decimal number	V-Binary	V-CDBS	F-Binary	F-CDBS
1	1	00001	00001	00001
2	10	0001	00010	00010
3	11	001	00011	00100
4	100	0011	00100	00110
5	101	01	00101	01000
6	110	01001	00110	01001
7	111	0101	00111	01010
8	1000	011	01000	01100
9	1001	0111	01001	01110
10	1010	1	01010	10000
11	1011	10001	01011	10001
12	1100	1001	01100	10010
13	1101	101	01101	10100
14	1110	1011	01110	10110
15	1111	11	01111	11000
16	10000	1101	10000	11010
17	10001	111	10001	11100
18	10010	1111	10010	11110

Step 4: To encode number 5, the code size of number 0 is 0, and the code size of number 10 is 1. This is Case (b) in Algorithm 2.1 where size(S_L) < size(S_R). Thus, based on Line 3 in Algorithm 2.1, the V-CDBS code of number 5 is "01" ("1" → "01").

Step 5: To encode number 15, the tenth code (S_L) is "1" now with size 1 bit, and the nineteenth code (S_R) is empty now with size 0. This is Case (a) in Algorithm 2.1 where size(S_L) $\geq$ size(S_R). Therefore, based on Lines 1 and 2 in Algorithm 2.1, the V-CDBS code of number 15 is "11" ("1" $\odot$ "1" → "11").

Step 6: Next we encode the middle numbers between 0 and 5, between 5 and 10, between 10 and 15, and between 15 and 19, which are numbers 3, 8, 13, and 17, respectively. The encodings of these numbers are still based on Case (a) or Case (b) in Algorithm 2.1.

In this way, all the numbers except 0 will be encoded because the round function will reach the larger value (divided by 2), and we need to discard the V-CDBS code for number 19 since number 19 does not exist actually.

Also the decimal numbers 1–18 can be encoded with fixed length binary numbers, called F-Binary (F-Binary column of Table 2.1). Since 18 needs 5 bits to store, zero or more "0"s should be concatenated before each code of V-Binary. On the other hand, when representing our CDBS using fixed length, called F-CDBS, we concatenate "0"s after the V-CDBS codes (F-CDBS column of Table 2.1). Because F-CDBS is that some "0"s are concatenated after the V-CDBS codes, we focus on V-CDBS to introduce the algorithm.

2.4.4.3 V-CDBS Encoding Algorithm

Algorithm 2.2 is the V-CDBS encoding algorithm. We use the procedure V-CDBS-SubEncoding to get all the codes of the numbers. Finally number 0 and number (TN + 1) should be discarded since they do not exist actually.

Algorithm 2.2: V-CDBS Encoding (TN)
Input: A positive integer TN
Output: The V-CDBS codes for numbers 1 to TN

```
Suppose there is one more number before the first number, called number 0, and
  one more number after the last number, called number (TN + 1);
Define an array codeArr[0, TN + 1] // the size of codeArr is TN+2; // each
  element of the codeArr is empty at beginning;
V-CDBS-SubEncoding (codeArr, 0, TN + 1);
Discard the 0th and (TN + 1)-th elements of codeArr;
```

Procedure V-CDBS-SubEncoding (codeArr, P_L, P_R)

```
$P_M$ = round(($P_L$ + $P_R$) / 2)(($P_L$ + $P_R$) / 2);
if $P_L + 1 < P_R$ then
    codeArr[$P_M$] = assignMiddleBinaryString(codeArr[$P_L$], codeArr[$P_R$]);
    V-CDBS-SubEncoding(codeArr, $P_L$, $P_M$);
    V-CDBS-SubEncoding(coderArr, $P_M$, $P_R$);
end
```

V-CDBS-SubEncoding is a recursive procedure, the input of which is an array codeArr, the left position "P_L" and the right position "P_R" in array codeArr. This procedure assigns codeArr[P_M] (corresponding to S_M in Algorithm 2.1) using the AssignMiddleBinaryString (Algorithm 2.1), then it uses the new left and right positions to call the V-CDBS-SubEncoding procedure itself, until each (except 0) element of the array codeArr has a value.

Theorem 2.3 *Given a positive integer TN, Algorithm 2.2 can encode all the numbers from 1 to TN with V-CDBS codes.*

Proof The V-CDBS encoding is like the binary search. As we know, the binary search will not miss any values in the search. Therefore, Algorithm 2.2 can encode each number without missing.

Example 2.9 The V-CDBS codes in Table 2.1 are lexicographically ordered from top to bottom.

The CDBS codes are ended with "1", and lexicographically ordered; therefore, we can insert without relabeling in updates based on CDBS.

2.4.4.4 Size of Different Encodings

V-Binary

For V-Binary, one number is stored with one bit ("1", see in Table 2.1), two numbers are stored with 2 bits ("10" and "11"), and four numbers are stored with 3 bits ("100", "101", "110", and "111"), …; therefore, the total size of V-Binary is

$$1 \times 1 + 2 \times 2 \times 2^2 \times 3 + 2^3 \times 4 + \cdots + 2^n \times (n + 1) = n \times 2^{n+1} + 1 \qquad (2.1)$$

Suppose the total number is N, which should be equal to $2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$. Thus, formula (2.1) becomes

$$N \log(N + 1) - N + \log(N + 1)N \qquad (2.2)$$

V-CDBS

When considering V-CDBS, it has one code ("1") stored with one bit, two codes ("01" and "11") stored with two bits, and four codes ("001", "011", "101", and "111") stored with three bits, …; therefore, V-CDBS has the same code size as V-Binary.

In addition, since V-Binary and V-CDBS have variable lengths, we need to store the size of each code. A fixed length number of bits are used to store the size of the codes. The maximal size for a code is log(N). To store this size, the bits required are log(log(N)), and the total bits required to store the sizes of all the variable codes are Nlog(log(N)). Taking formula (2.2) into account, the total sizes of V-Binary and V-CDBS both are

$$N \log(N + 1) + N \log(\log(N)) - N + \log(N + 1) \qquad (2.3)$$

F-Binary

To store N numbers with fixed lengths, the size required is

$$N \log(N) \qquad (2.4)$$

The size of the F-Binary code also needs to be stored, but needs to be stored only once with size log(log(N)). Therefore, the total size for F-Binary is

$$N \log(N) + \log(\log(N)) \qquad (2.5)$$

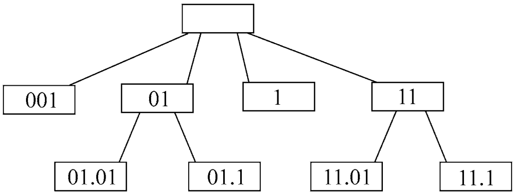


Fig. 2.8 V-CDBS-Prefix scheme (for Fig. 2.2)

F-CDBS

F-CDBS has the same total code size as Formula (2.5).

Theorem 2.4 *V-CDBS and F-CDBS are the most compact variable and fixed length dynamic binary string encodings.*

Proof As we know, the V-Binary and F-Binary are encodings for the consecutive decimal numbers, and there are no gaps between any two consecutive numbers; thus, V-Binary and F-Binary are the most compact encodings. In addition, from the above size analysis, we know that V-CDBS and F-CDBS have the same total sizes as V-Binary and F-Binary, respectively. Therefore, V-CDBS and F-CDBS are also the most compact. Though the size of F-CDBS is smaller than the size of V-CDBS, it is easier for F-CDBS to encounter the overflow problem. Later we'll talk about this overflow problem.

2.4.4.5 Applying CDBS to Different Labeling Schemes

In this section, we mainly illustrate how V-CDBS can be applied to different labeling schemes. F-CDBS is similar since it only concatenates zeros to V-CDBS codes.

When we replace the "start" and "end" values 1–18 of the containment scheme in Fig. 2.1 with the V-CDBS codes in Table 2.1 and based on the lexicographical comparison, a V-CDBS-based containment labeling scheme is formed, called V-CDBS-Containment.

Similarly, we can replace the decimal numbers (see Fig. 2.2) in the prefix labeling scheme with V-CDBS codes, then a V-CDBS-based prefix labeling scheme is formed, called V-CDBS-Prefix. We use the following example to show V-CDBS-Prefix.

Example 2.10 From Fig. 2.2, we can see that the root has four children. To encode four numbers based on Algorithm 2.2, the V-CDBS codes will be "001", "01", "1", and "11". Similarly if there are two siblings, their self-labels are "01" and "1". Figure 2.8 shows V-CDBS-Prefix.

Similarly we can apply V-CDBS to the prime labeling scheme to record the document order. But because prime employs the modular and division operations to determine the ancestor–descendant, etc., relationships, its query efficiency is quite bad. Therefore, we do not discuss in detail how V-CDBS is applied to prime.

2.4.4.6 CDQS Encoding

The size of each V-CDBS code is stored with fixed length. If many nodes are inserted into the XML tree, the size of the length field is not enough for the new labels; then we have to relabel all the existing nodes. Even if we increase the size of the length field to a larger number, it still cannot completely avoid relabeling, and it will waste the storage space. This is called the overflow problem.

To solve the overflow problem, we observed that the size of V-CDBS is used only to separate different V-CDBS codes. After separation, we can directly compare the V-CDBS codes from left to right. Therefore, to solve the overflow problem, the way is to find a separator which can separate different V-CDBS codes; meanwhile this separator will not encounter the overflow problem. In binary string, there are only two symbols "0" and "1"; if we use "0" or "1" as the separator, only one symbol is left and CDBS will not be dynamic. Therefore, we introduce a quaternary string encoding which improved from CDBS can help to completely avoid relabeling in XML leaf node updates.

2.4.4.7 CDQS Encoding

Four symbols "0", "1", "2", and "3" are used in the quaternary string, and each symbol is stored with two bits, that is, "00", "01", "10", and "11". CDQS code is a special quaternary string; "0" is used as the separator and only "1", "2", and "3" are used in the CDQS code itself. Because "0" is used as the separator, it is not appropriate to concatenate "0"s for the fixed length CDQS, that is, F-CDQS. Therefore, when we talk about CDQS, it is equivalent to V-CDQS.

Algorithm 2.3 shows how to insert a quaternary string between two CDQS codes (two quaternary strings).

Algorithm 2.3: AssignInsertedQuaternaryString(S_L, S_R)
Input: $S_L < S_R$, S_L and S_R are ended with "2" or "3"
Output: S_M such that $S_L < S_M < S_R$ lexicographically

```
if size(S_L) > size(S_R) then                    // Case (a)
    if the last symbol of S_L is "2" then
        S_M = S_L with the last symbol changed from "2" to "3";
    else if the last symbol of S_L is "3" then
        S_M = S_L ⊙ "2";                         //⊙ means concatenation
    end
else if size(S_L) = size(S_R) then               // Case (b)
    S_M = S_L ⊙ "2";
else if size(S_L) < size(S_R) then               // Case (c)
    S_M = S_R with the last symbol "2" changed to "12";
end
return S_M;
```

Table 2.2 CDQS encoding

Decimal number	CDQS	Decimal number	CDQS
1	112	10	223
2	12	11	23
3	122	12	232
4	13	13	3
5	132	14	312
6	2	15	32
7	212	16	322
8	22	17	33
9	222	18	332

Still based on the 18 numbers in Table 2.1, firstly, we use examples to show how CDQS works (see Table 2.2).

Step 1: In the encoding of the 18 numbers based on CDQS, we suppose there is one more number before number 1, say number it 0, and one more number after number 18, say number 19.

Step 2: The (1/3)-th number is encoded with "2", and the (2/3)-th number is encoded with "3". The (1/3)-th number is number 6, which is calculated in this way, $6 = \text{round}(0 + (19 - 0)/3)$. The (2/3)-th number is number 13 ($13 = \text{round}(0 + (19 - 0) \times 2/3)$).

Step 3: The (1/3)-th and (2/3)-th numbers between number 0 and number 6 are number 2 ($2 = \text{round}(0 + (6 - 0)/3)$) and number 4 ($4 = \text{round}(0 + (6 - 0)2/3)$). The CDQS code of number 0 (S_L) is now empty with size 0 bit, and the CDQS code of number 6 (S_R) is now "2" with size 2 bits. This is Case (b) where $\text{size}(S_L) < \text{size}(S_R)$. In this case, the (1/3)-th code is that we change the last symbol "2" of S_R to "12", that is, the code of number 2 is "12" ("2" → "12"), and the (2/3)-th code is that we change the last symbol "2" of S_R to "13", that is, the code of number 4 is "13" ("2"→"13").

Step 4: The (1/3)-th and (2/3)-th numbers between numbers 6 and 13 are numbers 8 ($8 = \text{round}(6 + (13 - 6)/3)$) and 11 ($11 = \text{round}(6 + (13 - 6) \times 2/3)$). The CDQS code of number 6 (S_L) is "2" with size 2 bits and the code of number 13 (S_R) is "3" with size 2 bits. This is the Case (b) in algorithm; in this case, the (1/3)-th code is that we directly concatenate one more "2" after the S_L, that is, the code of number 8 is "22" ("2"$\odot$"2"→"22"), and the (2/3)-th code is that we change the last symbol of 8 (S_L), that is, the code of number 11 is "23".

Step 5: The (1/3)-th and (2/3)-th numbers between numbers 13 and 19 are numbers 15 ($15 = \text{round}(13 + (19 - 13)/3)$) and 17 ($17 = \text{round}(13 + (19 - 13) \times 2/3)$). The code of number 13 (S_L) is "3" with size 2 bits and the code of number 19 (S_R) is empty now with size 0 bit. This is the Case (a) in Algorithm 2.3. Therefore, the CDQS code of number 15 is "32" ("3"$\odot$"2"→"32"), and the code of number 17 is "33" (change the last "2" of number 15 to "3").

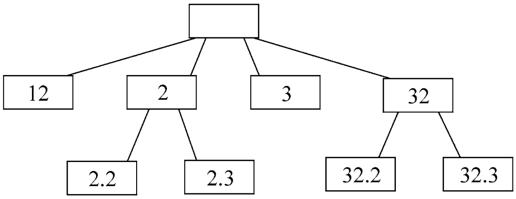


Fig. 2.9 CDQS-Prefix scheme

The formal algorithm of CDQS is similar to the V-CDBS algorithm (Algorithms 2.1 and 2.2). The difference is that CDQS is based on the (1/3)-th and (2/3)-th positions rather than the (1/2)-th position in V-CDBS. The above Steps 1–5 are an illustration of the formal algorithms for CDQS.

Theorem 2.5 *Algorithm 2.3 guarantees that a quaternary string can be inserted between two consecutive CDQS codes with the orders kept and without re-encoding any existing numbers.*

Proof When we check Algorithm 2.3, all the conditions can guarantee that $S_L < S_M < S_R$ lexicographically.

Corollary 2.2 *Algorithm 2.3 guarantees that infinite number of quaternary strings can be inserted between any two consecutive CDQS codes.*

Proof When recursively using Algorithm 2.3 for the insertions.

Theorem 2.6 *CDQS can completely avoid relabeling in XML leaf node updates.*

Proof We use “0” as the separator to separate different CDQS codes, and “0” will never encounter the overflow problem. Also Corollary 2.2 guarantees that infinite number of quaternary strings can be inserted between any two consecutive CDQS codes.

The existing labeling schemes cannot completely avoid the relabeling in internal node updates; this is the drawback of the existing labeling schemes.

2.4.4.8 Application of CDQS Encoding

For the prefix scheme, we use one separator “0” as the delimiter to separate different components of a label (e.g., separate “2” and “3” in “2.3”; the delimiter “0” is equivalent to “.”; note “.” cannot be stored together with numbers in the implementation) and use two consecutive “0”, that is, “00”, as the separator to separate different labels (e.g., separate “2.2” and “2.3”).

Example 2.11 To store the first three nodes “12”, “2”, and “2.2” in Fig. 2.9 (except the root which is empty) in the hard disk, they are stored as “120020020200”. Based on the separator “00”, we can separate “12”, “2”, and “202”, and if necessary, we can separate different components of a label, for example, separate “2” and “2” in “202” based on the delimiter “0”.

2.4.4.9 Extensions of CDBS and CDQS

By further extending CDQS, we can use octal and hex string encodings to process updates, called CDOS and CDHS, respectively. The CDQS grossly wastes 1/4 of the total numbers for the separator. If we use CDOS and CDHS encodings, only 1/8 and 1/16 of the total numbers are wasted grossly. Thus, CDOS and CDHS encodings will be more compact when the total number is large. On the other hand, the separator sizes of CDOS and CDHS encodings are three and four bits, respectively, which will make CDOS and CDHS encodings not so compact as expected.

2.5 Summary

In this chapter, we present existing XML labeling schemes which are classified into static and dynamic labeling schemes.

Static labeling schemes facilitate efficient determination of various relationships of the nodes in an XML tree which is essential for XML query processing. We introduce three types of static labeling schemes: containment, Dewey ID, and extended Dewey labeling schemes. Dynamic labeling scheme supported the dynamic update performance efficiently. In addition, we introduce three types of dynamic labeling schemes: region-based, prefix-based, and prime labeling scheme. We also pointed out the limitations of existing labeling schemes for XML data assuming documents are frequently updated.

At last, we introduce a compact dynamic binary string (CDBS) encoding, proposed in [LLH08], which is orthogonal to specific labeling schemes; therefore, it can be applied broadly to different labeling schemes, for example, containment, prefix, and prime schemes, to maintain the document order when XML is updated.

References

[ABJ89] Agrawal, R., Borgida, A., Jagadish, H.V.: Efficient management of transitive relationships in large data and knowledge bases. In: Proceeding of the ACM SIGMOD International Conference on Management of Data (SIGMOD'89), Portland, Oregon, pp. 253–262 (1989)

[AYU03] Amagasa, T., Yoshikawa, M., Uemura, S.: QRS: a robust numbering scheme for XML documents. In: Proceeding of the 19th International Conference on Data Engineering (ICDE'03), Bangalore, pp. 705–707 (2003)

[BBC+05] Berglund, A., Boag, S., Chamberlin, D., Fernandez, M.F., Keay, M., Robie, J., Simon, J.: XML path language (XPath) 2.0. W3c Working Draft 04, Apr 2005

[BCF+05] Boag, S., Chamberlin, D., Fernandez, M.F., Florescu, D., Robie, J., Simon, J.: XQuery 1.0: an XML query language. W3C Working Draft 04, Apr 2005

[BKS02] Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. Technical Report, Columbia University (2002)

[CKM02] Cohen, E., Kaplan, H., Milo, T.: Labeling dynamic XML trees. In: PODS, Madison, pp. 271–281 (2002)

[Die82] Dietz, P.F.: Maintaining order in a linked list. In: Proceeding of the 14th Annual ACM Symposium on Theory of Computing (STOC'82), San Francisco, pp. 122–127 (1982)

[HBG+03] Halverson, A., Burger, J., Galanis, L., Kini, A., Krishnamurthy, R., Rao, A.N., Tain, F., Viglas, S., Wang, Y., Naughton, J.F., DeWitt, D.J.: Mixed mode XML query processing. In: Proceeding of the 29th International Conference on Very Large Data Bases (VLDB'03), Berlin, Germany, pp. 225–236 (2003)

[JAC+02] Jagadish, H.V., Al-khalifa, S., Chapman, A., Lakshmanan, L.V.S., Nierman, A., Paparizos, S., Patel, J.M., Srivastava, D., Wiwatwattana, N., Wu, Y., Yu, C.: TIMBER: a native XML database. VLDB J. **11**(4), 274–291 (2002)

[LL05] Li, C., Ling, T.W.: QED: a novel quaternary encoding to completely avoid re-labeling in XML updates. In: Proceeding of the 14th International Conference on Information and Knowledge Management (CIKM'05), Bremen, pp. 501–508 (2005)

[LLCC05] Lu, J., Ling, T.W., Chan, C., Chen, T.: From region encoding to extended Dewey: on efficient processing of XML twig pattern matching. In: Proceedings of 31th International Conference on Very Large Data Bases (VLDB), Trondheim, pp. 193–204 (2005)

[LLH08] Li, C., Ling, T.W., Hu, M.: Efficient updates in dynamic XML data: from binary string to quaternary string. VLDB J. **17**(3), 573–601 (2008)

[LM01] Li, Q., Moon, B.: Indexing and querying XML data for regular path expressions. In: Proceedings of the 27th VLDB Conference, Rome, Italy, pp. 361–370 (2001)

[OOP+04] O'Neil, P.E., O'Neil, E.J., Pal, S., Cseri, I., Schaller, G., Westbury, N.: ORDPATHs: insert-friendly XML node labels. In: Proceeding of the ACM SIGMOD International Conference on Management of Data (SIGMOD'04), Paris, pp. 903–908 (2004)

[SHYY05] Silberstein, A., He, H., Yi, K., Yang, J.: BOXes: efficient maintenance of order-based labeling for dynamic XML data. In: Proceeding of the 21st International Conference on Data Engineering (ICDE'05), Tokyo, pp. 285–296 (2005)

[TIHW01] Tatarinov, I., Ives, Z.G., Halevy, A.Y., Weld, D.S.: Updating XML. In: Proceeding of the ACM SIGMOD International Conference on Management of Data (SIGMOD'01), Santa Barbara (2001)

[TVB+02] Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J., Zhang, C.: Storing and querying ordered XML using a relational database system. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Madison, pp. 204–215 (2002)

[WLH04] Wu, X., Lee, M., Hsu, W.: A prime number labeling scheme for dynamic ordered XML trees. In: Proceedings of ICDE, Boston, pp. 66–78 (2004)

[XT04] Xing, G., Tseng, B.: Extendible range-based numbering scheme for XML document. In: Proceeding of the International Conference on Information Technology: Coding and Computing (ITCC'04), Las Vegas, pp. 140–141 (2004)

[YASU01] Yoshikawa, M., Amagasa, T., Shimura, T., Uemura, S.: XRel: a path-based approach to storage and retrieval of XML documents using relational databases. ACM Trans. Internet Tech. **1**(1), 110–141 (2001)

[ZND+01] Zhang, C., Naughton, J.F., Dewitt, D.J., Luo, Q., Lohman, G.M.: On supporting containment queries in relational database management systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Santa Barbara, pp. 425–436 (2001)

Chapter 3
XML Data Indexing

Abstract The emergence of the Web has increased interests in XML data. XML query languages such as XQuery and XPath use label paths to traverse the irregularly structured data. Without a structural summary and efficient index, query processing can be quite inefficient due to an exhaustive traversal on XML data. To overcome the inefficiency, several path indexes have been proposed in the research community. DataGuides and the 1-Index can be viewed as covering indexes, for simple path expressions over tree- or graph-structured XML data. By representing both XML documents and XML queries in structure-encoded sequences, querying XML data is equivalent to finding subsequence matches. We will also introduce the above index structures in this chapter.

Keywords DataGuides • 1-Index • ViST • PRIX

3.1 Introducing XML Data Indexing

The emergence of the Web has increased interests in XML data. XML query languages such as XQuery and XPath use label paths to traverse the irregularly structured data. Without a structural summary and efficient index, query processing can be quite inefficient due to an exhaustive traversal on XML data. To overcome the inefficiency, several path indexes have been proposed in the research community.

With the rapidly increasing popularity of XML for data representation, there is a lot of interest in query processing over data that conforms to a labeled-tree or labeled-graph data model. DataGuides [GW97] and the 1-Index [MS99] can be viewed as covering indexes (traditional relational query acceleration techniques), for simple path expressions over tree- or graph-structured XML data. DataGuides and 1-Indexes are in the category of generalized path indexes that represent all paths starting from the root in XML data. They are generally useful for processing queries with path expressions starting from the root. While the forward-and-backward index

J. Lu, *An Introduction to XML Query Processing and Keyword Search*,
DOI 10.1007/978-3-642-34555-5_3,

[KBNK02] already proposed in the literature can be viewed as a structure analogous to a summary table or covering index, it is also the smallest such index that covers all branching path expression queries.

By representing both XML documents and XML queries in structure-encoded sequences, querying XML data is equivalent to finding subsequence matches. ViST is a novel index structure for searching XML documents with that method. And PRIX is a new way of indexing XML documents and processing twig patterns in an XML database by transforming XML document in the database into a sequence of labels .We will also introduce those index structures in this chapter.

3.2 Indexes on XML Tree Structure

Many database researchers developed various path indexes to support label path expressions. Goldman and Widom [GW97] provided a path index, called the strong DataGuides. The strong DataGuides is restricted to a simple label path and is not useful in complex path queries with several regular expressions. The building algorithm of the strong DataGuides emulates the conversion algorithm from the nondeterministic finite automaton (NFA) to the deterministic finite automaton (DFA). This conversion takes linear time for tree-structured data and exponential time in the worst case for graph-structured data.

Milo and Suciu [MS99] provided another index family (1-Index). Their approach is based on the backward simulation and the backward bisimulation which are originated from the graph verification area. The 1-Index coincides with the strong DataGuides on tree- structured data. The 1-Index can be considered as a non-deterministic version of the strong DataGuides. The forward-and-backward index (F&B-Index) can be viewed as a covering index for branching path expression queries. We also show that among a large and natural class of indexes, it is the smallest index that can cover all branching path expression queries.

3.2.1 DataGuides

3.2.1.1 Foundations

Object Exchange Model

DataGuides [GW97] is based on the Object Exchange Model (OEM for short) [PGW95], a simple and flexible data model that originates from the Tsimmis project at Stanford University [PGW95]. OEM itself adapts easily to any graph-structured data model. In OEM, each object contains an object identifier (oid) and a value. A value may be atomic or complex.

Figure 3.1 presents a very small sample OEM database, representing a portion of an imaginary eating guide database. Each object has an integer oid. The database

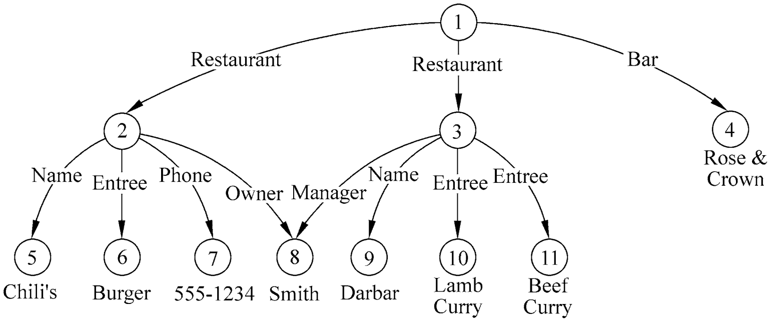


Fig. 3.1 A sample OEM database

contains one complex root object with three subobjects, two Restaurants, and one Bar. Each Restaurant is a complex object and the Bar is atomic.

Definition 3.1 Here are definitions to describe an OEM database and define DataGuides:

1. A label path of an OEM object o is a sequence of one or more dot-separated labels, $l_1, l_2, \ldots, l_n$, such that we can traverse a path of n edges $(e_1, \ldots, e_n)$ from o where edge e_i has label l_i.
2. A data path of an OEM object o is a dot-separated alternating sequence of labels and oids of the form $l_1{\cdot}o_1, l_2{\cdot}o_2, \ldots, l_n{\cdot}o_n$ such that we can traverse from o a path of n edges $(e_1, \ldots, e_n)$ through n objects $(x_1, \ldots, x_n)$ where edge e_i has label l_i and object x_i has oid o_i.
3. A data path d is an instance of a label path l if the sequence of labels in d is equal to l.
4. In an OEM object s, a *target set* is a set t of oids such that there exists some label path l of s where $t = \{o|\ l_1{\cdot}o_1, l_2{\cdot}o_2, \ldots, l_n{\cdot}o_n$ is a data path instance of $l\}$. That is, a target set t is the set of all objects that can be reached by traversing a given label path l of s. We also say that t is *the target set of l in s*, and we write $t = T_s(l)$. We say that l reaches any element of t, and likewise each element of t is *reachable* via l.

Lorel Query Language

Lorel (for Lore language) was developed at Stanford to enable queries over semistructure OEM databases. Lorel is based on OQL (Object Query Language) [Cat93], with modifications and enhancements to support semistructure data. As an extremely simple example, in Fig. 3.1 the Lorel query

Select Restaurant.Entree

returns all entrees served by any restaurant, the set of objects {6, 10, 11}.

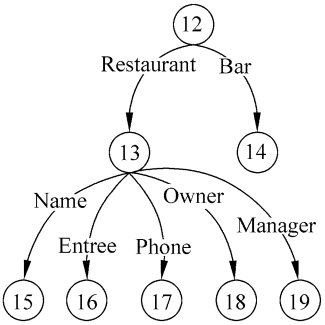


Fig. 3.2 A DataGuides for Fig. 3.1

3.2.1.2 Strong DataGuides

DataGuides

We are now ready to define a DataGuides, intended to be a *concise*, *accurate*, and *convenient* summary of the structure of a database. Figure 3.2 shows a DataGuides for the source OEM database shown in Fig. 3.1.

Definition 3.2 (DataGuides) A DataGuides for an OEM source object s is an OEM object d such that every label path of s has exactly one data path instance in d, and every label path of d is a label path of s.

Using a DataGuides, we can check whether a given label path of length n exists in the original database by considering at most n objects in the DataGuides. For example, in Fig. 3.2 we need only to examine the outgoing edges of objects 12 and 13 to verify that the path Restaurant Owner exists in the database. In Fig. 3.2, the five different labeled outgoing edges of object 13 represent all possible labels that ever follow Restaurant in the source.

Annotations

Beyond using a DataGuides to summarize the structure of a source, we may wish to keep additional information in a DataGuides. For example, consider a source with a label path l. To aid query formulation, we might want to present to a user sample database values that are reachable via l. Finally, for query processing, direct access through the DataGuides to all objects reachable via l can be very useful. The following definition classifies all of these examples.

Definition 3.3 (Annotation) In a source database s, given a label path l, a property of the set of objects that comprise the target set of l in s is said to be an *annotation* of l. That is, an annotation of a label path is a statement about the set of objects in the database reachable by that path.

Building Strong DataGuides

We define a class of DataGuides that supports annotations as described in the previous subsection. Intuitively, we are interested in DataGuides where each set of label paths that share the same (singleton) target set in the DataGuides is exactly the set of label paths that share the same target set in the source.

Definition 3.4 (Strong DataGuides) Consider OEM objects s and d, where d is a DataGuides for a source s. Given a label path l of s, let $T_s(l)$ be the target set of l in s, and let $T_d(l)$ be the (singleton) target set of l in d. Let $L_s(l) = \{m \mid T_s(m) = T_s(l)\}$. That is, $L_s(l)$ is the set of all label paths in s that share the same target set as l. Similarly, let $L_d(l) = \{m \mid T_d(m) = T_d(l)\}$. That is, $L_d(l)$ is the set of all label paths in d that share the same target set as l. If, for all label paths l of s, $L_s(l) = L_d(l)$, then d is a strong DataGuides for s.

Theorem 3.1 *Suppose d is a strong DataGuides for a source s. If an annotation p of some label path l is stored on the object o reachable via l in d, then p describes the target set in s of each label path that reaches o.*

Proof Suppose otherwise. Then there exists some label path m that reaches o, such that p incorrectly describes the target set of m in s. This implies that $T_s(m) \neq T_s(l)$, since we know by Definition 3.3 that p is a valid property of $T_s(l)$. We reuse the notation from the definition of a strong DataGuides: let $L_d(l)$ denote the set of label paths in d whose target set is $T_d(l)$, and let $L_s(l)$ denote the set of label paths in s whose target set is $T_s(l)$. By construction, $L_d(l)$ contains both l and m. By definition of a strong DataGuides, $L_d(l) = L_s(l)$. Therefore, l and m are both elements of $L_s(l)$. But this means that $T_s(m)$, the target set of m in s, is equal to $T_s(l)$, a contradiction to $T_s(l) \neq T_s(m)$, derived above.

Theorem 3.2 *Suppose d is a strong DataGuides for a source s. Given any target set t of s, t is by definition the target set of some label path l. Compute $T_d(l)$, the target set of l in d, which has a single element o. Let F describe this procedure, which takes a source target set as input and yields a DataGuides object as output. In a strong DataGuides, F induces a one-to-one correspondence between source target sets and DataGuides objects.*

Proof We show that F is (1) a function, (2) one-to-one, and (3) onto.

1. To show F is a function we prove that for any two source target sets t and u, if $t = u$ then $F(t) = F(u)$. t is the target set of some label path l, and u is the target set of some label path m, so $t = T_s(l)$ and $u = T_s(m)$. If $t = u$, then l and m are both elements of $L_s(l)$, the set of label paths in s that share $T_s(l)$. Since d is strong, $L_s(l) = L_d(l)$. Therefore, m is also an element of $L_d(l)$, $T_d(l) = T_d(m)$, and their single elements are equal. Hence, $F(t) = F(u)$.
2. We show that F is one-to-one using the same notation and a symmetrical argument. If $F(t) = F(u)$, by construction we know that $T_d(l) = T_d(m)$. l and m are therefore both elements of $L_d(l)$ and by definition of a strong DataGuides are also elements of $L_s(l)$. Therefore, $T_s(l) = T_s(m)$, that is, $t = u$.

3. Finally, we see that the accuracy constraint of any DataGuides guarantees that *F* is onto. Any object in *d* must be reachable by some label path *l* that also exists (and therefore has a target set) in *s*.

3.2.1.3 Building a Strong DataGuides

Strong DataGuides are easy to create. In a depth-first fashion, we examine the source target sets reachable by all possible label paths. Each time we encounter a new target set *t* for some path *l*, we create a new object *o* for *t* in the DataGuides—object *o* is the single element of the DataGuides target set of *l*. Theorem 3.2 guarantees that if we ever see *t* again via a different label path *m*, rather than creating a new DataGuides object, we instead add an edge to the DataGuides such that *m* will also refer to *o*. A hash table mapping source target sets to DataGuides objects serves this purpose.

The algorithm is specified in Algorithm 3.1. Note that we must create and insert DataGuides objects into targetHash before recursion, in order to prevent a cyclic OEM source from causing an infinite loop. Also, since we compute target sets to construct the DataGuides, we can easily augment the algorithm to store annotations in the DataGuides.

Algorithm 3.1: Algorithm to create a strong DataGuides

```
// MakeDataGuide: algorithm to build a strong DataGuides over a source database
// Input: o, the oid of the root of a source database
// Effect: dg is set to be the root of a strong DataGuides for o
targetHash = global empty hash table, to map source target sets to DataGuides
    object
dg = global oid
MakeDataGuide(o) {
    dg = NewObject()
    targetHash.Insert({o}, dg)
    RecursiveMake({o}, dg)
}
RecursiveMake(t1, d1) {
    p = set of <label, oid> children pairs of each object in t1
    foreach(unique label l in p) {
        t2 = set of oids paired with l in p
        d2 = targetHash.Lookup(t2)
        if(d2 != nil) {
            add an edge from d1 to d2 with label l
        } else {
            d2 = NewObject()
            targetHash.Insert(t2, d2)
            add an edge from d1 to d2 with label l
            RecursiveMake(t2, d2)
        }
    }
}
```

3.2.2 1-Index

1-Index [MS99] is a novel, general index structure for semistructure databases. We start by reviewing the basic framework on databases and queries.

3.2.2.1 Review: Data Model and Query Languages

Semistructure data is modeled as a labeled graph, in which nodes correspond to the objects in the database and edges to their attributes. Unlike the relational or object-oriented data models, the labeled graph model carries both data and schema information, making it easy to represent irregular data and treat data coming from different sources in a uniform manner. We assume an infinite set D of data values and an infinite set N of nodes.

Definition 3.5 A data graph DB $= (V; E; R)$ is a labeled rooted graph, where $V \subseteq N$ is a finite set of nodes, $E \subseteq V$; D, V is a set of labeled edges, and $R \subseteq V$ is a set of root nodes. *W. l. o. g.* we assume that all the nodes in V are reachable from some root in R. We will often refer to such a data graph as a database.

Definition 3.6 (Path Expressions) We assume a set of base predicates $p1, p2, \ldots,$ over the domain of values D and denote with F the set of Boolean combinations of such predicates. We assume that we have effective procedures for evaluating the truth values of sentences $f(d)$ and $\exists x f(x)$, for $f \in F$ and $d \in D$.

Definition 3.7 (Path Templates) A path template t has the form $T_1 x_1 T_2 x_2 \ldots T_n x_n$ where each T_i is either a regular path expression or one of the following two place holders: $\boxed{P}$ and $\boxed{F}$. A query path q is obtained from t by instantiating each of the $\boxed{P}$ place holders by some regular path expressions and each of the $\boxed{F}$ place holders by some formula. The query path thus obtained is called an instantiation of the path template t. The set of all such instantiations is denoted inst(t).

Example 3.1 Consider the path template (*.Restaurant) $x_1 \boxed{P} x_2$ Name $x_3 \boxed{F} x_4$. The following three query paths are possible instantiations:

$q_1 =$ (*.Restaurant) x_1 * x_2 Name x_3 Fridays x_4
$q_2 =$ (*.Restaurant) x_1 * x_2 Name x_3 _ x_4
$q_3 =$ (*.Restaurant) x_1 $(\varepsilon \mid \_)$ x_2 Name x_3 Fridays x_4

Given a path template t, our goal is to construct an index structure that will enable an efficient evaluation of queries in inst(t).

The templates are used to guide the indexing mechanism to concentrate on the more interesting parts of the data graph. For example, if we know that the database contains a restaurants directory and that most common queries refer to the restaurant and its name, we may use a path template such as the one above. As another example, assume we know nothing about the database, but users never ask for more than k objects on a path. Then we may take $t = \boxed{P}\, x_1 \boxed{P}\, x_2 \ \ldots \ \boxed{P}\, x_k$ and build the corresponding index.

3.2.2.2 Naive Index

Our goal here is to compute efficiently queries $q \in \text{inst}(\boxed{P}\, x)$. A naive way is the following: For each node v in DB, let Lv(DB), or Lv in short, be the set of words on paths from some root node to v:

$$Lv(\text{DB}) \stackrel{\text{def}}{=} \left\{ w | w = a_1, \ldots, a_n, \exists \text{ a path } v_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} v, \text{ with } v_0 \text{ a root node} \right\}$$

Next, define the language equivalence relation, $v \equiv u$ on nodes in DB to be

$$v \equiv u \Leftrightarrow Lv = Lu$$

We denote with $[v]$ the equivalence class of v. Clearly, there are no more equivalence classes than nodes in DB. Language equivalence is important because two nodes v, u in DB can be distinguished by a query path in $\text{inst}(\boxed{P}\, x)$ iff $v \not\equiv u$. A naive index can be constructed as follows: it consists of the collection of all equivalence classes s_1, s_2, ..., each accompanied by (1) an automaton/regular expression describing the corresponding language and (2) the set of nodes in the equivalence class. We call this set the extent of s_i, and denote it by extent (s_i).

Given the naive index, a query path of the form Px can be evaluated. By iterating over all the classes s_i, if the language of that class has a nonempty intersection with $W(P)$ when testing, the answer of the query is the union of all extent (s_i) for which this intersection is not empty.

This naive approach is inefficient, for two reasons:

1. Construction Cost: the construction of the index is very expensive since computing the equivalence classes for a given data graph is a PSPACE complete problem [SM73].
2. Index Size: the automaton/regular expressions associated with different equivalence classes have overlapping parts which are stored redundantly. This also results in inefficient query evaluation, since we have to intersect $W(P)$ with each regular language.

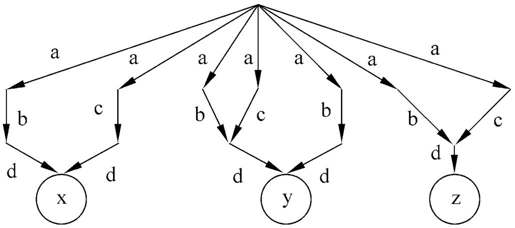


Fig. 3.3 A data graph on which the relations $\equiv$, $\approx_\mathrm{s}$, and $\approx_\mathrm{b}$ differ

3.2.2.3 Refinements

To tackle the construction cost, we consider refinements. An equivalence relation $\approx$ is called a refinement if:

$$v \approx u \Rightarrow v \equiv u$$

We discuss here two choices for refinements: bisimulation $\approx_\mathrm{b}$ and simulation $\approx_\mathrm{s}$. Both are discussed extensively in the literature [HHK95, Mil89, PT87]. The idea that these can be used to approximate the language equivalence dates back to the modeling of reactive systems and process algebras [HHK95]. For completeness, we revise their definitions here. Unlike standard definitions in the literature, we need to traverse edges *backward*, because L_v refers to the set of paths leading into v.

Definition 3.8 Let DB be a data graph. A binary relation $\sim$ on its nodes is a backwards bisimulation if:

1. If $v \sim v'$ and v is a root, then so is v'.
2. Conversely, if $v \sim v'$ is a root, then so is v.
3. If $v \sim v'$, then for any edge $u \xrightarrow{a} v$. there exists an edge $u' \xrightarrow{a} v'$, s.t. $u \sim u'$.
4. Conversely, if $v \sim v'$, then for any edge $u' \xrightarrow{a} v'$, there exists an edge $u \xrightarrow{a} v$, s.t. $u \sim u'$.

A binary relation $\leq$ is a backwards simulation, if it satisfies conditions 1 and 3. Since we consider only backwards simulations and bisimulations, we safely refer to them as simulation and bisimulation.

Two nodes v, u are bisimilar, in notation $v \approx_\mathrm{b} u$, iff there exists a bisimulation $\sim$ s.t. $v \sim u$. Two nodes v, u are similar, in notation $v \approx_\mathrm{s} u$, if there exists two simulations$\leq$, $\leq'$ s.t. $v \leq u$, and $v \leq' u$.

We have $v \approx_\mathrm{b} u \Rightarrow v \approx_\mathrm{s} u \Rightarrow v \equiv u$, hence both $\approx_\mathrm{b}$ and $\approx_\mathrm{s}$ are refinements. The implications are strict as illustrated in Fig. 3.3, where $x \equiv y \equiv z$, $x \not\approx_\mathrm{x} y$ $y \approx_\mathrm{s} z$, and $x \not\approx_\mathrm{b} y \not\approx_\mathrm{b} z$.

In constructing our indexes, we will use either a bisimulation or a simulation. We prove a slightly more general statement. Let us say that a database DB has unique incoming labels if for any node x, whenever a, b are labels of two distinct edges entering x, then $a \neq b$. In particular, tree databases have unique incoming labels.

Proposition 3.1 *If DB is a graph database with unique incoming labels, then* $\equiv$, $\approx_\mathrm{s}$, *and* $\approx_\mathrm{b}$ *coincide.*

Proof Recall that we only consider accessible graph databases, that is, in which every node is accessible from some root. We will show that $\equiv$ is a bisimulation: this proves that $v \equiv u \Rightarrow v \approx_\mathrm{b} u$, and the proposition follows. We check the four conditions in Definition 3.8. If $v \equiv u$ and v is a root, then $\varepsilon \in L_v$, hence $\varepsilon \in L_u$, so u is a root too. This proves items 1 and 2. Let $v \equiv u$ and let $v' \xrightarrow{a} v$ be some edge. Hence $L_v = L_1 \cdot a \cup L_2$, where $L_1 = L_{v'}$, while L_2 is a language which does not contain any words ending in a (because DB has unique incoming labels). It follows that $L_u = L_1 \cdot$ a $\cup$ L_2. Since v' is an accessible node in DB, we have $L_1 \neq \emptyset$, hence there exists some edge $u' \xrightarrow{a} u$ entering u and it also follows that $L_{v'} = L_{u'}$. Now we can define 1-Indexes.

Definition 3.9 Given a database DB and a refinement $\approx$, the 1-Index I(DB) is a rooted, labeled graph defined as follows. Its nodes are equivalence classes $[v]$ of $\approx$; for each edge $v \xrightarrow{a} v'$ in DB, there exists an edge $[v] \xrightarrow{a} [v']$ in I(DB); the roots are $[r]$ for each root r in DB. When DB is clear from the context, we omit it and simply write I.

We store I as follows. First we associate an oid s to each node in I and store I's graph structure in a standard fashion. Second, we record for each node s the nodes in DB belonging to that equivalence class, which we denote extent(s). That is, if s is an oid for $[v]$, then extent(s) $= [v]$. The space for I incurs two costs: the space for the graph I and that for the extents. The graph is at most as large as the data graph DB, but we will argue that in practice it may be much less. The extents contain each node in DB exactly once. This is similar to an unclustered index in a relational databases, where each tuple id occurs exactly once in the index.

We describe now how to evaluate a query path Px. Rather than evaluating it on the data graph DB, we evaluate it on the index graph I(DB). Let $\{s_1, s_2, \ldots, s_k\}$ be the set of nodes in I(DB) that satisfy the query path. Then the answer of the query on DB is extent(s_1) $\cup$ extent(s_2) $\cup$ … $\cup$ extent(s_k). The correctness of this algorithm follows from the following proposition.

Proposition 3.2 *Let $\approx$ be a refinement on DB. Then, for any node v in DB,* $L_v(\mathrm{DB}) = L_{[v]}(I(\mathrm{DB}))$.

Proof The inclusion $L_v \subseteq L_{[v]}$ holds for any equivalence relation $\approx$, not only refinements: this is because any path $v_0 \xrightarrow{a_1} v_1 \xrightarrow{a_2} v_2 \cdots$ in DB, with v_0 a root node, has a corresponding path $[v_0] \xrightarrow{a_1} [v_1] \xrightarrow{a_2} [v_2] \cdots$ in I. For the converse, we prove by induction on the length of a word w that if $w \in L_{[v]}$, then $w \in L_v$. When $w = \varepsilon$ (the empty word), then $[v]$ is a root of I. Hence $v \approx r$, for some root r. This implies $L_v = L_r$, so $\varepsilon \in L_v$. When $w = w_1 \cdot$ a, then we consider the last edge in I: $s \xrightarrow{a} [v]$, with $w_1 \in L_s$. By definition there exists nodes $v_1 \in [s]$ and $v' \in [v]$ and an edge $v_1 \xrightarrow{a} v'$ and, by induction, $w_1 \in L_{v1}$. This implies $w_1 \in L_{v'}$. Now we use the fact that $\approx$ is a refinement, to conclude that $w \in L_v$.

Example 3.2 Figure 3.4a illustrates a graph data DB and Fig. 3.4b its 1-Index I. Considering the query $q = t.a\ x$, its evaluation follows the two paths $t.a$ in I (rather than the 5 in DB) and unions their extents: $\{7, 13\} \cup \{8, 10, 12\}$.

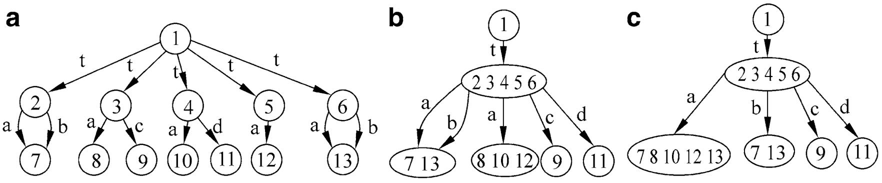


Fig. 3.4 A data graph(**a**), its 1-Index(**b**), and its Strong DataGuides(**c**)

The Size of a 1-Index

The storage of a 1-Index consists of the graph I and the sum of all extents. Query performance is dictated by the former, which we discuss here. On the experimental side, we computed I(DB) for a variety of databases, obtaining results which show that in common scenarios I is significantly smaller than DB. On the theoretical side, we identified two parameters which alone control the size of I. These are (1) the number of distinct labels in DB and (2) the longest simple path.

3.2.3 F&B-Index

In this subsection, we will discuss an F&B-Index, which is proposed in [KBNK02].

3.2.3.1 Foundation

The Labeled Graph Data Model

We model XML or other semistructure data as a directed, node-labeled tree with an extra set of special edges called *idref* edges. More formally, consider a directed graph $G = (V_G, E_T, E_{\mathrm{Ref}}, root, \Sigma_{G,}\ nodelabel, oid, value)$. V_G is the node set. E_T denotes the set of tree edges. The graph induced by E_T on V_G denotes the underlying spanning tree. Each edge in E_T indicates an object–subobject or object–value relationship. When we talk about parent–child and ancestor–descendant relationships, we refer to the tree edges. E_{Ref} is the set of idref edges each of which indicates an idref relationship. *Simple* nodes in V_G have no outgoing edges and are given a value via the *value* function. Each node in V_G is labeled with a string literal from Σ_G via the *nodelabel* function and with a unique identifier via the *oid* function, with simple objects given the distinguished label, VALUE. There is a single root element with the distinguished label, ROOT.

Example 3.3 Figure 3.5 shows a portion of a hypothetical *metroguide*, represented as a data graph. The solid edges represent the tree edges. The numeric identifiers in nodes represent oid’s. Non-tree edges (shown dashed) may be implemented with the id/idref construct or XLink syntax. The nodes labeled feature and star are attributes of their parent elements and indicate, respectively, whether a museum has a featured

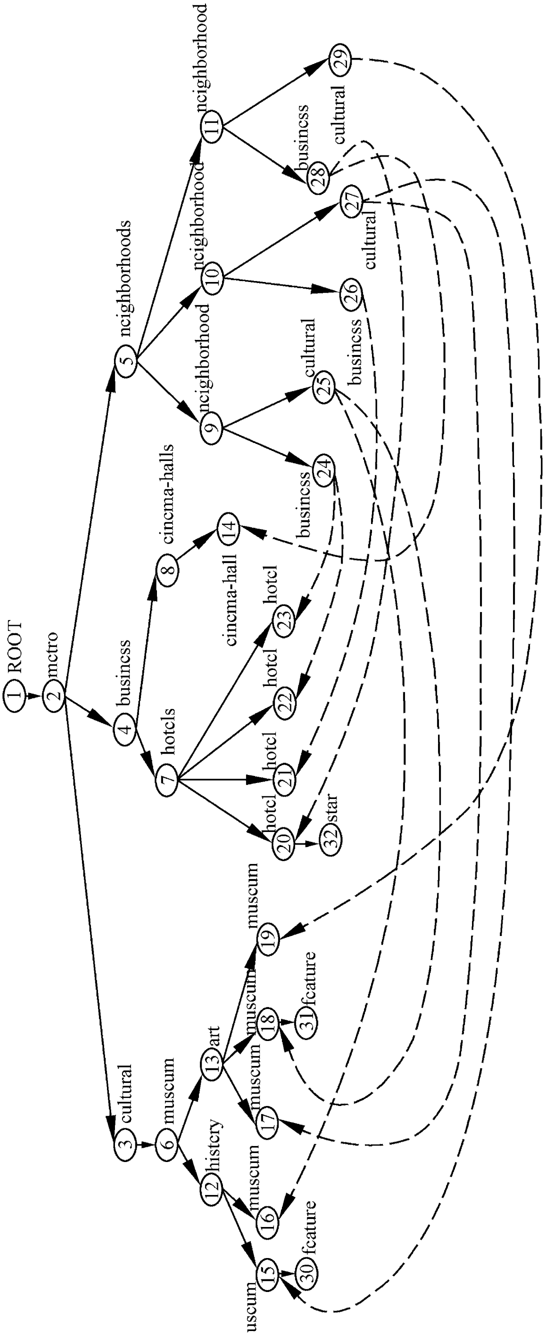


Fig. 3.5 An example graph-structured database

exhibit and whether a hotel is starred. This guide could be a large XML document, the output of publishing a relational database, or the result of decomposing an XML document into relations for the purpose of storage and querying. Attributes like *name* and *address* are suppressed from the data graph.

Branching Path Expression

A label path is a sequence of labels $l_1, \ldots, l_p$ $(p \geq 1)$, separated by forward separators /, \\, $\Leftarrow$. A node path in G is a sequence of nodes, $n_1, \ldots, n_p$, again separated by /, //, $\Rightarrow$ or \, \\, $\Leftarrow$ such that, for $1 \leq i \leq p-1$, if n_i and n_{i+1} are separated by a:

1. /, then n_i is the parent of n_{i+1}
2. //, then n_i is an ancestor of n_{i+1}
3. $\Rightarrow$, then n_i points to n_{i+1} through an idref edge
4. \, then n_i is a child of n_{i+1}
5. \\, then n_i is a descendant of n_{i+1}
6. $\Leftarrow$, then n_i is pointed to by n_{i+1} through an idref edge

A node path $n_1, \ldots, n_p$ matches a label path $l_1, \ldots, l_p$ if the corresponding separators are the same and label$(n_i) = l_i$, for $1 \leq i \leq p$. Label paths and node paths where the separators are restricted to be the forward separators are called forward label paths and forward node paths, respectively. We can similarly define backward label paths and backward node paths. Label paths that involve both forward-and-backward separators are called mixed paths.

Example 3.4 In Fig. 3.5, the path ROOT/metro/neighborhoods/neighborhood/ business $\Rightarrow$ hotel is a forward label path, and the node path 1/2/5/9/24 $\Rightarrow$ 23 matches it. The label path ROOT/metro/business/hotels/hotel $\Leftarrow$ business\neighborhood is a mixed label path and the node path 1/2/4/7/23 $\Leftarrow$ 24\9 matches it.

We define branching path expressions by the following grammar sketch:

bpathexpr → fwdlabelpath [orexpr] fwdsep bpathexpr | fwdlabelpath

orexpr → andexpr 'or' orexpr | andexpr

andexpr → bpathexpr2 'and' andexpr | notbpathexpr2 'and' andexpr
  | bpathexpr2 | notbpathexpr2

notbpathexpr2 → 'not' bpathexpr2

bpathexpr2 → labelpath2 [orexpr] bpathexpr2 | labelpath2

labelpath2 → fwdsep labelpath | backsep labelpath

fwdlabelpath → forward label paths

labelpath → label paths

fwdsep → / | // | $\Rightarrow$

backsep → \ | \\ | $\Leftarrow$

3.2.3.2 Index Graphs

In this section we are concerned with index graphs. An index graph for data graph G is a graph $I(G)$ where we associate an extent with each node in I. If A is a node in the index graph $I(G)$, then $\text{ext}^I(a)$, the extent of A, is a subset of V_G. We add the constraint that the extents of two index nodes should never overlap. The index graph result of executing a branching path expression P on $I(G)$ is the union of the extents of the index nodes that result from evaluating P on $I(G)$. An index graph $I(G)$ covers a branching path expression query P if the index result of P is accurate, that is, it is the same as the result on G.

Bisimilarity

A symmetric, binary relation $\approx$ on V_G is called a bisimulation if, for any two data nodes u and v with $u \approx v$, we have that (a) u and v have the same label, (b) if par_u is the parent of u and par_v is the parent of v, then $\text{par}_u \approx \text{par}_v$, and (c) if u' points to u through an idref edge, then there is a v' that points to v through an idref such that $u' \approx v'$, and vice versa. Two nodes u and v in G are said to be bisimilar, denoted by $u \approx^{\text{b}} v$, if there is some bisimulation $\approx$ such that $u \approx v$.

3.2.3.3 Properties of the Forward-and-Backward Index

Let us consider the following process for an edge-labeled data graph (a similar process can be applied to node-labeled graphs too). While the discussion below does not distinguish between tree and idref edges, the definitions and properties we talk about can be easily tweaked to accommodate the same:

1. For every (edge) label l, add a new label l^{-1}.
2. For every edge e-labeled l from node u to node v, add an (inverse) edge e^{-1} with label l^{-1} from v to u.
3. Compute the 1-Index (or DataGuides) on this modified graph.

The above is a structural summary that captures information about paths both entering and leaving nodes in the data graph. With the 1-Index used in step 3 above, we obtain a partition of the data nodes which can be used to define an index graph. Let us call this the forward-and-backward index (F&B-Index).

In this section, we prove some new theorems about the F&B-Index that are important in the context of families of covering indexes. To do so, we first give an alternative definition of the F&B-Index based on the notion of the stability of one set of graph nodes with respect to another. For a set of nodes, A, let Succ (A) denote the set of successors of the nodes in A, that is, the set $\{v \mid \text{there is a node } u \in A$ with and edge in G from u to $v\}$. We can define the predecessor set of A, Pred (A) analogously.

Definition 3.10 Given two sets of data graph nodes A and B, A is said to be succ-stable with respect to B if either A is a subset of Succ(B) or A and Succ(B) are disjoint.

A succ-stable partition has the property that if we build an index graph from it, then whenever there is an edge from index node A to index node B, there is an edge from every data node in ext(A) to some node in ext(B). Let us call a label grouping a partition of the data nodes that corresponds to their node labels (i.e., two nodes are in the same equivalence class if they have the same label).

Now consider the following procedure over data graph G. We work with a current partition of the data nodes which is initialized to the label grouping:

1. Reverse all edges in G.
2. Compute the bisimilarity partition.
3. Set the current partition to what is output by the previous step.
4. Reverse edges in G again, obtaining the original G.
5. Compute the bisimilarity partition (again initializing the computation with the current partition).
6. Set the current partition to what is output by the previous step.
7. Repeat the above steps till the current partition does not change.

It is not hard to see that the F&B-Index is the index graph obtained by using this final partition.

Theorem 3.3 *The F&B-Index over a data graph G covers all branching path expressions over G.*

Theorem 3.4 *For data graph G, any index graph that covers all branching path expressions over G must be a refinement of the F&B-Index.*

Theorem 3.5 *For data graph G, the F&B-Index is the smallest index graph that covers all branching path expressions over G.*

As a result, whenever two nodes are not in the same extent of the F&B-Index, there is some branching path query that distinguishes between the two.

3.2.3.4 Covering Index Definition Scheme

The F&B-Index is often big. This size problem with the F&B-Index leads us to look for a way to cut down the size of this index. Since it is the smallest index that handles all branching path expressions, the only way out is to compromise on the class of queries to be covered. This motivates the index definition scheme we propose in the section. The index definition scheme is designed toward eliminating branching path expressions that are deemed less important, so that we arrive at an index that is much smaller and can handle the remaining branching path expressions more efficiently. We use four different approaches toward this goal based on the following intuitions:

1. There are many tags in data that are of lesser interest and so need not be indexed.
2. Branching path expressions (in the manner defined by us) give more importance to tree edges over idref edges. In particular, // matches only tree edges and there is no equivalent for idref edges. It may be desirable to reflect this in the index.
3. Not all structure is interesting. Our intuition is that queries on long paths are rare. Instead, short paths are more common. Thus, it may be useful to exploit local similarity to cut down the index size.
4. Restricting the *tree-depth* of the branching path expressions for which the index is accurate might help. We will explain this in more detail later.

Tags to be Indexed

It is often the case that there are tags in the data that are never queried using branching path expressions and thus need not be indexed. We do so by altering the data graph so that all nodes that have tags that are not to be indexed are labeled with a unique label, other. In addition, if a node labeled other does not appear in the tree path to any node that is being indexed, it can be assumed to be absent from the data graph for purposes of indexing. This simple technique can have a lot of effect in practice.

Example 3.5 For example, in the XMark data, there are text tags such as bold and emph that appear in the description of categories and items that are being auctioned. It may be worthwhile to build an index that ignores these tags. The F&B-Index on the tree edges of the XMark file of size 100 MB (which has about 1.43 million nodes) has about 436,000 nodes. But on ignoring the text nodes, the number of nodes in the index drops to about 18,000.

Tree Edges vs. Idref Edges

The XPath standard [CD99] for path expressions gives a higher priority to tree edges. In particular, // matches only tree edges, and there is no equivalent ancestor operation for idref edges. On the other hand, all occurrences of idref *s* in path expressions have to be explicit. This is also reflected in the way we have defined our branching path expressions. The effect on the index size of idref *s* can be tremendous.

Example 3.6 For example, on the XMark data of size 100 MB, the F&B-Index on just the tree edges has 18,000 nodes (ignoring text nodes), while the size is about 1.35 million when all idref *s* are incorporated (again ignoring text nodes). Thus, it is desirable to have some way of giving priority to tree edges. We specify the set of idref edges to be indexed as part of the index definition. We do so by specifying the source and target labels of the idref edges we wish to retain.

Figure 3.6 shows the F&B-Index constructed on only the tree edges of the data graph in Fig. 3.5 (again we only show extents that have more than one data node).

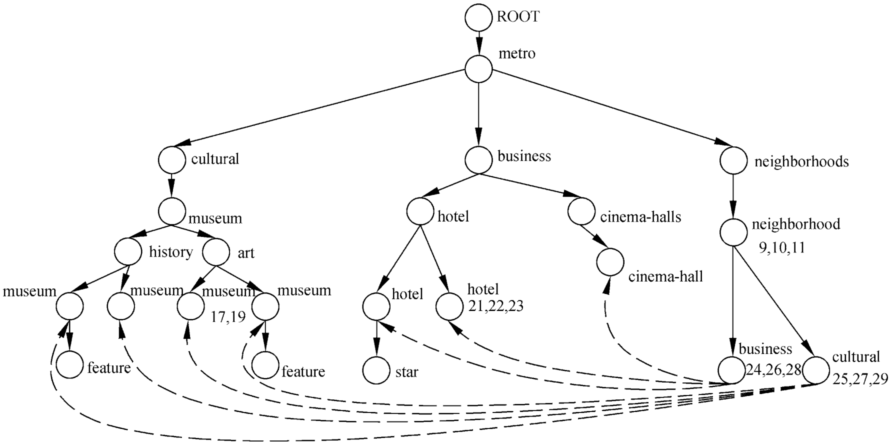


Fig. 3.6 F&B-Index on tree edges

As we can see, all three nodes labeled neighborhood are merged together in this index, showing that they cannot be distinguished by any branching path expression query on the tree edges alone. However, this index cannot necessarily be used to cover a branching path expression query that refers to an idref edge.

Exploiting Local Similarity

An important approach in controlling index sizes lies in exploiting local similarity. Our intuition is that most queries refer to short paths and seldom ask for long paths. As a result, it may not be desirable to split the index partition along long paths.

For example, in the data graph in Fig. 3.5, it may not be desirable to split nodes labeled neighborhood based on whether they contain a museum that has a featured exhibit. This can be achieved by looking at paths of length up to 2 (here, length refers to the number of edges). This process is reminiscent of summary tables for OLAP workloads, where picking a larger number of attributes to aggregate yields a smaller summary that is accurate for queries covered by it while picking a smaller number of attributes to aggregate makes the summary more generic.

We have the notion of k-bisimilarity (defined in [Mil80]) which groups nodes based on paths of length up to k. We reproduce it below.

Definition 3.11 ($\approx^k$ ***k*-bisimilarity**): This is defined inductively:

1. For any two nodes, u and v, $u \approx^0 v$ iff u and v have the same label.
2. Node $u \approx^k v$ iff $u \approx^{k-1} v$, $\text{par}_u \approx^{k-1} \text{par}_v$ where par_u and par_v are, respectively, the parents of u and v, and for every u' that points to u through an idref edge, there is a v' that points to v through an idref edge such that $u' \approx^{k-1} v'$, and vice versa.

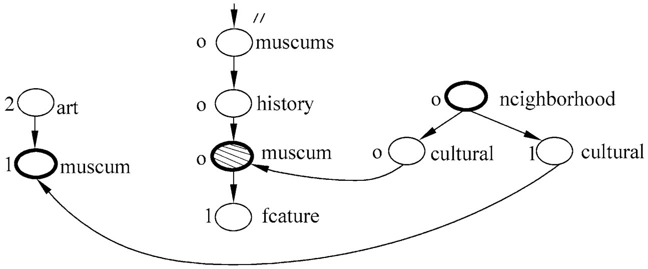


Fig. 3.7 Example for tree-depth

By a simple induction, we can see that this definition ensures the weaker condition it sets out to achieve. Note that k-bisimilarity defines an equivalence relation on the nodes of a graph. We call this the k-bisimulation. This partition is used in [KSBG02] for incoming path queries, to define an index graph, the $A(k)$-index, where k is a parameter. Thus, an $A(2)$-index is accurate for the query //neighborhood/cultural ⇒ museum, but not necessarily for the query //neighborhood/cultural ⇒ museum/featured.

In our context, it should be possible to specify, say, that in the forward direction, we only want local similarity for paths of length at most 1 and in the backward direction. We want *global* similarity. This way, we get a covering index where, for example, the condition *museum with a featured exhibit* or the condition *hotel that is a star hotel* can be imposed since these only involve paths of length 1, whereas the condition *neighborhood with a museum that has a featured exhibit* cannot be imposed since it involves a longer path. We argue that by thus restricting the queryable path length, we can still index a large and interesting subset of branching path expression queries.

Restricting Tree-Depth

We defined the F&B-Index as a transitive closure of the following sequence of computations:

1. Reverse all edges in G.
2. Compute the bisimilarity partition (with the current partition as the initialization).
3. Reverse edges in G again, obtaining the original G.
4. Compute the bisimilarity partition (again initializing the computation with the current partition).

We now define the notion of tree-depth of a node in a query. Consider the query//museums/history/museum[/featured and ⇐ cultural\neighborhood [/cultural ⇒ museum [\art]]] that asks for history museums that have a featured exhibit and also have an art museum in the same neighborhood. The query is shown in Fig. 3.7 in graph format. The numbers to the side indicate the tree-depths of the nodes. The idea is that all nodes on the primary path and having a path to some node in the primary path have tree-depth 0. All nodes that do not have tree-depth 0 and have a path from some node in the primary path have tree-depth 1, nodes that do not have tree-depth 1 and have a path to some node of tree-depth 1 have tree-depth 2, nodes that do not have tree-depth 2 and have a path from some node of tree-depth 2 have tree-depth 3, and so on. Intuitively, odd tree-depths correspond to outgoing path conditions, while even tree-depths correspond to incoming path conditions. Nodes that have an edge to or from a node of higher tree-depth are called branching points. The nodes in the example query graph that are branching points are indicated in bold. The return node is shaded and, in this case, also happens to be a branching point. Note that tree-depth is different from the nesting level of a node in the query text. In particular, the neighborhood node has tree-depth 0 although it is nested in the query text. In general, the same query can be written in more than one way. Tree-depths of nodes do not depend on how the query is written (whereas nesting levels do). The tree-depth of a query is the maximum tree-depth of its nodes.

3.2.3.5 Putting It Together

An index definition consists of the following parts:

1. A set of tags to be indexed. Call this set *T*.
2. For each of the forward-and-backward directions.
3. Set of idref edges to be indexed (call them ref_{fwd} and ref_{back})
4. A parameter *k* indicating the extent of local similarity desired (call them k_{fwd} and k_{back}).
5. The number of iterations in the F&B-Index computation to be performed. Call this td, the tree-depth.

The parameters k_{fwd}, k_{back}, and td can be set to be ∞, referring to a transitive closure computation. The index obtained for a given index definition *S* is called the BPCI(S) (for branching path expression covering index). The algorithm for computing the BPCI(S) is shown below.

Algorithm 3.2: Algorithm for Computing the BPCI(S)
Procedure compute_ partition(G,S)

```
G → data graph, S → index definition
begin
Convert all tags in G not in T into special tag other
Remove any occurrence of a node labeled other if it is not on some tree path from the root to any node with a label that is to be indexed
Let P be a list of sets of nodes    //representing a partition of the nodes of G
P ← label –grouping of G
for i = 1 to td do                  //forward direction
    Retain idref edges in $\text{ref}_{\text{fwd}}$
    Reverse all edges in G
```

```
8.     Compute the $k_{fwd}$-bisimulation on G initializing the computation with P
9.     P ← partition of nodes of G corresponding to the above $k_{fwd}$-bisimulation
    //backward direction
10. Restore G
11. Retain idref edges in $ref_{back}$
12. Compute the $k_{back}$-bisimulation on G initializing the computation with P
13. P ← partition of nodes of G corresponding to the above $k_{back}$-bisimulation
end
```

```
Procedure compute_ index(G, S)
begin
1.  compute partition(G,S)
2.  foreach equiv. class in P do
3.      create an index node I
4.      ext[I] = data nodes in the equiv. class
5.  foreach edge from u to v in G do
6.      I[u] = index node containing u
7.      I[v] = index node containing v
8.  if there is no edge from I[u] to I[v] then
9.      add an edge from I[u] to I[v]
end
```

The subset of branching path expressions for which an arbitrary covering index is accurate depends, of course, on the index definition. The following are some example index definitions and the indexes they generate:

1. The F&B-Index can be obtained by indexing all tags and all idref edges, with $k_{\text{fwd}} = k_{\text{back}} = \text{td} = \infty$.
2. The 1-Index can be generated by indexing all tags and all idref edges, with $k_{\text{fwd}} = 0$, $k_{\text{back}} = \infty$, and $\text{td} = 0$.

Index Selection

We now discuss the issue of how to arrive at a reasonable index definition that covers a set of queries. Note that this is analogous to the problem of choosing an appropriate covering index for a given query workload. Consider the following queries on the data in Fig. 3.5:

1. Hotels that have a museum in the same neighborhood. This could be written as a branching path expression query in the following way: //neighborhood[/cultural ⇒ museum]/business ⇒ hotel

   Note that the primary path length is 3 and the tree-depth is 1.
2. Starred hotels: //hotel[/star]. The primary path length is 1, and the tree-depth is 1.
3. Neighborhoods with an art museum: //neighborhood[/cultural ⇒ museum[\art]].

   Here, the primary path length is 1 and the tree-depth is 2.

The following constraints hold for any index that covers the above queries. First of all, the tags involved in this query must be indexed. Since the maximum tree-depth is 2, the index tree-depth must be at least 2. Since the maximum path length for path conditions is 2 (//neighborhood[/cultural ⇒ museum]), $k_{\text{fwd}} \geq 2$. Similarly, the constraint on k_{back} is that $k_{\text{back}} \geq 2$. Given these constraints, one straightforward thing to do is to pick the minimum values needed to cover the queries. Call this index I_{min}. However, there is a tradeoff—by picking larger values for the parameters, we get a more generic index that can potentially cover more queries, although the performance of this index for these queries may be worse than that of I_{min}. This example serves to illustrate the issues involved in choosing an index definition to cover a set of branching path expression queries. It is possible to give simple heuristics (e.g., output I_{min}) to create a covering index. However, a good choice of index definition depends heavily on the data and the queries.

Testing if an Index Covers a Query

We now discuss how to test whether a given index covers a branching path expression query. Intuitively, the conditions that must be satisfied are:

1. The tags and idref s should match—all tags and idref s referenced in the query must be indexed.
2. The tree-depth should match—the tree-depth of the index should be at least as large as the tree-depth of the query.
3. If the index uses local similarity, then all relevant path lengths must be bounded by the extent of local similarity captured in the index.

In more detail, let Q_G be the query graph. A path in Q_G has tree-depth i if all nodes in the path, with the possible exception of either end, have tree-depth i (not all paths necessarily have a tree-depth). The procedure shown in Algorithm 3.3 tests whether a given index covers a given branching path expression query. This procedure can be implemented using a depth-first search of the query graph and takes time linear in the query size.

Algorithm 3.3: Checking if BPCI(S) Covers a Query
Procedure cover(J, Q)

```
J ← index
Q ← branching path express
begin
    convert Q into a query graph $Q_G$
    check if all tags in Q are indexed in J
    check if the tree-depth of Q $\leq$ td, the tree-depth of J
    check if all paths in $Q_G$ with even tree-depth have length $\leq k_{\text{back}}$;
  if $k_{\text{back}} < \infty$, no separator in these paths should be //
    check if all paths in $Q_G$ with odd tree-depth have length $\leq k_{\text{fwd}}$;
  if $k_{\text{fwd}} < \infty$, no separator in these paths should be //
end
```

3.3 Index Based on XML Sequencing

3.3.1 PRIX: Indexing and Querying XML Using Prüfer Sequences

3.3.1.1 Overview of PRIX Approach

In this subsection, we will present the Prüfer's method [RM03] that constructs a one-to-one correspondence between trees and sequences and describe how Prüfer's sequences are used for indexing XML data and processing twig queries in the PRIX system.

Prüfer Sequences for Labeled Trees

Prüfer (1918) proposed a method that constructed a one-to-one correspondence between a labeled tree and a sequence by removing nodes from the tree one at a time [Pru18]. The algorithm to construct a sequence from tree T_n with n nodes labeled from 1 to n works as follows. From T_n, delete a leaf with the smallest label to form a smaller tree T_{n-1}. Let a_1 denote the label of the node that was the parent of the deleted node. Repeat this process on T_{n-1} to determine a_2 (the parent of the next node to be deleted) and continue until only two nodes joined by an edge are left. The sequence $(a_1, a_2, \ldots, a_{n-2})$ is called the Prüfer sequence of tree T_n. From the sequence $(a_1, a_2, \ldots, a_{n-2})$, the original tree T_n can be reconstructed.

The length of the Prüfer sequence of tree T_n is $n-2$. In our PRIX approach, however, we construct a Prüfer sequence of length $n-1$ for T_n by continuing the deletion of nodes till only one node is left. (The one-to-one correspondence is still preserved.)

Indexing by Transforming XML Documents into Prüfer Sequences

In the discussions to follow, each XML document is represented by a labeled tree such that each node is associated with its element tag and a number.

Example 3.7 For example, in Fig. 3.8, the root element of the XML document has (A, 15) as its tag number pair. Any numbering scheme can be used to label an XML document tree as long as it associates each node in the tree with a unique number between one and the total number of nodes. This guarantees a one-to-one mapping between the tree and the sequence. In our PRIX system, without loss of generality, we have chosen to use postorder to uniquely number tree nodes and will continue further discussions based on the postorder numbering scheme.

With tree nodes labeled with unique postorder numbers, a Prüfer sequence can be constructed for a given XML document using the node removal method described before. This sequence consists entirely of postorder numbers and is called NPS (numbered Prüfer sequence). If each number in an NPS is replaced by its

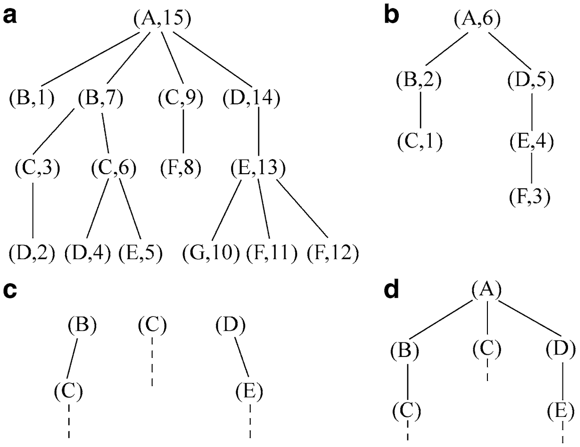


Fig. 3.8 XML document tree and query twig. (**a**) Tree *T*, (**b**) tree *Q*, (**c**) disconnected graph, (**d**) connected graph

corresponding tag, a new sequence that consists of XML tags can be constructed. We call this sequence LPS (labeled Prüfer sequence). The set of NPS's are stored in the database together with their unique document identifiers.

Example 3.8 In Fig. 3.8a, tree T has LPS(T) = $A\ C\ B\ C\ C\ B\ A\ C\ A\ E\ E\ E\ D\ A$, and NPS($T$) = 15 3 7 6 6 7 15 9 15 13 13 13 14 15.

Processing Twig Queries by Prüfer Sequences

A query twig is transformed into its Prüfer sequence like XML documents. Non-matches are filtered out by subsequence matching on the indexed sequences, and twig matches are then found by applying a series of refinement strategies. These filtering and refinement phases will be described in the next section. Figure 3.9 shows an architectural overview of the indexing and query processing units in PRIX as described before. With this high-level overview of our system, we shall now move on to explain the process of finding twig matches.

3.3.1.2 Finding Twig Matches

To simplify our presentation of concepts in this section, we shall use the notations listed in Table 3.1. Formally the problem of finding twig matches can be stated as follows: Given a collection of XML documents Δ and a query twig Q, report all the occurrences of twig Q in Δ. We restrict to handling twig Q with equality predicates only.

We will initially deal with the problem of finding all occurrences of twig Q without wildcards "//" and "*". Later in the next section, we explain how query twigs with wildcards can be processed. In addition, we will first address the problem

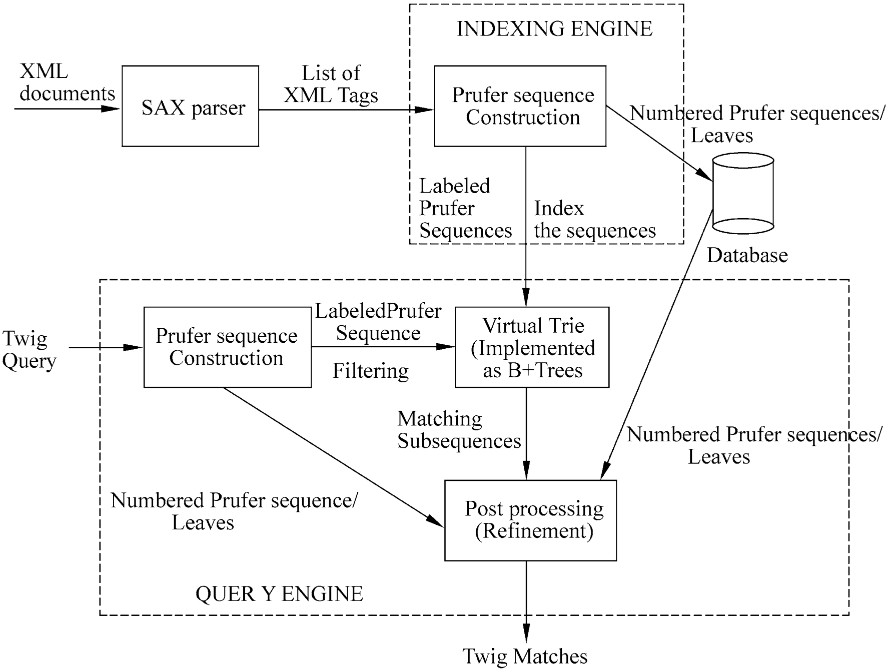

INDEXING ENGINE
XML documents
SAX parser
List of XML Tags
Prufer sequence Construction
Numbered Prufer sequences/ Leaves
Labeled Prufer Sequences
Index the sequences
Database
Twig Query
Prufer sequence Construction
LabeledPrufer Sequence
Filtering
Virtual Trie (Implemented as B+Trees
Matching Subsequences
Numbered Prufer sequences/ Leaves
Numbered Prufer sequence/ Leaves
Post processing (Refinement)
QUER Y ENGINE
Twig Matches

Fig. 3.9 Architectural overview of PRIX

Table 3.1 Notations used

Symbol	Description
Q	Query twig
Δ	A collection of XML documents
Γ	A set of labeled Prüfer sequences of Δ
Θ	A set of subsequences in Γ that are identical
LPS(T)	Labeled Prüfer sequence of tree T
NPS(T)	Numbered Prüfer sequence of tree T

of finding ordered twig matches. We will explain how unordered twig matches can be found.

Finding twig matches in the PRIX system involves a series of filtering and refinement phases, namely, (1) filtering by subsequence matching, (2) refinement by connectedness, (3) refinement by structure, and (4) refinement by leaf nodes.

Filtering by Subsequence Matching

The filtering phase involves subsequence matching. The classical definition of a subsequence is stated below.

Definition 3.12 (Subsequence) A subsequence is any string that can be obtained by deleting zero or more symbols from a given string.

In this phase, given a query twig Q, we find all the subsequences In Γ (the set of LPS's) that match LPS(Q). We shall discuss the significance of subsequence matching using the following lemma and theorem.

Lemma 3.1 *Given a tree T with n nodes, numbered from* 1 *to n in postorder, the node deleted the ith time during Prüfer sequence construction is the node numbered i.*

As a result, if a and b are two nodes of a tree such that a has a smaller postorder number than b, then node a is deleted before node b during Prüfer sequence construction. Based on Lemma 3.1, we can state the following theorem.

Theorem 3.6 *If tree Q is a subgraph of tree T, then LPS(Q) is a subsequence of LPS(T).*

From Theorem 3.6, it is evident that by finding every subsequence in Γ that matches LPS(Q), we are guaranteed to have no false dismissals.

Example 3.9 Consider trees T and Q in Fig. 3.8a, b. T has LPS(T) = $A\ C\ B\ C\ C\ B\ A\ C\ A\ E\ E\ E\ D\ A$ and NPS(T)=15 3 7 6 6 7 15 9 15 13 13 13 14 15. Q has LPS(Q) = $B\ A\ E\ D\ A$ and NPS(Q) = 2 6 4 5 6. Q is a (labeled) subgraph of T, and LPS(Q) matches a subsequence S of LPS(T) at positions (6, 7, 11, 13, 14). The postorder number sequence of subsequence is 7 15 13 14 15. Note that there may be more than one subsequence in LPS(T) that matches LPS(Q).

Refinement by Connectedness

The subsequences matched during the filtering phase are further examined for the property of connectedness. This is because, only for some of the subsequences, all the labels in the subsequence correspond to nodes that are connected (representing a tree) in the tree. Formally we state a necessary condition for any subsequence S to satisfy the connectedness property.

Theorem 3.7 *Given a tree T, let N_T be the NPS of T. Let S be a subsequence of LPS(T) and let N be the postorder number sequence of S. Then the tree nodes in T corresponding to the labels of S are connected (representing a tree) only if for every element of N, that is, N_i, if $N_T \neq \max(N_1, N_2, \ldots, N_{|N|})$ and $\nexists\ (j > i)$ s.t. $N_j = N_i$ then $N_i = N_T[N_i]$.*

The intuition for the above theorem is as follows. Let i be the index of the last occurrence of a postorder number n in an NPS. This last occurrence is a result of deletion of the last child of n during Prüfer sequence construction. Hence, the next child to be deleted (based on Lemma 3.1) is the node n itself. Hence, the number at the $(i+1)$th index in the NPS, say m, is the postorder number of the parent of node n. Thus, n followed by m indicates that there is an edge between node m and node n.

Example 3.10 Consider two subsequences S_A and S_B of LPS(T) where T is the tree in Fig. 3.8a. Let S_A be $C\ B\ C\ E\ D$ whose postorder number sequence N_A is 3 7 9 13 14. Let S_B be $C\ B\ A\ C\ A\ E\ D\ A$ whose postorder number sequence N_B is 3 7 15 9 15 13 14 15. Let N_T be the NPS of T. Then N_T is 15 3 7 6 6 7 15 9 15 13 13 13 14 15. The nodes represented by labels of S_A form a disconnected graph as shown in Fig. 3.8c. In this case, $\max(N_{A1}, N_{A2}, \ldots, N_{A5}) = 14$. The last occurrence of postorder number 7 in N_A is at the 2nd position since there is no index $j > 2$ such that $N_{Aj} = 7$. However, N_{A2} is not followed by N_T, that is, $N_{A3} \neq 15$. Hence, the necessary condition of Theorem 3.7 is not satisfied. The nodes represented by elements of S_B represent a tree as shown in Fig. 3.8c because the necessary condition of Theorem 3.7 is satisfied. We shall refer to sequences that satisfy Theorem 3.7 as tree sequences.

Refinement by Twig Structure

The tree sequences obtained in the previous refinement phase are further refined based on the query twig structure. In this phase we would like to determine if the structure of the tree represented by a tree sequence matches the query twig structure.

Gaps. Before we delve into details of refinement by structure, we shall first introduce the notion of gap between two tree nodes and gap consistency and frequency consistency between two tree sequences.

Definition 3.13 The gap between two nodes a and b in a tree is defined as the difference between the postorder numbers of the nodes a and b.

The gap between tree nodes can be computed using the NPS of the tree.

Definition 3.14 Tree sequence A is said to be gap consistent with respect to tree sequence B if:

1. A and B have the same length n.
2. For every pair of adjacent elements in A and the corresponding adjacent elements in B, their gaps, g_A and g_B, have the same sign, and if $|g_A| > 0$, then $|g_A| \leq |g_B|$, else $|g_A| = |g_B| = 0$.

Note that gap consistency is not a symmetric relation.

Definition 3.15 Tree sequences A and B are frequency consistent if A and B have the same length n, let NA and NB be the postorder number sequences of A and B, respectively. Let nAi and nBi be the ith element in NA and NB, respectively. For every i from 1 to n, nAi occurs k times in NA at positions $\{p1, p2, \ldots, pk\}$, iff nBi occurs k times in NB at positions $\{p1, p2, \ldots, pk\}$.

Note that frequency consistency is an equivalence relation. It should be noted that the LPS of a tree contains only the non-leaf node labels. Thus, in addition to the

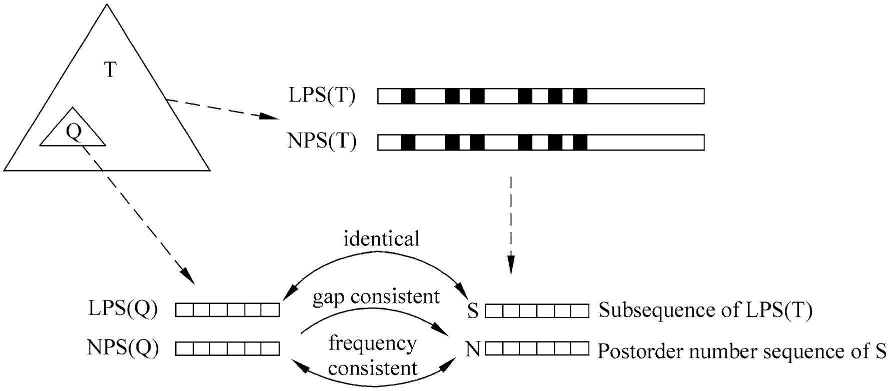


Fig. 3.10 Data and query sequences

LPS and NPS, the label and postorder number of every leaf node should be stored in the database. Since the LPS of a tree contains only non-leaf node labels, filtering by subsequence matching followed by refinement by connectedness and structure can only find twig matches in the data tree whose tree structure is the same as the query tree and whose non-leaf node labels match the non-leaf node labels of the query twig. Let us call such matches as partial twig matches. To find a complete twig match, the leaf node labels of a partially matched twig in the data should be matched with the leaf node labels of the query twig.

We now state a necessary and sufficient condition for a partial twig match.

Theorem 3.8 *Tree Q has a partial twig match in tree T iff*:

1. *LPS*(*Q*) *matches a subsequence S of LPS*(*T*) *such that S is a tree sequence.*
2. *LPS*(*Q*) *is gap consistent andfrequency consistent with subsequence S.*

The different relationships between the data and query sequences as described in this section are illustrated in Fig. 3.10. Consider the tree *T* (XML document) and its subgraph tree *Q*(query twig) in the figure. The dark regions in LPS(*T*) and NPS(*T*) correspond to the deletion of nodes in *T* during Prüfer sequence construction that are also in *Q*(except root of *Q*).The dark regions in LPS(*T*) and NPS(*T*) form sequences *N* and *S*, respectively. From the lemmas and theorems described before, we can conclude the following: LPS(*Q*) and *S* are identical, NPS(*Q*) is gap consistent with *N*, and NPS(*Q*) and *N* are frequency consistent.

Refinement by Matching Leaf Nodes

In the final refinement phase, the leaf node labels of the query twig are tested with the leaf node labels of partially matched twigs in the data to find complete twig matches.

Example 3.11 The leaf nodes of tree T in Fig. 3.8, that is, (D, 2), (D, 4), (E, 5), (G, 10), (F, 11), and (F, 12) are stored in the database. Let tree Q (Fig. 3.8b) be the query twig. LPS(Q) matches a subsequence $S = B\ A\ E\ D\ A$ in LPS(T) at positions $P = (3, 7, 11, 13, 14)$. The postorder number sequence of S is $N = 7\ 15\ 13\ 14\ 15$. LPS(Q) is gap consistent and frequency consistent with S. We can match leaf (F, 3) in Q as follows. Since the leaf has postorder number 3, its parent node matches the node numbered 13 (i.e., the 3rd element of N) in the data tree. Also because this node numbered 13 occurs at the 11th position (3rd element in P) in LPS(T), it may have a leaf (F, 11). And indeed, we have (F, 11) in the leaf node list of T. Similarly we can match the leaf (C, 1) of Q. The parent of (C, 1) in Q matches node 7 (1st element in N) at position 3 in NPS(T). Hence, the child of node 7 in T, that is, node 3 matches leaf (C, 1), except that the labels may not match (partial twig match). Since there are no nodes with number 3 in the leaf list of T, we search LPS(T) and NPS(T) to find (C,3) in T. And indeed, we have this pair at the 2nd position in LPS/NPS of T.

However, this refinement phase can be eliminated by special treatment of leaf nodes in the query twig and the data trees. The key idea is to make the leaf nodes of the query twig and the data trees appear in their LPS's, so that all the nodes of the query twig are examined during subsequence matching and refinement by connectedness and structure phases.

Processing Wildcards

We shall explain the processing of wildcards "//" and "*" with the following example.

Example 3.12 Let us find the query pattern $Q = //A//C/D$ in tree T (in Fig. 3.8a). Q is transformed to its Prüfer sequences by ignoring the wildcards. As a result, LPS(Q) $= C\ A$, and NPS(Q) $= 2\ 3$. The wildcard at the beginning of the query is handled by our current method as it allows finding occurrences of a query tree anywhere in the data tree. To process the wildcard in the middle of the query, we do a simple modification to the refinement-by-connectedness phase. LPS(Q) matches a subsequence $S = C\ A$ at positions (2, 7) in LPS(T). The postorder number sequence of S is $N = 3\ 15$. Based on Theorem 3.7, this subsequence would be discarded as the last occurrence of 3 in N is not followed by 7 (parent of node numbered 3 in T).To avoid this, we check instead if the last occurrence of node 3 in N can lead to node 15 (15 follows 3 in N) by following a series of edges in T. Recall that the ith element in an NPS is the postorder number of the parent of node i in a tree (Lemma 3.1). Let $n_0 = 3$ and let N_T be NPS(T). We recursively check if $n_1(N_T[n_0])$ equals 15, then if $n_2(N_T[n_1])$ equals 15 and so on until for some i, $n_{n+1}(N_T[n_i])$ equals 15. In the above example, we find a match at $i = 2$. For processing wildcard "*", we simply test whether the match is found at $i = 2$. Thus, all the subsequences that pass the above test will move to the next phase.

3.3.1.3 Implementation Issues in the PRIX System

Given the theoretical background in section before, we shall move on to explain the implementation aspects of the PRIX system.

Building Prüfer Sequences

In the PRIX system, Prüfer sequences are constructed for XML document trees (with nodes numbered in postorder) using the method described in section before. Our proposed tree-to-sequence transformation causes the nodes at the lower levels of the tree to be deleted first. This results in a bottom-up transformation of the tree. We shall show in our experiments that the bottom-up transformation is useful to process query twigs efficiently.

Indexing Sequences Using B^+-Trees

The set of labeled Prüfer sequences of the XML documents is indexed in order to support fast subsequence matching for query processing. Maintaining an in-memory index for the sequences like a trie is unsuitable, as the index size grows linearly with the total length of the sequences. In essence, we would like to build an efficient disk-based index. In fact, Prüfer sequences can be indexed using any good method for indexing strings. In the current version of our PRIX system, we index labeled Prüfer sequences using B$^+$-trees in the similar way that Wang et al. build a virtual trie using B$^+$-trees [WPFY03].

Virtual Trie. We shall briefly explain the process of indexing sequences using a virtual trie. Essentially, we provide positional representations for the nodes in the trie by labeling them with ranges. Each node in the trie is labeled with a range (LeftPos, RightPos) such that the containment property is satisfied [LM01]. Typically, the root node can be labeled with a range (1, MAX_INT). The child nodes of the root can be labeled with subranges such that these subranges are disjoint and are completely contained in (1, MAX_INT). This containment property is recursively satisfied at every non-leaf node in the trie. We can then obtain all the descendants of any given node *A* by performing a range query that finds nodes whose LeftPos falls within the (LeftPos, RightPos) range of node *A*.

In the PRIX system, for each element tag *e*, we build a B$^+$-trees that indexes the positional representation of every occurrence of element *e* in the trie using its LeftPos as the key. We call this index Trie-Symbol index. In addition, we store each document (tree) identifier in a separate B$^+$-trees and index it using the LeftPos of the node where the LPS ends in the virtual trie as the key. This index is called Docid *index*. Note that it is sufficient to store only the LPS's in the virtual trie. The suffixes of the LPS's need not be indexed at all, since all the subsequences can be found by performing range queries on the Trie-Symbol indexes to be described.

Space Complexity. The size of a trie grows linearly with the total length of the sequences stored in it. In the PRIX system, the length of a Prüfer sequence is linear in the number of nodes in the tree.

Filtering by Subsequence Matching

Let $Q_s = Q_{s1}Q_{s2} \dots Q_{sk}$ (a sequence of length k) denote the LPS of a query twig Q. The process of finding all occurrences of Q_s using the Trie-Symbol indexes is shown in Algorithm 3.4. The algorithm is invoked by FindSubsequence (Q_s, 1, 0, MAX_INT). A range query in the open interval (q_l, q_r) is performed on the T_{Qs1} (Trie-Symbol index of Q_{si}) (Line 1). For every node id r returned from the range query (Line 1), if the sequence Q_s is found, then all the documents in the closed interval [r_1, r_r] are fetched from the Docid index (Line 5). (r_l, r_r) is the positional representation of node id r. (In this case $r_l = r$) otherwise, FindSubsequence(.) is recursively invoked for the next element Q_{si+1} in the sequence using the range (r_l, r_r). In Line 3, the position of match of the ith element of Q_s (i.e., level of node in the trie) is stored in S. The solutions of the range query in Line 1 are the ids of nodes Q_{si+1} that are descendants of nodes Q_{si} in the virtual trie. In Line 4, the algorithm outputs a set of document (tree) identifiers D and a list S. S contains the positions in the LPS's of trees corresponding to tree identifiers in D where Q_s has a subsequence match.

Algorithm 3.4: Filtering Algorithm
Input: {Q_s, I, (q_l, q_r)}, Q_s is a query sequence; index i; (q_l, q_r) is a range
Output: (D, S), D is a set of document(tree) identifiers; S denotes the positions of subsequence match.

```
Procedure FindSubsequence ($Q_s$, I, ($q_l$, $q_r$))
1.  R = RangeQuery($T_{Qsi}$, ($q_l$, $q_r$));
2.  Foreach r in R do
3.     $S_i$ = Level(r);
4.     if(i = $|Q_s|$) then
5.        D = RangeQuery(DocidIndex, ($r_l$, $r_r$))
6.        Output(D, S);
7.     else FindSubsequence($Q_s$, i+1, ($r_l$, $r_r$))
8.  end
```

Optimized Subsequence Matching

In order to speed up subsequence matching further, it is desired to reduce the number of range queries to be performed by Algorithm 3.4 without causing any false dismissals. We can achieve this by pruning some nodes (r in Line 2 of Algorithm

3.4) with an additional requirement on the gap between elements corresponding two adjacent nodes in the query sequence. In this regard, we have developed an upper-bounding distance metric based on the property of Prüfer sequences.

Given a collection Δ of XML document trees and node label e in Δ, we define the distance metric on the pair (e, Δ) as follows.

Definition 3.16 MaxGap(e, Δ**)** Maximum postorder gap of a node label e is defined as the maximum of the difference between the postorder numbers of the first and the last children of the node labeled e in Δ.

The following theorem summarizes the use of the MaxGap as an upper-bounding distance metric for pruning the search space and shortening the subsequence matching process.

Theorem 3.9 *Given a query twig Q and the set* Θ *of LPSs for* Δ*, let A and B denote adjacent labels in LPS(Q) such that A occurs before B*:

1. *In case node A is a child of node B in Q, any subsequence AB in* Θ *cannot result in a twig match, if its position pair* (i, j) *is such that* $j - I > MaxGap(A, \Delta) + 1$.
2. *In case node A is an ancestor of node B in Q, any subsequence AB in* Θ *cannot result in a twig match, if its position pair* (i, j) *is such that* $j - I \geq MaxGap\ (A, \Delta)$.

It is straightforward to extend Algorithm 3.4 to incorporate the upper-bounding distance metric by computing $(S_i - S_{i-1})$ (after Line 3) and testing the appropriate condition in Theorem 3.9 using MaxGap of label Q_{i-1}. Note that the MaxGap metric can be defined at different levels of granularity. Finer-grained MaxGap values can be stored in every occurrence of a symbol in the virtual trie.

The Refinement Phases

The set of ordered pairs (D, S) returned by Algorithm 3.4 is further examined during the refinement phases. The steps for the refinement phases are shown in Algorithm 3.5. The NPS and the set of leaf nodes of D are read from the database and passed as input to this algorithm. The input subsequence is refined by connectedness (Theorem 3.9) in Lines 1–4. Note that this algorithm does not handle wildcards but can be easily extended by modifying Line 4. Next, the subsequence is refined by structure by testing for gap consistency (Definition 3.14) in Lines 5–11. The subsequence is then tested for frequency consistency (Definition 3.15) in Lines 12–15. Finally, the algorithm matches leaf nodes of the query twig in Lines 16–18. This step can be eliminated by special treatment of leaf nodes in the query twigs and the data trees [RM03]. In Line 19 we report a twig match.

Algorithm 3.5: Refinement Algorithms
Input: $\{N_D, D_Q, L_D, L_Q, S\}$: N_D is the NPS of tree D; D_Q is the NPS of query twig; L_D is a list of leaves in tree D; L_Q is a list of leaves in Q; S is the position of subsequence match in LPS(D);

```
Output: report twig match;
Procedure RefineSubsequence ($N_D$, $D_Q$, $L_D$, $L_Q$, S)
//Test for connectedness (Refinement by Connectedness)
maxN=max($N_D[S_1]$, $N_D[S_2]$, ..., $N_D[S_{|S|}]$);
for i=1 to |S| do
    if $N_D[S_i] \neq$ maxN and $\nexists$(j > i) s.t. $N_D[S_i]=N_D[S_j]$ then
        if $N_D[S_i] \neq S_{i+1}$then return;
        end if
    end if
// Test for gap consistency(Refinement By Structure)
for i=1 to |S| – 1 do
    dataGap=$N_D[S_i] - N_D[S_{i+1}]$;
    queryGap=$N_Q[i] - N_Q[i + 1]$;
    if ((dataGap=0 and queryGap $\neq$ 0) or (queryGap=0 and dataGap $\neq$ 0))
    then return;
    else if dataGap * queryGap < 0 then return;
    else if |queryGap| > |dataGap| then return;
    end if
end for
//Test for frequency consistency(Refinement By Structure)
for i=1 to |S| do
    for j=1 to |S| and j $\neq$ i do
        if $N_Q[i] = N_Q[j]$ and $N_D[S_i] \neq N_D[S_j]$ then
            return;
    end for
end for
//Match leaves(Refinement By matching Leaves)
foreach l in $L_Q$do
    if l not found in $L_Q$ then
        if l not found in LPS/NPS of D then return;
    end if
end for
report twig match; return;
```

Extended Prüfer Sequences

The Prüfer sequence of a tree contains only the labels of non-leaf nodes. We call this sequence Regular-Prüfer sequence. If we extend the tree by adding a dummy child node to each of its leaf nodes, the Prüfer sequence of this extended tree will contain the labels of all the nodes in the original tree. We shall refer to this new sequence as Extended-Prüfer sequence. In the case of XML, all the value nodes (strings/character data) in the XML document tree are extended by adding dummy child nodes before transforming it into a sequence. Similarly, query twigs are also extended before transforming them into sequences. We refer to the index based on Regular-Prüfer sequences as RPIndex and the index based on Extended-Prüfer sequences as EPIndex.

Indexing Extended-Prüfer sequences is useful for processing twig queries with values. Since queries with value nodes usually have high selectivity, Extended-Prüfer sequences provide higher pruning power than Regular-Prüfer sequences during subsequence matching. As a result, during subsequence matching, a fewer root-to- leaf paths are explored in the virtual trie of EPIndex than in the virtual trie of RPIndex for queries with values. If twig queries have no values, then indexing Regular-Prüfer sequences is recommended. Note that Extended-Prüfer sequences are longer than Regular-Prüfer sequences and the increase in length is proportional to the number of leaf nodes in the original tree.

In the PRIX system, both RPIndex and EPIndex can coexist. A query optimizer can choose either of the indexes based on the presence or absence of values in twig queries. It is easy for a query optimizer to detect values in queries since SAX parsers already have separate callback routines for values, attributes, and elements.

Ordered and Unordered Twig Matches

In PRIX, the Prüfer sequence constructed after numbering a query twig in postorder can be used to find all the ordered twig matches. In order to find unordered matches, Prüfer sequence for different arrangements of the branches of the query twig should be constructed and tested for twig matches. Since the number of twig branches in a query is usually small, only a small number of configurations (arrangements) need to be tested.

3.3.2 ViST: A Dynamic Index Method for Querying XML Data by Tree Structures

3.3.2.1 Structure-Encoded Sequence

In this section, we introduce structure-encoded sequences, a sequential representation of both XML data and XML queries [WPFY03]. We show that querying XML is equivalent to finding subsequence matches.

The purpose of modeling XML queries through sequence matching is to avoid as many unnecessary join operations as possible in query processing. That is, we use structure-encoded sequences, instead of nodes or paths, as the basic unit of query. Through sequence matching, we match structured queries against structured data as a whole, without breaking down the queries into sub-queries of paths or nodes and relying on join operations to combine their results. Many XML databases, such as DBLP [Ley] and the Internet movie database IMDB [IMDB00], contain

Table 3.2 Preorder sequence of the XML purchase record example (Fig. 3.10)

P S N v_1 I M v_2 N v_3 I M v_4 I N v_5 L v_6 B L v_7 N v_8

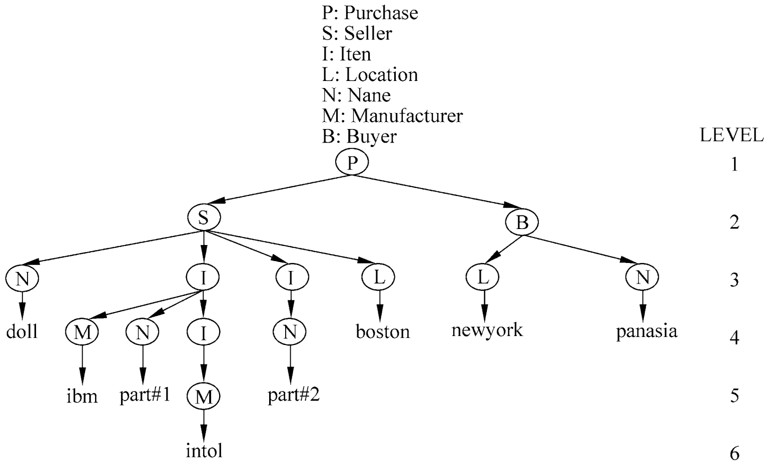


Fig. 3.11 A single purchase record

a large set of records of the same structure. Other XML databases may not be as homogeneous. A synthetic XMARK [XMark02] dataset consists of one (huge) record. However, each substructure in XMARK's schema, items, closed auction, open auction, person, etc., contains a large number of instances in the database and justifies to have an index of its own. Our sequence matching approach ensures that queries confined within the same structure are matched as a whole.

Mapping Data and Queries to Structure-Encoded Sequences

Consider the XML purchase record shown in Fig. 3.10. We use capital letters to represent names of elements/attributes, and we use a hash function, $h()$, to encode attribute values into integers. Suppose, for instance, $v_1 = h(dell)$ and $v_2 = h(ibm)$. We then use v_1 and v_2 to represent *dell* and *ibm*, respectively.

We represent an XML document by the preorder sequence of its tree structure. For the purchase record example, its preorder sequence is shown in Table 3.2.

Since isomorphic trees may produce different preorder sequences, we enforce an order among sibling nodes. The DTD schema embodies a linear order of all elements/attributes defined therein. If the DTD is not available, we simply use the lexicographical order of the names of the elements/attributes. For example, under lexicographical order, the Buyer node will precede the Seller node under Purchase. For multiple occurring child nodes (such as the Item nodes under Seller), we order them arbitrarily. As we shall see later, branching queries require special handling when multiple occurring child nodes are involved (Fig. 3.11).

$$\begin{aligned} &D = \\ &(P,\varepsilon), (S,P), (N,PS), (v_1,PSN), (I,PS), (M,PSI), \\ &(v_2,PSIM), (N,PSI), (v_3,PSIN), (I,PSI), (M,PSII), \\ &(v_4,PSIIM), (I,PS), (N,PSI), (v_2,PSIN), (L,PS), \\ &(v_6,PSL), (B,P), (L,PB), (v_7,PBL), (N,PB), (v_8,PBN) \end{aligned}$$

Fig. 3.12 The structure-encoded sequence of the purchase record document (Fig. 3.10). The underlined noncontiguous subsequence of D matches query Q_2 (Table 3.3)

Table 3.3 XML queries in path expression and sequence form

Path expression	Structure-encoded sequences
Q_1:/Purchase/Seller/Item/Manufacturer	(P, ε) (S, P) (I, PS) (M, PSI)
Q_2:/Purchase[Seller[Loc=v_5]]/Buyer[Loc=v_7]	(P, ε) (S, P) (L, PS)(v_5, PSL) (B, P) (L, PB) (v_7, PBL)
Q_3: /Purchase/*[Loc=v_5]	(P, ε) (L, P*) (v_5, P*L)
Q_4: /Purchase//Item[Manufacturer=v_3]	(P, ε) (I, P//) (M, P//I)(v_3, P//IM)

To reconstruct trees from preorder sequences, extra information is needed. Our structure-encoded sequence, defined below, is a two-dimensional sequence, where the second dimension preserves the structure of the data.

Definition 3.17 (Structure-Encoded Sequence) A structure-encoded sequence, derived from a prefix traversal of a semistructure XML document, is a sequence of (symbol, prefix) pairs:

$$D = (a_1, p_1), (a_2, p_2), \ldots, (a_n, p_n)$$

where a_i represents a node in the XML document tree (of which $a_1, \ldots, a_n$ is the preorder sequence) and p_i is the path from the root node to node a_i.

Based on the definition, the XML purchase record (Fig. 3.10) can be converted to the structure-encoded sequence in Fig. 3.12. The prefixes in the sequential representation contain much redundant information; however, as we shall see, since we do not store duplicate (symbol, prefix) pairs in the index and that the prefix can be encoded easily, it will not create problems in index size or storage.

Querying XML Through Structure-Encoded Sequence Matching

The purpose of introducing structure-encoded sequences is to model XML queries through sequence matching. In other words, querying XML is equivalent to finding (noncontiguous) subsequence matches. We show this by queries Q_1, Q_2, Q_3, Q_4 (Table 3.3).

The structure-encoded sequence of Q_1 is a subsequence of D, and we can see Q_1 is a sub-tree of the XML purchase record that D represents. The sequence of Q_2 is a noncontiguous subsequence of D, and again, Q_2 is a sub-tree of the XML purchase record. The same can be said for query Q_3 and Q_4, where prefix paths contain wildcards "*" and "//"—if we simply match "*" with any single symbol in the path and "//" with any portion of the path. The obvious benefits of modeling XML queries through sequence matching are that structural queries can be processed as a whole instead of being broken down to smaller query units (paths or nodes of XML document trees), as combining the results of the sub-queries by join operations is often expensive. In other words, we use structures as the basic unit of query. Most structural XML queries can be performed through direct subsequence matching. The only exception occurs when a branch has multiple identical child nodes. For instance, in $Q_5 = /A[B/C]/B/D$, the two nodes under the branch are the same: B. In this case, the tree isomorphism problem cannot be avoided by enforcing sibling orders, since the two nodes are identical. As a result, the preorder sequences of XML data trees that contain such a branch can have two possible forms. In order to find all matches, we convert Q_5 to two different sequences, namely, (A, ε) (B, A) (C, AB) $(B, A)(D, AB)$ and (A, ε) (B, A) (D, AB) (B, A) (C, AB). We find matches for these two sequences separately and union their results. On the other hand, we may find false matches if the indexed documents contain branches with identical child nodes. Then, we ask multiple queries and compute set difference on their results. If, in the unlikely case, the query contains a large number of same child nodes under a branch, we can choose to disassemble the tree at the branch into multiple trees and use join operations to combine their results. For instance, Q_5 can be disassembled into two trees: (A, ε) (B, A) (C, AB) and (A, ε) (B, A) $(D, AB)^2$.

After both XML data and XML queries are converted to structure-encoded sequences, it is straightforward to devise a brute force algorithm to perform (noncontiguous) sequence matching. We will focus on building a dynamic index structure so that such matches can be found efficiently.

3.3.2.2 The ViST Approach

ViST(The Virtual Suffix Tree). The sole purpose of the suffix tree is to provide a labeling mechanism to encode S-Ancestorships. Suppose a node x is created for element d_i during the insertion of sequence $d_1, \ldots, d_i, \ldots, d_k$. If we can estimate (1) how many different elements will possibly follow d_i in future insertions and (2) the occurrence probability of each of these elements, then we can label x's child nodes right away, instead of waiting until all sequences are inserted. It also means (1) the suffix tree itself is no longer needed, because its sole purpose of providing a labeling mechanism can be accomplished on the y, and (2) we can support dynamic data insertion and deletion.

ViST uses a dynamic labeling method to assign labels to suffix tree nodes. Once assigned, the labels are fixed and will not be affected by subsequent data insertion or deletion.

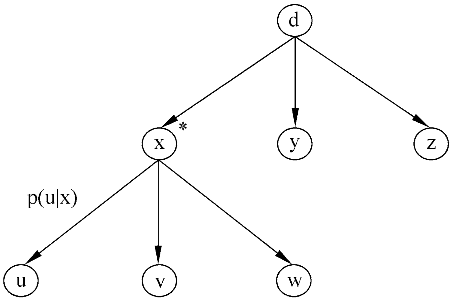


Fig. 3.13 A simple XML schema

Dynamic Virtual Suffix Tree Labeling

We present a dynamic method for labeling suffix tree nodes without building the suffix tree. The method relies on rough estimations of the number of attribute values and other semantic/statistical information of the XML data. To the authors' knowledge, the only dynamic labeling scheme available was recently proposed by Cohen et al. [CKM02] to label XML document trees. Our dynamic scheme is designed to label suffix trees built for structure-encoded sequences derived from XML document trees.

Top–Down Scope Allocation

A tree structure defines nested scopes: the scope of a child node is a subscope of its parent node, and the root node has the maximum scope which covers the scope of each node. Initially, the suffix tree contains a single node (root), and we let it cover the entire scope (0, Max), where Max is the maximum value that the machine can represent under certain precision.

Semantic and Statistical Clues

Semantic and statistical clues of structured XML data can often assist subscope allocation. Figure 3.13 shows a sample XML schema. We use $p(u \mid x)$ to denote, in an XML document, the probability that node u occurs given that node x occurs. For a multiple occurring node v, $p(v \mid x)$ denotes the probability that at least one v occurs given x occurs in an XML document.

If x is the parent of u, usually it is not difficult to derive or estimate, from the semantics of the XML structure or the statistics of a sample dataset, the probability $p(u \mid x)$. For instance, if each Buyer has a name, then $p(\text{Name} \mid \text{Buyer})=1$. If we know that roughly 10 % of the items contain at least a subitem, then $p(\text{SubItem} \mid \text{Item}) = 0.1$.

We start with two assumptions: (1) we know probability $p(u \mid x)$ for all u, where x is the parent of u and (2) in XML document trees, sibling nodes occur independently of each other. We will see how assumption (2) can be relaxed. If node x appears in an XML document based on the schema in Fig. 3.13, then each of the following symbols can appear immediately after x in the sequence derived from the document: u, v, w, y, z, and ε (empty, x is the last element). These symbols form the follow set of x.

Definition 3.18 (Follow Set) Given a node x in an XML scheme, we define the follow set of x as a list, that is, follow(x) = $y_1, \ldots, y_k$, where y_i satisfies the following condition: $x \prec y_i \prec y_{i+1}$ (according to prefix traversal order) and the parent of y_i is either x or an ancestor node of x.

It is straightforward to prove that only symbols in x's follow set can appear immediately after x. Suppose follow(x) = $y_1, \ldots, y_k$, based on the assumption that subnodes occur independently, we have

$$p(y_i|x) = p(y_i|d), \text{ where } d \text{ is the parent of } y_i \tag{3.1}$$

Equation (3.1) is trivial if $d = x$. If $d \neq x$, then based on the definition of the follow set, d must be an ancestor of x, so we have $p(y_i \mid x) = p(y_i \mid x, d)$. Since x and y_i are in different branches under d, according to our assumption, they occur independently of each other, which means $p(y_i \mid x, d) = p(y_i \mid d)$.

Let follow(x) = $y_1, \ldots, y_k$. The probability that x is followed immediately by y_1 is $p(y_1 \mid x)$, by y_2 is $(1 - p(y_1 \mid x))\, p(y_2 \mid x)$.

The probability that x is followed immediately by y_i is

$$P_x(y_i) = p(y_i|\, x) \prod_{k=1}^{i-1} (1 - p(y_k|x)) \tag{3.2}$$

We allocate subscopes for the child nodes in the suffix tree according to the probability. More formally, if x's scope is $[l, r)$, the size of the subscope assigned to y_i, the ith symbol in x's follow set is

$$s_i = (r - l - 1)\, P_x(y_i)\, / C \tag{3.3}$$

where $C = \Sigma y \in \text{follow}(x) - \{\varepsilon\} Px(y)$ is a normalization factor (we do not allocate any scope to ε). In other words, we should assign a subscope $[li, ri) \subseteq [l, r)$ to y_i, where

$$l_i = l + 1 + (r - l - 1) \sum_{j=1}^{i} P_x(y_i); \quad r_i = l_{i+} s_i \tag{3.4}$$

In the following situations, the follow set and Eq. (3.2) need to be adjusted:

1. A same node can occur multiple times under its parent node. Let follow(x) = $y_1, \ldots, y_k$. If x occurs multiple times under its parent, then x also appears in

follow(x), that is, follow(x) $= y_1, \ldots, y_k$, where the symbols before x are the descendants of x. Let the probability that an XML document contains n occurrences of x under d is $p_n(x \mid d)$, then the probability that the $(n-1)$-th x is followed immediately by the nth x is $p_n(x \mid x) \prod_{k=1}^{i-1} (1 - p(y_k|x))$.

2. Nodes do not occur independently. Equation (3.2) is derived based on the assumption that nodes occur independently. However, this may not be true. Suppose for instance, in Fig. 3.13, either u or v must appear under x, and $p(u \mid x) = p(v \mid x) = .8$. We have follow($x$) $= u, v$, since if either u or v must occur, there is no possibility that any of w, y, z, ε can immediately follow x.

   Thus, we have $Px(u) = p(u \mid x) = .8$, $Px(v) = (1 - p(u \mid x))P(v \mid \bar{u}, x) = .2 \mid 1 = .2$.

Dynamic Scope Allocation Without Clues

Assume we do not have any statistical information of the data, or any semantic knowledge about the schema, and all that we can rely on is a rough estimation of the number of different elements that follow a given element. The best we can do is to assume each of these elements occurs at roughly the same rate. This situation usually corresponds to attribute values. For instance, in a certain dataset, we roughly estimate the number of different values for attribute CountryOfBirth to be 100.

Suppose node x is assigned a scope of [l, r). Node x itself will then take l as its ID, and the remaining scope $[l+1, r)$ is available for x's child nodes. Assume the expected number of child nodes of x is λ. Without the knowledge of the occurrence rate of each child node, we allocate $1/\lambda$ of the remaining scope to x's first inserted child, which will have a scope of size $r - l - 1/\lambda$. We allocate $1/\lambda$ of the remaining scope to x's second inserted child, which will have a scope of size $(r - l - 1 - (r - l - 1)/\lambda)/\lambda = (r - l - 1)(\lambda - 1)/\lambda^2$. The third inserted child will use a scope of size $(r - l - 1)(\lambda - 1)^2/\lambda^3$ and so forth.

More formally, according to the above procedure, for a given node x with a range of $[l, r)$, the size of the subrange assigned to its k-th child is $s_k = \left(r - l - 1 - \sum_{i=1}^{k-1} s_i\right)/\lambda$. It is easy to prove that

$$s_k = \frac{(r - l - 1)(\lambda - 1)^{k-1}}{\lambda^k} \tag{3.5}$$

In other words, we should assign a subrange $[l_k, r_k) \subseteq [l, r)$ to the k-th child of node x, where

$$l_k = l + 1 + (r - l - 1)\left(1 - (\lambda - 1)^{k-1}\right)/\lambda^{k-1}; \quad r_k = l_k + s_k \tag{3.6}$$

Based on the above discussion, we introduce the following definition of dynamic scope.

Definition 3.19 (Dynamic Scope of a Suffix Tree Node) The dynamic scope of a node is a triple $< n$, size, $k >$, where k is the number of subscopes allocated inside the current scope. Let the dynamic scopes of x and y be $s_x = < n_x$, size$_x$, $k_x >$ and $s_y = < n_y$, size$_y$, $k_y >$, respectively. Node y is a descendant of x if $s_{y\subset}$ s_x, that is, $[n_y$, $n_y +$ size$_y)_\subset$ $[n_x, n_x +$ size$_x)$.

Scope Underflow

Let $T = t_1, \ldots, t_k$ be a sequence. Each t_i corresponds to a node in the suffix tree. Assume the size of the dynamically allocated scopes decreases on average by a factor of Y every time we descend from a parent node to a child node. As a result, the size of t_i's scope comes to Max/Y^{i-1}, where Max is the size of the root node's scope. Apparently, for a large enough i, Max/$Y^{i-1} \rightarrow 0$. This problem is called scope underflow.

As we have mentioned, XML databases such as DBLP [Ley] and IMDB [IMDB00] are composed of records of small structures. For databases with large structures, such as XMARK [XMark02], we break down the structure into small substructures and create index for each of them. Thus, we limit the average length of the derived sequences.

If scope underflow still occurs for a given sequence $T = t_1, \ldots, t_k$ at t_i, we allocate a subscope of size $k - i + 1$ from node t_{i-1} and label each element $t_i, \ldots,$ t_k sequentially. If node t_{i-1} cannot spare a subscope of size $k - i + 1$, we allocate a subscope of size $k - i + 2$ from node t_{i-2}, and so forth. Intuitively, we borrow scopes from the parent nodes to solve the scope underflow problems for the descendent nodes. In order to do this, we preserve certain amount of scope in each node for this unexpected situation, so that it does not interfere with the dynamic labeling process prescribed by Eqs. (3.3), (3.4), (3.5), and (3.6). Using this method, the involved nodes are labeled sequentially (each node is allocated a scope for only one child), and they can't be shared with other sequences, but they are still properly indexed for matching.

The Algorithms

In this section, we present the dynamic labeling algorithm and the index construction algorithm of ViST. ViST uses the sequence matching algorithm as Algorithm 3.5.

Algorithm 3.6 Search: Noncontiguous Subsequence Matching Using B$^+$Tree
Input: Q – $q_1, \ldots, q_k$, a query sequence, D-Ancestor B$^+$ Tree, index of (symbol, prefix) pairs, S-Ancestor B$^+$ Tree, index of <n, size> labels, DocId B$^+$ Tree, mapping between the n values in node labels and document IDs
Output: all occurrences of Q in the XML data

```
Search(<0, size>,1);   /* <0,size> is the label of the root node of the suffix tree */
Function Search(<n, size>, i);
if i ≤ |Q| then
  T ← retrieve, from the D-Ancestor B$^+$ Tree , the S-Ancestor B$^+$ Tree
  that represents $q_1$;
  N ← retrieve from T, the S-Ancestor B$^+$ Tree, all nodes with range
  inside (n, n+size];
  foreach node c∈N do
    Assume c is labeled <n', size'>;
    Search(<n', size'>,i+1) ;
  end for
else
  Perform a range query [n, n+size) on the DocId B$^+$ Tree to output all document IDs in that range
end if
```

Algorithm 3.7 outlines the top–down dynamic range allocation method described above. The labeling is based on a virtual suffix tree, which means it is not materialized.

Algorithm 3.7 Subscope(parent, e): Create a Subscope Within the Parent Scope for e

Input: p: parent scope; e: symbol for which a subscope is to be created

Output: s, a subscope inside the parent scope p; p, updated parent scope

```
Assume p =<n, size, k>;
if semantical/statistical clues for e is available then
  Assume e is the ith symbol in the follow set of e′s parent node;
  s ← <$l_i$, $s_i$, 0>; /* $l_i$ and $s_i$ are defined in Equation (3.4) and (3.3)
  respectively */
else
  s ← <$l_k$, $s_k$, 0>; /* $l_k$ and $s_k$ are defined in Equation (3.6) and (3.5)
  respectively */
end if
p ← <n, size, k+1>;
return s;
```

Example 3.13 We use an example to demonstrate the process of inserting a structure-encoded sequence into the index structure. Suppose, before the insertion, the index structure already contains the following sequence:

$$\mathrm{Doc}_1 = (P, \varepsilon)\,(S, P)\,(N, PS)\,(v_1, PSN)\,(L, PS)\,(v_2, PSL)$$

The sequence to be inserted is

$$\mathrm{Doc}_2 = (P, \varepsilon)\,(S, P)\,(L, PS)\,(v_2, PSL)$$

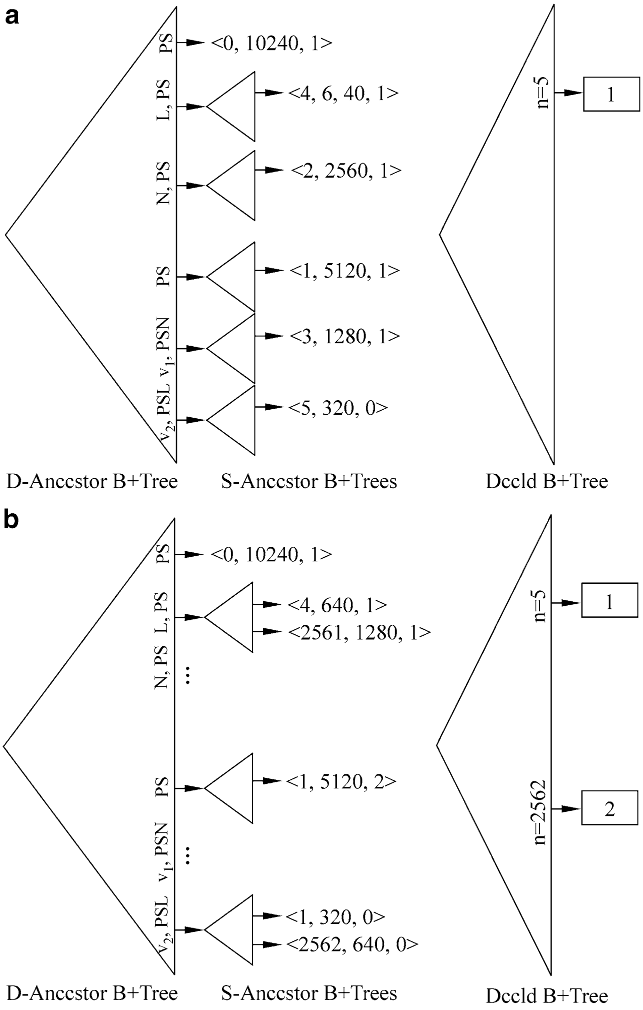


Fig. 3.14 Index structure before and after insertion. (**a**) Index containing Doc$_1$, (**b**) changes caused by the insertion of Doc$_2$

The index before the insertion of Doc$_2$ is shown in Fig. 3.14a. For presentation simplicity, we make two assumptions: (1) Max = 20480, that is, the root node covers a scope of [0, 20480); and (2) there is no semantic/statistical clue available and the top–down dynamic scope allocation method uses a fixed parameter $\lambda = 2$ for all nodes.

The insertion process is much like that of inserting a sequence into a suffix tree—we follow the branches, and when there is no branch to follow, we create one. We start with node (P, ε), and then (S, P), which has scope <1, 5120, $1>$. Next, we search in the S-Ancestor B$^+$ Tree of (L, PS) for all entries that are within the scope of [2, 5120). The only entry there < 4, 640, 1> is apparently not an immediate child[5] of <1, 5120, 1>. As a result, we insert a new entry <2561, 1280, 1>, the 2nd child of (*S*, *P*), in the S-Ancestor B$^+$ Tree of (*L*, *PS*). The scope for the (*S*, *P*) node is updated to <1, 5120, 2> as it has a new child now. Similarly, when we reach (v_2, PSL), we insert a new entry <2562, 640, 0>. Finally, we insert key 2562 into the DocId B$^+$ Tree for Doc$_2$. The resulting index is shown in Fig. 3.14b.

Algorithm 3.8 details the process of inserting an XML sequence into the index structure.

Algorithm 3.8: Index an XML Document
Input: T: a structure-encoded sequence; id: ID of the XML document represented by T
Output: updated index file F

```
Assume T=$(a_1, l_1), \ldots, (a_i, l_i), \ldots, (a_k, l_k)$;
s ← <0, Max, k>; /* s is the scope of the root node of the virtual suffix tree */
i ← 1;
while $i \le k$ do
    Search key($a_i$, $l_i$) in the D-Ancestor B$^+$ Tree;
    if found then
        e ← the S-Ancestor B$^+$ Tree associated with ($a_i$, $l_i$);
    else
        e ← new S-Ancesotr B$^+$ Tree;
        Insert e into the D-Ancestor B$^+$ Tree with key ($a_i$, $l_i$);
    end if
    Search in e for scope r such that r is an immediate child scope of s;
    if not found then
        r ← <n, size, k> ← subScope(s, $a_i$);
        Insert (n, size) into S-Ancestor B$^+$ Tree e with n as key;
    end if
    s ← r;
    i ← i + 1;
end while
Assume s=<n, size, k>;
Insert (n,id) into the DocId B$^+$ Tree;
```

3.3.3 *APEX: An Adaptive Path Index for XML Data*

3.3.3.1 Overview of APEX

In this subsection, we discuss APEX, proposed in [CMS02]. We show an example and the formal definition of APEX as the following. An example of APEX for

a

```
<MovieDB>
      <actor id = “a1”>
            <name> actor1 </name></actor>
      <actor id = “a2”,movie = “m1”>
            <name> actor2 </name></actor>
      <director id = “d1”>
            <name> director1 </name>
            <movie id = “m1” director = “d1”>
                  <title> movie2 </title></movie></director>
      <director id = “d2”>
            <name> director2</name></director>
      <movie id = “m2” actor = “a1”, director = “d2”>
            <title> movie1</title></movie>
</MovieDB>
```

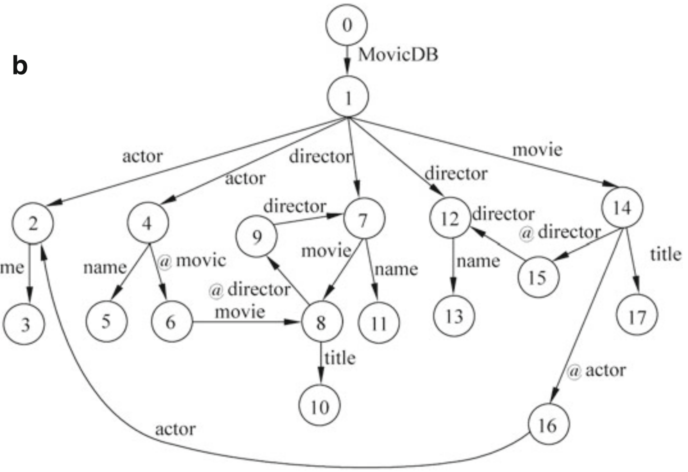


Fig. 3.15 A sample XML data. (**a**) XML data, (**b**) XML data structure

Fig. 3.15 is shown in Fig. 3.16 when the required paths $= A\cup$ {director.movie, @movie.movie, actor.name} (see Definition 3.20). It is not necessary to know how to construct APEX at this point. The purpose of this example is to help understanding the definitions in this section.

As shown in Fig. 3.16, APEX consists of two structures: a graph structure (G_{APEX}) and a hash tree (H_{APEX}). G_{APEX} represents the structural summary of XML data. It is useful for query pruning and rewriting. H_{APEX} represents incoming label paths to nodes of G_{APEX}. H_{APEX} consists of nodes, called the hnode and each hnode contains a hash table. In an hnode, each entry of the hash table points to another hnode or a node of G_{APEX} but not both. That is, each node of G_{APEX} maps to an entry of an hnode of H_{APEX}. H_{APEX} is a useful structure to find a node of G_{APEX} for given label path. Furthermore, H_{APEX} is useful at the incremental update phase.

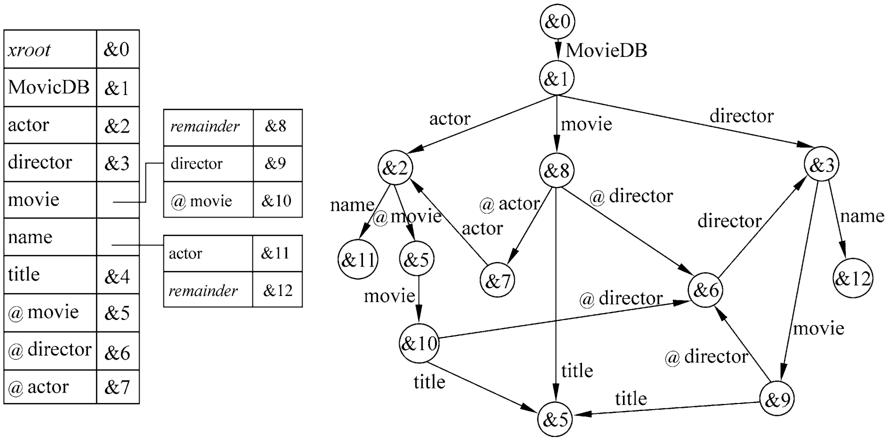


Fig. 3.16 An example of APEX

Also, each node of G_{APEX} corresponds to an extent. The extent is similar to the materialized view in the sense that it keeps the set of edges whose ending nodes are the result of a label path expression of the query.

Since making an effective index structure for all the queries is very hard, APEX changes its structure according to the frequently used paths. To extract frequently used paths, we assume that a database system keeps the set (=workload) of queries (=label paths). Furthermore, we adopt the support concept of the sequential pattern mining to identify frequently used paths [PT87, XMark02].

Let the support of a label path $p = l_i \ldots l_j$, denoted by sup(p), be the ratio of the number of queries having p as a subpath to the total number queries. Also, let minSup denote the user-specified minimum support.

Definition 3.20 A label path $p = l_i \ldots l_j$ in G_{XML} is a frequently used path if sup(p) $\geq$ minSup. Let p be a required path if it is either a frequently used path or the length of p is one.

Definition 3.21 For a label path p of the form $l_i.l_{i+1} \ldots l_j$ in G_{XML}, an edge set, $T(p)$, is $\{<o_{j-1}, o_j> \mid l_i.o_i \ldots l_{j-1}.o_{j-1}.l_j.o_j\}$ is a data path in G_{XML} g. That is, an edge set $T(p)$ is a set of pairs of nids for the incoming edges to the last nodes that are reachable by traversing a given label path p.

Definition 3.22 Let Q_{XML} be a set of label paths of the root node in G_{XML}. For each label path p in the required path set R, let $Q_G(p) = \{l \mid l \in Q_{\mathrm{XML}}$ s.t. p is a suffix of $l\}$, let $Q_A(p) = \{l \mid l{\in}Q_{\mathrm{XML}}$ s.t. every path $q(\neq p)$ R having p as a suffix of $l\}$, and let $Q(p) = Q_G(p) - Q_A(p)$. Finally, a target edge set, $T^R(p) \cup_{r \in Q(p)} T(r)$.

Now, we will define APEX.

Definition 3.23 Given a G_{XML} and a required path set R, APEX can be defined as follows. We introduce the root node of G_{APEX}, x_{root} in APEX which corresponds to the root node of G_{XML}. By considering every required path $p \in R$, we introduce a node of G_{APEX}, with an incoming label path p, that keeps $T^R(p)$ as an extent only if $T^R(p)$ is not empty. For each edge $v.l.v'$ in G_{XML}, there is an edge $x.l.x'$ in APEX where the target edge set of x’ contains $<v, v'>$ and the target edge set of x contains $<u, v>$ where u is a parent node of v.

The following theorem proves that APEX is sufficient for the path index. By the definition of the simulation [BDFS97], if there is a simulation from G_{XML} to G_{APEX}, all the label paths on G_{XML} exist on G_{APEX}. Thus, all queries based on label paths can be evaluated on APEX.

Theorem 3.10 *There is a simulation from G_{XML} to G_{APEX}.*

Proof Given $G_{\text{APEX}}=(V_x, E_x, \text{xroot}, A)$ and $G_{\text{XML}}=(V, E, r, A)$, there is a simulation from r to xroot. Suppose, there is a simulation from $v \in V$ to $x \in V_x$, a full label path q to v is $l_1, \ldots, l_m$, and $\exists\ v.l_{m+1}.v' \in E$. By Definition 3.23, there is a node x' for $T(p' = l_1, \ldots, l_{m+1}$, where $1 \le I \le m)$ whose incoming path is p', and $\exists x.$ $l_{m+1}.x' \in E_x$. Therefore, there is a simulation from G_{XML} to G_{APEX}.

Furthermore, G_{APEX} satisfies the following theorem.

Theorem 3.11 *All the label paths whose lengths are 2 on G_{APEX} are on G_{XML}.*

Proof Recall that APEX groups the edges with respect to the incoming label paths. By Definition 3.23, $\forall$edge $x.l_j.x' \in E_x$, $\exists v.l_j.v' \in E$. By Definition 3.20, an incoming label path of x is a label of incoming edge of x. Therefore, by letting the label of incoming edge of u be l_i, the label path $l_i.l_j$ exists on G_{XML}.

APEX is a general path index since APEX minimally keeps all the label paths whose length is 2 and maximally keeps all the label paths on G_{XML} corresponding to the frequently used paths. Thus, any label path query can be evaluated by lookup of H_{APEX} and/or joins of extents.

3.3.3.2 Construction and Management of APEX

The architecture of the APEX management tool is illustrated in Fig. 3.17. As shown in the figure, the system consists of three main components: the initialization module, the frequently used path extraction module, and the update module.

The initialization module is invoked without query workload only when APEX is built first. This module generates APEX^0 that is the simplest form of APEX and is used as a seed to build a more sophisticated APEX. As query workload is collected with the use of the current APEX, the frequently used paths are computed and used to update the current APEX dynamically into a more detailed version of APEX. The last two steps are repeated whenever query workload changes.

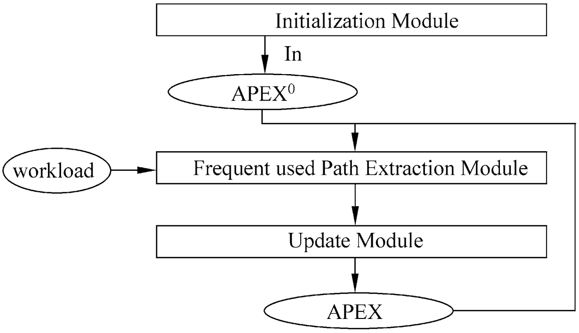


Fig. 3.17 Architecture of APEX management tool

label	xnode
xroot	&0
MovieDB	&1
actor	&2
director	&3
movie	&8
name	&9
title	&4
@movie	&5
@director	&6
@actor	&7

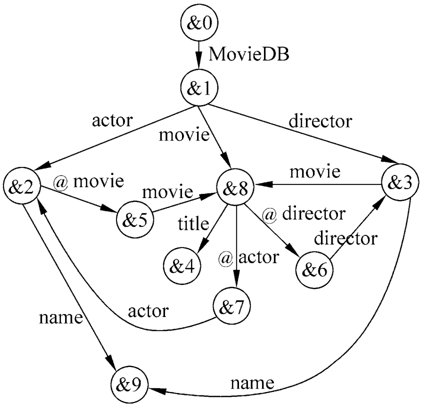


Fig. 3.18 An example of APEX^0

APEX^0 : Initial Index Structure

APEX^0 is the initial structure to build APEX. This step is executed only once at the beginning. Since there is no workload at the beginning, the required path set has paths of size one that is equivalent to the set of all labels in the XML data.

Example 3.14 An example of the APEX^0 for the XML data in Fig. 3.15 is presented in Fig. 3.18. The structure of APEX^0 is similar to the 1-Representative Object(1-RO) proposed as a structural summary in [NUWC97]. As 1-RO contains all paths of size two in the XML data, APEX^0 includes every required path of size two. However, in APEX, we have not only the structural summary in G_{APEX} but also the extents in the nodes of G_{APEX}.

The algorithm of building APEX^0 is shown in Algorithm 3.9. Each node in APEX^0 represents a set of edges that have the same incoming label. Basically, we traverse every node in XML data (G_{XML}) in the depth-first fashion. We first visit the root node of XML data (G_{XML}) and generate the root node for G_{APEX} first.

We add an edge <NULL, root> to the extent of the root node in G_{APEX}. Since each node in $\mathrm{G}_{\mathrm{APEX}}$ represents a unique label and there is no incoming label for the root node of APEX, we represent the root node with a special incoming label *xroot* for convenience. We then call the function explore APEX^0 with the root node in G_{APEX} and the extent of the root node in G_{APEX}.

Algorithm 3.9 Algorithm to Build $\mathbf{APEX^0}$

```
Procedure build APEX⁰(root)
begin
xnode := hash('xroot')
xnode.extent := {< NULL, root >}
exploreAPEX0(xnode, xnode.extent)
end
```

```
Procedure exploreAPEX0(x, ΔESet)
begin
EdgeSet := ∅
for each <u, v> ∈ ΔESet do
   ESet := ESet ∪ {o|o is an outgoing edge from v}
for each unique label l in ESet do {
   y := hash(l)
   if (y=NULL) {
      y := newXNode()
      insert y into hash table
   }
  make edge(x, y, l)
  ΔnewESet := a set of edges having l in EdgeSet – y.extent
  y.extent := y.extent ∪ ΔnewESet
  exploreAPEX0(y, ΔnewESet)
}
end
```

In each invocation of exploreAPEX0, we have two input arguments: the newly visited node x in G_{APEX} and new edges just added to the extent of x in the previous step. We traverse all outgoing edges from the end point of the edges in ΔESet and group them by labels. Then we process edges in each group having the same label l one by one. Let us assume that the node representing the label l in G_{APEX} is y.

Intuitively, we need to put the edges having the label l to the extent of y and connect x and y in G_{APEX}. Then, we call exploreAPEX0 recursively with y and the newly added edges to the extent of y. We give only newly added edges at this step as ΔESet for recursive invocation of exploreAPEX0 since the outgoing edges from the edges included previously to the extent have been all traversed already.

When we consider the edges for each partition with a distinct label, we have to check whether the node y exists already. To find this, we can maintain a hash table and use it. In case the node y for the label l does not exist, we generate a new node and put it to G_{APEX}. We also make sure that the new node can be located quickly by inserting the node into H_{APEX}. The hash at Line (5) in Algorithm 3.9 is the hash function which returns a node for a given label. The procedure make edge makes an edge from x to y with label l. For preventing the infinite traversal of cyclic data, we do not consider the edges which are already in the extent of the node y.

Frequently Used Path Extraction

Any sequential pattern mining algorithms such as the one in [AS95, GRS99] may be used to extract frequently used paths from the path expressions appearing in query workload. While we need to use the traditional algorithms with the anti-monotonicity property [NLHP98] for pruning, we have to modify them.

Consider a mail order company. Assume that many customers buy A first, then B, and finally C. In the traditional sequential pattern problem, when a sequence of (A, B, C) is frequent, all subsequences of size 2 (i.e., (A, B), (A, C), (B, C)) including (A, C) are frequent. However, for the problem of finding frequently used path expressions, it is not valid any more. In other words, even though the path expression of $A.B.C$ is frequently used, the path expression of $A.C$ may not be frequent. Therefore, in case that we want to use traditional data mining algorithms which use the anti-monotonicity pruning technique, we need a minor modification to handle the subtle difference.

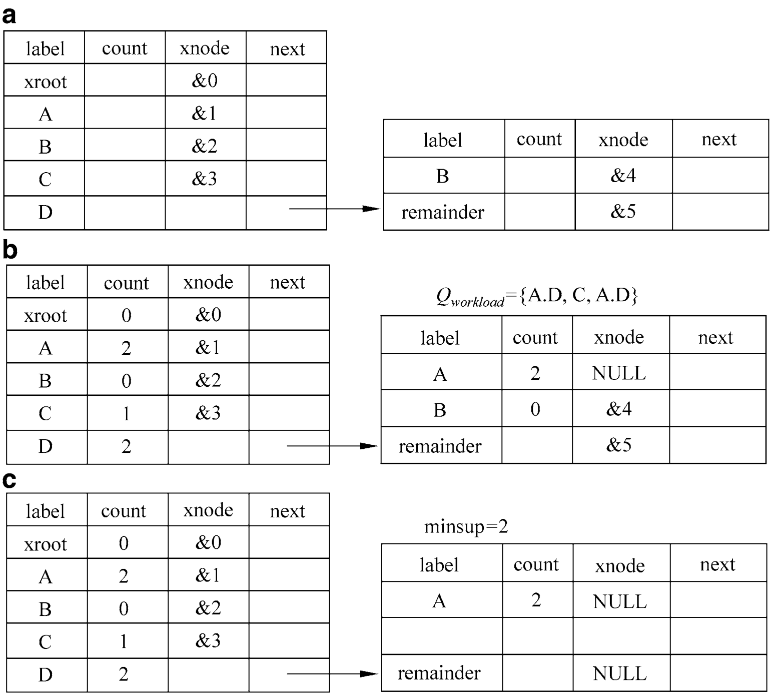
Actually, we also found that the size of query workload is not so large as that of data for sequential pattern mining applications. Thus, we used a naive algorithm in our implementation that simply counts all sequential subsequences that appear in query workload by one scan.

The basic behavior of the frequently used path extraction module is described in Fig. 3.19. Suppose the required path set was $\{A, B, C, D, B.D\}$, the current entry state of H_{APEX} is represented as Fig. 3.19a. A label path $B.D$ is represented as an entry in the subnode of D entry in the root node (HashHead) of H_{APEX}.

Each entry of hash table in a node of H_{APEX} consists of five fields: label, count, new, xnode, and next. The label field keeps the key value for the entry. The count field keeps the frequency of label path which is represented by the entry. The new field is used to check a newly created entry in a node of H_{APEX}. The xnode field points to a node in G_{APEX} whose incoming label path is represented by the entry. Finally, the next field points another node in H_{APEX}. For simplicity, we omit new fields in Fig. 3.19.

Let the workload Q_{workload} become $\{A.D, C, A.D\}$. We first count the frequency of each label path which appeared in Q_{workload} and store the counts in H_{APEX}. When we count, we do not use the remainder entry for counting. Figure 3.19b shows the status of H_{APEX} after the frequency count.

Finally, we prune out the label paths whose frequency is less than minSup. The status of H_{APEX} after pruning is illustrated in Fig. 3.19c. Assume that minSup is 0.6, the label path whose frequency is less than 2 is removed. Thus, a label path $B.D$ is pruned. However, label paths B and C still remain since a label path of size 1 is



a

label	count	xnode	next
xroot		&0	
A		&1	
B		&2	
C		&3	
D			

label	count	xnode	next
B		&4	
remainder		&5	

b

label	count	xnode	next
xroot	0	&0	
A	2	&1	
B	0	&2	
C	1	&3	
D	2		

$Q_{workload}$={A.D, C, A.D}

label	count	xnode	next
A	2	NULL	
B	0	&4	
remainder		&5	

c

label	count	xnode	next
xroot	0	&0	
A	2	&1	
B	0	&2	
C	1	&3	
D	2		

minsup=2

label	count	xnode	next
A	2	NULL	
remainder		NULL	

Fig. 3.19 The behavior of frequently used path extraction. (**a**) current status of H_{APEX}, (**b**) status of H_{APEX} after the frequency count, (**c**) status of H_{APEX} after pruning

always in the required path set. Also, in the pruning step, the xnode fields, which are not valid any more by the change of frequently used paths, are set to NULL. The content for $T^R(D)$ in Fig. 3.19a represents *remainder*. D in H_{APEX} is the set of edges whose end nodes are reachable by traversing a label path D but not $B.D$. However, the content for $T^R(D)$ in Fig. 3.19c should be changed to the set of edges whose end nodes are reachable by traversing a label path D but not $A.D$. Thus, we set this remainder entry to NULL to update it later.

The algorithm of frequently used path extraction is presented in Algorithm 3.10. To extract frequently used paths in the given workload (Q_{workload}), the algorithm first sets count fields to 0 and new fields to FALSE of all entries in H_{APEX}.

The algorithm consists of two parts: the first part counts frequencies and the second part is the pruning phase. H_{APEX} is used to keep the change of the workload. The algorithm invokes the procedure frequencyCount to count the frequency of each label path and its subpaths in Q_{workload}. In this procedure, the new field of a newly created entry in a node in H_{APEX} sets to TRUE to identify a newly created entry in a node in H_{APEX} during pruning phase.

The function pruning H_{APEX} removes the hash entry whose frequency is less than the given threshold minSup(Line (4)–(5)). Even though the frequency of an entry in the root node (HashHead) of H_{APEX} is less than minSup, it should not be removed since a label path of size 1 should be always in the required path set by Definition 3.20. If the frequency of an entry of the node in H_{APEX} is less than minSup and the entry is not in the root node, the entry is removed from the hash node by the function hnode.delete (Line (6)–(7)). If all entries in a node of H_{APEX} except remainder entry are removed by the function hnode.delete, hnode.delete returns TRUE. And then we remove this node of H_{APEX} (Line (10)–(11)).

Finally, it sets the xnode field to NULL because it points to the wrong node in G_{APEX}. As mentioned earlier, contents of some nodes of G_{APEX} may be affected by the change of frequently used paths. There are two cases. A label path q was maximal suffix previously but it is not anymore. This is captured by that an entry has not NULL value in both xnode and next fields (Line (12)–(13)). In this case, the algorithm sets the xnode field to NULL to update the xnode field appropriately later. The second case is when a new frequently used path influences the contents of the node of G_{APEX} for the remainder entry in the same node of H_{APEX} since the contents of *remainder* is affected by the change of frequently used path. This is represented by that a new entry is appeared in the node of H_{APEX} and remainder entry in this node points to a node in G_{APEX} using the xnode field (Line (14)–(15)). In this case, the algorithm sets the content of the xnode field in remainder entry to NULL to update it later.

Algorithm 3.10: Frequently Used Path Extraction Algorithm

```
Procedure frequentlyUsedPathExtraction()
begin
1.  reset all count fields to 0 and new fields to FALSE
2.  frequencyCount()
3.  pruning$H_{\mathrm{APEX}}$ (HashHead)
end

Function pruning$H_{\mathrm{APEX}}$ (hnode)
begin
1.  is empty :=FALSE
2.  if hnode=NULL return is empty
3.     for each entry t$\in$hnode do
4.        if (t.count < minsup) {
5.           t.next :=NULL
6.           if ($t \notin$HashHead) {
7.              is empty :=hnode.delete(t)
8.           }
9.        } else {
10.          if (pruning$H_{\mathrm{APEX}}$ (t.next)=TRUE)
11.             t.next :=NULL
12.          if (t.next $\neq$ NULL) and (t.xnode $\neq$ NULL)
13.             t.xnode :=NULL
14.          if (t.new=true) and (hnode.remainder.xnode $\neq$ NULL)
15.             hnode.remainder=NULL
```

```
    }
  }
  return is empty
end
```

The Update with Frequently Used Paths

After the entries in H_{APEX} was updated with frequently used paths computed from the changes of query workload, we have to update the graph G_{APEX} and xnode fields of entries in the nodes of H_{APEX} that locates the corresponding node in G_{APEX}.

Each entry in a node of H_{APEX} may have a pointer to another node of H_{APEX} in the next field or a pointer to the node of G_{APEX} in the xnode field, but the entry may not have non-NULL value for both next and xnode fields. If the entry of a node n in H_{APEX} has a pointer to another node *m* of H_{APEX}, there exists a longer frequently used path represented in *m* whose suffix is represented by the entry in the node *n* of H_{APEX}.

The basic idea of update is to traverse the nodes in G_{APEX} and update not only the structure of G_{APEX} with frequently used paths but also the xnode field of entries in H_{APEX}. While visiting a node in G_{APEX}, we look for the entry of the maximum suffix path in H_{APEX} from the root to the currently visiting node in G_{APEX}. Note that an entry for the maximum suffix path always exists in H_{APEX} since the last label of the path to look for in H_{APEX} always exists by the definition of the required path (see Definition 3.20).

Algorithm 3.11: Algorithm to Update APEX

```
Procedure updateAPEX(xnode, ΔESet, path)
begin
if (xnode.visited=TRUE) and (ΔESet=∅), return
xnode.visited :=TRUE
EdgeSet :=∅
if ΔESet=∅ {
   foreach e that is an outgoing edge of xnode do {
      newpath :=concatenate(path, e.label)
      xchild :=hash(newpath)
      if (xchild=NULL) xchild :=new XNode()
      if (xchild !=e.end) {
        if (EdgeSet=∅) {
          foreach < u, v >∈xnode.extent do
             EdgeSet :=EdgeSet ∪ {o|o is an outgoing edge from v}
        }
        subEdgeSet :=a set of edges with the label e.label in EdgeSet
        ΔEdgeSet :=subEdgeset - xchild.extent
        xchild.extent :=xchild.extent ∪ ΔEdgeSet
```

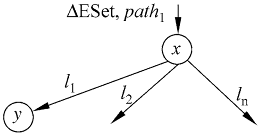
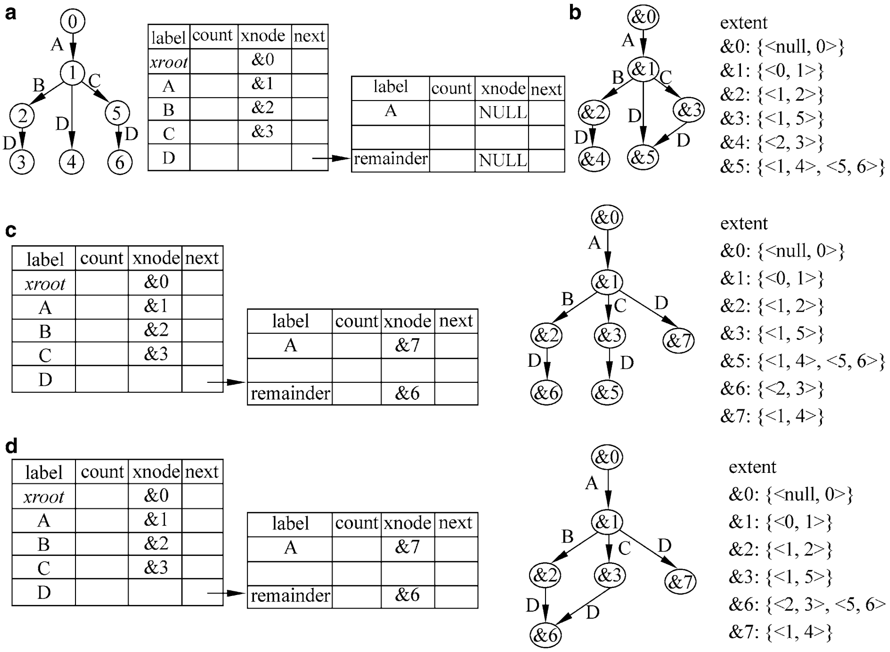


Fig. 3.20 Visiting a node by UpdateAPEX

```
17.          make edge(xnode, xchild, e.label);
18.          hash.append(newpath, xchild);
19.        }
20.        else ΔEdgeSet :=Ø
21.          updateAPEX(xchild, ΔEdgeSet, newpath);
22.      }
23.    } else {
24.     foreach < u, v >∈ΔESet do
25.        EdgeSet :=EdgeSet ∪ {o|o is an outgoing edge from v}
26.        foreach unique label l in EdgeSet do {
27.          newpath :=concatenate(path, e.label)
28.          xchild :=hash(newpath)
29.          if (xchild=NULL) xchild :=new XNode()
30.          subEdgeSet :=set of edges labeled l in EdgeSet
31.          ΔEdgeSet :=subEdgeset - xchild.extent
32.         xchild.extent :=xchild.extent ∪ ΔEdgeSet
33.         make edge(xnode, xchild, l)
34.         hash.append(newpath, xchild)
35.         updateAPEX(xchild, ΔEdgeSet, newpath);
36.        }
37.    }
end
```

Now, we present the updateAPEX in Algorithm 3.11 that does the modification of APEX with frequently used paths stored in H_{APEX}. Before calling the updateAPEX, we first initialize *visited* flags of all nodes in G_{APEX} to FALSE. The updateAPEX is executed with the root node in G_{APEX} by calling updateAPEX(xroot, Ø, NULL) where xroot is the root node of G_{APEX}.

Suppose that we visit a node x in G_{APEX} with a label path path$_l$ and an edge set ΔESet as illustrated in Fig. 3.20. ΔESet is a newly added edge to the extent of x just before visiting the current node. If x was previously visited and ΔESet is empty, then we do nothing since all edges and their subgraphs of x were traversed before (Line (1)). If x is newly visited and ΔESet is empty, we should traverse all outgoing edges of x in G_{XML} to verify the all subnodes of x according to H_{APEX}. For each ending vertex (i.e., e.end) in the outgoing edges of the visiting node in G_{APEX}, we get the pointer xchild that represents a node in G_{APEX} with the maximal suffix stored in H_{APEX} of the label path to the visiting node from the root by calling hash function (Line (6)–(7)). If the value of xchild is NULL, it means that the valid node in G_{APEX} does not exist. Thus, we allocate a new node of G_{APEX} and set to xchild (Line (8)).



a

label	count	xnode	next
xroot		&0	
A		&1	
B		&2	
C		&3	
D			

label	count	xnode	next
A		NULL	
remainder		NULL	

c

label	count	xnode	next
xroot		&0	
A		&1	
B		&2	
C		&3	
D			

label	count	xnode	next
A		&7	
remainder		&6	

d

label	count	xnode	next
xroot		&0	
A		&1	
B		&2	
C		&3	
D			

label	count	xnode	next
A		&7	
remainder		&6	

Fig. 3.21 An example for update APEX. (**a**) G_{XML} (**b**) APEX before update, (**c**) snapshot of updating, (**d**) APEX after update

If xchild is not NULL, the entry in H_{APEX} points to node in G_{APEX} for a given label path. Thus, we insert the edge set to the extent of xchild.

If xchild and e.end are different, we compute edge set which should be added to xchild (Line (10)–(14)). We insert the newly added edges in the extent of xchild to ΔESet and update the extent of xchild (Line (15)–(16)). We next make edge by invoking *make edge* from this xnode to xchild with label e.label, if the edge does not exist, and set xchild to H_{APEX} with the given label path by calling *make edge* (Line (17)–(18)). If xnode has an outgoing edge to a node in G_{APEX} which is different from xchild with label e.label, make edge removes this edge. If xchild and e.end are equal, there is no change of the extent of the node xchild (Line(20)). Thus, ΔESet is set to empty set. Now, we call updateAPEX recursively for the child node xchild.

Whether an xnode is previously visited or not, if there is a change of the extent of it, we should update the subgraph rooted at xnode (Line (23)–(37)). In this case, we obtain outgoing edges from the end point of edges in ΔESet (Line (24)–(25)). In order to update the subgraph rooted at xnode, we partition the edges based on the labels of edges in ΔESet and update H_{APEX} and G_{APEX} similarly as we processed for the case when ΔESet was empty set (Line (26)–(36)).

Let us consider the H_{APEX}, G_{APEX}, and G_{XML} in Fig. 3.21a, b. Assume that we invoke up updateAPEX (xroot, Ø, NULL). Since the ΔESet is empty, the code in Line (4)–(23) in Algorithm 3.11 will be executed. Since there is only one outgoing edge with the label A and the ending node &1, we check the entry with H_{APEX} for the path of A. The xnode field of the entry returned by hash points the node &1. Thus, we do nothing and call updateAPEX (&1, Ø, A) recursively. This recursive call visits the node &1 with label path A. We have three outgoing edges from the node &1. Consider node &2 first in the for-loop in Line (5), we check the entry in H_{APEX} with the path of $A.B$ and find that the entry points to the node &2. Thus, we do nothing again and invoke updateAPEX (&2, Ø, $A.B$) recursively. Inside of this call, we check outgoing edges of &2. As illustrated in Fig. 3.21b, the end node of an outgoing edge of &2 with label D is &4. However, for the input of $A.B.D$, hash returns NULL that is the xnode field of the entry for remainder.D, which should point to the node in G_{APEX} representing all label paths ending with D except $A.D$. Thus, we make a new node &6 for remainder.D, compute extent of &6, and change an outgoing edge of &2 with label D to point out &6. Since there is no outgoing edge, we return back to &1 from recursive call. We next consider the outgoing edge with end node of &5 with label $A.D$ from &1.

The H_{APEX} and G_{APEX} including the extents after traversing every node in G_{APEX} with updateAPEX are illustrated in Fig. 3.21d.

3.4 Summary

In this chapter, we present XML index structures. Two phases are introduced in detail: in the index based on XML tree structure section, we show three index structures: DataGuides [GW97], 1-Indexes [MS99], and F&B-Index [KBNK02]. The strong DataGuides and the 1-Indexes can be viewed as covering indexes, since it is possible to answer queries over those indexes directly without consulting the base data. The F&B-Index already can be viewed as a structure analogous to a summary table or covering index. It is also shown that it is the smallest such index that covers all branching path expression queries. In addition, we also show three index structures: PRIX [RM03], ViST [WPFY03], and APEX [CMS02]. PRIX presents a new paradigm for XML pattern matching. Unlike most state-of-the-art techniques, PRIX approach processes twig queries without breaking them into root-to-leaf paths and processing them individually. ViST uses the structures as the basic unit of query, which enables us to process, through sequence matching, structured queries as a whole and, as a result, to avoid expensive join operations. In addition, ViST supports dynamic insertion of XML documents through the top–down scope allocation method. Finally, APEX does not keep all paths starting from the root and utilizes frequently used paths to improve the query performance. To support efficient query processing, APEX consists of two structures: the graph structure G_{APEX} and the hash tree H_{APEX}. G_{APEX} represents the structural summary of XML data with extents. H_{APEX} keeps the information for frequently used paths and their corresponding nodes in G_{APEX}. Given a query, H_{APEX} can be used to locate the nodes of G_{APEX} that have extents required to evaluate the query.

References

[AS95] Agrawal, R., Srikant, R.: Mining sequential patterns. In: Proceedings of the 11th International Conference on Data Engineering, Taipei, pp. 3–14, Mar 1995

[BDFS97] Buneman, P., Davidson, S.B., Fernandez, M.F., Suciu, D.: Adding structure to unstructured data. In: Proceedings of the 6th International Conference on Database Theory, Delphi, pp. 336–350, Jan 1997

[Cat93] Cattell, R.G.G. (ed.): The Object Database Standard: ODMG-93. Morgan Kaufmann, San Francisco (1994)

[CD99] Clark, J., Derose, S.: XML path language (XPath) 1.0. W3C Recommendation. World Wide Web Consortium, http://www.w3.org/TR/xpath, Nov 1999

[CKM02] Cohen, E., Kaplan, H., Milo, T.: Labeling dynamic XML trees. In: PODS, Madison, pp. 271–281 (2002)

[CMS02] Chung, C-W., Min, K-K., Shim, K.: APEX: an adaptive path index for XML data. In: SIGMOD, Madison, pp. 121–132 (2002)

[GRS99] Garofalakis, M.N., Rastogi, R., Shim, K.: SPIRIT: sequential pattern mining with regular expression constraints. In: Proceedings of 25th International Conference on Very Large Data Bases, Edinburgh, pp. 223–234, Sept 1999

[GW97] Goldman, R., Widom, J.: DataGuides: enabling query formulation and optimization in semistructured databases. In: VLDB, Athens, pp. 436–445 (1997)

[HHK95] Henziner, M., Henziner, T., Kopke, P.: Computing simulations on finite and infinite graphs. In: Proceedings of 20th Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, pp. 453–462 (1995)

[IMDB00] The internet movie database: http://www.imdb.com (2000)

[KBNK02] Kaushik, R., Bohannon, P., Naughton, J.F., Korth, H.F.: Covering indexes for branching path queries. In: Proceedings of SIGMOD 2002, Madison, (2002)

[KSBG02] Kaushik, R., Shenoy, P., Bohannon, P., Gudes, E.: Exploiting local similarity for efficient indexing of paths in graph structured data. In: Proceedings of ICDE, San Jose (2002)

[Ley] Ley, M.: DBLP database web site. http://www.informatik.uni-trier.de/ley/db (2000)

[LM01] Li, Q., Moon, B.: Indexing and querying XML data for regular path expressions. In: Proceedings of the 27th VLDB Conference, pp. 361–370, Sept 2001

[Mil80] Milner, R.: A Calculus for Communicating Processes. Lecture Notes in Computer Science, vol. 92. (1980)

[Mil89] Milner, R.: Communication and Concurrency. Prentice Hall, New York (1989)

[MS99] Milo, T., Suciu, D.: Index structures for path expressions. In: ICDT, Jerusalem, pp. 277–295 (1999)

[NLHP98] Ng, R.T., Lakshmanan, L.V.S., Han, J., Pang, A.: Exploratory mining and pruning optimizations of constrained association rules. In: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, Seattle, pp. 13–24, June 1998

[NUWC97] Nestorov, S., Ullman, J.D., Wiener, J.L., Chawathe, S.S.: Representative objects: concise representations of semistructured, hierarchial data. In: Proceedings of the 13th International Conference on Data Engineering, Birmingham, pp. 79–90, Apr 1997

[PGW95] Papakonstantinou, Y., Garcia-molina, H., Widom, J.: Object exchange across heterogeneous information source. In: Proceeding of the 11th International Conference on Data Engineering, Taipei, pp. 251–260 (1995)

[Pru18] Prüfer, H.: Neuer Beweis eines Satzes über Permutationen. Arch. Math. Phys. **27**, 742–744 (1918)

[PT87] Paige, R., Tarjan, R.: Three partition refinement algorithms. SIAM J. Commun. **16**, 973–988 (1987)

[RM03] Rao, P., Moon, B.: PRIX: indexing and querying XML using Prüfer sequences. Technical Report TR 03-06, University Of Arizona, Tucson, AZ 85721. http://www.cs.arizona.edu/research/reports.html, July 2003

[SM73] Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time. In: 5th STOC. ACM, Austin, pp. 1–9 (1973)

[WPFY03] Wang, H., Park, S., Fan, W., Yu, F.S.: ViST: a dynamic index method for query XML data by tree structures. In: Proceeding of the 2003 ACM-SIGMOD Conference, San Diego, CA, June 2003

[XMark02] XMARK.: The XML-benchmark project. http://monetdb.cwi.nl/xml (2002)

Chapter 4
XML Tree Pattern Processing

Abstract Finding all the occurrences of a twig pattern in an XML database is a core operation for efficient evaluation of XML queries. In this chapter, we present two kinds of join algorithms: XML structural join and XML holistic twig pattern processing. The structural join decomposes twig pattern into many binary relationships, which can be either parent–child or ancestor–descendant relationships, while holistic twig join processes the twig pattern as a whole, which has been proved more efficient. In addition, we showed two new streaming schemes: Tag+level scheme and Prefix-Path Streaming, which can provide optimal solutions for a larger class of twig patterns.

Keywords Algorithm • XML • Twig pattern • Structural join • Holistic • Stream

4.1 Introducing XML Tree Pattern Processing

XML employs a tree-structured data model, and naturally, XML queries specify patterns of selection predicates on multiple elements related by a tree structure. Finding all occurrences of such a twig pattern in an XML database is a core operation for XML query processing.

According to method the twig patterns being processed, we classify these algorithms [AJP+02, BKS02, ZND+01] into two main categories: structural join, also called binary join, and holistic twig join. The structural join decomposes twig pattern into many binary relationships, which can be either parent–child or ancestor–descendant relationships, while holistic twig join is to process the twig pattern as a whole, which has been proved more efficient.

In this chapter, we first show the structural join algorithms called tree-merge and stack-tree algorithms [AJP+02], which are proposed to match the binary relationships and finally stitch together basic matches to get the final results; however, the main disadvantage of this decomposition based approach is that intermediate result

J. Lu, *An Introduction to XML Query Processing and Keyword Search*,
DOI 10.1007/978-3-642-34555-5_4,

sizes can get very large, even when the input and the final result sizes are much more manageable. Thus, some holistic twig join algorithms are proposed. Without decomposition and join, they save intermediate costs obviously. In this chapter, we will show PathStack [BKS02], TwigStack [BKS02], TwigStackList [LCL04], and TJFast [LLC+05] algorithms, to process the twig pattern queries.

In addition, the algorithms above use a tag scheme. By using it, an XML document is clustered into element streams which group all elements with the same tag name together and assign each element a containment label. But these algorithms are not optimal processing. Therefore, Chen, Lu, and Ling [CLL05] propose two new streaming schemes: Tag+level scheme and Prefix-Path Streaming. Based on these two streaming schemes, they develop a holistic twig join algorithm iTwigJoin. In the last section of this chapter, we will show the optimality of this new algorithm to process twig patterns with parent–child or ancestor–descendant, even with one branchnode.

4.2 XML Structural Join

XML employs a tree-structured model for representing data. Quite naturally, queries in XML query languages typically specify patterns of selection predicates on multiple elements that have some specified tree-structured relationships. For example, the XQuery path expression: book[title = ‘XML’]//author[. = ‘jane’] matches author elements that (1) have as content the string value “jane” and (2) are descendants of book elements that have a child title element whose content is the string value “XML”. This XQuery path expression can be represented as a node-labeled tree pattern with elements and string values as node labels.

A complex query tree pattern can be decomposed into a set of basic binary structural relationships such as parent–child and ancestor–descendant between pairs of nodes. The query pattern can then be matched by (1) matching each of the binary structural relationships against the XML database and (2) “stitching” together these basic matches. For example, the basic structural relationships corresponding to the query tree pattern of Fig. 4.1a are shown in Fig. 4.1b.

More recently, Zhang et al. [ZND+01] proposed a variation of the traditional merge join algorithm, called the multi-predicate merge join (MPMGJN) algorithm, for finding all occurrences of the basic structural relationships (they refer to them

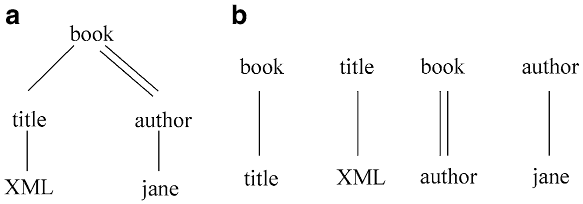


Fig. 4.1 (**a**) Tree pattern, (**b**) structural relationships

a

```
<book>
   <title> XML </title>
   <allauthors>
      <author> jane </author>
      <author> john </author>
   </allauthors>
   <year> 2000 </year>
   <chapter>
      <head> origins </head>
      <section>
         <head> ... </head>
         <section> ... </section>
      </section>
      <section> ... </section>
   </chapter>
   <chapter> ... </chapter>
</book>
```

b

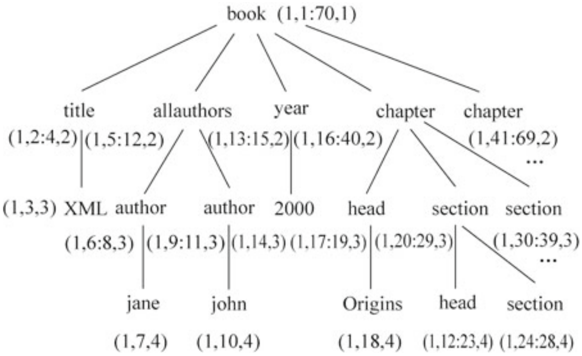


Fig. 4.2 (**a**) A sample XML document fragment, (**b**) tree representation

as containment queries). The key to the efficiency of the MPMGJN algorithm is the (DocId, StartPos: EndPos, LevelNum) representation of positions of XML elements and the (DocId, StartPos, LevelNum) representation of positions of string values that succinctly capture the structural relationships between elements (and string values) in the XML database. Checking that structural relationships in the XML tree, like ancestor–descendant and parent–child (corresponding to containment and direct containment relationships, respectively, in the XML document representation), are present between elements amounts to checking that certain inequality conditions hold between the components of the positions of these elements.

The MPMGJN algorithm performs a lot of unnecessary computation and I/O for matching basic structural relationships, especially in the case of parent–child relationships (or direct containment queries). In this chapter, we will introduce algorithms proposed by Al-Khalifa et al. [AJP+02],who take advantage of the (DocId, StartPos: EndPos, LevelNum) representation of positions of XML elements and string values to devise novel I/O and CPU optimal join algorithms for matching structural relationships against an XML database.

In the 3-tuple (DocId, StartPos, LevelNum), (1) DocId is the identifier of the document; (2) StartPos and EndPos can be generated by counting word numbers from the beginning of the document with identifier DocId until the start of the element and end of the element, respectively; and (3) LevelNum is the nesting depth of the element (or string value) in the document. Figure 4.2b depicts a 3-tuple with each tree node, based on this representation of position. (The DocId for each of these nodes is chosen to be 1.)

Structural relationships between tree nodes (elements or string values) whose positions are recorded in this fashion can be determined easily—(1) ancestor–descendant: a tree node n_2 whose position in the XML database is encoded as

(D_2, S_2:E_2, L_2) is a descendant of a tree node n_1 whose position is encoded as (D_1, S_1:E_1, L_1) if and only if $D_1 = D_2$, $S_1 < S_2$, and $E_1 < E_2$ (for leaf strings, EndPos is the same as StartPos); and (2) parent–child: a tree node n_2 whose position in the XML database is encoded as (D_2, S_2:E_2, L_2) is a child of a tree node n_1 whose position is encoded as (D_1, S_1:E_1, L_1) if and only if $D_1 = D_2$, $S_1 < S_2$ and $E_2 < E_1$, and $L_1 + 1 = L_2$.

For example, in Fig. 4.2b, the author node with position $(1, 6:8, 3)$ is a descendant of the book node with position $(1, 1:70, 1)$, and the string "jane" with position (1, 7, 4) is a child of the author node with position $(1, 6:8, 3)$.

Consider an ancestor–descendant (or parent–child) structural relationship (e_1, e_2), for example, (book, author) (or (author, jane)). Let AList = [a_1, a_2, ...] and DList = [d_1, d_2, ...] be the lists of tree nodes that match the node predicates e_1 and e_2, respectively, each list sorted by the (DocId, StartPos) values of its elements. There are a number of ways in which the AList and the DList could be generated from the database that stores the XML data. For example, a native XML database system could store each element node in the XML data tree as an object with the attributes: ElementTag, DocId, StartPos, EndPos, and LevelNum. An index could be built across all the element tags, which could then be used to find the set of nodes that match a given element tag. The set of nodes could then be sorted by (DocId, StartPos) to produce the lists that serve as inputs to these join algorithms.

Given these two input lists, AList of potential ancestors (or parents) and DList of potential descendants (resp., children), the algorithms in each family can output a list OutputList = [(a_i, d_j)] of join results, sorted either by (DocId, a_i.StartPos, d_j.StartPos) or by (DocId, d_j.StartPos, a_i.StartPos). Both variants are useful, and the variant chosen may depend on the order in which an optimizer chooses to compose the structural joins to match the complex XML query pattern. Here, two families of join algorithms, tree-merge and stack-tree algorithm, are introduced.

4.2.1 Tree-Merge Join Algorithms

4.2.1.1 Algorithm of Tree-Merge Join

Algorithm 4.1: Tree-Merge-Anc(AList, DList)

```
/* Assume that all nodes in AList and DList have the same DocId        */
/* AList is the list of potential ancestors, in sorted order of StartPos    */
/* DList is the list of potential descendants in sorted order of StartPos */
begin-desc = DList→firstNode;    OutputList = NULL;
for (a = AList→firstNode; a ! = NULL;a = a→nextNode) {
   for (d-begin-desc; (d!=NULL && d.StartPos<a.StartPos); d=d→nextNode)
      {/* skipping over unmatchable d's */
      begin-desc = d;
   for (d=begin-desc; (d!=NULL && d.EndPos<a.EndPos); d=d→nextNode) {
        if ((a.StartPos < d.StartPos) && (d.EndPos < a.EndPos)
            [&& (d.LevelNum = a.LevelNum + 1)]) {
        /* the optional condition is for parent-child relationships */
          append (a,d) to OutputList;   }
    }
}
```

Algorithm 4.2: Tree-Merge-Desc(AList, DList)

```
/* Assume that all nodes in AList and DList have the same DocId */
/* AList is the list of potential ancestors, in sorted order of StartPos */
/* DList is the list of potential descendants in sorted order of StartPos */
begin-anc = AList→firstNode; OutputList = NULL;
for (d=DList→firstNode d=DList->firstNode; d !=NULL; d=d→nextNode) {
  for (a=begin-anc; (a!=NULL && a.EndPos<d.StartPos); a=a→nextNode)
            { /* skipping over unmatchable a's   */   }
          begin-anc = d;
  for (a=begin-anc; (a!=NULL && a.StartPos<a.StartPos); a=a→nextNode) {
            if ((a.StartPos < d.StartPos) && (d.EndPos < a.EndPos)
        [&& (d.LevelNum = a.LevelNum + 1)]) {
        /* the optional condition is for parent-child relationships */
        append (a,d) to OutputList; }
  }
}
```

The algorithms in the tree-merge family are a natural extension of traditional relational merge joins (which use an equality join condition) to deal with the multiple inequality conditions that characterize the ancestor–descendant or the parent–child structural relationships, based on the (DocId, StartPos:EndPos, LevelNum) representation.

The basic idea here is to perform a modified merge join, possibly performing multiple scans through the "inner" join operand to the extent necessary. Either AList or DList can be used as the inner (resp., outer) operand for the join: the results are produced sorted (primarily) by the outer operand. In Algorithm 4.1, we present the tree-merge algorithm for the case when the outer join operand is the ancestor. Similarly, Algorithm 4.2 deals with the case when the outer join operand is the descendant. For ease of understanding, both algorithms assume that all nodes in the two lists have the same value of DocId, their primary sort attribute. Dealing with nodes from multiple documents is straightforward, requiring the comparison of DocId values and the advancement of node pointers as in the traditional merge join.

4.2.1.2 An Analysis of the Tree-Merge Algorithms

Traditional merge joins that use a single equality condition between two attributes as the join predicate can be shown to have time and space complexities O(|input|+|output|), on sorted inputs, while producing a sorted output. In general, one cannot establish the same time complexity when the join predicate involves multiple equality and/or inequality conditions. This section identifies the criteria under which tree-merge algorithms have asymptotically optimal time complexity.

Algorithm Tree-Merge-Anc for Ancestor–Descendant Structural Relationship

Theorem 4.1 *The space and time complexities of Algorithm 4.1 are* O(|AList|+|DList|+|OutputList|), *for the ancestor–descendant structural relationship.*

The intuition is as follows. Consider first the case where no two nodes in AList are themselves related by an ancestor–descendant relationship. In this case, the size of OutputList is O(|AList|+|DList|). Algorithm 4.1 makes a single pass over the input AList and at most two passes over the input DList. Thus, the above theorem is satisfied in this case.

Consider next the case where multiple nodes in AList are themselves related by an ancestor–descendant relationship. This can happen, for example, in the (section, head) structural relationship for the XML data in Fig. 4.2. In this case, multiple passes may be made over the same set of descendant nodes in DList, and the size of OutputList may be O(|AList|*|DList|), which is quadratic in the size of the input lists. However, we can show that the algorithm still has optimal time complexity, O(|AList|+|DList|+|OutputList|).

Algorithm Tree-Merge-Anc for Parent–Child Structural Relationship

When evaluating a parent–child structural relationship, the time complexity of Algorithm 4.1 is the same as if one were performing an ancestor–descendant structural relationship match between the same two input lists. However, the size of OutputList for the parent–child structural relationship can be much smaller than the size of the OutputList for the ancestor–descendant structural relationship. In particular, consider the case when all the nodes in AList form a (long) chain of length n, and each node in AList has two children in DList, one on either side of its child in AList; this is shown in Fig. 4.3a. In this case, it is easy to verify that the size of OutputList is O(|AList|+|DList|), but the time complexity of Algorithm 4.1 is O((|AList|+|DList|^2)); the evaluation is pictorially depicted in Fig. 4.3b, where each node in AList is associated with the sublist of DList that needs to be scanned. The I/O complexity is also quadratic in the input size in this case.

An Analysis of Algorithm Tree-Merge-Desc

There is no analog to Theorem 4.1 for Algorithm 4.2, since the time complexity of the algorithm can be O((|AList|+|DList|+|OutputList|)^2) in the worst case. This happens, for example, in the case shown in Fig. 4.3c, when the first node in AList

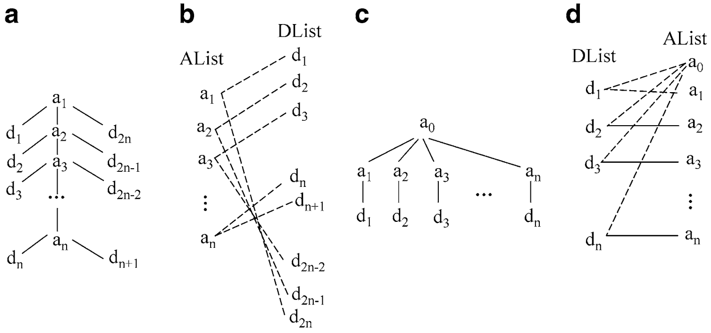


Fig. 4.3 (**a**), (**b**) Worst case for Tree-Merge-Anc and (**c**), (**d**) Worst case for Tree-Merge-Desc

is an ancestor of each node in DList. In this case, each node in DList has only two ancestors in AList, so the size of OutputList is O(|AList|+|DList|), but AList is repeatedly scanned, resulting in a time complexity of O(|AList|*|DList|); the evaluation is depicted in Fig. 4.3d, where each node in DList is associated with the sublist of AList that needs to be scanned.

While the worst-case behavior of many members of the tree-merge family is quite bad, on some datasets and queries they perform quite well in practice.

4.2.2 *Stack-Tree Join Algorithms*

A depth-first traversal of a tree can be performed in linear time using a stack of size as large as the height of the tree. In the course of this traversal, every ancestor–descendant relationship in the tree is manifested by the descendant node appearing somewhere higher on the stack than the ancestor node. Therefore, a family of stack-based structural join algorithms, stack-tree join algorithms, are proposed, with better worst-case I/O and CPU complexity than the tree-merge family, for both parent–child and ancestor–descendant structural relationships.

Unfortunately, the depth-first traversal idea, even though appealing at first glance, cannot be used directly since it requires traversal of the whole database. We would like to traverse only the candidate nodes provided as part of the input lists. The algorithms here have no counterpart in traditional join processing.

4.2.2.1 Stack-Tree-Desc Algorithm

Consider an ancestor–descendant structural relationship (e_1, e_2). Let AList = $[a_1, a_2, \ldots]$ and DList = $[d_1, d_2, \ldots]$ be the lists of tree nodes that match node predicates e_1 and e_2, respectively, sorted by the (DocId, StartPos) values of its elements.

First, we discuss the stack-tree algorithm for the case when the output list [(a_i, d_j)] is sorted by (DocID, d_j.StartPos, a_i.StartPos). This is both simpler to understand and extremely efficient in practice. The algorithm is presented in Algorithm 4.3 for the ancestor–descendant case.

The basic idea is to take the two input operand lists, AList and DList, both sorted on their (DocId, StartPos) values and conceptually merge (interleave) them. As the merge proceeds, we determine the ancestor–descendant relationship, if any, between the current top of stack and the next node in the merge, that is, the node with the smallest value of StartPos. Based on this comparison, we manipulate the stack and produce output.

The stack at all times has a sequence of ancestor nodes, each node in the stack being a descendant of the node below it. When a new node from the AList is found to be a descendant of the current top of stack, it is simply pushed on to the stack. When a new node from the DList is found to be a descendant of the current top of stack, we know that it is a descendant of all the nodes in the stack. Also, it is guaranteed that it won't be a descendant of any other node in AList. Hence, the join results involving this DList node with each of the AList nodes in the stack are output. If the new node in the merge list is not a descendant of the current top of stack, then we are guaranteed that no future node in the merge list is a descendant of the current top of stack, so we may pop stack and repeat our test with the new top of stack. No output is generated when any element in the stack is popped.

Algorithm 4.3: Stack-Tree-Desc (AList, DList)

```
/* Assume that all nodes in AList and DList have the same DocId        */
/* AList is the list of potential ancestors, in sorted order of StartPos   */
/* DList is the list of potential descendants in sorted order of StartPos */
a = AList→firstNode;   d = DList→firstNode;   OutputList = NULL;
while (the input lists are not empty or the stack is not empty)   {
  if ((a.StartPos>stack→top.EndPos) && (d.StartPos>stack→top.EndPos)) {
      /* time to pop the top element in the stack */
      Tuple = stack→pop();   }
  else if (a.StartPos < d.StartPos)   {
      stack→push(a)
       a = a→nextNode }
  else {
       for (a1 = stack→bottom; a1 ! = NULL; a1 = a1→up)   {
           append (a1,d) to OutputList
       }
       D = d→nextNode
  }
}
```

The basic idea is to take the two input operand lists, AList and DList, both sorted on their (DocId, StartPos) values and conceptually merge (interleave) them. As the merge proceeds, we determine the ancestor–descendant relationship, if any,

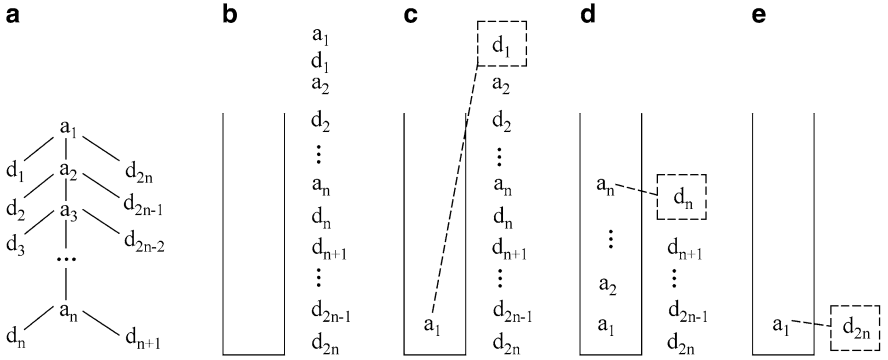


Fig. 4.4 (**a**) Dataset; (**b**)–(**e**) steps during evaluation of Stack-Tree-Desc

between the current top of stack and the next node in the merge, that is, the node with the smallest value of StartPos. Based on this comparison, we manipulate the stack and produce output.

The stack at all times has a sequence of ancestor nodes, each node in the stack being a descendant of the node below it. When a new node from the AList is found to be a descendant of the current top of stack, it is simply pushed on to the stack. When a new node from the DList is found to be a descendant of the current top of stack, we know that it is a descendant of all the nodes in the stack. Also, it is guaranteed that it won't be a descendant of any other node in AList. Hence, the join results involving this DList node with each of the AList nodes in the stack are output. If the new node in the merge list is not a descendant of the current top of stack, then we are guaranteed that no future node in the merge list is a descendant of the current top of stack, so we may pop stack and repeat our test with the new top of stack. No output is generated when any element in the stack is popped.

The parent–child case of Algorithm 4.3 is even simpler since a DList node can join only (if at all) with the top node on the stack. In this case, the "for loop" inside the "else" case of Algorithm 4.3 needs to be replaced with

$$\begin{array}{c}\text{if } (d.\text{LevelNum} = \text{stack-} > \text{top.LevelNum} + 1)\\ \text{append } (\text{stack-} > \text{top}, d) \text{ to OutputList}\end{array}$$

Example 4.1 Some steps during an example evaluation of Algorithm 4.3, for a parent–child structural relationship, on the dataset of Fig. 4.4a, are shown in Fig. 4.4b–e. The a_i's are the nodes in AList and the d_j's are the nodes in DList. Initially, the stack is empty, and the conceptual merge of AList and DList is shown in Fig. 4.4b. In Fig. 4.4c, a_1 has been put on the stack, and the first new element of the merged list, d_1, is compared with the stack top (at this point (a_1, d_1) is output). Figure 4.4d illustrates the state of the execution several steps later, when a_1, a_2, ..., a_n are all on the stack, and d_n is being compared with the stack top (after this point, the OutputList includes (a_1, d_1),(a_2, d_2), ...,(a_n, d_n). Finally, Fig. 4.4e shows the state of the execution when the entire input has almost been processed. Only a_1 remains on the stack (all the other a_i's have been popped from the stack), and d_{2n} is compared with a_1. Note that all the desired matches have been produced while making only a single pass through the entire input. Recall that this is the same dataset of Fig. 4.3a, which illustrated the suboptimality of Algorithm 4.1, for the case of parent–child structural relationships.

4.2.2.2 Stack-Tree-Anc Algorithm

We next discuss the stack-tree algorithm for the case when the output list [(a_i, d_j)] needs to be sorted by (DocID,a_i.StartPos,d_j.StartPos).

It is not straightforward to modify Algorithm 4.3 to produce results sorted by ancestor because of the following: if node a from AList on the stack is found to be an ancestor of some node d in the DList, then every node a' from AList that is an ancestor of a (and hence below a on the stack) is also an ancestor of d. Since the StartPos of a' precedes the start position of a, we must delay output of the join pair (a, d) until after (a', d) has been output. But there remains the possibility of a new element d' after d in the DList joining with a' as long a' is on stack, so we cannot output the pair (a, d) until the ancestor node a' is popped from stack. Meanwhile, we can build up large join results that cannot yet be output. The solution to this problem is described in Algorithm 4.4 for the ancestor–descendant case.

Algorithm 4.4: Stack-Tree-Anc (AList, DList)

```
/* Assume that all nodes in AList and DList have the same DocId */
/* AList is the list of potential ancestors, in sorted order of StartPos */
/* DList is the list of potential descendants in sorted order of StartPos */
a = AList→firstNode; d = DList→firstNode;   OutputList = NULL;
while (the input lists are not empty or the stack is not empty)   {
   if ((a.StartPos>stack→top.EndPos) && (d.StartPos>stack→top.EndPos))
{
      /* time to pop the top element in the stack */
      Tuple = stack→pop();
      if (stack→size == 0) {    /* we just popped the bottom element */
         append tuple.inherit-list to OutputList }
      else {
           append tuple.inherit-list to tuple.self-list
           append the resulting tuple.self-list to stack→top.inherit-list
      }
   }
   else if (a.StartPos < d.StartPos) {
      stack→push(a)
      a = a→nextNode }
   else {
        for (a1 = stack→bottom; a1 ! = NULL;a1 = a1→up) {
            if (a1 == stack→bottom)    append (a1,d) to OutputList
            else append (a1,d) to the self-list of a1
        }
        D = d→nextNode
    }
```

As with Algorithm 4.3, the stack at all times has a sequence of ancestor nodes, each node in the stack being a descendant of the node below it. Now, we associate two lists with each node on the stack: the first, called self-list, is a list of result elements from the join of this node with appropriate DList elements; the second, called inherit-list, is a list of join results involving AList elements that were descendants of the current node on the stack. As before, when a new node from the AList is found to be a descendant of the current top of stack, it is simply pushed on to the stack. When a new node from the DList is found to be a descendant of the current top of stack, it is simply added to the self-list of the nodes in the stack. Again, as before, if no new node (from either list) is a descendant of the current top of stack, then we are guaranteed that no future node in the merge list is a descendant of the current top of stack, so we may pop stack and repeat the test with the new top of stack. When the bottom element in stack is popped, we output its self-list first and then its inherit-list. When any other element in stack is popped, no output is generated. Instead, we append its inherit-list to its self-list and append the result to the inherit-list of the new top of stack.

An optimization to the algorithm (incorporated in Algorithm 4.4) is as follows: no self-list is maintained for the bottom node in the stack. Instead, join results with the bottom of the stack are output immediately. This results in a small space savings and renders the stack-tree algorithm partially non-blocking.

4.2.2.3 An Analysis of Algorithm Stack-Tree-Desc

Algorithm 4.3 is easy to analyze. Each AList element in the input may be examined multiple times, but these can be amortized to the element on DList, or the element at the top of stack, against which it is examined. Each element on the stack is popped at most once and when popped, causes examination of the new top of stack with the current new element. Finally, when a DList element is compared against the top element in stack, then it either joins with all elements on stack or none of them; all join results are immediately output. In other words, the time required for this part is directly proportional to the output size. Thus, the time required for this algorithm is $O(|\text{input}|+|\text{output}|)$ in the worst case. Putting all this together, we get the following result:

Theorem 4.2 *The space and time complexities of Algorithm 4.3 are* $O(|\text{AList}|+|\text{DList}|+|\text{OutputList}|)$, *for both ancestor–descendant and parent–child structural relationships. Further, Algorithm 4.3 is a non-blocking algorithm.*

Clearly, no competing join algorithm that has the same input lists, and is required to compute the same output list, could have better asymptotic complexity.

The I/O complexity analysis is straightforward as well. Each page of the input lists is read once, and the result is output as soon as it is computed. Since the maximum size of stack is proportional to the height of the XML database tree, it is quite reasonable to assume that all of stack fits in memory at all time. Hence, we have the following result:

Theorem 4.3 *The I/O complexity of Algorithm 4.3 is* O(|AList|/B+|DList|/B+|OutList|/B), *for ancestor–descendant and parent–child structural relationships, where B is the blocking factor.*

4.2.2.4 An Analysis of Algorithm Stack-Tree-Anc

The key difference between the analyses of Algorithms 4.4 and 4.3 is that join results are associated with nodes in the stack in Algorithm 4.4. Obviously, the list of join results at any node in the stack is linear in the output size. What remains to be analyzed is the appending of lists each time the stack is popped. If the lists are implemented as linked lists (with start and end pointers), these append operations can be carried out in unit time and require no copying. Thus, one comparison per AList input and one per output are all that are performed to manipulate stack. Combined with the analysis of Algorithm 4.3, we can see that the time required for this algorithm is still O(|input|+|output|) in the worst case.

The I/O complexity analysis is a little more involved. Certainly, one cannot assume that all the lists of results not yet output fit in memory. Careful buffer management is required. It turns out that the only operation we ever perform on a list is to append to it (except for the final read out). As such, we only need to have access to the tail of each list in memory as computation proceeds. The rest of the list can be paged out. When list x is appended to list y, it is not necessary that the head of list x be in memory: the append operation only establishes a link to this head in the tail of y. So all we need is to know the pointer for the head of each list, even if it is paged out. Each list page is thus paged out at most once and paged back in again only when the list is ready for output. Since the total number of entries in the lists is exactly equal to the number of entries in the output, we thus have that the I/O required on account of maintaining lists of results is proportional to the size of output (provided that there is enough memory to hold in buffer the tail of each list: requiring two pages of memory per stack entry—still a requirement within reason). All other I/O activity is for the input and output. This leads to the desired linearity result.

Theorem 4.4 *The space and time complexities of Algorithm 4.4 are* O(|AList|+|DList|+|OutputList|), *for both ancestor–descendant and parent–child structural relationships. The I/O complexity of Algorithm 4.4 is* O(|AList|/B+|DList|/B+|OutList|/B), *for both ancestor–descendant and parent–child structural relationships, where B is the blocking factor.*

4.3 XML Holistic Twig Pattern Processing

As introduced in last section, we have typically decomposed the twig pattern into a set of binary (parent–child and ancestor–descendant) relationships, and twig matching is achieved by (1) using structural join algorithms to match the binary relationships against the XML database and (2) stitching together these basic matches. A limitation of this approach for matching twig patterns is that intermediate result sizes can get large, even when the input and output sizes are more manageable. Therefore, from this part, we will introduce holistic twig join algorithms, which will process the twig patterns as a whole to reduce the intermediate costs, for twig pattern processing. We will introduce four-twig join algorithms: PathStack [BKS02], TwigStack [BKS02], TwigStackList [LCL04], and TJFast [LLC+05].

Some notions will be used in these algorithms:

Given a query twig pattern Q and an XML database D, a match of Q in D is identified by a mapping from nodes in Q to nodes in D, such that (1) query node predicates are satisfied by the corresponding database nodes (the images under the mapping) and (2) the structural (parent–child and ancestor–descendant) relationships between query nodes are satisfied by the corresponding database nodes. The answer to query Q with n nodes can be represented as an n-ary relation where each tuple $(d_1, \ldots, d_n)$ consists of the database nodes that identify a distinct match of Q in D.

Let q (with or without subscripts) denote twig patterns as well as (interchangeably) the root node of the twig pattern. Associated with each node q in a query twig pattern there is a stream T_q. The stream contains the positional representations of the database nodes that match the node predicate at the twig pattern node q (possibly obtained using an efficient access mechanism, such as an index structure).

The node data is coding as (DocId, StartPos, EndPos, LevelNum), and the nodes in the stream are sorted by their values.

Some twig node operations are used as follows: isLeaf: Node→Bool, isRoot: Node→Bool, parent: Node→Node, children: Node→{Node}, and subtreeNodes: Node→{Node}. Path queries have only one child per node; otherwise, children(q) returns the set of children nodes of q. The result of subtreeNodes(q) is the node q and all its descendants.

The operations over streams are eof, advance, next, nextBegin, and nextEnd. The last two operations return the StartPos and EndPos coordinates in the positional representation of the next element in the stream, respectively.

In stack-based algorithms, just like PathStack and TwigStack, we also associate with each query node q a stack S_q. Each data node in the stack consists of a pair: (positional representation of a node from T_q, pointer to a node in $S_{\text{parent}(q)}$).

The operations over stacks are empty, pop, push, topBegin, and topEnd. The last two operations return the StartPos and EndPos coordinates in the positional representation of the top element in the stack, respectively. At every point during the computation, (1) the nodes in stack S_q (from bottom to top) are guaranteed to

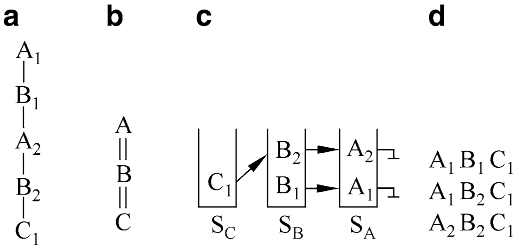


Fig. 4.5 Compact encoding of answers using stacks

lie on a root-to-leaf path in the XML database, and (2) the set of stacks contain a compact encoding of partial and total answers to the query twig pattern, which can represent in linear space a potentially exponential (in the number of query nodes) number of answers to the query twig pattern, as illustrated below.

Figure 4.5 illustrates the stack encoding of answers to a path query for a sample dataset. The answer [A_2, B_2, C_1] is encoded since C_1 points to B_2 and B_2 points to A_2. Since A_1 is below A_2 on the stack S_A, [A_1, B_2, C_1] is also an answer. Finally, since B_1 is below B_2 on the stack S_B and B_1 points to A_1, [A_1, B_1, C_1] is also an answer. Note that [A_2, B_1, C_1] is not an answer, since A_2 is above the node (A_2) on stack S_A to which B_1 points.

4.3.1 PathStack

Bruno et al. [BKS02] had proposed a holistic path join algorithm, called PathStack, to match XML query root-to-leaf paths efficiently. Path pattern query is twig pattern query without branch.

PathStack generalizes the Stack-Tree-Desc binary structural join algorithm [AJP+02]. After analysis, PathStack is I/O and CPU optimal among all sequential algorithms that read the entire input and has worst-case complexities linear in the sum of input and output sizes but independent of the sizes of intermediate results.

4.3.1.1 Algorithm of PathStack

Algorithm 4.5: PathStack(q)

```
while !end(q)
    $q_{min}$ = getMinSource(q)
    for $q_i$ in subtreeNodes(q)    // clean stacks
        while (!empty($S_{qi}$) $\wedge$ topEnd($S_{qi}$) < nextBegin($T_{qmin}$))
            pop($S_{qi}$))
    moveStreamToStack($T_{qmin}$, $S_{qmin}$, pointer to top($S_{parent(qmin)}$))
    if (isLeaf(qmin))
        showSolutions($S_{qmin}$,1)
        pop($S_{qmin}$)
```

Function end(q)

```
return $\forall$ $q_i \in$ subtreeNodes(q):isLeaf($q_i$) $\Rightarrow$ eof($T_{qi}$)
```

Function getMinSource(q)

```
return $q_i \in$ subtreeNodes(q) such that nextBegin($T_{qi}$) is minimal
```

Procedure moveStreamToStack (T_q, S_q, p)

```
push($S_q$, (next($T_q$),p))
advance ($T_q$)
```

Algorithm PathStack computes answers to a query path pattern for the case when the streams contain nodes from a single XML document. When the streams contain nodes from multiple XML documents, the algorithm is easily extended to test equality of Docld before manipulating the nodes in the streams and stacks.

The key idea of Algorithm 4.5 is to repeatedly construct (compact) stack encodings of partial and total answers to the query path pattern, by iterating through the stream nodes in sorted order of their StartPos values; thus, the query path pattern nodes will be matched from the query root down to the query leaf. Line 2, in Algorithm 4.5, identifies the stream containing the next node to be processed. Lines 3–5 remove partial answers from the stacks that cannot be extended to total answers, given the knowledge of the next stream node to be processed. Line 6 augments the partial answers encoded in the stacks with the new stream node. Whenever a node is pushed on the stack $S_{q^{\min}}$, where $q_{\min}$ is the leaf node of the query path, the stacks contain an encoding of total answers to the query path, and Algorithm showSolutions is invoked by Algorithm 4.5 (lines 7–9) to "output" these answers.

A natural way for Algorithm showSolutions to output query path answers encoded in the stacks is as *n*-tuples that are in sorted leaf-to-root order of the query path. This will ensure that, over the sequence of invocations of Algorithm showSolutions by Algorithm 4.5, the answers to the query path are also computed in leaf-to-root order. Procedure showSolutions shows such a procedure for the case when only ancestor–descendant edges are present in the query path.

Procedure showSolutions (SN, SP)

```
// Assume, for simplicity, that the stacks of the query
// nodes from the root to the current leaf node we
// are interested in can be accessed as S[1] . . . S[N].
// Also assume that we have a global array index[1 . . . n]
// of pointers to the stack elements.
// index[i] represents the position in the i'th stack that
// we are interested in for the current solution, where
// the bottom of each stack has position 1.
// Mark we are interested in position SP of stack SN.
index[SN]=SP
if (SN == 1)        // we are in the root
                    // output solutions from the stacks
```

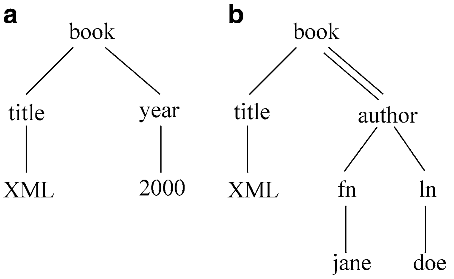


Fig. 4.6 Query twig patterns

```
4.       output (S[n].index[n] . . . ,S[1].index[1])
5.    else                      // recursive call
6.       for i = 1 to S[SN].index[SN].pointer_to_parent
7.          showSolutions (SN-1,i)
```

When parent–child edges are present in the query path, we also need to take the LevelNum information into account. PathStack does not need to change, but we need to ensure that each time showSolutions invoked, it does not output incorrect tuples, in addition to avoiding unnecessary work. This can be achieved by modifying the recursive call (lines 6–7) to check for parent–child edges, in which case only a single recursive call (showSolutions(SN-1, S[SN].index[SN].pointer_to_the_parent_stack)) needs to be invoked, after verifying that the LevelNum of the two nodes differ by one. Looping through all nodes in the stack S[SN-1] would still be correct, but it would do more work than is strictly necessary.

If we desire the final answers to the query path be presented in sorted root-to-leaf order (as opposed to sorted leaf-to-root order), it is easy to see that it does not suffice that each invocation of showSolutions output answers encoded in the stack in the root-to-leaf order. To produce answers in the sorted root-to-leaf order, we would need to "block" answers and delay their output until we are sure that no answer prior to them in the sort order can be computed. The details of how to achieve this naturally are presented by Bruno et al. [BKS02].

Example 4.2 Consider the leftmost path, book–title–XML, in each of the query twigs of Fig. 4.6. If we used the binary structural join algorithms of [ZND+01], we would first need to compute matches to one of the parent–child structural relationships: book–title or title–XML. Since every book has a title, this binary join would produce a lot of matches against an XML book database, even when there are only a few books whose title is XML. If, instead, we first computed matches to title–XML, we would also match pairs under chapter elements, as in the XML data tree of Fig. 4.7, which do not extend to total answers to the query path pattern. Using Algorithm 4.5, partial answers are compactly represented in the stacks, and not output. Using the XML data tree of Fig. 4.7, only one total answer, identified by the mapping [book→(1,1:150,1), title→(1,2:4,2), XML→(1,3,3)], is encoded in the stacks.

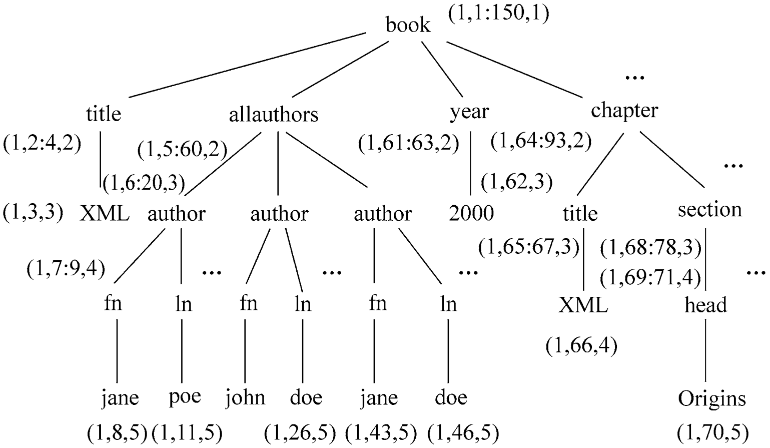


Fig. 4.7 A sample XML tree representation

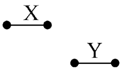
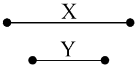
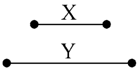
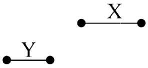
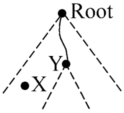
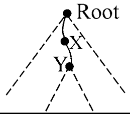
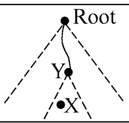
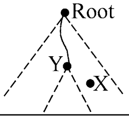
	Case 1	Case 2	Case 3	Case 4
Property	X.R<Y.L	X.R<Y.R X.L<Y.R	X.L<Y.L X.R<Y.R	X.L<Y.R
Segments				
Tree				

Fig. 4.8 Cases for PathStack and TwigStack

4.3.1.2 Analysis of Pathstack

The following proposition is a key to establishing the correctness of Algorithm 4.5

Proposition 4.1 *If we fix node Y, the sequence of cases between node Y and nodes X on increasing order of* StartPos (L) *is* (1|2)*3*4*. *Case 1 and Case 2 are interleaved, then all nodes in Case 3 before any node in Case 4, and finally all nodes in Case 4* (see Fig. 4.8).

Lemma 4.1 *Suppose that for an arbitrary node q in the path pattern query, we have that* getMinSource(q) $= q_N$. *Also, suppose that* t_{q^N} *is the next element in* q_N*'s stream. Then, after* t_{q^N} *is pushed on to stack* S_{q^N}, *the chain of stacks from* S_{q^N} *to Sq verifies that their labels are included in the chain of nodes in the XML data tree from* t_{q^N} *to the root.*

For each node $t_{q^{\min}}$, pushed onto stack $S_{q^{\min}}$, it is easy to see that the above lemma, along with the iterative nature of Algorithm showSolutions, ensures that all answers in which $t_{q^{\min}}$ is a match for query node $q_{\min}$ will be output. This leads to the following correctness result:

Theorem 4.5 *Given a query path pattern q and an XML database D, Algorithm 4.5 correctly returns all answers for q on D.*

We next show optimality. Given an XML query path of length n, PathStack takes n input lists of tree nodes sorted by (DocId, StartPos) and computes an output sorted list of n-tuples that match the query path. It is straightforward to see that, excluding the invocations to showSolutions, the I/O and CPU costs of PathStack are linear in the sum of sizes of the n input lists. Since the cost of showSolutions is proportional to the size of the output list, we have the following optimality result:

Theorem 4.6 *Given a query path pattern q with n nodes and an XML database D, Algorithm 4.5 has worst-case I/O and CPU time complexities linear in the sum of sizes of the n input lists and the output list. Further, the worst-case space complexity of Algorithm 4.5 is the minimum of (1) the sum of sizes of the n input lists and (2) the maximum length of a root-to-leaf path in D.*

It is particularly important to note that the worst-case time complexity of Algorithm 4.5 is independent of the sizes of any intermediate results.

4.3.2 TwigStack

A straightforward way of computing answers to a query twig pattern is to decompose the twig into multiple root-to-leaf path patterns, use PathStack to identify solutions to each individual path, and then merge-join these solutions to compute the answers to the query. This approach faces the same fundamental problem as the techniques based on binary structural joins, toward a holistic solution: many intermediate results may not be part of any final answer, as illustrated below.

Example 4.3 Consider the query sub-twig rooted at the author node of the twig pattern in Fig. 4.9b. Against the XML database in Fig. 4.10, the two paths of the query, author-fn-jane and author-In-doe, have two solutions each, but the query twig pattern has only one solution.

In general, if the query (root-to-leaf) paths have many solutions that do not contribute to the final answers, using PathStack (as a subroutine) is suboptimal, in that the overall computation cost for a twig pattern is proportional not just to the sizes of the input and the final output but also to the sizes of intermediate results. In this section, Bruno et al. [BKS02] proposed a new algorithm TwigStack to overcome this suboptimality.

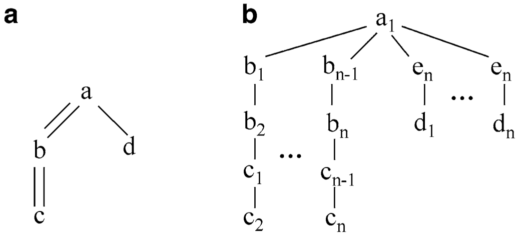


Fig. 4.9 Illustration to the suboptimality of TwigStack (**a**) query twig pattern, (**b**) an XML tree

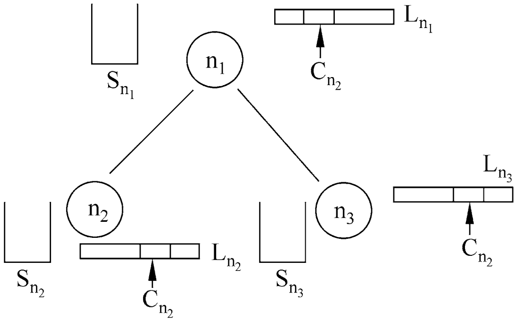


Fig. 4.10 Stacks S_n and lists L_n used in algorithm

4.3.2.1 Algorithm of TwigStack

Algorithm TwigStack computes answers to a query twig pattern for the case when the streams contain nodes from a single XML document. As with Algorithm 4.5, when the streams contain nodes from multiple XML documents, the algorithm is easily extended to test equality of DocId before manipulating the nodes in the streams and on the stacks.

Algorithm TwigStack operates in two phases. In the first phase (lines 1–11), some (but not all) solutions to individual query root-to-leaf paths are computed. In the second phase (line 12), these solutions are merge-joined to compute the answers to the query twig pattern.

The key difference between PathStack and the first phase of TwigStack is that before a node h_q from the stream T_q is pushed on its stack S_q, TwigStack (via its call to getNext) ensures that (1) node h_q has a descendant h_{q^i} in each of the streams T_{q^i}, for $q_i \in$ children(q), and (2) each of the nodes h_{q^i} recursively satisfies the first property. Algorithm 4.5 does not satisfy this property (and it does not need to do so to ensure (asymptotic) optimality for query path patterns). Thus, when the query twig pattern has only ancestor–descendant edges, each solution to each individual query root-to-leaf path is guaranteed to be merge joinable with at least one solution to each of the other root-to-leaf paths. This ensures that no intermediate solution is larger than the final answer to the query twig pattern.

The second merge join phase of Algorithm TwigStack is linear in the sum of its input (the solutions to individual root-to-leaf paths) and output (the answer to the query twig pattern) sizes, only when the inputs are in sorted order of the common prefixes of the different query root-to-leaf paths. This requires that the solutions to individual query paths be output in root-to-leaf order as well, which necessitates blocking; Procedure showSolutions, which outputs solutions in sorted leaf-to-root order, cannot be used.

Algorithm 4.6: TwigStack(q)

```
// Phase 1
while !end(q)
    $q_{act}$ = getNext(q)
    if (!isRoot($q_{act}$))
        cleanStack(parent($q_{act}$), nextL($q_{act}$))
    if (isRoot($q_{act}$) V !empty($S_{parent(qact)}$))
        cleanStack($q_{act}$ , next($q_{act}$))
        moveStreamToStack($T_{qact}$ ,$S_{qact}$ , pointer to top($S_{parent(qact)}$))
        if (isLeaf($q_{act}$))
            showSolutionsWithBlocking($S_{qact}$ ,1)
            pop($S_{qact}$)
    else advance($T_{qact}$)
// Phase 2
mergeAllPathSolutions()
```

Function getNext(q)

```
if (isLeaf(q))    return q
for $q_i$ in children(q)
    $n_i$ = getNext($q_i$)
    if ($n_i \neq q_i$)    return $n_i$
$n_{min}$ = minarg$_{ni}$nextL($T_{ni}$)
$n_{max}$ = maxarg$_{ni}$nextL($T_{ni}$)
while (nextR($T_q$) <nextL($T_{nmax}$))
    advance($T_q$)
if (nextL($T_q$) <nextL($T_{nmin}$)) return q
else return $n_{min}$
```

Procedure cleanStack(S, actL)

```
while (!empty(S)$\wedge$(topR(S) < actL))
    pop(S)
```

Example 4.4 Consider again the query of Example 4.3, which is the sub-twig rooted at the author node of the twig pattern in Fig. 4.9b and the XML database tree in Fig. 4.10. Before Algorithm TwigStack pushes an author node on the stack S_{author}, it ensures that this author node has (1) a descendant fn node in the stream T_{fn} (which in turn has a descendant jane node in T_{jane}), and (2) a descendant ln node in the stream T_{ln} (which in turn has a descendant doe node in T_{doe}). Thus, only one of the three author nodes (corresponding to the third author) from the XML data tree in Fig. 4.10 is pushed on the stacks. Subsequent steps ensure that only one solution to each of the two paths of this query, author-fn-jane and author-ln-doe, is computed. Finally, the merge join phase computes the desired answer.

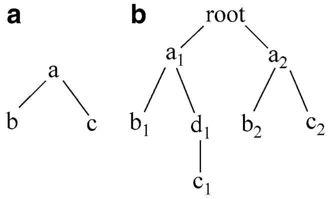


Fig. 4.11 An example to illustrate getNext algorithm (**a**) query (**b**) document

4.3.2.2 Analysis of TwigStack

Here we discuss the correctness of algorithm TwigStack for processing twig queries, and then we analyze its complexity. Most of the proofs in this section are omitted for lack of space and can be found in the paper by Bruno et al. [BKS02].

Definition 4.1 Consider a twig query Q. For each node $q\in$ subtreeNodes(Q), we define the head of q, denoted h_q, as the first element in T_q that participates in a solution for the sub-query rooted at q. We say that a node q has a minimal descendant extension if there is a solution for the sub-query rooted at q composed entirely of the head elements of subtreeNodes (q).

Proposition 4.1, based on Fig. 4.11, is important for establishing the following lemma:

Lemma 4.2 *Suppose that for an arbitrary node q in the twig query, we have that* getNext(q) $= q_N$. *Then, the following properties hold*:

1. *q_N has a minimal descendant extension.*
2. *For each node $q'\in$ subtreeNodes(q_N), the first element in $T_{q'}$ is $h_{q'}$.*
3. *Either* (a) $q = q_N$ *or* (b) parent(q_N) *does not have a minimal right extension because of q_N* (*and possibly other nodes*). *In other words, the solution rooted at $p =$* parent(q_N) *that uses h_p does not use h_q for node q but some other element whose L component is larger than that of h_q.*
4. *Using the lemma above, we can prove the paper by Bruno et al.* [BKS02] *that when some node q_N is returned by getNext, h_{q^N} is guaranteed to have a descendant extension in* subtreeNodes(q_N). *We can also prove that any element in the ancestors of that uses h_{q^N} in a descendant extension was returned by getNext before h_{q^N}. Therefore, we can maintain, for each node q in the query, the elements that are part of a solution involving other elements in the streams of* subtreeNodes(q). *Then, each time that $q_N =$* getNext(q) *is a leaf node, we output all solutions that use h_{q^N}.*

Theorem 4.7 *Given a query twig pattern q and an XML database D, Algorithm TwigStack correctly returns all answers for q on D.*

Proof [*Sketch*] In Algorithm TwigStack, we repeatedly find getNext(q) for query q (line 2). Assume that getNext(q) $= q_N$. Let A_{q^N} be the set of nodes in the query that axe ancestors of q_N. We know that getNext already returned all elements from the streams of nodes in A_{q^N} that are part of a solution that uses h_{q^N}. If $q \neq q_N$, in Line 4 we pop from parent(q_N)'s stack all elements that are guaranteed not to participate in any new solution. After that, in Line 5 we test whether h_{q^N} participates in a solution. We know that q_N has a descendant extension by Lemma 4.2, property 1. If $q \neq q_N$ and parent(q_N)' stack is empty, node q_N does not have an ancestor extension. Therefore, it is guaranteed not to participate in any solution, so we advance q_N in Line 11 and continue with the next iteration. Otherwise, node q_N has both ancestor and descendant extensions, and therefore it participates in at least one solution. We then clean q_N's stack (line 6) and push h_{q^N} to it (line 7). Finally, if q_N is a leaf node, we output the stored solutions from the stacks (lines 8–10).

While correctness holds for query twig patterns with both ancestor–descendant and parent–child edges, we can prove optimality only for the case where the query twig pattern has only ancestor–descendant edges. The intuition is simple. Since we push into the stacks only elements that have both a descendant and an ancestor extension, we are guaranteed that no element that does not participate in any solution is pushed into any stack. Therefore, the merge post-processing step is optimal, and we have the following result.

Theorem 4.8 *Consider a query twig pattern q with n nodes, and only ancestor–descendant edges, and an XML database D. Algorithm TwigStack has worst-case I/O and CPU time complexities linear in the sum of sizes of the n input lists and the output list. Further, the worst-case space complexity of Algorithm TwigStack is the minimum of* (*1*) *the sum of sizes of the n input lists and* (*2*) *n times the maximum length of a root-to-leaf path in D.*

It is particularly important to note that, for the case of query twigs with ancestor–descendant edges, the worst-case time complexity of Algorithm TwigStack is independent of the sizes of solutions to any root-to-leaf path of the twig.

4.3.3 TwigStackList

4.3.3.1 Limitations

As is introduced, TwigStack uses a chain of linked stacks to compactly represent partial results of individual query root-to-leaf paths, the approach is I/O and CPU optimal among all sequential algorithms that read the entire input for twigs with only ancestor–descendant edges, but for queries with parent–child edges, TwigStack can't control the size of intermediate results.

To get the better understanding of this limitation, Lu et al. [LCL04] experimented with TreeBank dataset which was downloaded from University of Washington XML repository [BKS02]. They use three twig query patterns (as shown in Table 4.1), each of which contains at least one parent–child edge. TwigStack operates two steps: (1) a list of intermediate path solutions is output as intermediate results and (2) the

Table 4.1 Number of partial path solutions produced by TwigStack against TreeBank data

Query	Partial paths	Merge joinable paths	Percentage of useless paths (%)
VP[/DT]//PRP_DOLLAR_	10,663	5	99.9
S[/JJ]/NP	70,988	10	99.9
S[//VP/IN]//NP	702,391	22,565	96.8

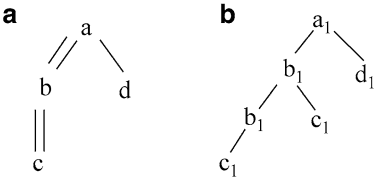


Fig. 4.12 The incorrect merge of two procedures (**a**) query twig pattern, (**b**) an XML tree

intermediate path solutions in the first step are merge-joined to produce the final solutions. Table 4.1 shows the numbers of intermediate path solutions output in the first step and the merge joinable paths among them in the second step. An immediate observation from the table is that TwigStack outputs too many partial paths that are not merge joinable. For all three queries, more than 95 % partial paths produced by TwigStack in the first step are "useless" to final answers.

To see an example, if we evaluate the twig pattern in Fig. 4.12a on the XML document in Fig. 4.12b, TwigStack will push a_1 into the stack and output all root-leaf path solutions: (a_1, b_1, c_1), $(a_1, b_1, c_2), \ldots, (a_1, b_{n-1}, c_n)$,$(a_1, b_n, c_n)$, because they match path *a//b//c*. Notice that in this example, there is no match at all! But TwigStack outputs $4n$ "useless" intermediate path solutions.

The main problem of TwigStack is that it only considers ancestor–descendant property between nodes. The level information of nodes, on the other hand, is not sufficiently exploited. As an illustration, see the query and data in Fig. 4.12a, b again. Since $<a, d>$ edge in the twig pattern is the parent–child relationship, node a_1 in the document contributes to the final answer only if a_1 has a child with name *d*. But TwigStack pushes a_1 into the stack only because a_1 has a descendant (not a child) with tag *d*. Thus, this algorithm outputs a large size of intermediate paths. Here, Lu et al. [LCL04] proposed a new holistic twig join algorithm, called TwigStackList; this method pushes a$_1$ into the stack only if a_1 or its descendant (with tag *a*) has a child with name *d*. In the document of Fig. 4.12b, although a_1 has many descendants with tag *d*, none of them has a child with tag *d*. Thus, this method does not push a_1 into the stack and thereby avoid outputting the "useless" intermediate path solutions.

In this section, we will introduce the algorithm of TwigStackList, which has the same performance as TwigStack for query patterns with only ancestor–descendant edges, but the TwigStackList is significantly more efficient than TwigStack for queries with the presence of parent–child edges. The main technique of TwigStackList is to make use of two data structures: stack and list for each node in query twigs.

A chain of linked stacks is used to compactly represent partial results of individual query root-leaf paths. The TwigStackList look-ahead reads some elements in input data streams and caches limited number of them in the list. The number of elements in any list is bounded by the length of the longest path in the XML document. The elements in lists help us to determine whether an element possibly contributes to final answers.

4.3.3.2 Algorithm of TwigStackList

In the algorithm, the function children(n) gets all child nodes of n, and PCRchildren(n), ADRchildren(n) returns child nodes which has the parent–child or ancestor–descendant relationship with n in the query twig pattern, respectively. That is, PCRchildren(n)$\cup$ADRchildren(n) = children(n). “node” refers to a tree node in the twig pattern (e.g., node n), while “element” refers to an element in the dataset involved in a twig join (e.g., element e).

There is a data stream T_n associated with each node n in the query twig. We use C_n to point to the current element in T_n. Function end(C_n) tests whether C_n is at the end of T_n. We can access the attribute values of C_n by C_n.start, C_n.end, and C_n.level. The cursor can be forwarded to the next element in T_n with the procedure advance(T_n). Initially, C_n points to the first element of T_n. This join algorithm will make use of two types of data structure: list and stack. Given a query twig, we associate a list L_n and a stack S_n for every node n in the twig, as shown in Fig. 4.10.

The use of stack in this algorithm is similar to that in TwigStack. For each list L_n, we declare an integer variable, say, p_n, as a cursor to point to an element in the list L_n; use L_n.elementAt(p_n) to denote the element pointed by p_n; and access the attribute values of L_n.elementAt(p_n) by L_n.elementAt(p_n).start, L_n.elementAt(p_n).end, and L_n.elementAt(p_n).level. At every point during computation, elements in each list L_n are strictly nested from the first to the end, that is, each element is an ancestor of the element following it. The operations over list L_n are delete(p_n) and append(e). The first operation deletes L_n.elementAt(p_n) in list L_n, and the last operation appends element at the end of L_n.

Algorithm TwigStackList, which computes answers to a query twig pattern, is presented in Algorithm 4.7. This algorithm operates in two phases. In the first phase (line 1–11), it repeatedly calls the getNext algorithm with the query root as the parameter to get the next node for processing. We output solutions to individual query root-to-leaf paths in this phase. In the second phase (line 12), these solutions are merge-joined to compute the answer to the whole query twig pattern.

Algorithm 4.7: TwigStackList

```
1.while !end() do
2.   $n_{act}$ = getNext(root)
3.   if (!isRoot($n_{act}$)) then
4.       cleanParentStack($n_{act}$, getStart($n_{act}$))
5.   end if
```

```
    if (isRoot($n_{act}$) ∨ !empty($S_{parent(nact)}$)) then
        clearSelfStack($n_{act}$, getEnd($n_{act}$))
        moveToStack ($n_{act}$, S $_{nact}$, pointertotop($S_{parent(nact)}$))
        if (isLeaf($n_{act}$)) then
            showSolutionsWithBlocking(S $_{nact}$,1)
            pop(S $_{nact}$)
        end if
        else
            proceed($n_{act}$)
    end if
end while
mergeAllPathSolutions()
```

Function end()

```
return ∀ $n_i$∈subtreeNodes(n):isleaf($n_i$) ∧ end($C_{ni}$))
```

Procedure moveToStack(n, S_n, p)

```
push (getElement(n),p) to stack $S_n$
proceed(n)
```

Procedure clearParentStack(n, actStart)

```
while (!empty($S_{parent(nact)}$)∧(topEnd($S_{parent(nact)}$) < actStart)) do
    pop($S_{parent(nact)}$)
end while
```

Procedure clearSelfStack(n, actEnd)

```
while (!empty(Sn)∧(topEnd(Sn) < actEnd)) do
    pop($S_n$)
end while
```

getNext(*n*) is a procedure called in the main algorithm of TwigStackList; it is explained as follows:

Algorithm 4.8: getNext(n) (in TwigStackList)

```
if isLeaf(n) return n
for all node $n_i$ in children(n) do
    $g_i$ = getNext($n_i$)
    if ($g_i \neq n_i$) return $g_i$
end for
$n_{max}$ = $\text{maxarg}_{ni\,\in chidren(n)}$ getStart($n_i$)
 $n_{min}$ = $\text{minarg}_{ni\in chidren(n)}$ getStart($n_i$)
while ( getEnd(n) < getStart($n_{max}$))    proceed(n)
    if ( getStart(n) > getStart($n_{min}$))    return $n_{min}$
MoveStreamToList(n, $n_{max}$)
for all node $n_i$in children(n) do
    if (there is an element $e_i$ in list $L_n$ such that $e_i$ is the parent of
       getElement($n_i$)) then
```

```
13.           if ($n_i$ is the only child of n) then
14.              move the cursor $p_n$ of list $L_n$ to point to $e_i$
15.           end if
16.        else
17.           return $n_i$
18.        end if
19.     end for
20.  return n
```

Procedure getElement(n)

```
if !empty(Ln) then
   return Ln.elementAt(pn)
else return Cn
```

Procedure getStart(n)
return the start attribute of getElement(n)

Procedure getEnd(n)
return the end attribute of getElement(n)

Procedure MoveStreamToList(n, g)

```
while $C_n$.start < getStart(g) do
   if $C_n$.end > getEnd(g) then
      $L_n$.append($C_n$)
   end if
   advance($T_n$)
end while
```

Procedure proceed(n)

```
if empty($L_n$) then
   advance($T_n$)
else
   $L_n$.delete($p_n$)
   $p_n = 0$ {Move $p_n$ to point to the beginning of $L_n$}
end if
```

getNext(n) returns a node n' (possibly $n' = n$) with three properties: assume that element $e_{n'}$=getElement(n'), then (1) $e_{n'}$ has a descendant e_{n^i} in each of stream T_{n^i} for $n_i \in$ children(n'), and (2) if n' is not a branching node in the query, element $e_{n'}$ has a child e_{n^i} in T_{n^i}, where $n_i \in$ PCRchildren(n') (if any), and (3) if n' is a branching node, there is an element e_{n^i} in each T_{n^i} such that there exists an element e_i (with tag n) in the path from $e_{n'}$ to $e_{n^{\max}}$, that is, the parent of e_{n^i}, where $n_i \in$ PCRchildren(n') (if any) and $e_{n^{\max}}$ has the maximal start attribute for all children(n').

At Lines 2–5, in Algorithm getNext, we recursively invoke getNext for each $n_i \in$ children(n). If any returned node g_i is not equal to n_i, we immediately return g_i (line 4). Otherwise, we will try to locate a child of n which satisfies the above three properties. Lines 6 and 7 get the max and min elements for the current head elements in lists or streams, respectively. Line 8 skips elements that do not contribute to results. If no common ancestor for all C_{n^i} is found, Line 9 returns the child node with the smallest start value, that is, $g_{\min}$.

Line 10 is an important step. Here we look-ahead read some elements in the stream T_n and cache elements that are ancestors of $C_{n^{\max}}$ into the list L_n. Whenever any element n_i cannot find its parent in list L_n for $n_i \in \text{children}(n')$, algorithm getNext returns node n_i (in Line 17). Note that this step manifests the key difference between TwigStackList and the previous algorithms (i.e., TwigStack). In this scenario, the previous ones return n instead of n_i, which may result in many "useless" intermediate paths. But this algorithm adopts a clever strategy: return n_i that has no parent in list L_n, since we make sure that n_i does not contribute to final results involved with the elements in the remaining parts of streams. Finally, if n is not a branching node, in Line 14, we need to move the cursor in the list L_n to point to the parent of getElement(n_i).

The main difference between two getNext algorithms in TwigStack and TwigStackList can be summarized as follows. In TwigStack, getNext(n) return n' if the head element $e_{n'}$ in stream $T_{n'}$ has a descendant e_{n^i} in each stream T_{n^i}, for $n_i \in \text{children}(n')$ (which is the same as the first property as mentioned above), but TwigStackList needs n' to satisfy three properties, illustrated as follows.

Example 4.5 Consider a query twig pattern $a[/b]/c$ on a dataset visualized in Fig. 4.11. A subscript is added to each element in the order of their start values for easy reference. Initially, the three elements are (a_1, b_1, c_1). The first call of getNext(root) returns node c, because element c_1 cannot find parent with tag a in the path from a_1 to it. But in this scenario, the first call of getNext(root) of TwigStack would return a_1, since a_1 has two descendants b_1 and c_1 in stream b and c, respectively. Because TwigStack returns a_1 instead of c_1, in the main algorithm, TwigStack will output the useless path solution <a_1, b_1>. Further, the second call of getNext(root) in TwigStackList returns b_1. In addition, the cursor of node a is forwarded to a_2. Right before the third call, TwigStackList reaches a cursor setup (a_2, b_2, c_2), which is actually the match of the query.

The main algorithm shows the main algorithm of TwigStackList. It repeatedly calls getNext(root) to get the next node n to process, as described next.

First of all, Line 2 calls getNext algorithm to identify the node n_{act} to be processed. Lines 4 and 7 remove partial answers from the stacks of parent(n_{act}) and n_{act} that cannot be extended to total answer. If n is not a leaf node, we push element getElement(n_{act}) into $S_{n^{\text{act}}}$ (Line 8); otherwise (Line 10), all path solution involving getElement(n_{act}) can be output. Note that path solutions should be output in root-leaf order so that they can be easily merged together to form final twig matches (Line 17). As a result, we may need to block some path solutions during output.

```
......
3.  if (!isRoot($n_{act}$) then
4.      cleanStack(parent($n_{act}$, getStart($n_{act}$))
```

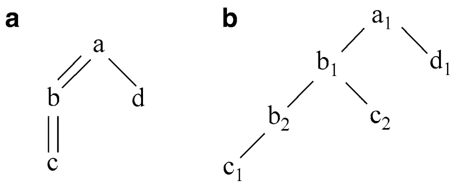


Fig. 4.13 An example to show the incorrectness of codes in Fig. 4.12. (**a**) Query twig pattern, (**b**) an XML tree

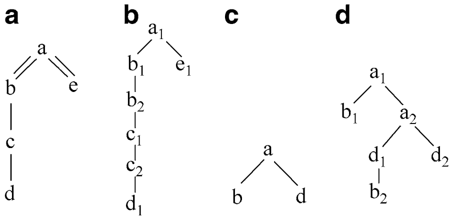


Fig. 4.14 Two examples to illustrate the benefits of Algorithm TwigStackList. (**a**) Q_1, (**b**) Doc1, (**c**) Q_2, (**d**) Doc2

```
endif
if (isRoot(($n_{act}$) ∨ !empty(Sparent$_{(nact)}$)) then
    cleanStack(($n_{act}$ , getStart(($n_{act}$))
......
```

Procedure cleanStack(n, actStart)

```
while (!empty($S_n$)∧(topEnd($S_n$)<actStart)) do
    pop($S_n$)
```

It is not correct to merge cleanParentStack and cleanSelfStack into one procedure cleanStack as Fig. 4.12. Consider a twig query *a*[*//b//c*]*/d* and a document in Fig. 4.13. Suppose the four elements are initially at (a_1, b_1, c_1, d_1). At the first call of getNext, node *a* is returned. At this point, note that the current element pointed by cursor p_b in list L_b is b_2, instead of b_1 (recall, Line 14 in Algorithm getNext). Then the next two calls of getNext return *b*, *c* once to consume b_1 and c_1. After that, the current elements are (b_1, c_2, d_1) (stream a has finished). The next call of getNext will return node b again. Here, if we used the algorithm shown in Fig. 4.12, then b_2 would not be popped from stack. Then, the property of stack (i.e., the upper element should be the descendant of the lower one) would not be hold.

Compared to the previous algorithm TwigStack, the benefit of the new algorithm TwigStackList can be illustrated with the following two examples.

Example 4.6 Consider the query twig pattern Q_1 in Fig. 4.14a and Doc1 in Fig. 4.14b. Initially, the first call of getNext() in TwigStack returns node *a*, but the first call of getNext() in TwigStackList returns node *c*. As a result, unlike Algorithm TwigStack, TwigStackList does not output the intermediate result (a_1, e_1), which does not contribute to any final answers.

Example 4.7 Consider Q_2 in Fig. 4.14c and Doc$_2$ in Fig. 4.14d. Initially, the first call of getNext in TwigStack and TwigStackList returns node *a*. After the second call of getNext, (a_1, b_1) is output as intermediate results in both TwigStack and TwigStackList. Subsequently, the third call of getNext in TwigStack returns node a again. But the third call of getNext in TwigStackList returns node b, since b_2 has not the parent in the stream T_a. Thus, unlike TwigStack, TwigStackList does not output the intermediate result (a_2, d_1) and (a_2, d_2), which do not contribute to any final answers.

Example 4.6 illustrates the fact that, in TwigStackList, when twig patterns contain only ancestor–descendant relationships below branching nodes, each solution to individual query root-leaf path is guaranteed to be merge joinable with at least one solution to each of the other root-leaf paths. On the other hand, Example 4.7 illustrates another fact that even if there exist parent–child relationships below branching nodes, TwigStackList is still superior to TwigStack in that it output less useless intermediate solutions.

4.3.3.3 Analysis of TwigStackList

Here, we discuss the correctness of Algorithm TwigStackList, and then we analyze its complexity. Finally, we compare TwigStackList with TwigStack in terms of the size of intermediate results.

Definition 4.2 (Head Elemente_n) In TwigStackList, for each node n in the query twig pattern, if the list L_n is not empty, then we say that the element indicated by the cursor p_n of lnL_n is the head element of n, denoted by e_n. Otherwise, we say that element C_n in the stream T_n is the head element of n.

Definition 4.3 (Child and Descendant Extension) We say that a node n has the child and descendant extension if the following three properties hold:

1. For each $n_i \in$ ADRchildren(n), there is an element e_{ni} (with tag n_i) which is a descendant of e_n.
2. For each $n_i \in$ PCRchildren(n), there is an element ei(with tag n) in the path from en to enmax such that e_i is the parent of e_{ni}, where $e_{n^{\max}}$ has the maximal start attribute value for all head elements of child nodes of n.
3. Each of the children of n has the child and descendant extension.

The above two definitions are important to establish the correctness of the following lemma.

Lemma 4.3 *Suppose that for an arbitrary node n in the twig query, we have* getNext(n) $= n'$. *Then the following properties hold*:

1. *n' has the child and descendant extension.*
2. *Either* (a) $n = n'$ *or* (b) parent(n) *does not have the child and descendant extension because of n'* (*and possibly a descendant of n'*).

Using Lemma 4.3, we can prove the following lemma.

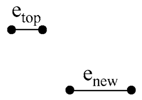
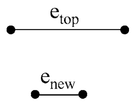
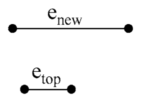
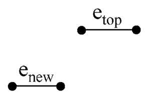
	Case (i)	Case (ii)	Case (iii)	Case (iv)
Property	$e_{\text{top}}\cdot\text{end} < e_{\text{new}}\cdot\text{start}$	$e_{\text{top}}\cdot\text{start} < e_{\text{new}}\cdot\text{start}$ $e_{\text{top}}\cdot\text{end} > e_{\text{new}}\cdot\text{end}$	$e_{\text{top}}\cdot\text{start} > e_{\text{new}}\cdot\text{start}$ $e_{\text{top}}\cdot\text{end} < e_{\text{new}}\cdot\text{end}$	$e_{\text{new}}\cdot\text{end} < e_{\text{top}}\cdot\text{start}$
Segment				

Fig. 4.15 Illustration to the proof of Lemma 4.5

Lemma 4.4 *Suppose* getNext(n) $= n'$ *returns a query node* $n'(n' \neq n)$ *in the Line 17 of Algorithm getNext. If the stack is empty, then the head element does not contribute to any final solutions.*

Proof [*Sketch*] Suppose that on the contrary, there is a solution using the head element. In Line 10 of algorithm getNext, we insert all elements with the name parent(n'), which are in the path from eparent(n') to enmax into the list Lparent(n'). According to Line 12, if the parent of en' is not in Lparent(n'), then using our hypothesis, we know that parent(en') also participates in the final solution. But using Lemma 4.3, we see that this is a contradiction, since the start attribute of parent(en') is less than that of eparent(n') and the stack Sparent(n') is empty.

Lemma 4.5 *At every point during computation of Algorithm TwigStackList, elements in each stack* S_n *are strictly nested, that is, each element is a descendant of the element below it.*

Proof This lemma is obvious in the previous TwigStack. But since Algorithm TwigStackList may change the cursor of the list, this lemma is nontrivial. In TwigStackList, we can insert elements into the stack only in Procedure moveToStack. There are four cases for relationship between the new element e_{new} to be pushed into stack and the existing top element e_{top} in stack (see Fig. 4.15).

Case (i): Since e_{top}.end $<$ e_{new}.end, the element e_{top} will be popped in Procedure cleanSelfStack. So this case is impossible.

Case (ii): In this case, e_{new} will be added into the stack safely.

Case (iii): Similar to case (i), since e_{top}.end $<$ e_{new}.end, the element e_{top} will be popped. We also ensure that e_{top} cannot participate in final answers any longer.

Case (iv): This case is impossible. Because in Algorithm TwigStackList, we can change the cursor of a list only in Line 14 of getNext. The new element indicated by the cursor is guaranteed to be a descendant of the previous one.

Therefore, this lemma holds in all cases.

Lemma 4.6 *In TwigStackList, any element that is popped from the stack* S_n *does not participate in any new solution any more.*

Proof Any element is popped from stack S_n in either Procedure cleanParentStack or cleanSelfStack. In the following, we prove the correctness of the lemma in these two cases respectively.

1. In cleanParentStack, suppose on the contrary, there is a new solution involving the popped element epop. According to Line 1 of cleanParentStack, epop.end<actStart, where actStart is the start attribute of the head element of parent(*n*) (i.e., eparent(*n*)). Using the containment property, epop cannot be contained by any element in the path from the root to eparent(*n*) and after eparent(*n*), which is a contradiction.
2. In cleanSelfStack, using the containment property, we see that cleanSelfStack pops elements that are descendants of en, where *en* is the head element of node *n*. The popped element does not participate in new answers any more. This is because, at this point, *n* has only one child with parent–child relationship. Thus, the start value of any child of epop is less than that of the head element of node children(*n*). Thus, there is no element that is a child of epop in the remaining portion of the stream Tchild(*n*). Therefore, epop does not participate in any new solutions.

Theorem 4.9 *Given a query twig pattern q and an XML database D, Algorithm TwigStackList correctly returns all answers for q on D.*

Proof [*Sketch*] Using Lemma 4.4, we know that when getNext returns a query node *n* in the Line 17 of getNext, if the stack Sparent(*n*) is empty, then the head element *en* does not contribute to any final solutions. Thus, any element in the ancestors of *n* that use *en* in the descendant and child extension is returned by getNext before *en*. By using Lemma 4.5 and Lemma 4.6, we can maintain, for each node *n* in the query, the elements that are involve in the root-leaf path solution in the stack S_n. Finally, each time that $n =$ getNext(root) is a leaf node, we output all solutions that use *en*.

While correctness holds for query twig patterns with both ancestor–descendant and parent–child relationships in any edges, we can prove optimality only for the case where parent–child relationships appear only in edges below non-branching nodes. The intuition is that we push into stacks only elements that have the child and descendant extension. If there is a parent–child relationship below the non-branching node, according to Lemma 4.3, we are guaranteed that e_n is pushed into stack only if e_n has a child element in the stream $T_{\text{child}(n)}$. Therefore, we have the following result.

Theorem 4.10 *Consider a query twig pattern with m nodes, and there are only ancestor–descendant relationships below branching nodes (in other words, this pattern may have parent–child relationships below non-branching nodes) and an XML database D. Algorithm TwigStackList has the worst-case I/O complexities linear in the sum of sizes of the m input lists and the output list.*

Since the worst-case size of any stack and list in TwigStackList is proportional to the maximal length of a root-leaf path in the XML database, we have the following results about the space complexity of TwigStackList.

Theorem 4.11 *Consider a query twig pattern with m nodes and an XML database D. The worst-case space complexity of Algorithm TwigStackList is proportional to m times the maximal length of a root-leaf path in D.*

It is particularly important to note that, even for the case where query patterns contain parent–child relationships below branching nodes, Algorithm TwigStackList usually outputs much less intermediate path solutions than TwigStack. The reason is simple. TwigStackList pushes any element into stack that has both descendant and child extension, but TwigStack pushes any element that has only the descendant extension into the stack. Therefore, TwigStackList pushes fewer elements that do not contribute to final answers to the stack and thereby output less intermediate results.

4.3.4 TJFast

Many algorithms, proposed to match twig patterns, always (1) first, develop a labeling scheme to capture the structural information of XML documents and then (2) perform twig pattern matching based on the labels alone without traversing the original XML documents.

Various methods have been proposed that are based on tree-traversal order (e.g., extended preorder [LM01]), textual positions of the start and end tags (e.g., region encoding [BKS02]), path expressions (e.g., Dewey ID [TVB+02], PID [BG03]), or prime numbers (e.g., [WLH04]). By applying these labeling schemes, one can determine the relationship (e.g., ancestor–descendant) between two elements in XML documents from their labels alone. However, information contained by a single label is very limited. As introduced in Chap. 2, in which extended Dewey ID is proposed, we can derive the names of all elements in the path from the root to this element, from the label of an element alone. Based on extended Dewey labeling scheme, Lu et al. [LLC+05] develop a novel holistic twig join algorithm to match twig patterns, called TJFast (i.e., a Fast Twig Join Algorithm). The algorithm only scans elements for query *leaf* nodes. This feature brings us two immediate benefits: (1) TJFast typically access much fewer elements than algorithms based on region encoding, and (2) TJFast can efficiently process queries with wildcards in internal nodes.

Some data structures and notations used in this section are introduced as follows:

Let q denote a twig pattern and p_n denote a path pattern from the root to the node $n \in q$ and also denote some functions: isleaf: Node→Bool, isBranching: Node→Bool, leafNodes: Node→{Node}, and directBranchingOrLeafNodes: Node→{Node}. leafNodes(n) returns the set of leaf nodes in the twig rooted with n. directBranchingOrLeafNodes(n)(for short, dbl(n)) returns the set of all branching nodes b and leaf nodes f in the twig rooted with n such that in the path from n to b or f (excluding n, b, or f), there is no branching nodes. For example, in the query Q_1 of Fig. 4.16, dbl(a) = {b, c} and dbl(c) = {f, g}.

Associated with each leaf node f in a query twig pattern, there is a stream T_f. The stream contains extended Dewey labels of elements that match the node type f. The elements in the stream are sorted by the ascending lexicographical order. For example, "1.2" precedes "1.3" and "1.3" precedes "1.3.1."

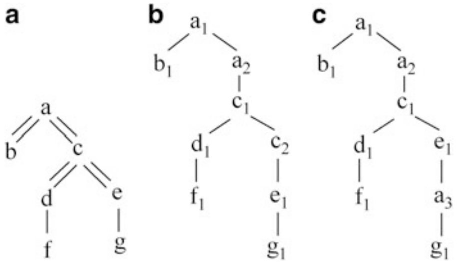


Fig. 4.16 Example twig query and documents. (**a**) Q1 (**b**) Doc1 (c) Doc2

The operations over a stream T_f include current(T_f), advance(T_f), and eof(T_f). The function current(T_f) returns the extended Dewey label of the current element in the stream T_f. The function advance(T_f) updates the current element of the stream T_f to be its next element. The function eof(T_f) tests whether we are in the end of the stream T_f. We make use of two self-explanatory operations over elements in the document: ancestors(e) and descendants(e), which return the ancestors and descendants of e, respectively (both including e). Algorithm TJFast keeps a data structure during execution: a set S_b for each branching node b. Each two elements in set S_b have an ancestor–descendant or parent–child relationship. So the maximal size of S_b is no more than the length of the longest path in the document. Each element cached in sets likely participates in query answers. Set S_b is initially empty.

4.3.4.1 Algorithm of TJFast

Algorithm TJFast, which computes answers to a query twig pattern q, is presented in Algorithm 4.9. TJFast operates in two phases. In the first phase (Lines 1–9), some solutions to individual root-leaf path patterns are computed. In the second phase (Line 10), these solutions are merge-joined to compute the answers to the query twig pattern.

Given the extended Dewey label of an element, according to the Ancestor Name Vision property, it is easy to check whether its path matches the individual root-leaf path pattern. Thus, the key issue of TJFast is to determine whether a path solution can contribute to the solutions for the whole twig. In the optimal case, we only output the path solution that is merge joinable to at least one solution of other root-leaf paths. Intuitively, if two path solutions can be merged, the necessary condition is that they have the common element to match the branching query node.

For example, consider a simple query $a[./b]/c$ and two path solution (a_1, b_1) and (a_2, c_1). Observe that two solutions can be merged only if $a_1 = a_2$. Therefore, in TJFast, in order to determine whether a path solution contributes to final answers, we try to find the most likely elements that match branching nodes b and store them in the corresponding set S_b.

Algorithm 4.9: TJFast

```
1. for each f∈leafNodes(root)
2.     locateMatchedLabel(f)
3. end for
4. while (!end(root)) do
5.     $f_{act}$ = getNext(topBranchingNode)
6.     outputSolutions($f_{act}$)
7.     advance($T_{fact}$)
8.     locateMatchedLabel($f_{act}$)
9.  end while
10. mergeAllPathSolutions()
```

Procedure locateMatchedLabel(f)

```
/* Assume that the path from the root to element */
/* get($T_f$) is $n_1/n_2/\ldots/n_k$ and $p_f$ denotes the path pattern */
/* from the root to leaf node f */
1. while !(( $n_1/n_2/\ldots/n_k$ matches pattern $p_f$)∧($n_k$ matches f)) do
2.     advance($T_f$)
3. end while
```

Function end(n)

```
return∀f∈leafNodes(root)→eof($T_f$)
```

Procedure outputSolutions(f)

```
Output path solutions of current($T_f$) to pattern $p_f$ such that in each solution s,
∀e∈s:(element e matches a branching node b→e∈$S_b$)
```

It is not difficult to understand the main procedure of TJFast (see Algorithm 4.9). In Lines 1–3, for each stream, we use Procedure locateMatchedLabel to locate the first element whose path matches the individual root-leaf path pattern. In Line 5, we identify the next stream $T_{f_{act}}$ to be processed by using getNext(topBranchingNode) Algorithm, where topBranchingNode is defined as the branching node that is an ancestor of all other branching nodes (if any). In Line 6, we output some path matching solutions in which each element that match any branching node b can be found in the corresponding set S_b. We advance $T_{f_{act}}$ in Line 7 and locate the next matching element in Line 8. (Note that the second condition "n_k matches f" in Line 1 of locateMatchedLabel is necessary, which avoids outputting duplicate solutions. For example, consider the element e (with tag name b) with the path "$a_1/b_1/c_1/d_2$" and the path query "a/b". "$a_1/b_1/c_1/d_2$" can matches "a/b", but this solution has been output by another element ends with b_1.)

Algorithm getNext (see Algorithm 4.10) is the core function called in TJFast, in which we accomplish two tasks. The first is to identify the next stream to process, and the second is to update the sets S_b associated with branching nodes b, discussed as follows.

For the first task to identify the next processed stream, Algorithm 4.10 returns a query leaf node f according to the following recursive criteria: (1) if n is a leaf node, return n (Line 2); else (2) if n is a branching node, then for each node $n_i \in \text{dbl}(n)$, (1) if the current elements cannot form a match for the subtree rooted with n_i, we immediately return f_i (Line 7); (2) if the current element from stream T_{f^i} does not participate in the solution involving in the future elements in other streams, we return f_i (Line 14); and (3) otherwise we return $f_{\min}$ such that the current element $e_{\min}$ has the minimal label in all e_i by lexicographical order (Line 20).

For the second task, we update set e_b. This operation is important, since the elements in e_b decide which path solution can be output in Procedure outputSolutions. In Line 18 of Algorithm 4.10, before an element e_b is inserted to the set S_b, we ensure that e_b is an ancestor of (or equals) each other element e_{b^i} to match node b in the corresponding path solutions.

Example 4.8 Consider Q_1 and Doc1 in Fig. 4.16a, b. A subscript is added to each element in the order of preorder traversal for easy reference. There are three input streams T_b, T_f, and T_g. Initially, getNext(a) recursively calls getNext(b) and getNext(c) (for b, $c \in \text{dbl}(a)$ in Q_1). Since b is a leaf node in Q_1, getNext(b) = b. Observe that MB(f, c) = $\{c_1\}$ and MB(g, c) = $\{c_1, c_2\}$, so $e_{\max} = g$ and $e_{\min} = f$ in Lines 10 and 11 of Algorithm 4.10. In Line 18, c_1 is inserted to set S_c. Then, getNext(c) = f. Subsequently, a_1 is inserted to S_a and getNext(a) = b. Finally path solutions (a_1, b_1), (a_1, c_1, d_1, f_1), and (a_1, c_1, e_1, g_1) are output and merged. Note that although (a_1, c_2, e_1, g_1) matches the individual path pattern *a//c//e/g*, it is not output for $c_2 \notin S_c$.

Note that the second phase (Line 10 of Algorithm 4.9) of TJFast can be performed efficiently, only when the intermediate path solutions are output in sorted order. To achieve this purpose, we would need to “block” some answers.

Algorithm 4.10: getNext(n) (in TJFast)

```
1. if (isLeaf(n)) then
2.     return n
3. else
4.     for each $n_i \in$dbl(n) do
5.         $f_i$ = getNext($n_i$)
6.         if (isBranching($n_i$)$\wedge$empty($S_{ni}$))
7.             return $f_i$
8.         $e_i = \max\{p|p \in MB(n_i,n)\}$
9.     end for
10.    max = maxarg$_i\{e_i\}$
11.    min = minarg$_i\{e_i\}$
12.    for each $n_i \in$dbl(n) do
13.        if ($\forall e \in MB(n_i,n)$:$e \notin$ancestors($e_{max}$))
14.            return $f_i$;
15.        end if
16.    end for
17.    for each $e \in MB(n_i,n)$
```

```
18.       if (e$\in$ancestors($e_{max}$)) updateSet($S_n$, e)
19.    end for
20.    return $f_{min}$
21.end if
```

Function MB(n, b)

```
1. if (isBranching(n)) then
2.    Let e be the maximal element in set $S_n$
3. else
4.    Let e = current($T_n$)
5. end if
6. Return a set of element a that is an ancestor of e such that a can match node b in
   the path solution of e to path pattern $p_n$
```

Procedure clearSet (S, e)

Delete any element a in the set S such that a$\notin$ancestors(e) and a$\notin$descendants(e)

Procedure updateSet(S, e)

```
1. clearSet(S, e)
2. Add e to set S
```

Analysis of TJFast

Here, we show the correctness of TJFast and then analyze its complexity.

Lemma 4.7 *In Procedure clearSet of Algorithm 4.9, any element e that is deleted from set S_b does not participate in any new solution.*

Lemma 4.8 *In Line 18 of Function getNext, if element e*$\notin$ancestor($e_{\max}$) *and* $e\notin S_n$, *then e is guaranteed to not involve in any final solution.*

Lemma 4.7 shows that any element deleted from sets does not participate in new solutions, so the deletion is safe. Lemma 4.8 shows that for any element *e* that matches a branching node, if *e* participates in any final answer, then e occurs in the corresponding set. Thus, the insertion is complete. The two lemmas are important to establish the correctness of the following theorem.

Theorem 4.12 *Given a twig query Q and an XML database D, Algorithm TJFast correctly returns all the answers for Q on D.*

While the correctness holds for any given query, the I/O optimality holds only for the case where there are only ancestor–descendant relationships between branching nodes and their children.

Theorem 4.13 *Consider an XML database D and a twig query Q with only ancestor–descendant relationships between branching nodes and their children. The worst-case I/O complexity of TJFast is linear in the sum of the sizes of input and output lists. The worst-case space complexity is* $O(d^2*|b|+d*|f|)$, w*here* $|f|$ *is the number of leaf nodes in q*, $|b|$ *is the number of branching nodes in q, and d is the length of the longest label in the input lists.*

Proof [*Sketch*] We first prove the I/O optimality.
The following observation is important to prove the optimality of TJFast: if all branching edges are only ancestor–descendant relationships, then in Line 18 of getNext, since $e \in \text{ancestor}(e_{\max})$, $e \in \text{MB}(n_i, n)$ for each $n_i \in \text{dbl}(n)$. That is, e is guaranteed to be a common element in each current path solution. Note that we only output path solutions, in which elements that match branching nodes occur in the corresponding set (Line 6 of Algorithm 4.9). Therefore, each intermediate path solution output in TJFast is guaranteed to contribute to final results when the query contains only ancestor–descendant relationships in branching edges.

As for space complexity, our result is based on the observation that in the worst case, the number of elements in branching node set S_b is at most d, where d is the length of the longest label in the input lists. Considering each extended Dewey label repeats its prefix, the total space complexity of S_b is $O(d^2)$.

Theorem 4.13 holds only for query with ancestor–descendant relationships to connect branching nodes. Unfortunately, in the case where the query contains parent–child relationships between branching nodes and their children, Algorithm TJFast is no longer guaranteed to be I/O optimal. For example, consider a query $a[./b]/c$ and a data tree consisting of a_1, with children(in order) b_1, a_2, c_2 such that a_2 has children b_2, c_1. There are two streams T_b, T_c in TJFast and their first elements are b_1 and c_1, respectively. In this case, b_1 and c_1 are "locked" simultaneously, because we cannot advance any stream before knowing if it participates in a solution. Thus, optimality can no longer be guaranteed.

4.3.4.2 Comparison Among TJFast, TwigStack, and TwigStackList

In this section, we use the following example to illustrate the advantages of TJFast over TwigStack and TwigStackList.

Example 4.9 Consider the query and data tree Doc2 in Fig. 4.16a, c. There are three input streams T_b, T_f, and T_g in TJFast. Initially, the current elements are b_1, f_1, and g_1. TJFast does not insert c_1 to set S_c, since by reading the label of g_1 alone, we immediately identify that g_1 does not contribute to query answers (for $a_1/a_2/c_1/e_1/a_3/g_1$ does not match $a//c//e/g$). In contrast, TwigStack pushes c_1 to stack S_c and outputs two "useless" intermediate path solution $<a_1, b_1>$ and $<a_1, c_1, d_1, f_1>$. The behavior of TwigStack is also reasonable because based on region coding of g_1, one cannot decide whether g_1 has the parent tagged with e. But based on extended Dewey, one can easily identify that the parent of g_1 is tagged with a rather than e. This example shows the benefit of extended Dewey labeling scheme on efficient processing of XML twig pattern matching.

Compared to TwigStack, TwigStackList looks more "clever." In the above example, TwigStackList does not hastily push c_1 to stack, but first checks the parent–child relationship between e_1 and g_1. Then they find that e_1 is not the parent of g_1.

Table 4.2 XML datasets (XM: XMark, TB: TreeBank)

	XM	Random	DBLP	TB
Data size(MB)	582	90	130	82
Nodes (million)	8	5.1	3.3	2.4
Maz/Avg depth	12/5	10/5.1	6/2.9	36/7.8

Then TwigStackList caches e_1 in a list and reads more elements in T_e. In this simple case, e_1 is the only element in stream T_e. So unlike TwigStack, TwigStackList does not output any useless intermediate results. Compared to TJFast, TwigStackList is also I/O optimal in this example, but TwigStackList needs to read more elements from all non-leaf node streams, and its processing will be very complicated when g_1 has more than one ancestor tagged with e.

4.3.5 Experimental Evaluation

Here, we will use some experiments to evaluate the algorithms mentioned above: PathStack, TwigStack, TwigStackList, and TJFast.

4.3.5.1 Experimental Setup

We implemented the join algorithms in JDK 1.4 using the file system for storage. Only TJFast is based on extended Dewey labeling scheme, and the other three use region encoding. All experiments were run on a 1.7-G Pentium IV processor running Windows XP with 768 MB of main memory and 2 GB of disk space. We used four different datasets, including two synthetic and two real datasets. The first synthetic dataset is the well-known XMark benchmark data. The second is a random dataset with ten distinct labels (namely, A_1, A_2, ..., A_{10}). The node labels in the tree were uniformly distributed. The two real datasets are DBLP and TreeBank [BKS02]. We chose these two datasets since they have different characteristics: DBLP is a shallow and wide document, but TreeBank has very deep recursive structure. Table 4.2 summarizes the characteristics of the four datasets.

In the experiments, the extended Dewey labels are not stored by the dotted-decimal strings displayed (e.g., "1.2.3.4"), but rather a compressed binary representation. In particular, we used UTF-8 encoding as an efficient way to present the integer value, which was proposed by Tatarinov et al. [TVB+02]. The experimental results show that compared to the naive implementation, where each integer value is presented as a fixed number of bytes, the UTF-8 encoding can save about 50 % space cost.

Table 4.3 Labels size (XM: XMark, TB: TreeBank)

	XM	Random	DBLP	TB
Original Dewey(MB)	56.2	36.1	18.1	22.8
Region coding(MB)	71.9	45.2	21.6	23.3
Navie extension(MB)	92.9	55.8	27.7	41.9
Extended Dewey(MB)	72.6	43.3	19.5	28.7

Table 4.4 Path queries on XMark data

	Path queries
PQ_1	/site/closed_auctions/closed_aution/price
PQ_2	/site/regions/item/location
PQ_3	/site/people/person/gender
PQ_4	/site/open_auctions/open_auction/reserve

4.3.5.2 Experimental Results

Labels Size

We compared the labels sizes of four labeling schemes in Table 4.3. The first conclusion is that the size of the naive extension, which directly presents the element name sequence in number presentation ahead of the original Dewey labels, is generally larger than that of extended Dewey labeling scheme. The second conclusion is that when the document tree is shallow and wide (i.e., DBLP), the size of extended Dewey is smaller than that of region encoding. But when the document tree is deep (i.e., TreeBank), the size of region encoding is smaller. This is because extended Dewey is a variation of prefix labeling scheme, whose size is closely related to the average depth of documents. The third conclusion is that the size of extended Dewey is about 10–30 % more than that of original Dewey. As we will show in experiments, it is worth using this additional space-overhead, since it significantly improves the performance of XML twig pattern matching.

Path Queries

First, compare algorithm TJFast with the PathStack to match path queries without branching nodes. For this purpose we used XMark benchmark data and four path queries shown in Table 4.4. Figure 4.17 compares two algorithms in terms of the number of elements read, the size of disk files scanned, and execution time.

An immediate observation from the figures is that TJFast is more efficient than PathStack. In particular, PathStack could perform 400 % more disk I/Os than those required by TJFast (e.g., PQ_2).

In order to research the effect of query path length on TJFast and PathStack, we then used the random dataset consisting of ten distinct labels A_1, A_2, ..., A_{10} and

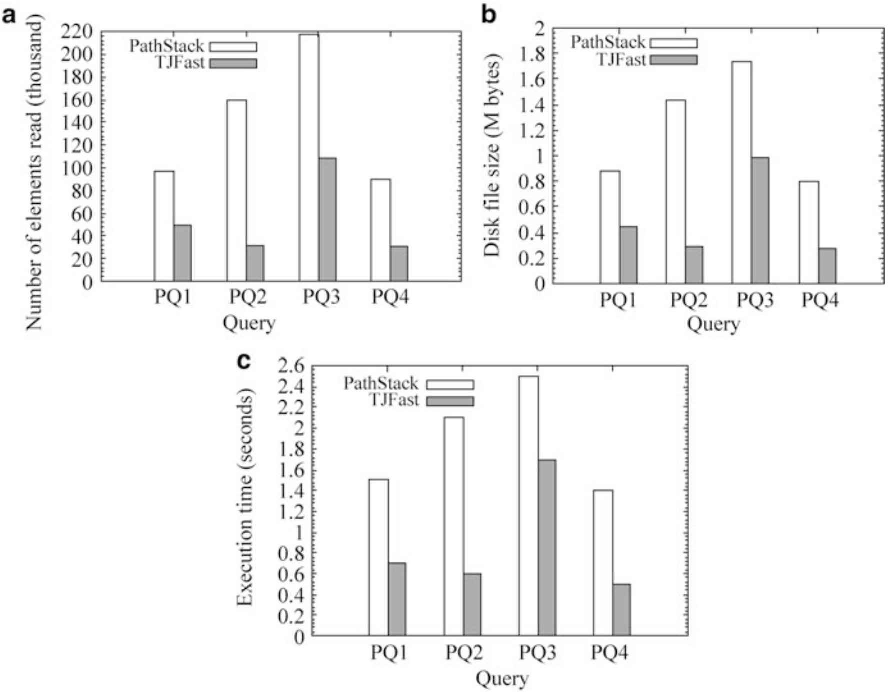


Fig. 4.17 PathStack versus TJFast using XMark data. (**a**) Number of elements read (**b**) size of disk files scanned (**c**) execution time

issue path queries of different lengths such as A_1, A_2, ..., A_{10}. Figure 4.17 shows the execution times of both techniques as well as the number of elements read and the size of disk files. Clearly, TJFast results in considerably better performance than PathStack. The performance of PathStack degrades significantly with the increase of the path length, but that of TJFast is almost not affected at all, as TJFast only scan data associated with the leaf node (Fig. 4.18).

Twig Queries

Now focus on twig queries and compare three holistic twig join algorithms TwigStack, TwigStackList, and TJFast; we tested several XML queries on DBLP and TreeBank data (see Table 4.5). These queries have different twig structures and combinations of parent–child and ancestor–descendant relationships. In particular, queries TQ_1 and TQ_2 contain only ancestor–descendant relationships, while TQ_4 contains only parent–child relationships. TQ_3 contains only ancestor–descendant relationships between the branching node and its children, while TQ_3 contains a branching node with both parent–child and ancestor–descendant relationships.

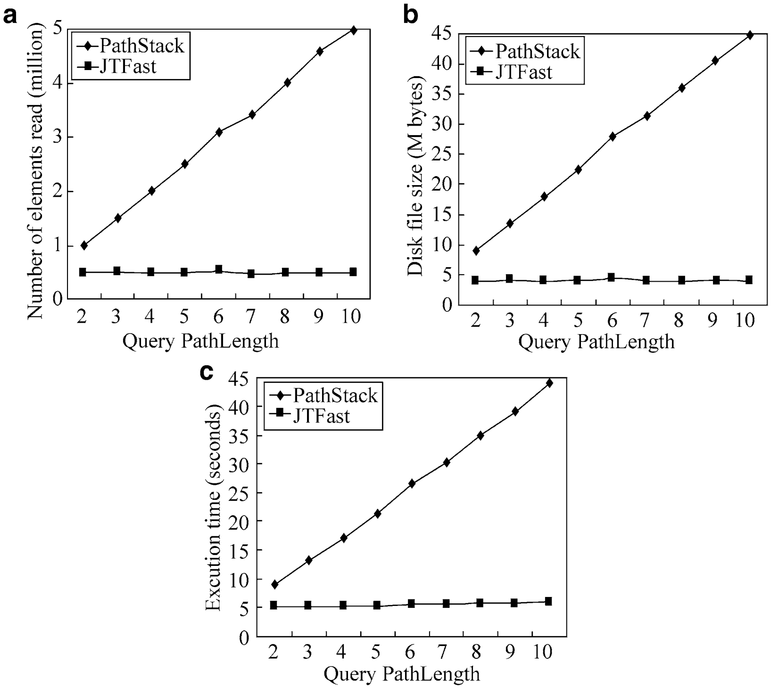


Fig. 4.18 PathStack versus TJFast using random data. (**a**) Number of elements scanned (**b**) size of disk files scanned (**c**) execution time

Table 4.5 Twig queries on DBLP and TreeBank(TB)

Twig	Data	Query
TQ$_1$	DBLP	//inproceedings//title[.//i]//sup
TQ$_2$	DBLP	//article[.//sup]//title//sub
TQ$_3$	TreeBank	/S[.//VP/IN]//NP
TQ$_4$	TreeBank	/S/VP/PP[IN]/NP/VBN
TQ$_5$	TreeBank	//VP[DT]//PRP_ DOLLAR_

TJFast Versus TwigStack

We first compare the performance between TJFast and TwigStack. From Figs. 4.19 and 4.20, we see that TJFast outperforms TwigStack for all queries. We now analyze the query performance under two scenarios, namely, the cost of disk access and the size of intermediate.

Cost of Disk Access. Figures 4.19a and 4.20a show that TJFast results.read far fewer elements than TwigStack. For example, for TQ$_1$, TwigStack reads 442,167 elements, but TJFast reads only 2,380 elements (over two orders of magnitude). This huge gap results from the fact that TwigStack scans the elements for all the queries nodes, but TJFast scans only elements for leaf nodes.

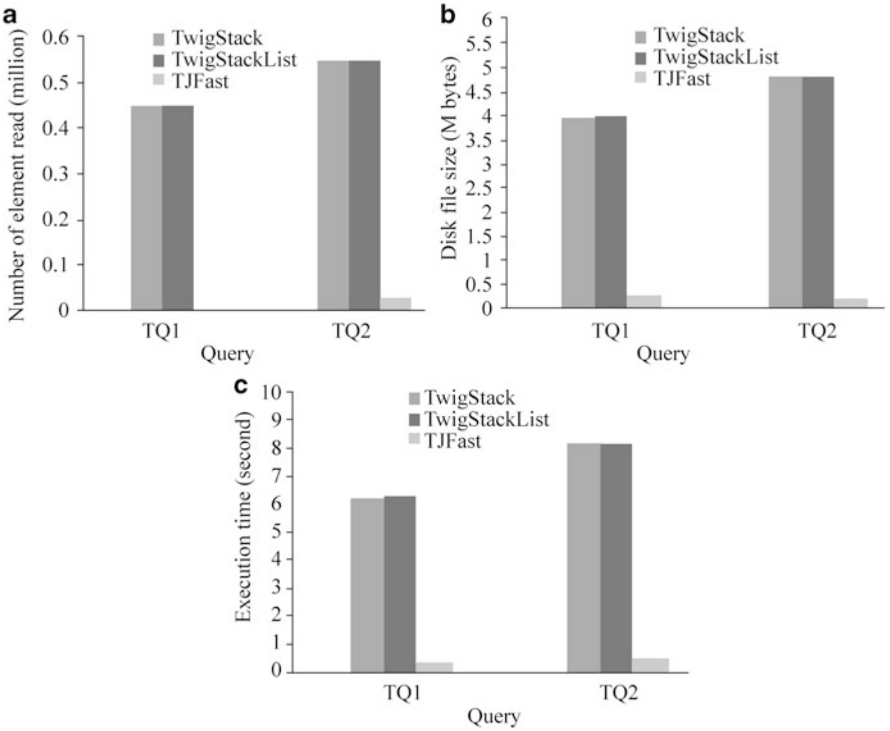


Fig. 4.19 TwigStack, TwigStackList versus TJFast on DBLP. (**a**) Number of elements read (**b**) size of disk files scanned (**c**) execution time

Size of Intermediate Results. Table 4.6 shows the number of intermediate path solutions output by different algorithms. The last column is the number of intermediate solutions that contribute to the final answers. An immediate observation is that TwigStack outputs many "useless" path solutions to queries with parent–child edges. For example, for TQ_3, TwigStack produced 702,391 intermediate paths, of which only 22,565 are useful. More than 95 % intermediate solutions output by TwigStack are "useless" to the final answers. In contrast, TJFast is optimal for query TQ_3 since the number of paths produced by TJFast is equal to the number of useful solutions.

TJFast Versus TwigStackList

From Figs. 4.22 and 4.23, TJFast also outperforms TwigStackList for all queries. This can be explained by the fact that TJFast reduces the I/O cost of TwigStackList by reading labels of only the *leaf* nodes.

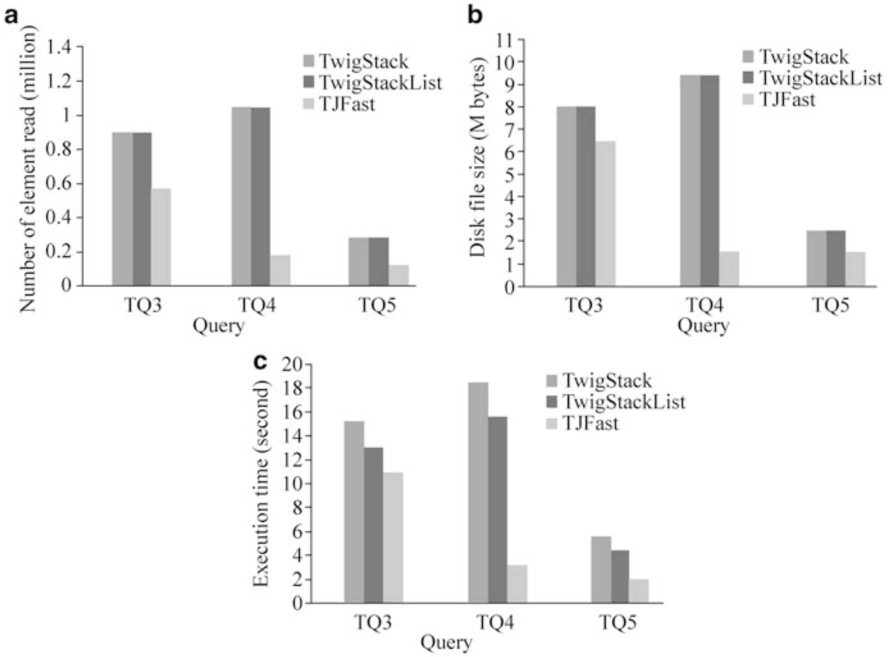


Fig. 4.20 TwigStack, TwigStackList versus TJFast on TreeBank, (**a**) Number of elements read, (**b**) size of disk files scanned, (**c**) execution time

Table 4.6 Number of intermediate path solutions

Query	TwigStack	TwigStackList	TJFast	Useful
TQ_3	702, 391	22, 565	22, 565	22, 565
TQ_4	2, 237	388	316	302
TQ_5	10, 663	9	9	5

When queries contain parent–child relationships between the branching node and its children (i.e., queries TQ_4, TQ_3), both TwigStackList and TJFast are suboptimal. Their suboptimality is evident from the observation that the number of intermediate path solutions by TwigStackList and TJFast is slightly larger than the number of useful solutions.

Wildcard Queries

Finally, we tested two wildcard queries Q_1://NP[.//CD]/*/V and Q_2://VP/*[PP-8]/PP-7 on TreeBank dataset. Q_1 is a twig query consisting of a wildcard in a non-branching node, but Q_2 is a branching wildcard twig query. For Q_1, all four algorithms can be applied. But the performance of TJFast is much better than the best algorithm based on region encoding (0.9s vs. 7.2s). For Q_2, the algorithms using region encoding are significantly affected by wildcards in branching nodes, as they do not know which elements can be used to match this wildcard.

Since there is no DTD available for TreeBank data, a brute-force solution is to access all elements to answer this query. Clearly, this method is unacceptably slow. In contrast, the existence of wildcard in branching nodes does not affect TJFast, which takes only 0.3s to answer Q_2. This shows that TJFast supports efficient processing of both non-branching as well as branching wildcard queries.

4.4 XML Query Processing Based on Various Streaming Schemes

An important assumption of the original holistic method is that an XML document is clustered into element streams which group all elements with the same tag name together and assign each element a containment label [BKS02]. We call this clustering method Tag Streaming. In recent years, there have been considerable amount of research on XML indexing techniques [CLO03, HY04, KSBG02, MS99] to speed up queries upon XML documents. In general, these XML indexes can be regarded as summary of XML source documents and thus much smaller in sizes. From another point of view, XML structural indexing can also be viewed as methods to partition XML documents for query processing. Interestingly, Tag Streaming used in TwigStack and TwigStackList can be regarded as a trivial XML indexing technique which groups all elements with the same tag together. Up till now, very little research has been done on performing holistic twig matching on XML documents partitioned by other structural indexing techniques than Tag Streaming. Furthermore, little is known about if more sophisticated XML structural indexing methods may allow optimal processing of other classes of twig pattern (besides A-D only twig patterns). Here, we call the combination of XML indexing methods with containment labels as XML streaming schemes.

We demonstrate that in general a more “sophisticated” (a formal definition will be given later) XML streaming scheme has the following two advantages than a simpler one in twig pattern processing: (1) reduce the amount of input I/O cost and (2) reduce the sizes of redundant intermediate results. The main reason behind is the increased “parallelism” to access elements with the same tag and the additional “context” information we know about each element.

Chen et al. [CLL05] proposed two XML streaming schemes: (1) a new Tag+Level scheme, which partitions elements according to their tags and levels, and (2) Prefix-Path Streaming (PPS), first proposed by Chen et al. [CLC04], which partitions elements according the label path from the root to the element; they show rigorously the impact of choosing XML streaming schemes on optimality of processing different classes of XML twig patterns.

Also, Chen et al. [CLL05] develop a holistic twig join algorithm “iTwigJoin”, which works correctly on any XML streaming scheme (as long as elements in a stream are ordered by their preorders). Applied on the Tag+Level scheme the

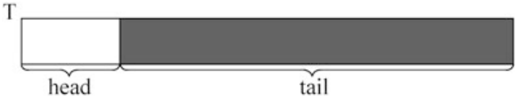


Fig. 4.21 XML stream model

algorithm can process A-D or P-C only twig patterns optimally; applied on the PPS scheme the algorithm can process A-D only or P-C only or 1-branchnode only twig patterns optimally. In this section, we will show these two streaming schemes and the Algorithm "iTwigJoin".

4.4.1 Tag+Level Streaming and Prefix-Path Streaming (PPS)

We use Q to denote a twig pattern and Q_A to denote the subtree of Q rooted at node A. We use node to refer to a query node in twig pattern and element to refer to a data node in XML data tree. We use M or $M = (e_1, e_2, \ldots, e_n)$ to denote a match to a twig pattern or sub-twig pattern where e_i is an element in the match tuple. We assume there is no node with identical tag in twig pattern Q. Since an element tag corresponds to a unique node in the twig, we use tag q and query node q interchangeably from now on.

Here, each XML "stream" is a posting list (or inverted list) accessed by a simple iterator. An XML streaming scheme is a combination of XML structural indexing techniques and element labeling scheme. More specifically, we partition an XML document into streams (in the terminology of XML structural indexing, the corresponding term for stream is extent). The only addition is to assign a region coding label to each element in the streams. We usually use T to denote a stream. A stream T has two parts: head(T) which is the stream's first element and tail(T) which is the rest of the stream. One can only read the head of a stream but not the tail portion of a stream (Fig. 4.21).

Before learning the new streaming schemes, we formally introduce various streaming techniques used and notations about XML streams.

Tag Streaming A Tag Stream in the Tag Streaming scheme contains all elements in the document tree with the same tag. For example, a stream TA contains all elements of tag A (see Fig. 4.22).

Tag+Level Streaming. The level of an element in an XML document tree is equal to the number of nodes from the root to the element. A Tag+Level stream contains all elements in the document tree with the same tag and level. A Tag+Level stream can be uniquely identified by the common tag and level of all its elements (see Fig. 4.23). Notation T_M^n denotes the stream for tag M and the level n. For example, a stream T_A^2 contains all elements of tag A and located in level 2.

Prefix-Path Streaming (PPS). The Prefix-Path of an element in an XML document tree is the path from the document root to the element. A Prefix-Path stream (PPS stream) contains all elements in the document with the same Prefix-Path,

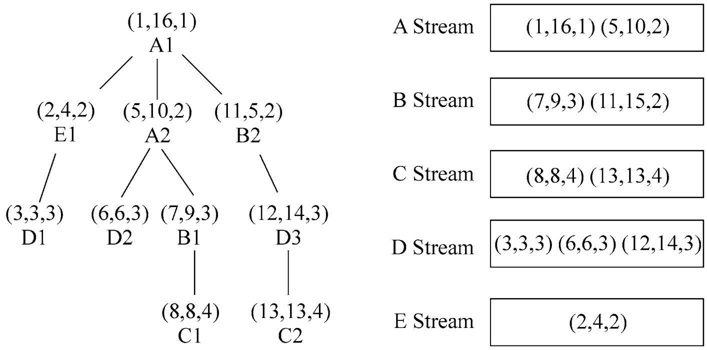


Fig. 4.22 A Sample XML document with Tag Streaming scheme

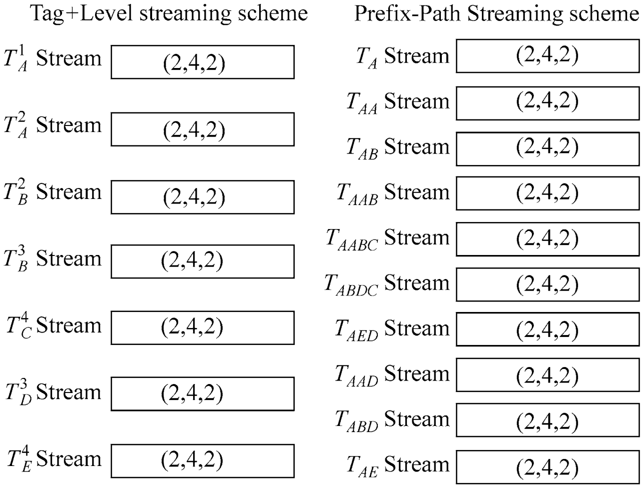


Fig. 4.23 Example of Tag+Level and PPS scheme

ordered by their begin numbers. A PPS stream T can be uniquely identified by its label, which is the common Prefix-Path of its elements. For example, a stream TABA contains all elements of tag A and of Prefix-Path ABA.

Independent of the XML streaming scheme used, we call a stream of class q if it contains elements of tag q. Identical to the notion of refinement [MS99] in XML structural indexes, we call a streaming scheme α is a refinement of streaming scheme β if for any two elements in a stream under α, the pair of elements are also in a stream under β. It can be proven that Tag+Level Streaming is a refinement over Tag Streaming and PPS is a refinement of both Tag and Tag+Level Streaming.

4.4.1.1 Notions of XML Streams Related to Twig Pattern Matching

The notions of ancestor/descendant streams and parent/child streams are defined as follows:

Under Tag+Level Streaming, a stream T_1 is an ancestor stream of stream T_2 if the level of T_1 is larger than that of that of T_2. T_2 is called the descendant stream of T_1. T_1 is the parent stream of T_2. The level of T_1 is equal to that of T_2 plus 1. T_2 is T_1's child stream.

Likewise under PPS, a PPS stream T_1 is an ancestor stream of PPS stream T_2, if label(T_1) is a prefix of label(T_2). T_1 is the parent stream of T_2, if label(T_1) is a prefix of label(T_2) and label(T_2) has one more tag than label(T_2). The definitions of descendant and child stream for PPS follow.

Given a P-C or A-D edge $<q_1, q_2>$ for which q_1 is the parent node in a twig pattern Q, two streams T_1 of class q_1 and T_2 of class q_2 are said to satisfy the structural relationship of edge $<q_1, q_2>$: (1) under Tag Streaming, the two streams automatically qualify, and (2) under Tag+Level Streaming or PPS, T_1 is the parent stream of T_2, if $<q_1, q_2>$ is a P-C edge; T_1 is the ancestor stream of T_2, if $<q_1, q_2>$ is an A-D edge. Intuitively, two streams are said to satisfy an edge if there exist two elements, one from each stream, that satisfy the P-C and A-D edge relationship.

Definition 4.4 (**Solution streams**) The solution streams of a stream T of class q for q's child edge $<q, q_i>$ in a twig pattern Q (or soln(T, q_i)) are the streams of class q_i which satisfy the structural relationship of edge $<q, q_i>$ with T.

4.4.1.2 Pruning XML Streams in Various Streaming Schemes

Using labels of Tag+Level or Prefix-Path stream labels, we can prune away streams which apparently contain no match to a twig pattern. The techniques used in stream pruning of Tag+Level and PPS streams are very similar.

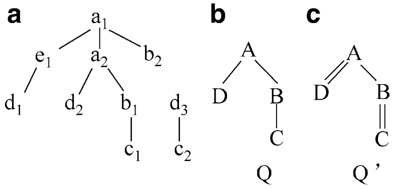
The following recursive formula helps us determine the useful streams for evaluating a twig pattern Q using the two streaming schemes. For a stream T of class q, we define U_T to be the set of all descendant streams of T (including T) which are useful for the sub-twig of Q_q except that we only use stream T (not any other stream of class q) to match node q:

$$U_T = \begin{cases} \{T\} & \text{if } q \text{ is } a \text{ leaf node;} \\ \{T\} \bigcup \{\bigcup_{q_i \in \text{child}(q)} C_i\} & \text{if none of } C_i \text{ is } \{\}; \\ \{\} & \text{if one of } C_i \text{ is } \{\}, \end{cases}$$

where $C_i = \bigcup_{T_c \in \text{so}\ln(T, q_i)} U_{T_c}$

Function child(q) returns the child nodes of q in the twig pattern Q. Apparently, under different streaming approaches, we just need to switch the definition of child/descendant stream and solution streams.

Fig. 4.24 A sample XML document and two queries



The base case is simple because if q is a leaf node, any stream of class q must contain matches to the trivial single node pattern Q_q. As for the recursive relationship, note that for a stream T of class q to be useful to the sub-twig Q_q, for each and every child node q_i of q, there should exist some nonempty set U_{T^c}which are useful to the sub-twig Q_{q^i}, and the structural relationship of T and T^c satisfies the edge between q and q_i. In the end the set U_{T^c} contains all the useful streams to a query pattern Q, where $T\Gamma$ is a stream of class root(Q). Notice that the above recursive relationship can be easily turned into an efficient algorithm using standard dynamic programming without worrying about excessive recomputation.

Example 4.10 For the XML document in Fig. 4.24a under Tag+Level Streaming, there are seven streams: T_A^1:$\{a_1\}$, T_A^2:$\{a_2\}$, T_E^2:$\{e_1\}$, T_B^2:$\{b_2\}$, T_B^3:$\{b_1\}$, T_D^3:$\{d_1, d_2, d_3\}$, and T_C^4:$\{c_1, c_2\}$. For the twig pattern in Fig. 4.24b, T_E^2 is obviously a useless stream, since there are no E node in the query.

Firstly, we have $U_{T_D^3}$ is $\{T_D^3\}$, $U_{T_C^4}$ is $\{T_C^4\}$(since B, C are leaf nodes), $U_{T_B^2}$ is $\{\}$, since child(B) $=C$, soln(T_B^2,C) $=\{T_C^3\}$, and there is no T_C^3 stream. Next, $U_{T_B^3}$ is$\{T_B^3, T_C^4\}$, $U_{T_A^1}$ is $\{\}$, since there is no T_D^2. Finally, $U_{T_A^2}$ is $\{T_A^2$,T_B^3,T_C^4,$T_D^3\}$, since child(A) $=\{D, B\}$, soln(T_A^2, D) $=\{T_D^3\}$ and soln (T_A^2, B) $=\{T_B^3\}$, $U_{T_D^3} = \{T_D^3\}$, and $U_{T_B^3} = \{T_B^2, T_C^4\}$; therefore, $U_{T_A^2} = \{T_A^2\} \cup U_{T_B^3} \cup U_{T_D^3} = \{T_A^2, T_B^3, T_C^4, T_D^3\}$.

So the final useful streams are $U_{T_A^2} \cup U_{T_A^1}$ and the two streams T_A^1 and T_B^2 are pruned.

4.4.1.3 Theoretical Foundation for Twig Pattern Matching

Based on the simple head-element-access-only XML streaming model, we have to decide if a head element is in a match to a given twig pattern before we can move to the next element in the stream. However, the difficulty to devise efficient XML twig pattern matching method lies in the fact that we cannot determine only from the head elements of various streams if any head element is in a match to a given twig pattern. Instead, the head elements of some streams may form a match to a given twig pattern with tail portions of other streams. However, since we cannot access the tail portions of streams, any premature declaration saying such head elements are indeed in some matches can result in misjudgment and in consequence redundant intermediate output.

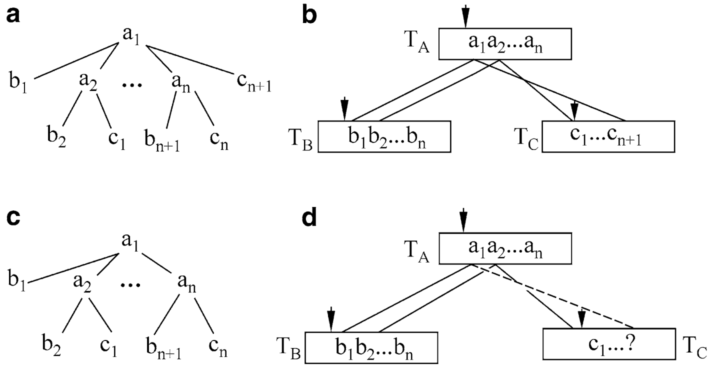


Fig. 4.25 The problem of twig join using Tag Streaming

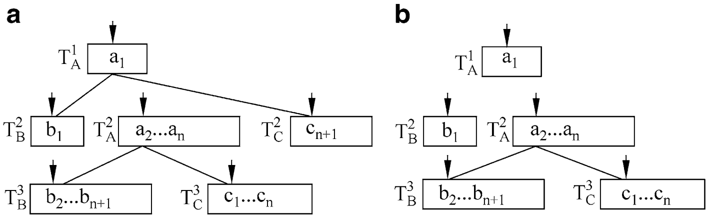


Fig. 4.26 Tag+Level Streaming for files in Fig. 4.25

Example 4.11 Suppose we evaluate the pattern $A[./B]/C$ on the XML document in Fig. 4.25a. Element a_1 is in match $<a_1, b_1, c_{n+1}>$. However, under the Tag Streaming scheme, with stream cursor positions shown in Fig. 4.25b, we cannot tell from the current head elements (e.g., a_1, b_1, c_1) that a_1 is indeed in a match. Indeed, we observe that under Tag Streaming, the XML document in Fig. 4.25c cannot be distinguished from the XML document in Fig. 4.25a because they have exactly the same set of streams and corresponding head elements. However, in the second document a_1 is not in any match. Consequently, for the document in Fig. 4.25a, we have to scan and store all the elements in the stream T_c until c_{n+1} before we are certain that a_1 is in a match. By doing so, we violate the bounded space requirement.

Noticeably, if we use Tag+Level Streaming (Fig. 4.26a) for the document in Fig. 4.25a, the above problem does not arise anymore because now we have a match $<a_1, b_1, c_{n+1}>$ which consists of only head elements of their respective streams. Therefore, a_1 is determined to be a match. Note that the documents in Fig. 4.25a, c now can be distinguished by Tag+Level Streaming, and we can determine for sure that a_1 is not in any match using Fig. 4.26b.

Example 4.11 shows that due to the introducing of P-C edge in twig patterns, matches to a twig pattern may not entirely consist of current head elements of streams under Tag Streaming. Because of our XML stream model, matches consisting of tail portions of streams can only be regarded as possible but not guaranteed matches. The existence of such possible matches means possibility of introducing of redundant intermediate results in Tag Streaming scheme if we do not want to lose any matches. We will define formally the concept of possible but not guaranteed match under this XML stream model as follows:

Definition 4.5 (Possible Match of a Twig Pattern Q) A tuple t with n fields is called a possible match of a twig pattern Q if:

Each field t_q of t corresponds to one node q of Q. The value of t_q can either be head(T_q) or tail(T_q) where T_q is a stream of class q; and

1. For each field t_q, for each child (if any) q_i of q, the streams T_q and T_{q^i} (which t_{qi} corresponds to) satisfy the edge $<q, q_i>$ and
   (a) If t_q and t_{qi} are of value head(T_q) and head(T_{q^i}), then head(T_{q^i}) is a parent or ancestor of head(T_q) depending on the edge $<q, q_i>$.
   (b) If t_q is of value head(T_q) and t_{qi} is tail(T_{q^i}), then head(T_q).end > head(T_{q^i}).start.
   (c) If t_q is of value tail(T_q) and t_{qi} is head(T_{q^i}), then head(T_q).start < head(T_{q^i}).start.
   (d) If t_q is of value tail(T_q) and t_{qi} is tail(T_{q^i}), there is no additional restriction.

An important difference between a possible match and a match to a twig pattern is that the former can always be told from the current head elements as seen from conditions in the definition whereas the later may not. Informally, the reason why we call the tuple satisfying the above conditions a possible match is that the tuple can be instantiated to match to Q. If a possible match of Q consists of only head elements, it is called a minimal match; otherwise it is called a blocked match. Analogously, we can have similar definition to possible match to a sub-twig Q_q of Q rooted at node q.

For the sake of simplicity, we always use the current head element of stream T instead of symbol head(T) in a possible match. We also append an imaginary "end" element (whose begin and end are all infinity) to the end of each stream so that the definitions of head(T) and tail(T) can be applied on single-element streams.

Example 4.12 For the XML document in Fig. 4.24a, under Tag Streaming, suppose all streams have not advanced, the tuple t: a_1, d_1, b_2,tail(T_C^4) > is not a possible match for the query Q in Fig. 4.24b because T_B^2 and T_C^4 do not satisfy edge B/C. However, the tuple is a possible but not minimal match for the query Q' in Fig. 4.24c because b_2.endPos>c_1.StartPos.

As another example, the tuple t':<tail(T_A^2), d_1, b_1, c_1> is not a possible match for Q because a_2.startPos>d_1,startPos. However, it is trivial that d_1 is a possible and also minimal match for Q_D here Q_D is a sub-query of Q rooted at D.

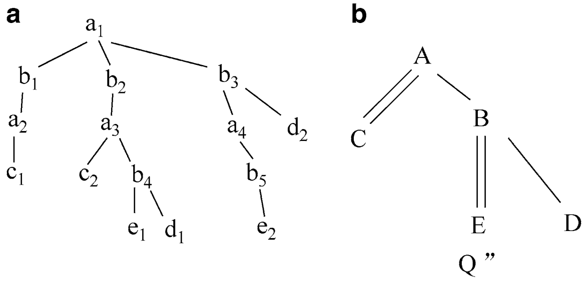


Fig. 4.27 A Sample XML Document for Which PPS Also Can't Help

Example 4.13 For the XML document in Fig. 4.27a, under PPS, there are 8 PPS streams: T_A:{a_1},T_{AB}:{b_1, b_2, b_3}, T_{ABA}:{a_2, a_3, a_4}, T_{ABD}:{d_2},T_{ABAC}:{c_1, c_2}, T_{ABAB}: {b_4, b_5}, T_{ABABD}:{d_1}, and T_{ABABE}:{e_1, e_2}.

Suppose no stream has yet moved, then the tuple <b_1, tail(T_{ABABE}),tail(T_{ABD})> is not a possible match for Q'_B.

Suppose now we move the stream T_{AB} to b_3 and T_{ABA} to a_3. The tuple <a_1, c_1, b_3, d_2,tail(T_{ABABE})> is a possible match for Q''. The tuple <a_3, Tail(T_{ABAC}), b_4, d_1, e_1> is also a possible match for Q$''$.

Using the concept of possible match, we classify the current head elements of useful streams to the following three types with respect to a twig pattern Q:

1. (Matching element) Element e of type E is called a matching element if e is in a minimal match to Q_E but not in any possible match to Q_P where P is the parent of E or E is the root of Q.
2. (Useless element) Element e is called a useless element if e is not in any possible match to Q_E.
3. (Blocked element) Otherwise e is a blocked element or we say e is blocked.

Informally, we can think a useless element as an element that can be thrown away safely. A matching element e is an element which is at least in a match to Q_E and we can tell if it is in a match to Q in an efficient way.

Notice that a head element e is blocked if e is in a possible but not in any minimal match to Q_E or e is in a minimal match to Q_E but is some possible match to Q_P where P is the parent of E. In the first case, e is not guaranteed to be in a match. In the later case, if e is in a possible match to Q_P, taking away e first can cause errors in determining if elements which are ancestor of e are indeed in possible matches to Q_P. Therefore, it is unsafe to advance the stream where e is the head.

For an optimal twig pattern matching algorithm to proceed, it should never happen that all current head elements are blocked because in such a situation we cannot advance any stream without storing elements which are not guaranteed in a match.

Notice that Example 4.11 gives an example where all head elements are blocked under Tag Streaming. Although better than Tag Streaming, the following example shows that there are also queries under which all head elements are blocked under Tag+Level Streaming.

Example 4.14 Under Tag+Level Streaming, for the XML file in Fig. 4.24a, Q in Fig. 4.24b, and Q′ in Fig. 4.24c, suppose no stream cursor has moved, we have the categories as follows.

Head	Q	Q'
a_1	Useless	Blocked
a_2	Blocked	Blocked
b_1	Blocked	Blocked
b_2	Useless	Blocked
c_1	Blocked	Blocked
d_1	Matching	Blocked

The following example shows that PPS can avoid some all blocked situation occurring in Tag+Level Streaming but also has its limitation.

Example 4.15 Under PPS, for the XML file in Fig. 4.24a and Q' in Fig. 4.24c, different from Tag+Level Streaming, we have both a_1 and a_2 as matching elements. The main reason is that d_1 and d_2 are now in two different PP streams T_{AED} and T_{AAD} and so are c_1 and c_2. However, for the document in Fig. 4.27a and the query in Fig. 4.27b, suppose the stream T_{AB} advances to b_3 and T_{ABA} advances to a_3. We have all head elements, namely, a_1, a_3, b_3, b_4, d_1, e_1, c_1 and c_2 blocked.

The above description shows some negative results and thus limits of various streaming schemes. Now we try to prove that for a certain class of twig pattern query, a particular streaming scheme can prevent the situation whereby all head elements are blocked. This can be seen as a necessary condition for optimal twig pattern matching. Now first we introduce an auxiliary lemma:

Lemma 4.9 *Given two streaming schemes α and β, suppose α is a refinement of β. Suppose we have a set of streams in β being further partitioned under α, we have*:

1. *A matching head element in β is also a matching head element in α.*
2. *A useless head element in β is also a useless head element in α.*

It is easy to find example to prove that the opposite direction in each of the two conclusions in the lemma is not true.

In proving the following lemmas, we use the following important observations: a blocked or useless element for sub-twig pattern Q_q of Q is also a blocked or useless element for Q; if e is a matching element for sub-twig pattern Q_q of Q but not in possible match to Q_q, it is also a matching element for Q.

Lemma 4.10 *Under Tag Streaming, for an A-D only query Q, there always exists a head element which is either a matching or useless element for Q.*

Proof We use induction on the sub-queries of Q.
(Base case) Suppose a node q in Q which is the parent of leaf nodes $q_1, q_2, \ldots, q_n$ and T_q has not ended. Note that if any stream T_{q^i} has ended, head(T_q) is a useless element for Q_q and we are done. Otherwise, all streams $T_{q^1}, \ldots, T_{q^N}$ are not end and

it is obvious that each sub-twig Q_{q^i} for i from 1 to n has a minimal match. There are the following cases:

1. If head(T_q).endPos < head(T_{q^i}).startPos for some q_i, then head(T_q) is a useless element for Q_q.
2. Else if head(T_q).endPos > head(T_{q^i}).startPos for some q_i, then head(T_{q^i}) is a matching element for Q_q but is not in possible match to Q_q.
3. Otherwise head(T_q) is ancestor of head(T_{q^i}) for each q_i and head(T_q) is a matching element for Q_q.

Note that in the base case we can find either a useless or matching element with respect to(w.r.t) sub-twig Q_q.

Suppose a node q in Q has child nodes $q_1, a_2, \ldots, q_n$, by the induction hypothesis, if there is a node q_i for which head(T_{q^i}) is not a matching element for Q_{q^i}, we must find either a useless element for Q_{q^i} (which is also a useless element for Q_q) under the above case 1 or a matching element for a sub-twig of Q_{q^i} but not in possible match to Q_{q^i} (which is also a matching element for Q_q) under case 2 and thus done. Otherwise each head(T_{q^i}) is a matching element for Q_{q^i}, we proceed with the same argument (with the three cases) in the base case. Note that the induction step ends when q is the root of Q and in such case head(T_q) is a matching element.

Lemma 4.11 *Under* Tag+Level *Streaming, for an A-D only or P-C only query Q, there always exists a head element which is either a useless or matching element for Q.*

Proof [*Sketch*] According to Lemma 4.9, for an A-D only twig pattern, the above statement is true.

Given a P-C only query Q with n nodes, we can partition the streams into a few groups, each of which has n streams and contains possible matches to Q. For example, streams $T_A^2, T_D^3, T_B^3, T_C^4$ form a group for the query $A[/D]/B/C$ in Fig. 4.24. Notice that it is impossible that two elements from streams of different groups can be in the same match. For each group of n streams, we can perform the same analysis as in the proof of Lemma 4.10 to find out either a useless or matching element for Q.

Lemma 4.12 *Under Prefix-Path streaming, for an A-D only or P-C only and one branching-node only queries, there always exists a head element which is either a useless or matching element.*

Proof [*Sketch*] According to Lemma 4.9, for an A-D or P-C only twig pattern, the above statement is true. For a one branching-node only query, we first prove the special case where the root node is also the branchnode. Suppose the PPS stream Tmin is the one whose head element with the smallest startPos among all streams.

1. $T_{\min}$ is of class q where q is not the root of Q and L is the label of $T_{\min}$. Suppose q_1 is a leaf node of Q and descendant of q. Suppose $T_{q1}^{\min}$ is a stream of class q_1 having label L' with the following properties: (1) $L' = L + L''$ and the string L'' matches the path query Q_q; (2) $T_{q1}^{\min}$ has the head element with the minimum

begin among all streams of class q_1 satisfying property (1). Now if head($T_{\min}$) is not an ancestor of head($T_{q1}^{\min}$), the head($T_{\min}$) is a useless element and we are done. Otherwise take any stream T_m of class q_m where q_m is a node between q and q_1 and having a label which is prefix of L' and for which L is a prefix. If for any such T_m, head(T_m).end < head($T_{q1}^{\min}$).begin, then head(T_m) is a useless element; otherwise head($T_{\min}$) is a matching element.

2. $T_{\min}$ is of class q where q is the root of Q. We can consider the leaf nodes for each branch of Q and repeat the same argument as (1) for each branch.

In the case where the branchnode is not the root node, we can reduce it to the first case. Interested readers may wonder if there is a streaming scheme whereby the all blocked situation never occurs; we point out that at least the costly FB−BiSimulation scheme proposed by Kaushik et al. [KBNK02] is able to do that because the scheme is so refined that from the label (or index node) of each stream we can tell if all elements in the stream are in a match or not.

4.4.2 iTwigJoin Algorithm

Here, we introduce a twig pattern matching method iTwigJoin proposed by Chen et al. [CLL05], which is applicable to any streaming schemes discussed so far based on the notion of possible match. The method can work correctly for all twig patterns, and meanwhile they are also optimal for certain classes of twig patterns depending on the streaming scheme used.

In the algorithm, there are two important components, namely, (1) a stream management system to control the advancing of various streams and (2) a temporary storage system to store partial matching status and output intermediate paths.

The role of the temporary storage system can be summarized as follows: it only keeps elements from streams which are in possible matches with elements which are still in the streams. The elements in the temporary storage system has dual roles: (1) they will be part of intermediate outputs, and (2) when a new element e with tag E is found to be in a possible match to sub-twig Q_E of twig pattern Q, we can know if e is in a possible match to Q by checking if e has a parent or ancestor element p in the temporary storage which is a possible match to Q_P where P is the parent node of E in Q. Similar to TwigStack, we associate each node q in a twig pattern with a stack S_q. At any time during computation, all elements in a stack are located on the same path in source XML documents. The property is ensured through the following push operation: when we push a new element e with tag E into its stack S_E, all the elements in the stack which are not ancestor of e will be pop out.

As for the stream management system, depending on different streaming schemes, each node q is associated with all useful streams of class q. Each stream has the following operations: head(T) returns the head element of stream T, and T.advance() moves the stream cursor to the next element.

4.4.2.1 Algorithm of iTwigJoin

Algorithm 4.11: iTwigJoin

```
Prune-Streams(Q)
while !end(root) do
   q = getNext(root);
   $T_{min}$ = the stream with the smallest startPos among all streams of class q
   pop out elements in $S_q$ and $S_{parent(q)}$ which are not ancestor of head($T_{min}$)
   if isRoot(q) $\vee$ existParAnc(head($T_{min}$),q) then
      push head($T_{min}$) into stack $S_q$
      if isLeaf(q) then
         showSolutionWithBlocking($S_q$); //See TwigStack for details
      end if
   end if
   advance($T_{min}$)
end while
mergeAllPathSolutions()
```

Function end(QueryNode q)

```
return true if all streams associated with leaf nodes of $Q_q$ end;
Otherwise return false;
```

Function existParAnc (Element e, Node q)

return true if e has a parent or ancestor element in stack
$S_{parent(q)}$ depending on edge $<$ parent(q), q $>$)

The flow of this algorithm iTwigJoin is similar to that of TwigStack. In each iteration, an element *e* is selected by the stream management system from the remaining portions of streams. To avoid redundant intermediate output, we always try to select a matching element unless all head elements are blocked. The element is then be used to update the contents of stacks associated with each query node in the twig pattern. The detail of updating process will be discussed shortly. During the update, partial matching paths will be outputted as intermediate results. The above process ends when all streams corresponding to leaf nodes of the twig pattern end. After that, the lists of intermediate result paths will be merged to produce final results.

Algorithm 4.12: getNext(q) (in iTwigJoin)

```
if isLeaf(q) then
     return q
end if
for i = 1 to n do
   $q_x$ = getNext($q_i$) //$q_i, \ldots, q_n$ are children of q
   if ($q_x \neq q_i$) then
      return $q_x$
   end if
end for
```

```
10. for each stream $T_q^j$ of class q do
11.    for i = 1 to n do
12.        $T_{qi}^{\min}$= min(soln($T_q^j$ ,qi));
13.    end for
14.    $T_{\max} = \max(\{T_{q1}^{\min}, \ldots, T_{qn}^{\min}\})$;
15.    while head($T_q^j$ ).endPos < head($T_{\max}$).startPos do
16.        $T_q^j$ .advance();
17.    end while
18.end for
19.$T_q^{\min}$=min(streams(q))
20.$T_{child}^{\min}$=min(streams($q_1 \cup \ldots \cup$streams($q_n$));
21.if $T_q^{\min}$.startPos <$T_{child}^{\min}$. startPos    then
22.    return q;
23.end if
24.return $q_c$ where $T_{\text{child}}^{\min}$ is of class $q_c$
```

Function min(a set of streams)
return the stream with the smallest startPos in the set

Function max(a set of streams)
return the stream with the largest startPos in the set

We divide this algorithm into two parts: one for the stream management system (Algorithm 4.12) and another for the temporary storage system (Algorithm 4.11).

The stream management system of iTwigJoin, in each iteration, discards useless elements and selects a head element *e* of tag *E* from the remaining portions of streams with the following two properties:

1. *e* is in a possible match to Q_E but not in a possible match to Q_P where *P* is the parent of *E* in *Q*. (Notice that *e* can be either a matching element or a blocked element but not a useless one.)
2. *e* is the element with the smallest begin among all non-useless head elements of tag E' where E' is a node in the sub-twig Q_E.

The first property guarantees that the element *e* is at least in a possible match to Q_E. Equally importantly, a parent/ancestor element in a match is always selected before its child/descendant by the property. The second property is important to ensure that the space used our temporary storage system is bounded as we will explain in the next section.

Before studying in detail how the stream management system works, let us first look at the temporary storage system in Algorithm 4.11. After the element e with tag *E* is selected (Line 3), we first pop out elements in S_E and $S_{\text{parent}(E)}$ whose end is smaller than *e*.startPos (Line 5) as they are guaranteed to have no more matches as we will prove shortly. In Line 6, we check in our temporary storage system if there is an element in stack S_p which is parent or ancestor of the selected element *e* (depending on edge <*P*, *E*>) where *P* is the parent node of *E*. If there is such

Table 4.7 PPS streaming scheme can provide an optimal solution

Step	Selected	Stack operation
1	a_1	push
2	d_1	push, output a_1/d_1
3	a_2	push
4	d_2	push, pop d_1,output a_1/d_2, a_2/d_2
5	b_1	push
6	c_1	push, output $a_2/b_1/c_1$
7	b_2	push
8	d_3	push, pop d_2, output a_1/d_3
9	c_2	push, pop c_1, output $a_1/b_2/c_2$

an element, e is then pushed into S_E (Line 7) and if E is a leaf node, a number of intermediate paths containing e as the leaf element are output (Line 9); otherwise, e is discarded.

Now we study how the stream management system works. The function call getNext(root) (Algorithm 4.12) plays the role of selecting an element from the remaining portions of the streams with the two aforementioned properties. It works recursively: (1) for the base case where q is a leaf node in Q, it just returns q (line 1–2). (2) Suppose q has children q_1, q_2, ..., q_n, we first call getNext(q_1), ..., getNext(q_n) (Line 5). If any of the recursive call does not return q_n, we have found the element because it satisfies the above two properties mentioned w.r.t Q_{q^i} and is not in a possible match to Q_{q^i}; thus, it also satisfies the two properties w.r.t Q_q and consequently Q. So it returns what the call returns (Lines 6–7). Otherwise, for each stream T_q^j of class q, for each child node q^i of q from 1 to n, in Lines 11–12 we find the stream $T_{q^i}^{\min}$ which is of class q_i and also has the smallest startPos among all solution streams of T_q^j for edge $<q, q_i>$. Let $T_{\max}$ be the stream whose head has the largest startPos among $T_{q^i}^{\min}$ for i from 1 to n (line 14). We next advance T_q^j until head(T_q^j).endPos > head($T_{\max}$).startPos (Lines 15–17). Notice that all elements skipped are useless elements because they are not in possible match to Q_q. After the advancing, head(T_q^j) is in a possible match to Q_q and can be either blocked or matching. Finally, let $T_q^{\min}$ be the current stream of class q with the smallest startPos and $T_{\text{child}}^{\min}$ be the stream with the smallest startPos among ALL streams of class q_c where q_c is any child node of q. If head($T_{\text{child}}^{\min}$).startPos<head($T_q^{\min}$).startPos, the element head($T_{\text{children}}^{\min}$) satisfies the two properties aforementioned: it will not be in any possible matches to Q_q because head($T_{\text{children}}^{\min}$).startPos < head($T_q^{\min}$).startPos and the satisfaction of property 2 is obvious. Thus, the child node is returned. Otherwise, q is returned and the recursion proceeds because head($T_q^{\min}$) may be in a match to Q$_{\text{parent}(q)}$.

Example 4.16 For the sample XML document in Fig. 4.24a and the twig pattern query Q' in Fig. 4.24c, the PPS scheme can provide an optimal solution. The following Table 4.7 traces the entire matching process with the elements selected in each iteration by getNext(root) and the corresponding stack operation. The word "push" means an element of tag E is pushed into its stack S_E.

Table 4.8 Algorithm iTwigJoin output less redundant intermediate

Step	Selected	Stack operation
1	a_1	push
2	c_1	push, output a_1/c_1
3	a_3	push
4	c_2	push, pop c_1,output a_3/c_2
5	b_4	push
6	e_1	push, output $a_3/b_4/e_1$, $a_1/b_4/e_1$
7	d_1	push, output $a_3/b_4/d_1$
8	b_3	push, pop b_4
9	e_2	push, pop e_1, output $a_1/b_3/e_2$
10	d_2	push, pop d$_1$, output $a_1/b_3/d_2$

Note that all elements selected are matching elements. As an example, when a_1 is selected, it is in the match $<a_1, d_1, b_2, c_2>$, which are all head elements of their Prefix-Path streams.

On the other hand, for the query Q'' in Fig. 4.27b and the document in Fig. 4.27a, iTwigJoin based on PPS scheme is no longer optimal because Q'' has two branchnodes. For example, when a_1 is selected, it is only in the nonminimal possible match $<a_1, c_1, b_3, d_2, \text{tail}(T_{ABABD})>$. Notice that the current head element of stream T_{ABABD} is e_1, which is also in a possible match. However, if the tail(T_{ABABD}) does not contain the element e_2, we will output two redundant intermediate paths: a_1/c_1 and $a_1/b_3/d_2$. Of course if we just discard a_1, we may lose the two paths if e_2 is there instead.

Even though the algorithm iTwigJoin is not optimal in this case, compared with TwigStack, it still output less redundant intermediate paths: note that TwigStack will also output the redundant path $a_1/b_2/e_1$ because $<a_1, c_1, b_2, e_1, d_1>$ is in a match to $A[//C]//B[//D\,]//E$ (Table 4.8).

iTwigJoin, applied on Tag Streaming, is essentially identical to TwigStack. Except the definition of solution streams (Algorithm 4.12, Line 12), iTwigJoin is independent of the underlying streaming scheme used. Thus, given a new streaming scheme (assuming elements in the stream are of the same tag and ordered in document-order) other than the three discussed, iTwigJoin is still applicable after we work out the new definition of solution streams which is often quite easy.

4.4.2.2 Analysis of Algorithm

First, prove the correctness of our algorithm. Next we are going to show that depending on streaming schemes used, this algorithm is optimal for several important classes of twig pattern queries.

Lemma 4.13 *The* getNext(root) *of Algorithm iTwigJoin returns all elements which are in matches to a given twig pattern Q.*

Essentially, we can show that the getNext(root) call of our algorithm returns all elements *e* of tag *E* which are in possible match to sub-query Q_E of *Q*, which is a superset of elements in matches to *Q*. The most important observation is that Property 1 of the element returned by getNext() guarantees that a parent/ancestor element in a possible match is always returned before its child/descendant element.

Lemma 4.14 *In Algorithm 4.11, when an element is popped out of its stack, all its matches have been reported.*

Proof An element *e* of tag *E* is popped out of its stack SE because we push into SE or child stack SC (Line 5 of Algorithm 4.11) an element e' and e'.startPos>*e*.endPos. Suppose *e* has some matches yet output, there must exist a child/descendant element *c* (with tag *C*) of *e* not yet be returned by getNext(root). It is easy to see that e' startPos >*c*.endPos too. Since *c* will also be in a possible match to *QC*, by the second property of the getNext(root) function, *c* will be returned before e' because *c*.startPos < e'.startPos.

The above two lemmas show that all elements in matches will be reported by our stream manager and no element will be removed from our temporary storage system before all its matches have been reported. Thus, we can come to the follow theorem:

Theorem 4.14 *The Algorithm iTwigJoin correctly reports all matches to a given twig pattern.*

The following lemma shows that the space complexity of our algorithm is bounded.

Lemma 4.15 *Algorithm iTwigJoin uses space bounded by* $|Q|$**L where L is the longest path in the XML source document and* $|Q|$ *is the number of nodes in Q.*

This is easy to see because all the elements in any stack are located on the same path of the source document.

The following lemma shows the optimality of our algorithm for certain classes of twig pattern queries depending on streaming schemes. The essential idea is that under the combination of twig pattern types and streaming schemes, every getNext(root) call only returns an element which is a matching element. Because a matching element *e* of tag *E* is in a real match, no redundant intermediate results will be outputted.

Lemma 4.16 *Depending on streaming schemes, this algorithm iTwigJoin is optimal in the following classes of queries*:

1. *Tag Streaming: A-D only twig pattern*
2. *Tag+Level Streaming: A-D or P-C only twig pattern*
3. *Prefix-Path Streaming: A-D or P-C or one branchnode only twig pattern*

Lemma 4.17 *The CPU time of iTwigJoin is* O(no_streams*$|Q|$*|INPUT+OUTPUT|) *where no streams is the total number of useful streams for the twig pattern query Q.*

In the actual implementation, for stream T_q^j of class q, we keep a number min(T_q^j, q_i) for each child edge of q:<q, q_i> to keep track of the minimum startPos of head elements of streams in soln(T_q^j, q_i). Notice the number is used in Line 12 of Algorithm 4.12. Notice that when a stream advances, at most O(no_streams) min numbers will be updated.

Since each element is scanned only once, we calculate the CPU time by bounding the time interval from the previous element scan event to the current one. There are two places to scan an element in the program: Line 12 of Algorithm 4.11 and Line 16 of Algorithm 4.12. Notice that if the current scan occurs in Line 16 of Algorithm 4.12, the time lapse after the previous scan event is at most O(no_streams) for updating those min(T_q^j, q_i) of various streams and at most O($|Q|$) on Lines 10–15 of Algorithm 4.12. If we scan an element in Line 12 of Algorithm 4.11, the maximum time interval is O($|Q|\times$no_streams) when the previous scan also occurs at Line 12 of Algorithm 4.11. Therefore, the total CPU time spent is O(no_streams× $|Q|\times$|INPUT+OUTPUT|) when added in the output size.

4.4.2.3 Experimental Evaluation

We first apply the two streaming schemes (i.e., Tag+Level and PPS) to XML documents with different characteristics to demonstrate their applicability of different kinds of XML files. Next we conduct a comprehensive study of twig pattern processing performances based on various streaming schemes and the algorithms we discussed. Our experiment results show significant advantages of new XML streaming schemes and twig join algorithms over the original TwigStack approach and its recent variant.

Experiment Settings and XML Datasets

We implemented all algorithms in Java 1.5. All our experiments were performed on a system with 2.4-GHz Pentium 4 processor and 512-MB RAM running on Windows XP. We used the following real-world (i.e., TreeBank [BKS02]) and synthetic datasets (i.e., XMark [CDF+94]) for our experiments: (1)XMark—it contains information about an auction site. Its DTD is recursive. (2)TreeBank—the DTD of Treebank is also recursive. TreeBank consists of encrypted English sentences taken from the Wall Street Journal, tagged with parts of speech. Their main characteristics can be found in Table 4.9.

The reason why we select the above two XML datasets is because they represent two important types of data: XMark is more "information oriented" and has many repetitive structures and fewer recursions, whereas TreeBank has inherent tree structure because it encodes natural language parse trees.

Table 4.9 XML datasets used in experiments

	XMark	Treebank
Size	113 MB	77 MB
Nodes	2.0 million	2.4 million
Tags	77	251
Max depth	12	36
Average depth	5	8
No. of streams using Tag+Level	119	2,237
No. of streams using PPS	548	338,740

Table 4.10 Queries used in experiments

	Query	Type
X-Q1	//site/people/person/name	Path query
X-Q2	//site/people/person[//name][//age]//income	Only A-D in branching edges
X-Q3	//text[/bold]/emph/keyword	Only P-C in all edges
X-Q4	//listitem[//bold]/text//emph	One-branching query
X-Q5	//listitem[//bold]/text[//emph]/keyword	Other kind query
T-Q1	S//ADJ[//MD]	Only A-D in branching edges
T-Q2	S[/JJ]/NP	Only P-C in all edges
T-Q3	S/VP/PP[/NP/VBN]/IN	Only P-C in all edges
T-Q4	S/VP//PP[//NP/VBN]/IN	One-branching query
T-Q5	S//NP[//PP/TO][/VP/_NONE_]/JJ	Other kind query

Number of Streams Generated by Various Streaming Techniques

Table 4.9 also shows the statistics of applying Tag+Level and Prefix-Path Streaming schemes. It is easy to see that on an information-oriented data source like XMark, the numbers of streams resulted from Tag+Level as well as Prefix-Path Streaming are small compared with the total number of nodes (two million) in the document. This shows that in the document, most of the elements with the same tag appear in relatively few different "contexts." On the other hand, in a much more deep recursive data like Treebank, Tag+Level still results in relatively few number of streams compared with element numbers (2.4 million). However, the number of streams under Prefix-Path Streaming is so large that it is nearly 16 % of the number of elements.

The above data shows that Tag+Level can be applied to a wider range of XML documents, whereas Prefix-Path Streaming is better to use in more information-oriented XML data.

Twig Pattern Matching on Various Streaming Schemes Queries

We select representative queries (shown in Table 4.10) which cover the classes of twig pattern query that fall within and outside the optimal sets of different streaming schemes.

Table 4.11 Numbers of streams before and after pruning for XMark and TreeBank datasets

	T+L	T+L pruning	PPS	PPS pruning
X-Q1	7	4	17	4
X-Q2	9	7	19	6
X-Q3	27	24	330	132
X-Q4	24	19	249	144
X-Q5	31	23	348	162
T-Q1	62	46	12,561	1,714
T-Q2	91	81	78,109	18
T-Q3	177	138	123,669	474
T-Q4	177	138	123,669	1,876
T-Q5	209	175	132,503	1,878

Performance Measures

We compare four algorithms: namely, TwigStack on Tag Streaming scheme, a recent proposed variant of TwigStack: TwigStackList and iTwigJoin on Tag+Level and Prefix-Path Streaming, respectively. TwigStackList is also based on Tag Streaming. It is different from TwigStack by allowing look ahead a limited amount of elements in a stream to avoid redundant intermediate paths. The method is shown not be optimal for P-C only or 1-branchnode only twig pattern. We consider the following performance metrics to compare the performance of twig pattern matching algorithms based on three streaming schemes: (1) number of elements scanned, (2) number of intermediate paths produced, and (3) running time. We also record the number of streams whose tags appear in the twig pattern and the number of useful streams after streaming pruning for each query under different streaming schemes in Table 4.11.

Scalability

We also test the four algorithms on datasets of different sizes. We present in Fig. 4.29 the performance results tested on XMark benchmark size of 11 MB, 40 MB, 80 MB, and 113 MB and 140 MB on the twig pattern XMark5.

Performance Analysis

In terms of number of bytes scanned (Fig. 4.28a, d), based on the XMark benchmark, we can see that both PPS and Tag+Level can prune large portions of irrelevant data: PPS from 40 to 300 % and Tag+Level from 4 to 250 %. Meanwhile, PPS can prune more data than Tag+Level. As for Treebank, Tag+Level saves fewer I/O (from 0 to 5 %) compared with PPS.

With respect to the numbers of intermediate paths output by various algorithms (Fig. 4.28b, e), iTwigJoin based on PPS and Tag+Level avoids redundant

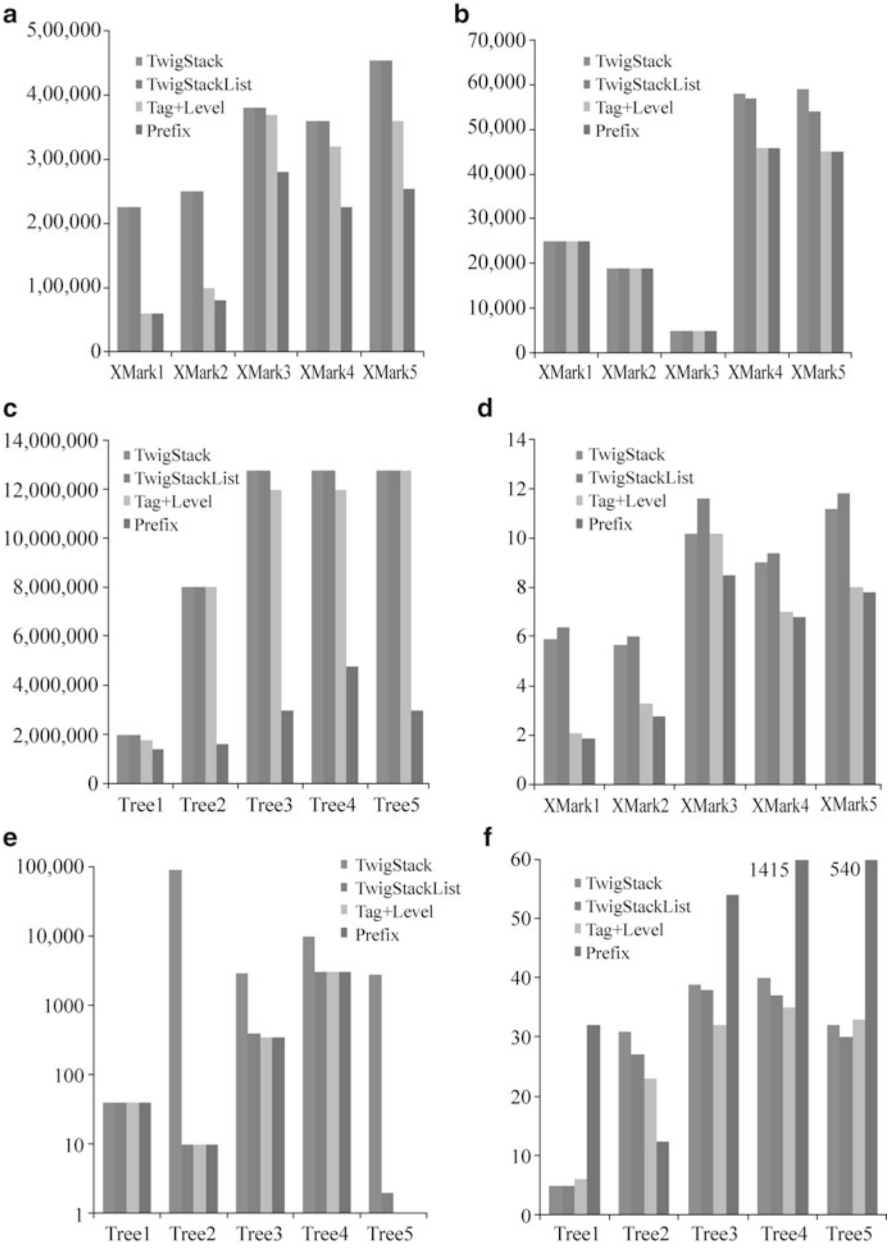


Fig. 4.28 Performance comparison over XMark(**a**–**c**) and TreeBank(**d**–**f**) datasets; The number of intermediate paths output by Tag+Level and PPS is also the number of merge joinable paths. (**a**) Bytes scanned (**b**) number of intermediate path (**c**) running time (**d**) bytes scanned (**e**) number of intermediate path (**f**) running time

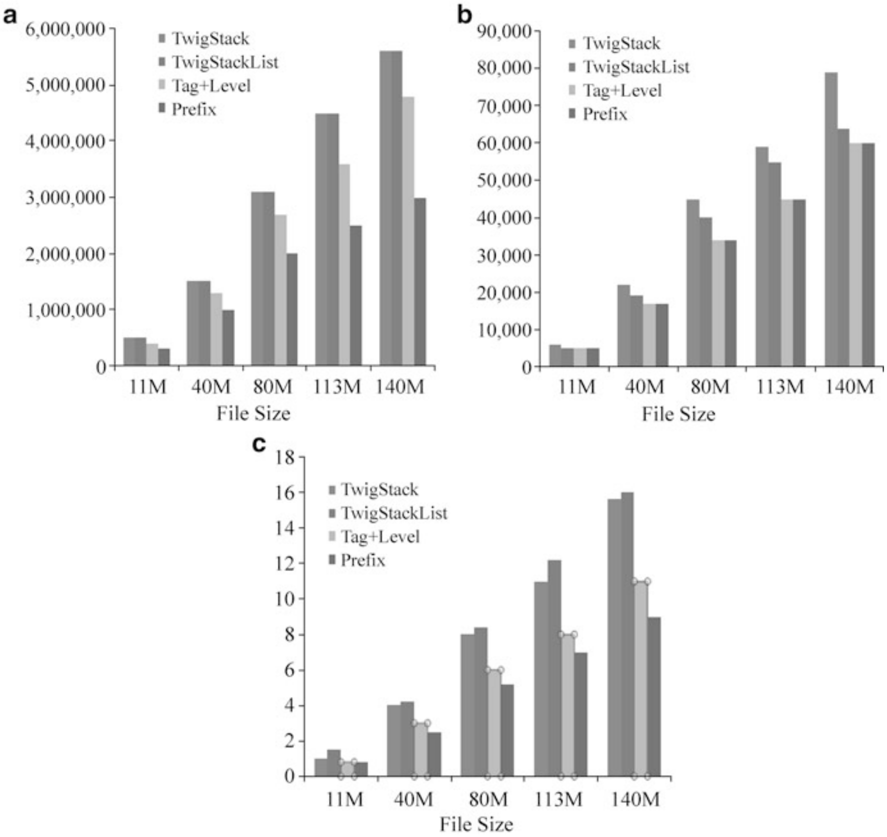


Fig. 4.29 Performance comparison of four algorithms over XMark of different sizes using XMark5. (**a**) Bytes scanned (**b**) number of intermediate paths (**c**) running time

intermediate paths produced by TwigStack and TwigStackList based on Tag Streaming. For XMark, the reduction ratio goes up to 25 % (X-Q5) and for Treebank as high as 79,800:10 (T-Q2). A somewhat surprising result is that although there are queries which fall outside of the theoretical optimal classes of Tag+Level and PPS (e.g., X-Q3, T-Q3, and T-Q4 for Tag+Level and X-Q5 for PPS), the numbers of intermediate paths output by Tag+Level and PPS for these queries are also the numbers of merge joinable paths! This shows that in real XML datasets, the theoretical worse cases seldom occur (Fig. 4.29).

Combining the savings in both input I/O cost and intermediate result sizes, iTwigJoin in general achieves faster running time (Fig. 4.28c, f). For XMark, iTwigJoin based on PPS is always faster than that based on Tag+Level Streaming which in turn is faster than that of TwigStack. For Treebank, iTwigJoin based on Tag+Level Streaming loses slightly (–5 %) in A-D only query (T-Q1) and the query T-Q5 where there are over 175 streams involved but wins in other cases;

however, except T-Q2 where there are 18 streams left after pruning, iTwigJoin based on PPS requires unacceptable running times (1412s for Tree4 and 540s for T-Q5) because it needs to join too many streams. iTwigJoin, based on Tag+Level or PPS, needs to join more streams than that of TwigStack. This makes iTwigJoin more CPU intensive than that of TwigStack. This experiment results show that even if the number of streams goes up to 150, iTwigJoin still has edges over Twigstack because of the saving in input I/O and intermediate outputs.

4.5 Summary

In this chapter, we present several join algorithm. First, we make research on structural join, and show two algorithm: tree-merge and stack-tree algorithms, which output useless intermediate results by using decomposition and join operations. Second, we propose holistic twig join algorithm.

Unlike two lists being joined in structural join, holistic twig join regards the twig patterns as whole. In this chapter, we introduce four algorithms: PathStack [BKS02], TwigStack [BKS02], TwigStackList [LCL04], and TJFast [LLC+05]. PathStack is I/O and CPU optimal among all sequential algorithms that read the entire input, and its worst-case time complexity is independent of the sizes of intermediate results, but many intermediate results may not be part of any final answer; TwigStack is proposed to solve this problem, but it only outputs the useful intermediate results for ancestor–descendant edges; unlike TwigStack, TwigStackList takes into account the level information of elements and consequently results in much smaller intermediate path solutions for query twig patterns with both ancestor–descendant and parent–child edges; as for TJFast, using the extended Dewey ID, this algorithm only scans elements for query leaf nodes, not only accesses much fewer elements, but can process queries with wildcards in internal nodes.

In addition, we show two new streaming schemes: Tag+level scheme and Prefix-Path Streaming, which can provide optimal solutions for a larger class of twig patterns. iTwigJoin [CLL05], based on these streaming schemes, can reduce input I/O cost and redundant intermediate result sizes to achieve better performance.

References

[AJP+02] Al-khalifa, S., Jagadish, H.V., Patel, J.M., Wu, Y., Koudas, N., Srivastava, D.: Structural joins: a primitive for efficient XML query pattern matching. In: Proceedings of the 20th International Conference on Data Engineering, San Jose, pp. 141–152 (2002)

[BG03] Bremer, J.M., Gertz, M.: An efficient XML node identification and indexing scheme. Technical Report CSE-2003-04, University of California at Davis (2003)

[BKS02] Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. Technical Report, Columbia University (2002)

[CDF+94] Carey, M.J., Dewitt, D.J., Franklin, M.J., Hall, N.E., Mcauliffe, M.L., Naughton, J.F., Schuh, D.T., SolomoN, M.H., Tan, C.K., Tsatalos, O.G., White, S.J., Zwilling, M.J.: Shoring up persistent applications. In: Proceedings of SIGMOD (1994)

[CLC04] Chen, T. Ling, T.W., Chan, C.Y.: Prefix path streaming: a new clustering method for optimal XML twig pattern matching. In: Proceedings of DEXA 2004, Zaragoza, pp. 801–811 (2004)

[CLL05] Chen, T., Lu, J., Ling, T.W.: On boosting holism in XML twig pattern matching using structural indexing techniques. In: Proceeding of the ACM SIGMOD International Conference on Management of Data, Baltimore (2005)

[CLO03] Chen, Q., Lim, A., Ong, K.W.: D(k)-index: an adaptive structural summary for graph-structured data. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, San Diego, pp. 134–144 (2003)

[HY04] He, H., Yang, J.: Multiresolution indexing of XML for frequent queries. In: Proceedings of ICDE 2004, Boston, pp. 683–694 (2004)

[KBNK02] Kaushik, R., Bohannon, P., Naughton, J.F., Korth, H.F.: Covering indexes for branching path queries. In: Proceedings of SIGMOD 2002, Madison (2002)

[KSBG02] Kaushik, R., Shenoy, P., Bohannon, P., Gudes, E.: Exploiting local similarity for efficient indexing of paths in graph structured data. In: Proceedings of ICDE 2002, San Jose, pp. 129–140 (2002)

[LCL04] Lu, J., Chen, T., Ling, T.W.: Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In: CIKM, Washington, DC, pp. 533–542 (2004)

[LLC+05] Lu, J., Ling, T.W., Chan, C., Chen, T.: From region encoding to extended Dewey: on efficient processing of XML twig pattern matching. In: Proceedings of 31th International Conference on Very Large Data Bases (VLDB), Trondheim, pp. 193–204 (2005)

[LM01] Li, Q., Moon, B.: Indexing and querying XML data for regular path expressions. In: Proceedings of the 27th VLDB Conference, Rome, pp. 361–370, Sept 2001

[MS99] Milo, T., Suciu, D.: Index structures for path expressions. In: Proceedings of ICDT 99, Jerusalem, pp. 277–295 (1999)

[TVB+02] Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J., Zhang, C.: Storing and querying ordered XML using a relational database system. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Madison, pp. 204–215 (2002)

[WLH04] Wu, X., Lee, M., Hsu, W.: A prime number labeling scheme for dynamic ordered XML trees. In: Proceeding of ICDE, Boston, pp. 66–78 (2004)

[ZND+01] Zhang, C., Naughton, J.F., Dewitt, D.J., Luo, Q, Lohman, G.M.: On supporting containment queries in relational database management systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Santa Barbara, pp. 425–436 (2001)

Chapter 5
Ordered and Generalized XML Tree Pattern Processing

Abstract Although holistic twig join approach algorithm has been proven an efficient way to match twig pattern, it cannot deal with some special patterns efficiently such as ordered XML tree pattern, generalized tree pattern, and extended XML tree pattern. Ordered XML twig query means a twig query which cares the order of the matching elements. In this chapter, we propose a holistic-processing algorithm called OrderedTJ to guarantee the I/O optimality. Generalized tree pattern is modeled by XQuery statements with multiple path expressions, and we introduce GTJFast to process this kind of patterns to reduce the I/O cost. Finally, extended XML tree pattern includes not only P-C/A-D relationships but also negation functions, wildcards, and order restriction. We also show holistic algorithms to efficiently process such patterns in details.

Keywords Algorithm • XML • Tree pattern

5.1 Introducing Ordered and Generalized XML Tree Pattern Processing

Holistic twig join approach has been taken as an efficient way to match twig pattern since this approach can efficiently control the size of intermediate results, like the Algorithms TwigStack (mentioned in Chap. 4) and iTwigJoin [CLL05]. However, the existing work on holistic twig query matching still has some disadvantages.

First, holistic twig join algorithms only considered unordered twig queries. But XPath defines four ordered axes: following-sibling, preceding-sibling, following, preceding. For the ordered twig queries, a straightforward approach that first matches the unordered twig queries and then prunes away the undesired answers is obviously not optimal in most cases. To handle ordered twig pattern, Lu et al. [LLY+05] proposed a holistic-processing algorithm, called OrderedTJ, which made the extension of TwigStackList [LCL04] to identify a large query class to guarantee the I/O optimality.

J. Lu, *An Introduction to XML Query Processing and Keyword Search*,
DOI 10.1007/978-3-642-34555-5_5,

Second, in XQuery, there may be multiple path expressions in the FOR, LET, WHERE, and RETURN clauses, all with different semantics. XQuery statements can be modeled as a generalized tree patterns. In this chapter, we will introduce an algorithm called GTJFast to efficiently process generalized tree pattern (GTP) by exploiting the non-output nodes in GTP to reduce the I/O cost.

Third, previous algorithms focus on XML tree pattern queries with only P-C and A-D relationships. Little work has been done on XML tree queries which may contain wildcards, negation function, and order restriction, all of which are frequently used in XML query languages such as XPath and XQuery. A large set of XML tree pattern, called extended XML tree patter was defined, which may include P-C and A-D relationships, negation functions, wildcards, and order restriction. Also, they developed a set of theorems, called "matching cross," to show the intrinsic reasons for the suboptimality of existing algorithms. Based on the theorems, a set of holistic algorithms to efficiently process the extended XML tree patterns were proposed. In this chapter, we will show the algorithms in detail and make experiments to demonstrate the effectiveness.

5.2 XML Ordered Query Processing

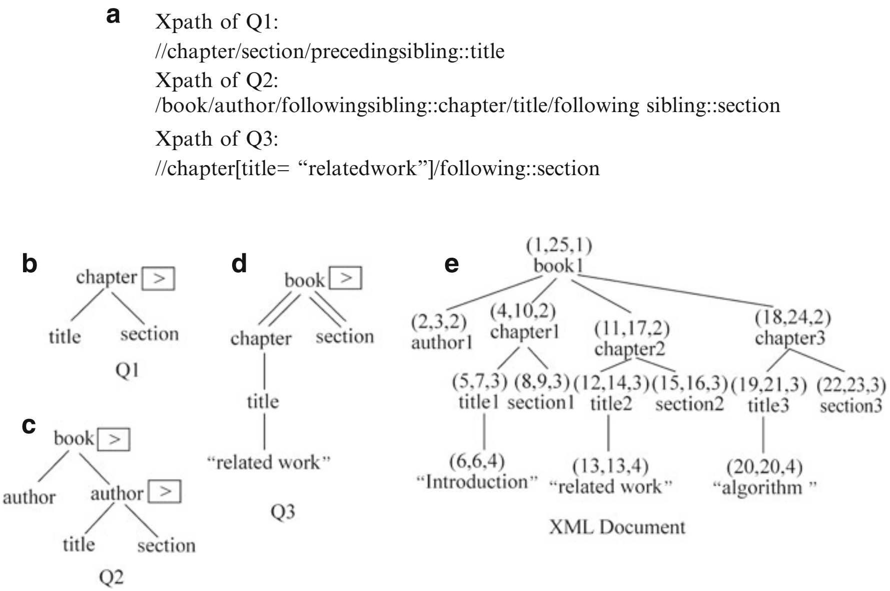
A twig query which cares the order of the matching elements can be called as an *ordered* twig query. On the other hand, a twig query that does not consider the order of matching elements can be called as an unordered query.

The existing work on holistic twig query matching only considered unordered twig queries. But XPath defines four ordered axes: following-sibling, preceding-sibling, following, and preceding. For example, XPath: //book/text/following-sibling::chapter is an ordered query, which finds all chapters in the dataset that are following siblings of text which should be a child of book.

To handle an ordered twig query, naively, the existing algorithms like TwigStack and TwigStackList (introduced in Chap. 4) output the intermediate path solutions for each individual root-leaf query path and then merge path solutions so that the final solutions are guaranteed to satisfy the order predicates of the query. Although existing algorithms are applied, such a post-processing approach has a serious disadvantage: many intermediate results may not contribute to final answers.

In this chapter, we will introduce a holistic algorithm, which is proposed by Lu et al. [LLY+05] for ordered twig queries, called OrderedTJ. The algorithm is based on the new concept of Ordered Children Extension (for short, OCE). With OCE, an element contributes to final results only if the order of its children accords with the order of corresponding query nodes. Thus, efficient holistic algorithm for ordered twigs can be leveraged.

If edges between branching nodes and their children are called as branching edges and the branching edge connecting to the nth child as the nth branching edge, OrderedTJ is I/O optimal among all sequential algorithms that read the entire input when the ordered twig contains only ancestor–descendant relationship from



a Xpath of Q1:
//chapter/section/precedingsibling::title
Xpath of Q2:
/book/author/followingsibling::chapter/title/following sibling::section
Xpath of Q3:
//chapter[title= “relatedwork”]/following::section

Fig. 5.1 (**a**) Three XPaths; (**b**)–(**d**) the corresponding ordered twig query; (**e**) an XML Tree

the second branching edge. In other words, the optimality of OrderedTJ allows the existence of parent–child relationships in non-branching edges and the first branching edges.

5.2.1 Data Model and Ordered Twig Pattern

An ordered XML data tree is shown in Fig. 5.1e. Each tree element is assigned a region code (start, end, level) based on its position. Each text is assigned a region code that has the same start and end values.

XML queries make use of twig patterns to match relevant portions of data in an XML database. The pattern edges are parent–child or ancestor–descendant relationships. Given an ordered twig pattern Q and an XML database D, a match of Q in D is identified by a mapping from the nodes in Q to the elements in D, such that (1) the query node predicates are satisfied by the corresponding database elements and (2) the parent–child and ancestor–descendant relationships between query nodes are satisfied by the corresponding database elements and (3) the *orders* of query sibling nodes are satisfied by the corresponding database elements. In particular, with region encoding, given any node $q \in Q$ and its right-sibling $r \in Q$ (if any), their corresponding database elements, say e_q and e_r in D, must satisfy that e_q.start < e_r.start.

The answers to query Q with n nodes can be represented as a list of n-ary tuples, where each tuple $(t_1, t_2, \ldots, t_n)$ consists of the database elements that identify a distinct match of Q in D.

Figure 5.1a shows three sample XPaths, and Fig. 5.1b–d shows the corresponding ordered twig patterns for the data of Fig. 5.1e. For each branching node, a symbol ">" is used in a box to mark its children ordered. Note that in Q_3, book is added as the root of the ordered query since it is the root of XML document tree. For example, the query solution for Q_3 is only <book1, chapter2, title2," related word", section$_3$>. But if Q_3 was an unordered query, section$_1$ and section$_2$ also would involve in answers.

5.2.2 XML Ordered Query Processing Algorithm

Here, we present the algorithm of OrderedTJ for finding all matches of an ordered twig pattern against an XML document. OrderedTJ makes the extension of TwigStackList algorithm proposed by Lu et al. [LCL04] to handle ordered twig pattern.

5.2.2.1 Data Structures and Notations

The data structures and notations to be used by OrderedTJ are as follows.

There is a data stream T_n associated with each node n in the query twig. C_n is used to point to the current element in T_n. The values of C_n can be got from C_n.start, C_n.end, and C_n.level. The cursor can advance to the next element in T_n with the procedure advance(T_n). Initially, C_n points to the first element of T_n.

The algorithm will use two types of data structures: list and stack. Associated with each node of queries, there are a list L_n and a stack S_n. At every point during computation, the nodes in stack S_n are guaranteed to lie on a root-leaf path in the database. Top(S_n) is denoted as the top element in stack S_n. Similarly, elements in each list L_n are also strictly nested from the first to the end, that is, each element is an ancestor or parent of that following it. For each list L_n, there is an integer variable, say p_n, as a cursor to point to an element in L_n. Initially, $p_n = 0$, which points to the first element of L_n.

The challenge to ordered twig evaluation is that, even if an element satisfies the parent–child or ancestor–descendant relationship, it may not satisfy the order predicate. So a new concept is introduced, namely, Ordered Children Extension (for short, OCE), which is important to determine whether an element likely involves in ordered queries.

Definition 5.1 (OCE) Given an ordered query Q and a dataset D, we say that an element *en* (with tag $n \in Q$) in D has an Ordered Children Extension (for short, OCE), if the following properties are satisfied:

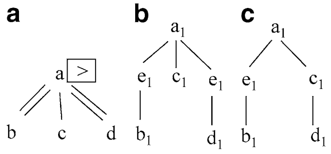


Fig. 5.2 Illustration to ordered child extension. (**a**) Query, (**b**) Doc1, (**c**) Doc2

1. For $n_i \in$ ADRChildren(n) in Q (if any), there is an element e_{n^i} (with tag n_i) in D such that e_{n^i} is a descendant of e_n and e_{n^i} also has OCE.
2. For $n_i \in$ PCRChildren(n) in Q (if any), there is an element e' (with tag n) in the path e_n to e_{n^i} such that e' is the parent of e_{n^i} and e_{n^i} also has OCE.
3. For each child n_i of n and $m_i =$ rightSibling(n_i) (if any), there are elements e_{n^i} and e_{m^i} such that e_{n^i} end $<$ e_{m^i} start and both e_{n^i} and e_{m^i} have OCE.

Properties (1) and (2) discuss the ancestor–descendant and parent–child relationship, respectively. Property (3) manifests the order condition of queries. For example, see the ordered query in Fig. 5.2a; in Doc1, a_1 has the OCE, since a_1 has descendants b_1, d_1, and child c_1 and more importantly, b_1, c_1, and d_1 appear in the correct order. In contrast, in Doc2, a_1 has not the OCE, since d_1 is the descendant of c_1, but not the following element of c_1 (i.e., c_1.end $\nless$ d_1.start).

Algorithm OrderedTJ

OrderedTJ, which computes answers to an ordered query twig, operates in two phases. In the first phase (Lines 1–7), the individual query root-leaf paths are output. In the second phase (Line 8), these solutions are merged and joined to compute the answers to the whole query. Here, we first explain getNext algorithm which is a core function and then presents the main algorithm in details.

getNext(n)(see Algorithm 5.1) is a procedure called in the main algorithm of OrderedTJ. It identifies the next stream to be processed and advanced. At Lines 4–8, the condition (1) of OCE is checked. Note that unlike the previous Algorithm TwigStackList, in Line 8, the maximal (not minimal) element that are not descendants of the current element in stream T_n is advanced, as it will be used to determine sibling order. Lines 9–12 check the condition (3) of OCE. Lines 11 and 12 return the elements which violate the query sibling order. Finally, Lines 13–19 check the condition (2) of OCE.

Algorithm 5.1 getNext (n)(in OrderedTJ)

```
1. if (isLeaf(n)) return n;
2. for all $n_i$ in children(n) do
3.   $g_i$=getNext($n_i$);   if ($g_i \neq n_i$)   return $n_i$;
4. $n_{max}$=$\text{maxarg}_{ni \in children(n)}$getStart(ni);
5.  $n_{min}$=$\text{minarg}_{ni \in children(n)}$getStart($n_i$);
6. While (getEnd(n) < getStart($n_{max}$))   proceed(n);
7. if (getStart(n) >getStart($n_{min}$))
```

```
8.    return $\text{maxarg}_{ni \in children(n) \wedge getStart(n) > getStart(ni)}$ getStart(ni);
9.    sort all $n_i$ in children(n) by start values;
10. // assume the new order are $n_1', n_2', \ldots, n_k'$
11. for each $n_i'(1 \le i \le n)$   do   //check children order
12.    if $(n_i' \neq n_i)$   return $n_i'$;
13.    else if ((i>1)(getEnd($n_{i-1}'$–1)>getStart($n_i'$))) return $n_{i-1}'$
14. MoveStreamToList(n, $n_{max}$);
15. for $n_i$ in PCRchildren(n) //check parent-child relationship
16.    if ($\exists$ e' $\in L_n$ such that e' is the parent of $C_{ni}$)
17.    if ($n_i$ is the first child of n)
18.     Move the cursor of list $L_q$ to point to e';
19.    else return $n_i$;
20. return n;
```

Now we discuss the main algorithm of OrderedTJ (Algorithm 5.2). First of all, Line 2 calls getNext algorithm to identify the next element to be processed. Line 3 removes partial answers that cannot be extended to total answer from the stack. In Line 4, when a new element is inserted to stack, it needs to be checked whether it has the appropriate right sibling. If *n* is a leaf node, the whole path solution will be output in Line 6.

Algorithm 5.2: OrderedTJ ()

```
1. While (!end( ))
2.    $q_{act}$ =getNext(root);
3.    if (isRoot($q_{act}$)$\vee$!empty($S_{parent(qact)}$))   cleanStack($q_{act}$, getEnd($q_{act}$));
4.    moveStreamToStack ($q_{act}$, $S_{qact}$);
5.    if (isLeaf($q_{act}$))
6.    showpathsolutions ($S_{qact}$, getElement($q_{act}$));
7.    else proceed($T_{qact}$);
8. mergeALLPathsolutions;
```

Function end()

```
return $\forall$ n$\in$subtreesNodes(root):isLeaf(n)$\vee$eof(Cn);
```

Procedure cleanStack (n, actEnd)

```
1. while (!empty($S_n$) and (topEnd($S_n$)< actEnd)) do pop($S_n$);
```

Procedure moveStreamToStack(n, S_n)

```
2. if((getEnd(n) < top($S_{rightsibling(n)}$.start)   //check order
3. push getElement(n) to stack $S_n$
4. proceed(n);
```

Procedure proceed(n)

```
5. if (empty($L_n$))   advance($T_n$);
6. else Ln.delete(pn);
7.    $p_n$=0;   //move $p_n$ to pint to the beginning of $L_n$
```

Procedure showpathsolutions(S_m, e)

```
index[m]= e
if (m== root)    //we are in root
Output(index[$q_1$], . . . ,index[$q_k$])    //k is the length of path processed
else    //recursive call
  for each element $e_i$ in $S_{parent(m)}$
    if $e_i$ satisfies the corresponding relationship with e
      showpathsolutions($S_{parent(m)}$, $e_i$)
```

Example 5.1 Consider the ordered query and data in Fig. 5.1d, e again. First of all, the five cursors are (book$_1$, chapter$_1$, title$_1$," related work", section$_1$). After two calls of getNext(book), the cursors are forwarded to (book$_1$, chapter$_2$, title$_2$," related work", section$_1$). Since section$_1$.start = 6 < chapter$_2$.start =9, we return section (in Line 11 of Algorithm 5.1) and forward to section$_2$. Then chapter$_2$.end =15 > section$_2$.start = 13. We return section again (in Line 12 of getNext) and forward to section$_3$. Then chapter$_2$.end =15 > section$_3$.start = 17. The following steps push book$_1$ to stack and output the individual two path solutions. Finally, in the second phase of main algorithm, two path solutions are merged to form one final answer.

5.2.3 Analysis of OrderedTJ

Here, we show the correctness of OrderedTJ and analyze its efficiency.

Definition 5.2 (Head Element e_n) In OrderedTJ, for each node in the ordered query, if List L_n is not empty, then we say that the element indicated by the cursor p_n of L_n is the head element of n, denoted by e_n. Otherwise, we say that element C_n in the stream T_n is the head element of n.

Lemma 5.1 *Suppose that for an arbitrary node n in the ordered query, we have* getNext(n) = n'. *Then the following properties hold*:

1. *n' has the OCE.*
2. *Either* (a) $n = n'$ *or* (b) parent(n) *does not have the OCE because of n'* (*and possibly a descendant of n'*).

Lemma 5.2 *Suppose* getNext(n) = n' *returns a query node in the Line 11 or 12 of Algorithm 5.1. If the current stack is empty, the head elementdoes not contribute to any final solution since it does not satisfy the order condition of query.*

Lemma 5.3 *In Procedure moveStreamToStack, any element e that is inserted to stack S_n satisfies the order requirement of the query. That is, if n has a right-sibling node n' in query, then there is an element $e_{n'}$ in stream $T_{n'}$ such that* $e_{n'}$.start > e_n.end.

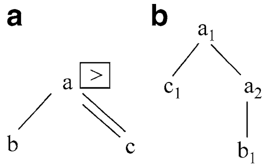


Fig. 5.3 Optimality example. (**a**) Query, (**b**) data

Lemma 5.4 *In OrderedTJ, when any element e is popped from stack, e is guaranteed not to participate a new solution any longer.*

Theorem 5.1 *Given an ordered twig pattern Q and an XML database D, Algorithm 5.2 correctly returns all answers for Q on D.*

Proof [*Sketch*] Using Lemma 5.2, we know that when getNext returns a query node n in Lines 11 and 12 of getNext, if the stack is empty, the head element e_n does not contribute to any final solutions. Thus, any element in the ancestors of n that use e_n in the OCE is returned by the getNext before e_n. By using Lemma 5.3, we guarantee that each element in stack satisfies the order requirement in the query. Further, by using Lemma 5.4, we can maintain that, for each node n in the query, the elements involve in the root-leaf path solution in the stack S_n. Finally, each time that $n =$ getNext(root) is a leaf node, we output all solution for e_n (Line 6 of Algorithm 5.2).

Now analyze the optimality of OrderedTJ. Recall that the unordered twig join Algorithm TwigStackList is optimal for query with only ancestor–descendant in all branching edges, but OrderedTJ can identify a little larger optimal class than TwigStackList for ordered query. In particular, the optimality of OrderedTJ allows the existence of parent–child relationship in the first branching edge, as illustrated below.

Example 5.2 Consider the ordered query and dataset in Fig. 5.3. If the query were an unordered query, then TwigStackList would scan a_1, c_1, and b_1 and output one useless solution (a_1, c_1), since before b_1 is advanced, we could not decide whether a_1 has a child tagged with b. But since this is an ordered query, it is immediate to identify that c_1 does not contribute to any final answer since there is no element with name b before c_1. Thus, this example tells us that unlike algorithms for unordered query, OrderedTJ may guarantee the optimality for queries with parent–child relationship in the first branching edge.

Theorem 5.2 *Consider an XML database D and an ordered twig query Q with only ancestor–descendant relationships in the nth $n \geq 2$ branching edge. The worst-case I/O complexity of OrderedTJ is linear in the sum of the sizes of input and output lists.*

The worst-case space complexity of this algorithm is that the number of nodes in Q times the length of the longest path in D.

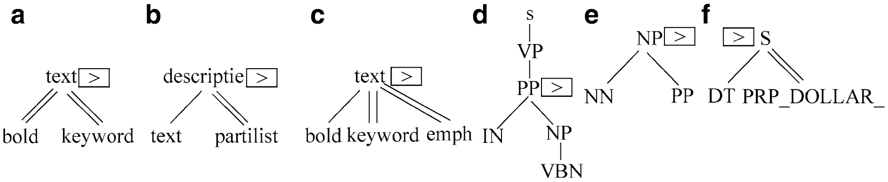


Fig. 5.4 Six tested ordered twig queries (Q_1, Q_2, Q_3 in XMark; Q_4, Q_5, Q_6 in TreeBank). (**a**) Q_1, (**b**) Q_2, (**c**) Q_3, (**d**) Q_4, (**e**) Q_5, (**f**) Q_6

5.2.4 Experimental Evaluation

5.2.4.1 Experimental Setup

Here, three ordered twig join algorithms are implemented: straightforward-TwigStack (for short, STW), straightforward-TwigStackList (STWL), and OrderedTJ. The first two algorithms use the straightforward post-processing approach. By post-processing, it means that the query is first matched as an unordered twig (by TwigStack and TwigStackList, respectively) and then merges all intermediate path solutions to get the answers for an ordered twig. This experiment uses JDK 1.4 with the file system as a simple storage engine. All experiments were run on a 1.7 G Pentium IV processor with 768 MB of main memory and 2 GB quota of disk space, running windows XP system. Two datasets are used for experiments. The first is the well-known benchmark data: XMark. The size of file is 115 M bytes with factor 1.0. The second is a real dataset: TreeBank [TB]. The deep recursive structure of this dataset makes this an interesting case for experiments. The file size is 82 M bytes with 2.4 million nodes.

For each data set, three XML twig queries are tested (see Fig. 5.4). These queries have different structures and combinations of parent–child and ancestor–descendant edges. These queries are chosen to give a comprehensive comparison of algorithms.

The following metrics will be used to compare the performance of different algorithms:

1. Number of intermediate path solutions: This metric measures the total number of intermediate path solutions, which reflects the ability of algorithms to control the size of intermediate results.
2. Total running time: This metric is obtained by averaging the total time elapsed to answer a query with six consecutive runs, and the best and worst performance results discarded.

5.2.4.2 Performance Analysis

Figure 5.5 shows the results on execution time. An immediate observation from the figure is that OrderedTJ is more efficient than STW and STWL for all queries. This

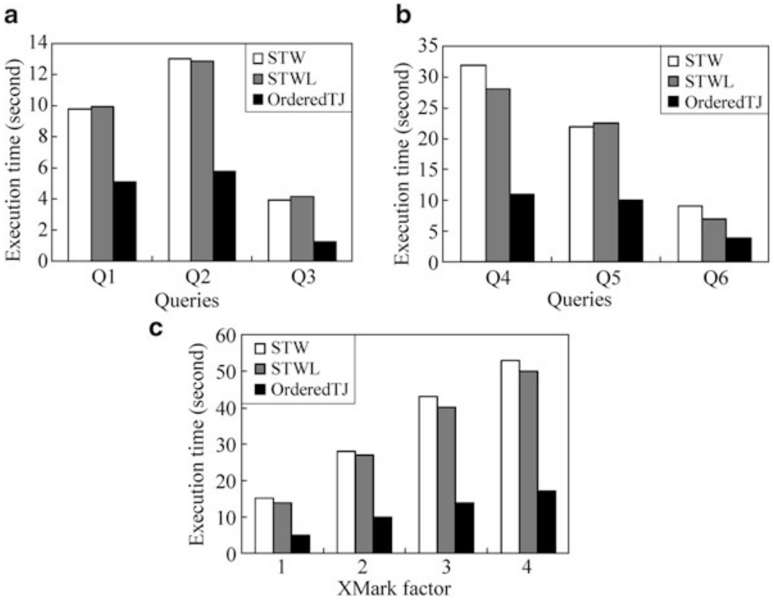


Fig. 5.5 Evaluation of ordered twig pattern on two datasets. (**a**) XMark, (**b**) TreeBank, (**c**) Varying XMark size

Table 5.1 The number of intermediate path solutions

Query	Dataset	STW	STWL	OrderedTJ	Useful solutions
Q_1	Xmark	71,956	71,956	44,382	44,382
Q_2	Xmark	65,940	65,940	10,679	10,679
Q_3	Xmark	71,522	71,522	23,959	23,959
Q_4	TreeBank	2,237	1,502	381	302
Q_5	TreeBank	92,705	92,705	83,635	79,941
Q_6	TreeBank	10,663	11	5	5

can be explained that OrderedTJ outputs much less intermediate results. Table 5.1 shows the number of intermediate path solutions. The last column shows the number of path solutions that contribute to final solutions. For example, STW and STWL could output 500 % more intermediate results than OrderedTJ (see XMark Q_2).

Queries XMark Q_2 are tested for scalability with XMark factor1(115 MB), 2(232 MB), 3 (349 M), and 4(465 M). As shown in Fig. 5.5c, OrderedTJ scales linearly with the size of the database. With the increase of data size, the benefit of OrderedTJ over STW and STWL correspondingly increases.

s is explained before; when there is any parent–child relationship in the *n*th branching edges $n \geq 2$, OrderedTJ is not optimal. As shown in Q_4 and Q_5 of Table 5.1, none of the algorithms is optimal, since all algorithms output some useless solutions. However, even in this case, OrderedTJ still outperforms STW and STWL by outputting less useless intermediate results.

According to the experimental results, there are two conclusions. First, this Algorithm OrderedTJ could be used to evaluate ordered twig pattern because they have obvious performance advantage over the straightforward approach: STW and STWL. Second, OrderedTJ guarantees the I/O optimality for a large query class.

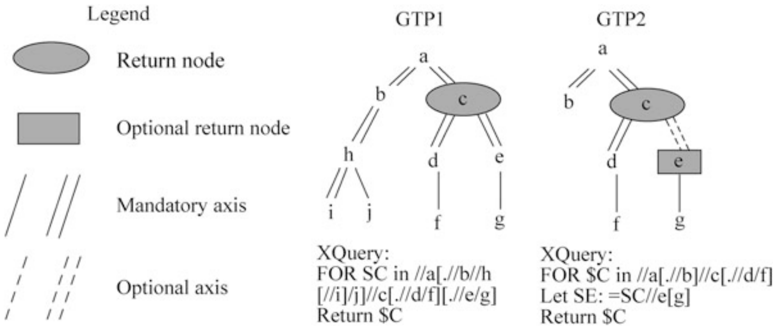
5.3 XML Generalized XML Tree Pattern

In XQuery [BCF+07], there may be multiple path expressions in the FOR, LET, WHERE, and RETURN clauses, all with different semantics. Existing works [ZND+01] show that XQuery can be modeled as a generalized tree pattern (GTP) in order to capture these semantics. Figure 5.6 depicts two sample XQuery statements and their respective GTPs.

For Fig. 5.6, in XQuery1, node *c* is a return node, that is, only its result is of interest. In XQuery2, node *e* is optional (in general, any expression in the LET or RETURN clauses is optional) in the sense that a *c* element can be a match even without any descendant *e* elements. Any matching *e* elements, however, must be grouped together under their common *c* ancestor elements. Most existing works on holistic twig query processing like TwigStackList, TJFast, and TwigList [QYD07] focus only on returning the entire twig results. In practice, however, returning the entire twig results is seldom necessary for either XPath or XQuery and may consequently cause duplicate elimination and/or ordering problems. Moreover, many XQuery statements in practice require grouping the results. Applying post-duplicate elimination, sorting and grouping operations to address these problems have already been shown to be expensive in many existing works [BKS02, JWLY03, LCL05].

Here, we introduce an Algorithm GTJFast [LU06] to efficiently process GTP pattern by exploiting the non-output nodes in GTP to reduce the I/O cost.

In the following, we first present the intuition in the optimization to process the non-return nodes and optional return nodes in GTP and then formally show the Algorithm GTJFast followed by the theoretical analysis.



```
XQuery:
FOR SC in //a[.//b//h
[//i]/j]//c[.//d/f][.//e/g]
Return $C
```

```
XQuery:
FOR $C in //a[.//b]//c[.//d/f]
Let SE: =SC//e[g]
Return $C
```

Fig. 5.6 Examples of GTP

5.3.1 *GTJFast Algorithm*

5.3.1.1 Optimization on Non-return Nodes

Non-return nodes do not appear in the final results, so all elements of such nodes are not required to be buffered during the processing. But in order to get how their existences determine the output nodes, it is important to judiciously and compactly record the existences of non-return elements.

Intuitively, see the example in GTP1 again in Fig. 5.6. All nodes in queries are classified to four categories according to their properties:

1. *Non-branching and non-return nodes.* An example is node *b* in GTP1. For this type of nodes, it is easy to derive the whole path by extended Dewey labeling scheme and determine whether they contribute to one path solution or not. Therefore, it is not necessary to buffer the elements of this type in the main memory.
2. *Non-branching and return nodes.* An example is node *e* in GTP2.This type of nodes should be stored in the main memory. However, as they are not branching nodes, their children information will not be remembered.
3. *Branching and non-return nodes.* An example is node *h* in GTP1.Those elements are not required to buffer in the main memory, but as they are branching nodes, the information of their descendants and children would be maintained, which can achieved by "BitArray" in the algorithm.
4. *Branching and return nodes.* An example is node *c* in GTP1. Those elements have to be buffered in the main memory, and the corresponding descendants/children relationship should be recorded (by "BitArray").

5.3.1.2 Optimization on Optional Return Nodes

The existence of optional return nodes in the final answer is not mandatory. Take GTP2 as an example; node *e* is an optional return node. That is, a *c* element can be a match even without any descendant *e* elements. Any matching *e* elements, however, must be grouped together under their common *c* ancestor elements. To process optional return nodes, the main approach is the same, and the only additional job is to mark the optional branch and check its existence when outputting the results.

5.3.1.3 Algorithm

Next, we give the description of this algorithm, starting with the introduction of some data structures.

GTJFast keeps a set S_b for branching node *b* during execution, and each two elements in set S_b have an ancestor–descendant or parent–child relationship. Different from TJFast, introduced in the Chap. 4, each element *e* with type *b* in the set is associated with a "BitArray(*e*)" and "outputList(*e*)." The length of

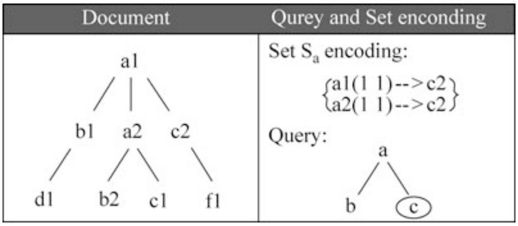


Fig. 5.7 Example of set encoding in GTJFast

a BitArray equals the number of children of b in the query. Given a child c of b, BitArray(e, c) = 1 if and only if the corresponding relationship (P-C or A-D relationship) between c and b is satisfied in the data. In addition, each element e' in outputList(e) is a potential final result related to e, where b' is a child/ descendant node of b in query if the type of e' is b', as illustrated in the following example:

Example 5.3 See the query and example document in Fig. 5.7, the BitArray of $a1$ is "11," which shows that $a1$ has the corresponding two children $b1$ and $c2$. Since c is the return node, $a1$ is associated with $c2$. Similarly, the BitArray of $a2$ is "11" and is associated with $c1$.

Algorithm 5.3: for GTJFast

```
for each f∈leafNodes(root)
    locateMatchedLabel(f)
end for
while (!end(root)) do
    $f_{act}$ = getNext(topBranchingNode)
    advance($T_{f_{act}}$)
    locateMatchedLabel($f_{act}$)
end while
emptyAllSets(root);
mergeAllPathSolutions();
```

Procedure emptyAllSets(q)

```
if (q is not a leaf node) then
    for each child c of q do
        emptyAllSets(c);
    end for
end if
for Each element e in $S_q$ do
    emptySet(q, e);
end for
```

Procedure emptySet(q, e)

```
if (all bits in BitArray(e) are '1') then
    if (q is an output node) then
        Add e to the output list of the nearest ancestor branching node of q;
    end if
    if (q is the top branching node) then
        output the outputList(e);
    end if
end if
```

Algorithm 5.4: updateSet(S,e)

Procedure updateSet(S, e)

```
clearSet(S,e);
add e to set S;
Assume that the query node type of S is q;
for $\forall$ q', s.t. q' is a child of q do
    if($\exists$ e' with type q's.t. e' satisfy the relationship between q and q')then
        BitArray(e, q')=1;
    else
        BitArray(e, q')=0;
    end if
end for
if (all bits in BitArray(e) are '1') then
    if (q is not the top branching node) then
        updateAncestorSet(q, e);
    else
        output the outputList(e);
    end if
end if
```

Procedure updateAncestorSet (q, e)

```
Assume the nearest ancestor branching node of q is q' in the query;
for any element e' in $S_{e'}$ do
    if (BitArray(e',q)=='0') then
        Set BitArray(q', q) be '1';
        Add all elements in outputList(e) to outputList(e');
        if((all bits in BitArray(b)are '1')$\wedge$(q is not the top branching node)) then
            updateAncestorSet (q', e);
        end if
    end if
end for
```

The main idea between GTJFast (see Algorithms 5.3 and 5.4) and TJFast is the same. That is, only the labels of leaf query nodes are accessed, and the partial matching results are maintained in sets, and finally, the results are output and merged. However, the differences between GTJFast and TJFast are summarized as follows:

1. In the procedure updateSet(S, e), GTJFast additionally bottom-up updates the BitArray's of e and its ancestor branching nodes as well.
2. In the new procedure emptyAllSets(q) which does not exist in TJFast, GTJFast clears the set S_q and recursively clears all sets $S_{q'}$, $q' \in$children(q).

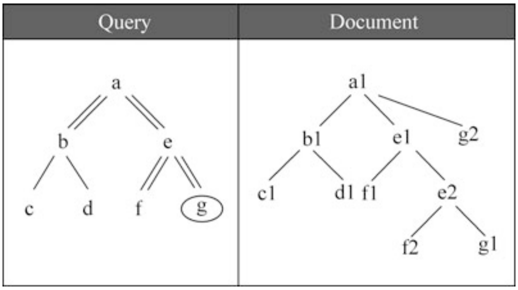


Fig. 5.8 An example to illustrate GTJFast

Table 5.2 Set encoding (g_2 does not satisfy query path pattern, so it is excluded)

Returned node from getNext	Set(a)	Set(b)	Set(e)
c_1	$\{a_1\ (1\ 0)\}$	$\{b_1\ (1\ 1)\}$	NULL
d_1	$\{a_1\ (1\ 0)\}$	$\{b_1\ (1\ 1)\}$	NULL
f_1	$\{a_1\ (1\ 1)\}$	$\{b_1\ (1\ 1)\}$	$\{e_1\ (1\ 1)\}$
f_2	$\{a_1\ (1\ 1)\}$	$\{b_1\ (1\ 1)\}$	$\{e_2\ (1\ 1), e_1\ (1\ 1)\}$
g_1	$\{a_1\ (1\ 1) \rightarrow g_1\}$	$\{b_1\ (1\ 1)\}$	$\{e_2\ (1\ 1), e_1\ (1\ 1)\}$

3. For each element e to delete in S_q, in the case where all bits in BitArray(e) are true, showing that the subtree rooted with q is satisfied, if q is an output nodes, GTJFast adds e to the corresponding outputList, and if q is the top branching nodes, GTJFast outputs elements in the corresponding outputList.
4. In the procedure mergeAllPathSolutions(), unlike TJFast where the output elements may be useless to the final results, all elements output in GTJFast are guaranteed to contribute to the final results. The merge algorithm is implemented by the traditional sorted multiple-way merge join.

Example 5.4 Consider the query and document in Fig. 5.8. g is the only return node. A set is associated with each branching node. Table 5.2 shows the states of the sets when the getNext() repeatedly returns the current elements. In the first iteration, c_1, d_1, f_1, and $g1$ are scanned, and c_1 is return from getNext. As b_1 has two children, its BitArray is "11." After d_1 is returned, b_1 is deleted from S_b, and its existence is recorded in $a_1(10)$, which shows that the subtree rooted with b_1 satisfies the query. Consequently, f_1, f_2, g_1, and g_2 are returned from getNext.

When g_1 is returned, it is an output node, and we add it to the result lists of e_1 and e_2. When g_1 is returned, it is transferred to S_a. As the BitArray of a_1 is "11," g_1 is a final result. Note that we do not explicitly process g_2, as it does not satisfy the query path pattern.

5.3.2 Analysis of GTJFast

In this chapter, we first show the correctness of GTJFast and then analyze its complexity.

Lemma 5.5 *In Algorithm 5.3, suppose any element e is removed from set S_q, then e matches the subtree rooted with q if and only if all bits in* BitArray(e) *are* "1."

Proof In Procedure updateSet of Algorithm GTJFast, when any new element e is inserted to S_q, BitArray(e, q') = 1, where q' is one of children of q, if and only if there is an element e' with the type q' such that e' and e satisfy the relationship between q and q' (Lines 4–10). Therefore, when all bits of BitArray(e) are "1," there exist elements to satisfy all P-C and A-D relationships for element e.

Furthermore, in Procedure updateAncestorSet, the matching relationships are recursively updated bottom-up to the root in Lines 2–10. Therefore, e matches the subtree rooted with q if and only if all bits in BitArray(e) are "1."

Theorem 5.3 *Given a generalized tree pattern Q and an XML database D, Algorithm GTJFast correctly returns all the answers for Q on D.*

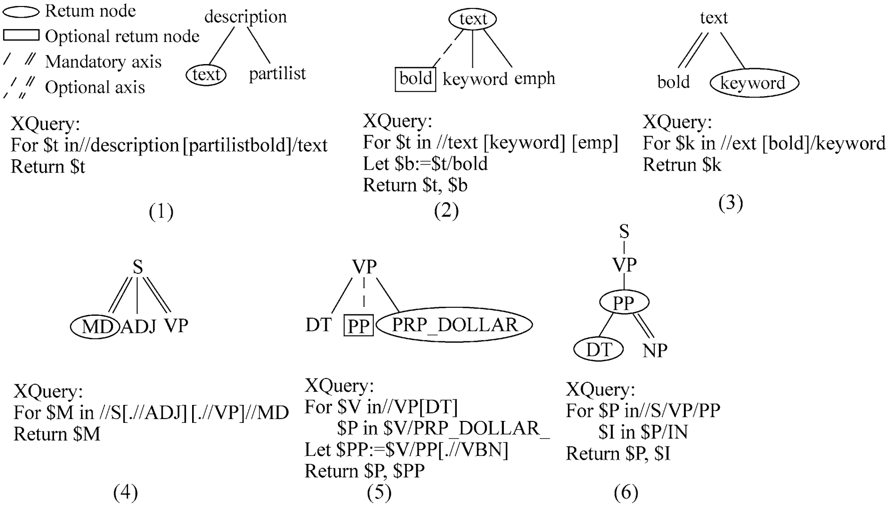
Proof In Procedure clearSet of Algorithm 5.3, any element e that is deleted from set S_b does not participate in any new solution. Function getNext guarantees that any element e that matches a branching node and participates in any final answer occurs in the set S_b. Thus, the insertion is complete. In Algorithm GTJFast, suppose any element e is removed from set S_q, where q is the top branching node in Procedure updateSet(q), then e matches the whole query if and only if all elements in outputList(e) belong to final query answers by Lemma 5.5, as all bits in BitArray(e) are "1" in this case. So, we get the correctness of the algorithm.

Theorem 5.4 *Consider an XML database D and a generalized tree query Q with only ancestor–descendant relationships between branching nodes and their children. The worst-case I/O complexity of GTJFast is linear in the sum of the sizes of input and output lists. The worst-case space complexity is* O($d2*|B| + d*|L|$), *where* $|L|$ *is the number of leaf nodes in Q*, $|B|$ *is the number of branching nodes in Q, and d is the length of the longest label in the input lists.*

Proof GTJFast makes an optimization for non-output nodes and optional axis, and its space and I/O cost are no greater than GTJFast in the worst case.

The above theorem holds only for query with ancestor–descendant relationships connecting branching nodes. When the query contains parent–child relationships between branching nodes and their children, the worst-case I/O complexity of GTJFast is still linear in the sum of the sizes of input and output lists. As for memory space complexity, however, Algorithm GTJFast may contain the whole document in the main memory, and thus its space cost is O($m*|D|$), where n =max($d2$,$|L|$), where d is the length of the longest label in the input lists and $|L|$ is the number of leaf nodes in Q.

However, we claim that the worst case will not practically happen unlike the traditional main memory XML database that always stores the entire DOM tree in the memory before processing [BBC+02, CLT+06, WLH07]. GTJFast only stores document elements that satisfy some part of GTP at runtime. More specifically, there are the following constraints on the data to be stored. First, the



(1)

```
XQuery:
For $t in//description [partilistbold]/text
Return $t
```

(2)

```
XQuery:
For $t in //text [keyword] [emp]
Let $b:=$t/bold
Return $t, $b
```

(3)

```
XQuery:
For $k in //ext [bold]/keyword
Retrun $k
```

(4)

```
XQuery:
For $M in //S[.//ADJ] [.//VP]//MD
Return $M
```

(5)

```
XQuery:
For $V in//VP[DT]
      $P in $V/PRP_DOLLAR_
Let $PP:=$V/PP[.//VBN]
Return $P, $PP
```

(6)

```
XQuery:
For $P in//S/VP/PP
     $I in $P/IN
Return $P, $I
```

Fig. 5.9 Six generalized XML tree patterns

elements that have labels matching with the query need to be stored. Second, when the selectivity of GTP is high, only small portion of elements will be pushed in the set. Third, only elements for output nodes which satisfy the query subtree are stored. The elements which satisfy the smaller subtree, but not the whole query tree, would be removed from the main memory during the runtime. Hence, it is unlikely to keep the entire document in the memory in practice. In addition, as shown in the next subsection, the space cost of GTJFast is often smaller than state-of-the-art Algorithm Twig^{2}Stack [CLT+06] which also require to buffer the whole document in the worst case.

5.3.3 Experiments

XML Tree Pattern Matching Algorithms. GTJFast and Twig^{2}Stack [CLT+06] in JDK 1.4 use the file system as a simple storage engine. GTJFast is based on extended Dewey labeling scheme, and the Twig^{2}Stack uses region encoding labeling scheme. All experiments were run on a 1.7 G Pentium IV processor with 768 MB of main memory and 2 GB quota of disk space, running windows XP system.

Experiments are made on generalized XML tree pattern queries and compare holistic join Algorithms GTJFast and Twig^{2}Stack.There are six XML queries on XMark and TreeBank data (see Fig. 5.9). The tested queries have different generalized XML tree structures and combinations of return nodes, optional return nodes, mandatory axis, and optional axis.

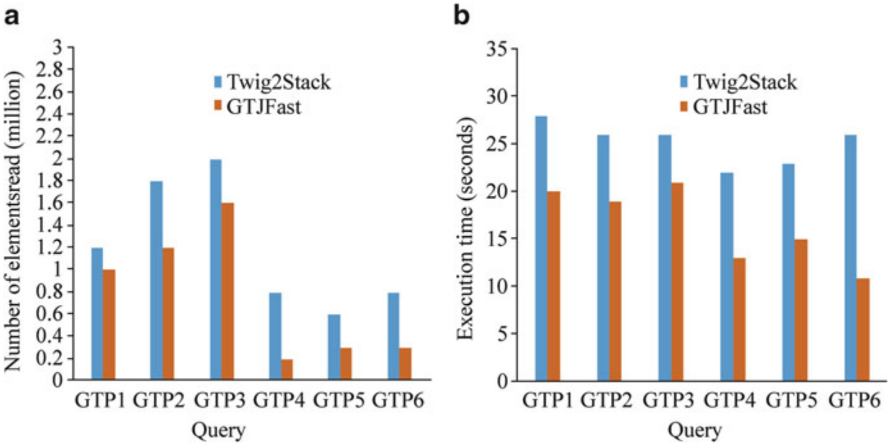


Fig. 5.10 GTJFast and Twig^{2}Stack. (**a**) # of element read, (**b**) execution time

5.3.3.1 GTJFast Versus Twig^{2}Stack

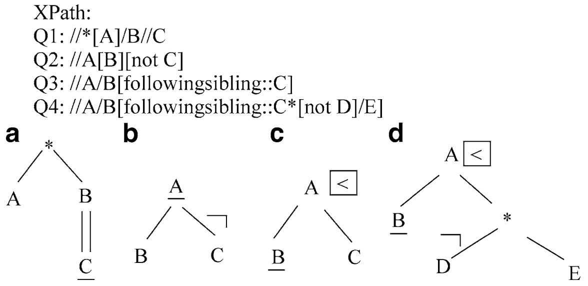
Compare GTJFast with Twig^{2}Stack. As shown in Fig. 5.10, GTJFast achieves better performance than Twig^{2}Stack. GTJFast maintains the BitArray in the main memory and avoids the output of useless results.

5.4 Extended XML Tree Pattern

Bruno et al. proposed a novel holistic approach named TwigStack [BKS02], which processes the tree pattern holistically without decomposing it into several small binary relationships. TwigStack guarantees that there are no "useless" intermediate results for queries with only ancestor–descendant (A-D) relationships. In other words, TwigStack is optimal for tree pattern queries with only A-D edges. After TwigStack, many other holistic algorithms such as TwigStackList, TSGeneric [JWLY03], iTwigJoin [CLL05], TJFast [LCL05], TwigStackList-No [YLL06], and Twig^{2}Stack [CLT+06] are proposed. These algorithms identify a larger class of query to guarantee the optimality than TwigStack. But there are the following three observations upon the above existing work.

First, previous XML tree pattern matching algorithms do not fully exploit the "optimality" of holistic algorithms. TwigStack guarantees that there is no useless intermediate result for queries with only A-D relationships. They called that TwigStack is optimal for queries with only A-D edges. But, this optimal query class can be enlarged! Another Algorithm TwigStackList enlarges its optimal class to contain P-C relationships in non-branching edges. A natural question is whether the optimal query subclass of TwigStackList can be further improved.

Fig. 5.11 Example extended XML tree pattern queries. "–" denotes the output node in query. (**a**) Q_1, (**b**) Q_2, (**c**) Q_3, (**d**) Q_4



XPath:
Q1: //*[A]/B//C
Q2: //A[B][not C]
Q3: //A/B[followingsibling::C]
Q4: //A/B[followingsibling::C*[not D]/E]

Note that theoretical analysis made by Choi et al. [CMW03] and Shalem and Bar-Yossef [SY08] shows that no algorithms are optimal for queries with any arbitrary combinations of A-D and P-C relationships. But the current open problems are (1) how to identify a large query class which can be processed optimally and (2) how to efficiently answer a query which cannot be guaranteed to process optimally.

Second, previous approaches solve the problem by producing the matching bindings for all nodes in a tree pattern. However, in a practical application, this requirement is not necessary. Take the XPath "*//A*[*B*]*//C*" as an example, only *C* element and its subtree are answers. In prior methods, like TwigStack, TJFast, and TSGeneric [JLW04], to answer this query, they first find the query answer with the combinations of all query nodes and then do an appropriate projection on those return nodes. Such a post-processing approach has an obvious disadvantage: it outputs many matching elements of non-return nodes that obviously are unnecessary for the final results. Here, we introduce a new encoding method proposed by somebody to record the mapping relationships and avoid outputting non-return nodes.

Third, previous algorithms focus on XML tree pattern queries with only P-C and A-D relationships. Little work has been done on XML tree queries which may contain wildcards, negation function, and order restriction, all of which are frequently used in XML query languages such as XPath and XQuery. Here, we define the extended XML tree pattern as an XML tree pattern with negation function, wildcards, and/or order restriction. Figure 5.11, for example, shows four extended XML tree patterns. Query (a) includes a wildcard node "*", which can match any single node in an XML database. Query (b) includes a negative edge, denoted by ".". This query finds *A* that has a child *B*, but has not child *C*. In XPath language, the semantic of negative edge can be presented with "not" Boolean function. Query (c) has order restriction, which is equivalent to an XPath "*//A/B*[following-sibling::*C*]". The "<" in a box shows that all children under *A* are ordered. The semantics of order-based tree pattern is captured by a mapping from the pattern nodes to nodes in an XML database such that the structural and ordered relationships are satisfied. Finally, query (d) is more complicated, which contains wildcards, negation function, and order restriction.

Thus, in general, given an extended XML tree pattern query which may include P-C and, A-D relationships, order restriction, negation function, and wildcards, we

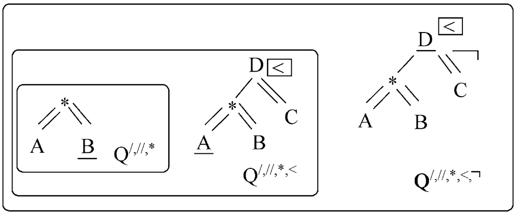


Fig. 5.12 Three categories of extended XML tree pattern and example optimal queries

consider the problem efficiently matching the extended XML tree query. Three categories of extended XML tree patterns will be investigated: (1) queries with P-C and A-D relationships and wildcards, denoted as $Q^{/,//,*}$; and (2) queries with P-C and A-D relationships, wildcards, and order restriction, denoted as $Q^{/,//,*,<}$; and (3) queries with P-C and A-D relationships, wildcards, order restriction, and negation function, denoted as $Q^{/,//,*,<,\neg}$. For each query class, we identify the respective a large subclass to be processed optimally (see Fig. 5.12). A holistic algorithm is "optimal" for a kind of query class, if it guarantees that any output intermediate results contribute to final answers. For example, previous Algorithm TwigStack is optimal for query class with only A-D edges, and TwigStackList is optimal for queries with only A-D relationships in branching edges.

In this chapter, we will first show the "matching cross," which will be introduced in the following, and then a holistic algorithm called TreeMatch, both of which are proposed by somebody.

"Matching cross" is the key reason to result in the suboptimality of holistic algorithms. Intuitively, matching cross describes a dilemma where holistic algorithms have to decide whether to output *useless* intermediate result or to miss useful results. The fact that TwigStack is optimal for queries with only A-D relationships can be perfectly explained that no matching cross can be found for any XML document with respect to queries with A-D edges. We classify matching cross to bound and unbounded matching cross (BMC and UMC). Theorems are developed to show that only part of UMC can force holistic algorithms to potentially output useless intermediate results.

Based on the theoretical analysis, the holistic Algorithm TreeMatch is proposed to achieve a large optimal query class for three categories of queries (i.e., $Q^{/,//,*}$, $Q^{/,//,*,<}$ and $Q^{/,//,*,<,\neg}$. The main technique is to use a concise encoding to present matching results and reduce useless intermediate results.

5.4.1 Extended Tree Pattern Query

An extended tree query Q describes a complex traversal of the XML document and retrieves relevant tree-structured portions of it. The nodes in Q include element tags, attributes, and character data. We use "*" to denote the wildcard, which can match any single tree element. There are four kinds of query edges, which are the four

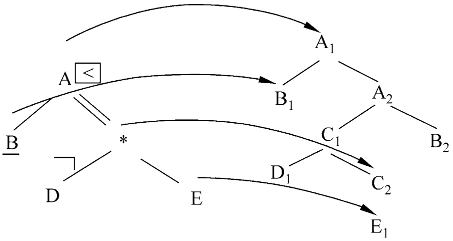


Fig. 5.13 Mapping relationship between an extended tree pattern and a document tree

combinations between (positive, negative) and (parent–child, ancestor–descendant). For example, in Fig. 5.11b, (A, B) is a positive parent–child edge, and (A, C) is a negative parent–child edge. We use a symbol "." to denote a negative edge. There are two kinds of query node: ordered and unordered node. We use "<" in a box to denote the ordered node; otherwise, it is an unordered node. For example, the node A in Fig. 5.11c, d is an ordered node. In each extended tree query pattern, there are one or multiple nodes which are assigned as the selected output node, denoted with an underline. For example, in Fig. 5.11a, C is the selected output node.

Given an extended tree query Q with n selected output nodes and an XML database D, a match of Q in D is identified by a mapping from the nodes in Q to the elements in D, such that (1) the query node types (i.e., tag name) are satisfied by the corresponding database elements and wildcards "*" can match any single database element; (2) the positive edge relationships (including positive parent–child and positives ancestor–descendant edges) between query nodes are satisfied by the corresponding database elements; (3) the negative edge relationships (including negative parent–child and negative ancestor–descendant edges) are satisfied, that is, no corresponding database element pairs exist; and (4) the order relationship of children of each ordered node is satisfied by the corresponding database elements. The answers of a query can be represented as a set of database elements, where each element identifies a distinct match of the selected output nodes on D. For example, Fig. 5.13 shows an example mapping relationship between an extended XML tree pattern and a document tree.

5.4.2 Matching Cross

A set of theories are given about "matching cross" which demonstrate the intrinsic reason for the suboptimality of existing holistic algorithms [LLB+11]. This theory also shows the possibility for holistic algorithms to identify a larger optimal query class than the existing algorithms do.

The existing holistic algorithms consist of two phases: (1) in the first phase, a list of element paths is output as intermediate path solutions, and each solution matches the individual root-to-leaf path expression; and (2) in the second phase, the element paths are merged to produce the final answers for the whole twig query. However, for

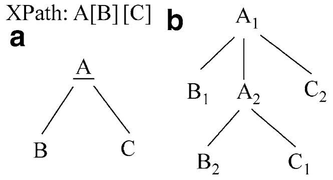


Fig. 5.14 Example XML tree query and document. "–" denotes the output node in query. The answers are A_1 and A_2. (**a**) Query, (**b**) data

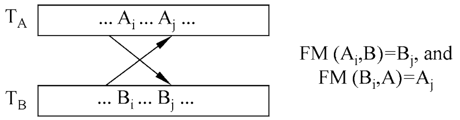


Fig. 5.15 Illustration to matching cross

queries with parent–child (P-C) relationships, the state-of-the-art algorithms cannot guarantee that each intermediate solutions output in the first phase is useful to merge in the second phase. In other words, many useless intermediate solutions may be produced in the first phase, which is called the suboptimality of algorithm, as shown in the following example.

Example 5.5 Consider the document and query in Fig. 5.18. First, A_1, B_1, and C_1 are scanned. We cannot determine whether or not A_1 is a query answer. Although A_1 has the child B_1, at this point, we do not know whether A_1 has a child C. Now holistic algorithms meet a dilemma, that is, whether to output possibly "useless" intermediate path (A_1, B_1) or to miss the potential correct answer A_1. In order to guarantee the completeness of query answers, previous methods buffer A_1 in the main memory and output the path (A_1, B_1), which may become "useless" intermediate path if there were no C_2 in join data.

The observation in Example 5.5 is formalized into an important concept: matching cross.

Definition 5.3 (First match) Given an XML database D and a query Q, assume that A, B are two query nodes in Q. Let A_i be an element in the label list T_A. We say that B_j in T_B is the first match of A_i, denoted as FM(A_i, B) = B_j, if and only if (A_i, B_j) appears in a match binding to query Q and there is no other element B_k, $k < j$ such that (A_i, B_k) is also in a match binding.

Note that in the above definition, all elements' labels in T_A and T_B are sorted by document order, and thus B_k is a preceding element of B_j as $k < j$. For example, in Fig. 5.14b, FM(B_1, C) = C_2 and FM(B_2, C) = C_1. In addition, note that FM(A_i, B) = B_j does not guarantee FM(B_j, A) = A_i.

Definition 5.4 (**Matching cross**) Given an XML database D and a query Q, assume that A, B are two query nodes in Q. Let A_i, A_j ($i < j$) be two elements in label list T_A and $B_{i'}$, $B_{j'}$ ($i' < j'$) be two elements in T_B. We say that the 4-tuple $<A_i, A_j, B_{i'}, B_{j'}>$ is a matching cross on D with respect to if and only if FM(A_i, B) = $B_{j'}$ and FM($B_{i'}$, A) = A_j. (See Fig. 5.15.)

It is easy to prove that if <A_i, A_j, $B_{i'}$, $B_{j'}$> is matching cross, then <$B_{i'}$, $B_{j'}$, A_i, A_j> is also a matching cross.

In Fig. 5.14b, <B_1, B_2, C_1, C_2> is a matching cross since the first match of B_1 is C_2 and that of C_1 is B_2. Note that B_1 and C_1 are not in the same match binding. The existence of matching cross forces holistic algorithms to output uncertain intermediate path solutions and may cause their suboptimality. When we read B_1 and C_1, we cannot know whether they can participate in a final match.

The following lemma identifies a query class, with respect to which we cannot find any document with matching cross.

Lemma 5.6 *Suppose Q is a tree pattern query with only ancestor–descendant (A-D) relationships in all edges, given any document D, there is no matching cross on D with respect to Q.*

Proof We prove it by contradiction. Assume that a matching cross <A_i, A_j, $B_{i'}$, $B_{j'}$> occurs when evaluating Q on document D. Let "$\prec$" denote preceding relationship in document order. Then $A_i \prec A_j$ and $B_{i'} \prec B_{j'}$. These are the following two cases:

1. A and B appear in the same path in the query Q. Without loss of generality, assume A is ancestor of B in Q. There are two sub-cases: case(1.1) A_i is an ancestor of A_j in document D. Since (A_j, $B_{i'}$) is a match binding, A_j is an ancestor of $B_{i'}$. Since all edges in query are A-D relationships, (A_i, $B_{i'}$) is also a match binding, which contradicts that A_j is the first match of $B_{i'}$. Case(1.2) A_i and A_j are in different data paths. Since (A_j, $B_{i'}$) is a match binding, A_j is an ancestor of $B_{i'}$. So $A_i \prec B_{i'}$ and $B_{i'}$ is not in the same data path with A_i. Since $B_{i'} \prec B_{j'}$, $B_{j'}$ is also not in the same data path with A_i, which contradicts that $B_{j'}$ is the first match of A_i.
2. Assume that A and B are in the different root-to-leaf paths in Q. Assume that node C is the lowest common ancestor of A and B in Q. Then there are two matching bindings (A_i, $B_{j'}$, C_1) and (A_j, $B_{i'}$, C_2). Consider two sub-cases (2.1): $C_1 = C_2$, then it is easy to see that <A_i, $B_{i'}$> is also a matching binding, which contradicts $B_{j'}$ is the first match of A_i. Case (2.2): $C_1 \neq C_2$. Without the loss of generality, assume that $C_1 \prec C_2$, then C_1 is an ancestor of C_2; otherwise, there is no overlap in C_1 and C_2 subtrees, which contracts that $B_{i'} \prec B_{j'}$. Then (A_i, $B_{i'}$, C_2) is also a matching binding, which contracts that $B_{j'}$ is the first match of A_i.

According to Lemma 5.6, no matching cross can occur during evaluating queries with only A-D relationships. This lemma perfectly explains why the previous Algorithm TwigStack can guarantee the optimality for queries with only A-D relationships, as there is no matching cross in such cases. But note that an existing Algorithm TwigStackList can identify a larger query class to guarantee the optimality than that of TwigStack. This fact implies that a certain kind of matching cross does not necessarily cause the suboptimality of holistic algorithms, as illustrated as follows.

Definition 5.5 (**Bounded matching cross**) Given a query Q and an XML database D, assume that <A_i, A_j, $B_{i'}$, $B_{j'}$> is a matching cross for D with respect to Q. If the

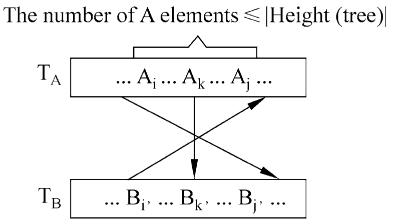

The number of A elements ≤ |Height (tree)|
T_A
... A_i ... A_k ... A_j ...
T_B
... $B_{i'}$ ... $B_{k'}$ ... $B_{j'}$ ...

Fig. 5.16 Illustration to bounded matching cross (The number of elements in T_A between A_i and A_j whose first match is after $B_{i'}$ is no more than the height of the tree)

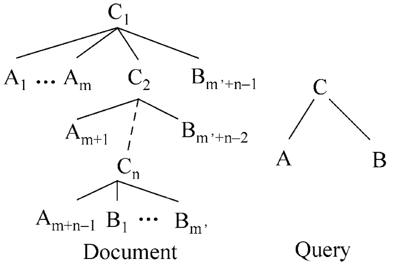


Fig. 5.17 Example of bounded matching cross (BMC) and unbounded matching cross (UMC). <C_1, C_n, B_1, $B_{m'+n-1}$> is a BMC, while <A_1, A_{m+n-1}, B_1, $B_{m'+n-1}$> is a UMC

number of distinct elements A_k, where $i \le k \le j$ and FM(A_k, B) $= B_{k'}(I' < k')$, is no more than the height of D, then we say that A has a bounded matching cross (BMC) with B; otherwise, it is unbounded matching cross (UMC) (See Fig. 5.16).

Since the number of distinct elements that have the first match after $B_{i'}$ is no more than |Height(D)|, we can buffer all such A_k's in the main memory and read A_j to find the matching element for $B_{i'}$.

Example 5.6 Consider the query and document in Fig. 5.17. <C_1, C_n, B_1, $B_{m'+n-1}$ >is a BMC, because the number of distinct elements $C_k(1 \leqq k \leqq n)$ that has the first match behind B_1 is no more than n, which is bounded by the height of the document, that is, C has a bounded matching cross with B. In contrast, <A_1, A_{m+n+1}, B_1, $B_{m'+n-1}$> is a UMC. This is because m or m' is not bounded by the height of the document and thus the number of distinct elements $A_k(1 \leqq \mathrm{k} \leqq m+n-1)$ (or similarly B_k, $(1 \leqq k \leqq m' + n + 1)$ that has the first match behind B_1 (or A_1) is possibly much greater than the height of documents.

As shown in Definition 5.5 and Example 5.6, matching cross can be classified as two classes according whether it can be solved by buffering limited elements. In particular, BMC can be solved by buffering bounded number of elements in the main memory. But we cannot guarantee to optimally process UMC with limited size of main memory, since it needs to buffer unbounded number of elements (note that we say it is unbounded in terms of the height of the document tree).

The following lemma identifies a query class, with respect to which no UMC occurs on any given XML document. In other words, this query class is guaranteed to be processed optimally by holistic algorithms. This lemma perfectly explains the optimal query class in TwigStackList, as the optimal query class in TwigStackList is the same as that in the following lemma.

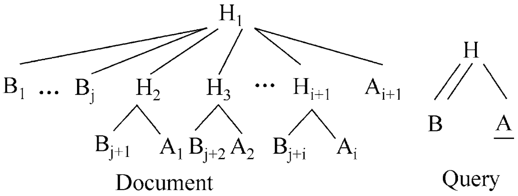


Fig. 5.18 An example to illustrate unbounded matching cross

Lemma 5.7 *Suppose Q is a tree pattern query with only ancestor–descendant (A-D) relationships to connect branching nodes and their children nodes, given any document D, there is no unbounded matching cross (UMC) on D with respect to Q.*

Proof Assume that $<A_i, A_j, B_{i'}, B_{j'}>$ is a matching cross on D. There are two cases: (1) A and B are two query nodes in Q, and A is an ancestor of B. We say that A_i and A_j are in the same data path, as A_i is an ancestor of $B_{j'}$ and A_j is an ancestor of $B_{i'}$. Thus, this is a BMC, as the number of elements between A_i and A_j is no more than |Height(D)|. (2) Assume A and B are in the different root-leaf path in Q and $<A_i, A_j, B_{i'}, B_{j'}>$ is a matching cross on D. We show that this case is impossible. Assume that the lowest common ancestor of A and B in Q is C. Since C connects its child nodes with only A-D relationships, $<A_i, B_{i'}>$ should also be a matching binding, as $<A_i, B_{j'}>$ is a matching binding and $B_{i'}$ precedes $B_{j'}$, which contracts the assumption that $<A_i, A_j, B_{i'}, B_{j'}>$ is a matching cross.

A natural question is whether all UMC definitely causes the suboptimality of holistic-processing algorithms. The answer is "no." Note that the query answers of an XML tree pattern usually include only part of query nodes; this observation can be used to identify a larger optimal query class. In order to understand this, let us first consider an XML tree in Fig. 5.18 and the XPath query "H[.//B]/A" (A is the selected output query node). $<B_1, B_{j+1}, A_1, A_{i+1}>$ is a UMC since i and j may be greater than the height of XML tree. But we observe that this UMC still can be efficiently processed by registering the information that H_1 has appropriate children B (e.g., B_1) and then scanning B_{j+1} and H_2. Then an exact match (H_2, B_{j+1}, A_1) without outputting any possibly useless intermediate paths can be got. This example shows that the existence of UMC does not necessarily result in the suboptimality of algorithm. Some UMC also can be solved by buffering limited information in the main memory. The following definition and lemma show that if there is a mediator node in UMC, then such UMC can be still processed optimally.

Definition 5.6 (**Mediator in UMC**) Given a query Q and an XML database D, assume that $<A_i, A_j, B_{i'}, B_{j'}>$ is an unbounded matching cross (UMC) for D with respect to Q and A is an output node in Q but B is a non-output node. We call the node $H \in Q$ as a mediator node (H may be an output node or not) if the first matches of all elements between $B_{i'}$ and $B_{j'}$ against node H are between H_m and H_n and the first matches of all elements between H_m and H_n are between A_i and A_j (see Fig. 5.19) and the number of elements between H_m and H_n that are the first matches of $B_k(i' \leq k \leq j')$ is no more than the height of D.

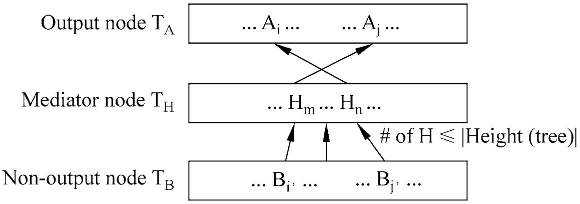
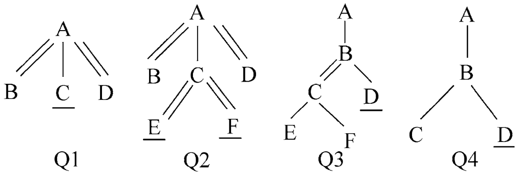


Fig. 5.19 Illustration to mediator node in UMC. $<B_i', B_j', A_i, A_j>$ is a UMC. H is mediator node, as the first matches of all elements between B_i' and B_j' are between H_m and H_n and the first matches of that between H_m and H_n are A_i and A_j

Fig. 5.20 Example queries to illustrate mediator subclass. Q_1, Q_2, and Q_3 are in mediator subclass, but Q_4 not, because of (B, C) edge



For example, consider Fig. 5.18 and the query "H[.//B]/A" again. $<B_1, B_{j+1}, A_1, A_{i+1}>$ is a UMC, and H is a mediator in this UMC, as the first matches of all elements between B_1 and B_{j+1} against node H are H_1 and the first matches of H_1 and H_2 are A_{i+1} and A_1 and $2 \leq$ Height(D).

Because of the existence of mediator node in UMC, we still can guarantee the optimality of algorithm by buffering limited elements of mediator nodes in the main memory. In the example of Fig. 5.18, only H_1 and H_2 need to be buffered to the main memory and be recorded that H_1 and H_2 have the matching element with node B. Note that we do not need to buffer $B_1, \ldots, B_j$ in the main memory as they are not output nodes.

The next definition and lemma identify a subclass of tree pattern queries, with respect to which, given any XML document, we can always find a mediator node in a UMC.

Definition 5.7 (**Mediator subclass**) A query Q belongs to mediator subclass if and only if given any output node N in Q and a branching node B in the path from N to root, the edge between B and its child C is ancestor–descendant relationship if C is not in the path from N to root.

For example, Fig. 5.20 shows four example queries. Q_1, Q_2, and Q_3 belong to mediator subclass, but Q_4 does not because of (B, C) edge.

Lemma 5.8 *Given a query Q that is in mediator subclass and a document D, for each UMC in D against Q, there exists a mediator node $H \in Q$ in this UMC.*

Proof Assume that $<A_i, A_j, B_{i'}, B_{j'}>$ is a UMC in D against Q. According to the proof of Lemma 5.7, A and B are in the different paths in Q (otherwise, it is a BMC). Assume that C is the lowest common ancestor of A and B, there are three cases. In Case (1), both A and B are output nodes; then according to Lemma 3.5, there is at least a P-C relationship as the child edge of C. Then this query does not satisfy the restriction of Q in the above lemma. In Case (2), either A or B is an output node. Without loss of generality, assume A is the output node and B is a non-output node. Next, we prove that C is a mediator node. There are two matching bindings (A_i, $B_{j'}$, $C_{i''}$) and (A_j, $B_{i'}$, $C_{j''}$). Then ($C_{i''}$, $C_{j''}$) has A-D relationship. Since B is an output node, assuming that the child of C, say D, which is also an ancestor of B, then (C, D) is an A-D relationship according to the prerequisite of lemma. Therefore, (A_j, $B_{j'}$, $C_{i''}$) is also a matching binding. Then the first match of all elements between $B_{i'}$ and $B_{j'}$ is between $C_{i''}$ and $C_{j''}$, and the first match of all elements between $C_{i''}$ and $C_{j''}$ is between A_i and A_j. Thus, C is a mediate node. In Case (3), neither A nor B is an output node; it is a trivial case of the lemma.

As a final remark of this chapter, it is important to note that the properties shown in the above theorems are independent of (1) any concrete labeling schemes and (2) any special data index structures, such as XB tree [BKS02], XR tree [JWLY03], and R tree [CVZ+02]. This is because (1) the proof of the above theorems does not rely on any specific labeling scheme, and (2) while special index structure can skip elements to accelerate processing in holistic XML query processing, these index structures cannot achieve the larger optimal query class, as the main bottleneck of optimality is the size of main memory.

5.4.3 Holistic Algorithms

In this chapter, we introduce the algorithm to evaluate an extended XML tree query. We first show the Algorithm TreeMatch to answer queries $Q^{/,//,*}$ and then naturally extend it to support queries with order restriction and negative edges (i.e., $Q^{/,//,*,<}$ and $Q^{/,//,*,<,\neg}$). The challenge in the algorithms is to achieve a large optimal query class according to aforementioned theorems.

The algorithms use the extended Dewey labeling scheme, proposed by Lu et al. [LLCC05], to assign each node in XML documents a sequence of integers to capture the structure information of documents.

The algorithms for XML tree pattern matching proposed have two basic yet important properties as follows.

5.4.3.1 Single-Direction Scan

A structure, named label list, is adopted, associated with each query node. The label list is a posting list (or inverted list) containing the extended Dewey labels of XML elements which have the same name, and all elements are ordered according to the document order. TA is used to denote the label list for query node A. There is a cursor for each list. It moves in the single direction to scan all elements once in increasing order. Each label in a list can be read only once.

5.4.3.2 Bounded Main Memory

The main memory requirement of this algorithm is linear to the number of nodes in the longest path of XML database, which is usually small. Therefore, this solution will be scalable to a very large document with a small main memory requirement. At the same time, it is also important to note that, if a large amount of main memory is available, the algorithm also can efficiently utilize such large memory to accelerate query processing.

5.4.3.3 Treematch for $Q^{/,//,*}$

Data Structures and Notations

There is an input list T_q associated with each query node q, in which all the elements have the same tag name q. Thus, we use e_q to refer to these elements. cur(T_q) denotes the current element pointed by the cursor of T_q. The cursor can be advanced to the next element in T_q with the procedure advance(T_q).

There is a set S_q associated with each branching query node q (not each query node). Each element e_q in sets consists of a 3-tuple (label, bitVector, outputList). label is the extended Dewey label of e_q. bitVector is used to determine whether the current element has the proper children or descendant elements in the document. Specifically, the length of bitVector(e_q) equals to the number of child nodes of q. Given a node $q_c \in$ children(q), bitVector(e_q, q_c) = "1" if and only if there is an element e_{qc} in the document such that the e_q and e_{qc} satisfy the query relationship between q and q_c. Finally, outputList contains elements that potentially contribute to final query answers. Next, we introduce two properties of elements in outputList and bitVector in details.

At every point during the computing, for each element e_q in set S_q, (1) if all bits in bitVector(e_q) are "1," then e_q is guaranteed to match the subtree rooted with q. Therefore, if q is the root, then e_q is guaranteed to match the whole query, and (2) element $e \in$ outputList(e_q) is the query answer if and only if e_q matches the whole tree query. Therefore, using both properties, we say that whether an element $e \in$ outputList(e_q) is a query answer can be accurately reflected by the corresponding bitVector(e_q), illustrated as follows.

Example 5.7 Figure 5.21 illustrates the set encoding S_A to query node A for an example document. There are two tuples in set S_A. Since A_1("0") has only one child B_1 and no child element to match C, bitVector(A_1) = "10." In contrast, bitVector (A_2) = "11" since A_2("0.1") has two children B_2 and C_1, which satisfy the P-C relationships in the query. Since all bits in bitVector(A_2) are "1," B_2("0.1.0") is guaranteed to be a query answer.

In this algorithm, we will frequently use the following two notations: (1) NAB(q) denotes the nearest ancestor branching nodes of q in the query pattern Q. Formally,

Fig. 5.21 Illustration to set encoding

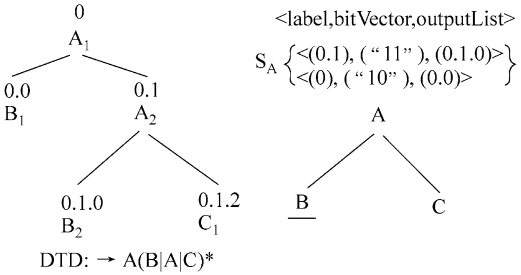


Table 5.3 Set encoding

Current elements	Set encoding of S_A
B_1, C_1	<0, "10", Ø>
B_2, C_1	<0, "10", Ø>,<0.1, "11", Ø>
B_2, C_2	<1, "12", 1>,<0.1, "11", 0.1>

$q' = \mathrm{NAB}(q)$ if and only if q' is a branching node and q' is an ancestor of q and there is no other branching node q'' s.t. q'' is in the path from q' to q. If there is no such ancestor of q, then NAB(q) denotes the top branching node in query. (2) NDB(q) denotes the nearest descendants branching nodes of q. Formally, $q' \in \mathrm{NDB}(q)$ if and only if q' is a branching or leaf node and q' is a descendant of q and there is no other branching or leaf node q'' s.t. q$''$ is in the path from q' to q. For example, see the query Q_3 in Fig. 5.20, $\mathrm{NAB}(E) = \{C\}$, $\mathrm{NAB}(D) = \{B\}$, and $\mathrm{NDB}(B) = \{C, D\}$.

Intuitive Example

Before we formally introduce the Algorithm TreeMatch, let us first see an example to intuitively understand this algorithm. Here the key point is set encoding of elements.

Example 5.8 Consider the data and query in Fig. 5.14. Note that A is the single output node. Firstly, we read B_1 and C_1. Since A_1 now has only one child B_1 and one descendant B_1 (not child), we insert A_1 to set S_A and bitVector(A_1) = "10" (see Table 5.3). Next, when we scan B_2, C_1, since A_2 has two children B_2 and C_1, we add A_2 to set and bitVector(A_2) = "11." Subsequently, when we scan C_2, we know that A_2 also has a child C_2, and thus we update bitVector(A_1) to be "11." Finally, we empty set S_A and output two elements A_1 and A_2 in the outputlists. Note that unlike previous algorithms such as TwigStack and TJFast, we use bitVector to accurately record matching results. If there were no C_2 elements, we would not output A_1, as bitVector(A_1) is "10," but TwigStack and TJFast would output two "useless" elements A_1 and B_1 in that case.

5.4.3.4 TreeMatch

We first introduce TreeMatch (see Algorithm 5.5) to evaluate a query with a single output node and then naturally extend it to evaluate a query with multiple output nodes.

Algorithm 5.5: Algorithm TreeMatch for class $Q^{/,//,*}$

```
1. locateMatchLabel(Q);
2. while (!end(root)) do
3.   $f_{act}$ = getNext(topBranchingNode);
4.   if ($f_{act}$ is an output node)
5.     addToOutputList(NAB($f_{act}$), cur($T_{f_{act}}$));
6.   advance($T_{f_{act}}$); // read the next element in $T_{f_{act}}$
7.   updateSet($f_{act}$); // update set-encoding
8.   locateMatchLabel(Q); // locate next element with matching path
9. emptyAllSets(root);
```

Now we go through Algorithm 5.5. Line 1 locates the first elements whose paths match the individual root-leaf path pattern. In each iteration, a leaf node fact is selected by getNext function (Line 3). The purpose of Lines 4 and 5 is to insert the potential matching elements to outputlist. Line 6 advances the list $T_{f^{\text{act}}}$, and Line 7 updates the set encoding. Line 8 locates the next matching element to the individual path. Finally, when all data have been processed, we need to empty all sets in Procedure EmptyAllSets (Line 9) to guarantee the completeness of output solutions.

In ProcedureaddToOutputList(q, e_{q_i}), we add the potential query answer e_{q_i} to the set of S_{e_q}, where q is the nearest ancestor branching node of q_i (i.e., NAB$(q_i) = q$). Procedure updateSet accomplishes three tasks. First, clean the sets according to the current scanned elements. Second, add e into set and calculate the proper bitVector. Finally, we need to recursively update the ancestor set of e. Because of the insertion of e, the bitVector values of ancestors of q need update.

Algorithm 5.6: Procedures and Functions in TreeMatch

Procedure locateMatchLabel(Q)

```
for each leaf q $\in$ Q, locate the extended Dewey label $e_q$ in list
$T_q$ such that $e_q$ matches the individual root-leaf path
```

Procedure addToOutputList(q, e_{qi})

```
1. for each $e_q \in S_q$ do
2.  if (satisfyTreePattern($e_{qi}$, $e_q$))
3.     outputList($e_q$).add($e_{qi}$);
```

Function satisfyTreePattern(e_{qi}, e_q)

```
4. if (bitVector($e_q$, $q_i$) = '1') return true;
5. else return false;
```

Procedure updateSet(q, e)

```
cleanSet(q, e);
add e to set $S_q$; //set the proper bitVector(e)
if (!isRoot(q) $\wedge$ (bitVector(e)="1 . . . 1"))
  updateAncestorSet(q);
```

Procedure cleanSet(q, e)

```
for each element $e_q \in S_q$ do
  if (satisfyTreePattern($e_q$, e))
    if (q is an output node)
      addToOutputList(NAB(q), e);
    if (isTopBranching(q))
      if (there is only one element in $S_q$)
      output all elements in outputList($e_q$);
    else merge all elements in outputList($e_q$) to
outputList($e_a$), where $e_a$ = NAB($e_q$);
delete $e_q$ from set $S_q$;
```

Procedure updateAncestorSet(q)

```
/*assume that q'=NAB(q)*/
for each $e \in S_{q'}$ do
if (bitVector(e, q)=0)
    bitVector(e, q)=1;
    if (!isRoot(q) $\wedge$ (bitVector(e)="1 . . . 1"))
      updateAncestorSet(q');
```

Procedure emptyAllSets(q)

```
if (q is not a leaf node)
  for each child c of q do EmptyAllSets(c);
for each element $e \in S_q$ do cleanSet(q, e);
```

Algorithm 5.7: getNext(n)

```
if (isLeaf(n)) then
  return n
else
  for each $n_i \in$NDB(n) do
    $f_i$=getNext($n_i$)
    if (isBranching($n_i$) $\wedge$!empty($S_{ni}$))
      return $f_i$
    else $e_i$ =max{p|p$\in$MB($n_i$,n)}
end for
max=maxarg$_i${$f_i$}
for each $n_i \in$ NDB(n) do
  if ($\forall$ e $\in$ MB($n_i$,n):e $\notin$ ancestors($e_{max}$)
    return $f_i$;
  end if
end for
```

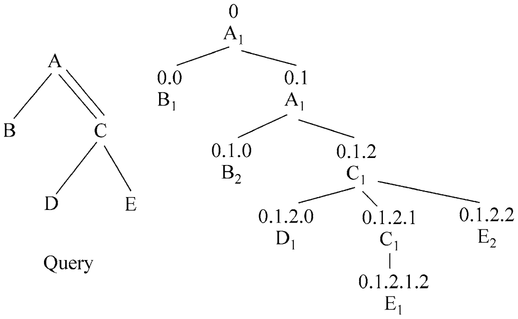


Fig. 5.22 Illustration to algorithm TreeMatch for class $Q^{/,//,*}$

```
44. min=minargi{$f_i$,$f_i$ is not an output node}
45. for each e∈MB($n_{min}$,n)
46.     if (e∈ancestors($e_{max}$)) updateSet($S_n$, e)
47.   end for
48.   return $f_{min}$
49. end if
50.
```

Function MB(n, b)

```
1. if (isBranching(n)) then
2.   Let e be the maximal element in set $S_n$
3. else
4.   Let e=cur($T_n$)
5. end if
6. Return a set of element a that is an ancestor of e such that a can match node b in
   the path solution of e to path pattern $p_n$
```

Algorithm getNext (see Algorithm 5.7) is the core function called in TreeMatch, in which we accomplish two tasks. For the first task to identify the next processed node, Algorithm getNext(n) returns a query leaf node f according to the following recursive criteria: (1) if n is a leaf node, $f = n$ (Line 2); else (2) n is a branching node, then suppose element e_i matches node n in the corresponding path solution (if more than one element that matches n, e_i is the deepest one by level) (Lines 7,8), we return $f_{\min}$ such that the current element $e_{\min}$ in $T_{f_{\min}}$ has the minimal label in all e_i by lexicographical order (Lines 13, 20).

For the second task of getNext, before an element e_b is inserted to the set S_b, we ensure that e_b is an ancestor (or parent) of each other element e_{b_i} to match node b in the corresponding path solutions (Line 13). If there are more than one element to match the branching node b, e_{b^i} is defined as their *deepest* (i.e., maximal) element (Line 8).

Example 5.9 We use the query and document in Fig. 5.22 to illustrate TreeMatch algorithm. Table 5.4 demonstrates the current access elements, the set encoding, and the corresponding output elements. There are two branching nodes in the query.

Table 5.4 Set encoding

Current elements	Set encoding S_A	Set encoding S_C
B_1, D_1, E_1	<0, “10”, Ø>	<0.1.2, “10”, Ø> <0.1.2.1, “01”, Ø>
B_1, D_1, E_2	<0, “11”, Ø>	<0.1.2, “11”, Ø> <0.1.2.1, “01”, Ø>
B_2, D_1, E_2	<0, “11”,0.0> <0.1, “11”, 0.1.0>	<0.1, “11”, Ø> <0.1.2.1, “01”, Ø>

When TreeMatch read E_2, it recursively updates the bitVector of C_1 and A_2. Finally, the algorithm outputs $\{B_1, B_2\}$ as the final results.

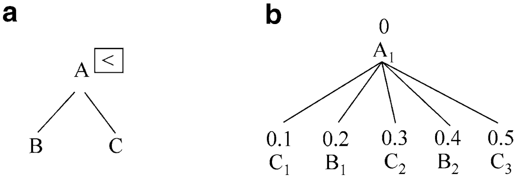
When there are multiple output nodes in a query, the Algorithm TreeMatch produces the corresponding outputList for each of them. TreeMatch first outputs the individual solution for each output node and then merges all these solutions to get the final result bindings. It is important to note the differences between TreeMatch and TJFast. Even if the available amount of main memory is large, TJFast outputs many path solutions that do not contribute to any final answers. However, TreeMatch can efficiently use such available main memory (by buffering potential useful elements in outputlist) to guarantee that each output element contributes to final answers. Therefore, TreeMatch not only identifies a larger optimal query class than TJFast but also has the ability to fully utilize the available amount of the main memory.

5.4.3.5 Extension for Order-Based Queries $Q^{/,//,/*,<}$

In this subsection, TreeMatch algorithm is extended to support a query including ordered nodes. In order to record the position information of elements, minChild and maxChild attributes are added for each tuple in sets. That is, each tuple in sets now is a 5-tuple: <label, bitVector, outputList, minChild, maxChild>. The length of minChild(e_q) and maxChild(e_q) is equal to the number of children of q. Assume that $q_1, \ldots, q_n$ is the children node of q (in order) in the query. Given an element $e_{qi}^{\min}$ in minChild(e_q) and $e_{qi}^{\max}$ in maxChild(e_q), $e_{qi}^{\min}$ is the minimal element that is greater than the element $e_{qi-1}^{\min}$ (if any) and $e_{qi}^{\max}$ is the maximal element that is smaller than $e_{qi+1}^{\max}$ (if any). Therefore, $e_{q1}^{\min}$ is the left most children of e_q, and $e_{qn}^{\max}$ is the right most children.

Example 5.10 See the query and document in Fig. 5.23. Table 5.5 shows the values of minChild and maxChild attributes in set. When we scan B_1 and C_1, since C_1 is before B_1, we do not insert C_1 as a minChild, as it is not greater than B_1. Only when we scan C_2, we insert C_2 to minChild. When B_3 and C_3 are scanned, they become the respective maxChild for node B and C.

Algorithm 5.8 describes the extended TreeMatch algorithm for answering ordered tree queries. The purpose of the extension is to maintain and check the



XPath: //A/B [following-sibling::C]

Fig. 5.23 An example ordered XML tree pattern query. (**a**) Ordered based query, (**b**) document

Table 5.5 Set encoding

	Set encoding S_A	
Current elements	minChild	maxChild
B_1, C_1	(0.2, Ø)	(0.2, Ø)
B_1, C_2	(0.2,0.3)	(0.2,0.3)
B_2, C_2	(0.2,0.3)	(0.4,0.3)
B_2, C_3	(0.2,0.3)	(0.4,0.5)

order relationship among the matching elements of query sibling nodes. In Line 2 of Procedure updateSet, we need to set the proper minChild and maxChild according to the current elements. In Function satisfyTreePattern, we also need to check the order restriction according to minChild and maxChild.

Algorithm 5.8: Algorithm TreeMatch for class $Q^{/,//,*,<}$
Procedure updateSet(q, e)

```
...
2. add e to set Sq; //set the proper bitVector, minChild and maxChild
...
```

Function satisfyTreePattern(q_i,e_q)

```
1. assume that child nodes of q in Q are $q_1, \ldots, q_n$ (in order)
2. if ($e_{qi}$< minChild($e_q$, $q_{i-1}$)) return false;
3. else if ($e_{qi}$> maxChild($e_q$, $q_{i+1}$)) return false;
4. else if (bitVectoreq($e_q$ , $q_i$)=‘1’) return true;
5. else return false;
```

5.4.3.6 Extension for Queries with Negative Edges $Q^{/,//,*,<,\neg}$

In this subsection, TreeMatch is extended to support negative edges (see Algorithm 5.9). negBitVector is added to record the matching information about negative child edge. Given a node $q_c \in$ negativeChildren(q), negBitVector(e_q, q_c) = “0” if and only if there is no element e_{qc} in the document such that the e_q and e_{qc} satisfy the query relationship in between q and q_c. In this way, in order to know whether all negative children of q are satisfied, we only check whether all children's negBitVectors are “0.” In Line 2 of Procedure updateSet, we need to set the proper negBitVector according to the current elements. In Function satisfyTreePattern, e_q is a valid element only if the negBitVector is “0.”

Algorithm 5.9: Algorithm TreeMatch for class $Q^{/,//,/*,<,\neg}$

Procedure updateSet(q, e)

```
. . .
2. add e to set $S_q$; //set the proper bitVector, negBitVector,minChild and maxChild
. . .
```

Function satisfyTreePattern(q_i, e_q)

```
1. if ($e_{qi}$< minChild($e_q$, $q_{i-1}$)) return false;
2. else if ($e_{qi}$> maxChild($e_q$, $q_{i+1}$)) return false;
3. else if ((bitVector($e_q$, $q_i$)='1') and (negBitVector($e_q$, $q_i$)='0'))
4.   return true;
5. else return false;
```

5.4.3.7 Analysis of Algorithm

We discuss the correctness of TreeMatch, then analyze its complexity.

Lemma 5.9 *In Algorithm TreeMatch, suppose any element e_q is popped from set S_q, where q is the top branching node in Procedure* CleanSet(q), *then e_q matches the whole query if and only if all bits in* bitVector(e_q) *are* "1" *and all children of e_q satisfy order condition* (*if any*) *and all bits in* negBitVector(e_q) *are* "0" (*if any*).

Lemma 5.10 *In Algorithm TreeMatch, suppose any element e_q is popped from set S_q, where q is the top branching node in Procedure* CleanSet(q), *then e_q matches the whole query if and only if all elements in* outputList(e_q) *belong to final query answers.*

Using Lemma 5.9 and 5.10, we can see that whether or not an element is a query answers is exactly reflected by the values of the corresponding bitVector, negBitVector and minChild, maxChild. Further, by Lines 5–7 in Procedure CleanSet, all correct solutions are output. In addition, each matching element is guaranteed to be inserted to the related sets in Procedure addToOutputList. Thus, the output solutions are also complete. Therefore, we have the following result.

Theorem 5.5 *Given an extended tree pattern query Q and an XML database D, Algorithm TreeMatch correctly returns all the answers for $\underline{Q}$ on D.*

While the correctness holds for any given query, the I/O optimality holds only for a subset of extended query class. In these cases, TreeMatch guarantees that each output element in Procedure CleanSet belongs to final query solutions. Next, we show the corresponding optimality query subclass for three categories of queries, that is, $Q^{/,//,*}$, $Q^{/,//,*,<}$ and $Q^{/,//,*,<,\neg}$.

Theorem 5.6 *Consider an XML database D and an extended tree pattern query $Q^{/,//,*}$ in mediator subclass* (*defined in Definition 5.7*), *the worst-case I/O complexity of TreeMatch is linear to the sum of the sizes of input and results. The worst-case space complexity is linear to the maximal depth in D times the number of nodes in Q.*

Proof [*Sketch*] Given any output node q in Q, let $b = \mathrm{NBA}(q)$, according to Definition 5.7, all edges except (b, q) between b and its children are ancestor–descendant relationships. b is a mediator node for any UMC involving in q. In Procedure addToOutputList, when each element e_q is inserted to outputList, it is guaranteed to satisfy the subtree rooted with q (Line 2). In Procedure UpdateAncestorSet, the elements in outputList are moved to its ancestor set only if the current subtree is satisfied. We recursively guarantee that each e_q in outputList satisfies the whole tree pattern. Therefore, each element e_q is inserted to outputList of e_b only if e_b satisfies the whole tree pattern. We can safely write each element in outputList to disk in Procedure CleanSet, and thus the worst-case I/O complexity of TreeMatch is linear to the sum of the sizes of input and results.

For queries with ordered node (i.e., $Q^{/,//,*,<}$), a larger optimal class can be identified. If node q is an order node in Q, the parent–child relationship between q and its first child does not affect the optimality of TreeMatch. Intuitively, this is because the order restriction stops some unbounded matching cross from happening.

Definition 5.8 **(Optimal Subclass for $Q^{/,//,/*,<}$)** We say that a query Q belongs to the optimal subclass for $Q^{/,//,/*,<}$ if and only if the parent–child relationship of Q occurs only in the following edges E: (1) given any output node q in Q, E is in the path from q to root, or (2) let $E = (a, b)$, then a should be an ordered node, and b is the first child of a.

Theorem 5.7 *Consider an XML database D and an extended tree pattern query $Q^{/,//,*,<}$ in the subclass defined in Definition 5.8, the worst-case I/O complexity of TreeMatch is linear to the sum of the sizes of input and results. The worst-case space complexity is linear to the maximal depth in D times the number of nodes in Q.*

Proof [*Sketch*] We need to show that the parent–child (P-C) relationship in the first branching edge of an ordered node does not affect the optimality of TreeMatch. Given an ordered branching node b, assume that the first child node of b is f and the output child node is q. If (e_f, e'_f, e_q, e'_q) is an unbounded matching cross, then we can prove that b is a mediator node in this matching cross. This is because, in function satisfyTreePattern, we use minChild and maxChild to test the order relationship, so we insert e_q to the outputList only if e_q is after e_f. Then we can safely advance the cursor in list T_f (note f is not an output node) and record the matching information to e_b in Procedure UpdateAncestorSet until we find e'_f. The number of elements e_b which are buffered in set S_b is no more than the depth of D. Therefore, b is the mediator node in this matching cross. Thus, each element e_q is inserted to outputList of b only if e_b satisfies the subtree pattern. Then we can safely output each element in outputList in Procedure CleanSet.

For queries with ordered nodes and negative edges (i.e., $Q^{/,//,*,<,\neg}$), the following results show that the existence of parent–child (or ancestor–descendant) edges in any negative edges does not affect the optimality of TreeMatch. Intuitively, this is because parent–child relationships in negative edges do not cause the matching cross.

Definition 5.9 (Optimal subclass for $Q^{/,//,*,<,\neg}$) We say that a query Q belongs to the optimal subclass for $Q^{/,//,*,<,\neg}$ if and only if the parent–child relationship of Q occurs only in the following edges E: (1) given any output node q in Q, E is in the path from q to root or (2) let $E=(a, b)$, then a should be an ordered node and b is the first child of a or (3) E is a negative edge.

Theorem 5.8 *Consider an XML database D and an extended tree pattern query $Q^{/,//,*,<,\neg}$ defined in Definition 5.9, the worst-case I/O complexity of TreeMatch is linear to the sum of the sizes of input and results. The worst-case space complexity is linear to the maximal depth in D times the number of nodes in Q.*

Proof [*Sketch*] We need to show the existence of negative P-C, and A-D relationship does not affect the optimality of TreeMatch. This is because the negative edges cannot generate the new matching cross. If (q, b) is a negative edge, then any e_q which is a descendant of e_b is guaranteed not to be final result. Therefore, we do not need to buffer e_q. In Procedure addToOutputList, all relationships about negative edges are satisfied, and in Function satisfyTreePattern, the negative edges are also checked. Therefore, each element e_q is inserted to outputList of e_b only if e_b satisfies the whole tree pattern. Then we can safely output each element in outputList in Procedure CleanSet.

Given any query pattern which does not belong to the optimal classes defined before, we cannot guarantee that each element that is inserted to outputList contributes to final results. In this case, when the main memory cannot hold all elements in outputList, some elements have to be output as intermediate results that may be useless to final query answers, and thus the optimality of TreeMatch cannot be guaranteed. But it is important to note that (1) the optimal query class of TreeMatch is larger than the existing algorithms such as TwigStack, TJFast, OrderedTJ [LLY+05], and TwigStackListNot [YLL06]; and (2) even in the cases when none of algorithm can guarantee the optimality, as shown in the following experiments, TreeMatch outputs much less intermediate results than other existing algorithms.

5.4.4 Experiments

In this subsection, we show an extensive experimental study of TreeMatch [LLB+11] on real-life and synthetic datasets. The results verify the effectiveness, in terms of accuracy and optimality, of the TreeMatch as holistic twig join algorithms for large XML datasets. These benefits become apparent in a comparison to previously four proposed algorithms TwigStack, TJFast, OrderedTJ, and TwigStackListNot. The reason that we choose these algorithms for comparison is that (1) similar to TreeMatch, both TJFast and TwigStack are two holistic twig pattern matching algorithms. But they cannot process queries with order restriction or negative edges, and (2) OrderedTJ is a holistic twig algorithm which can handle

Table 5.6 Characters of test datasets

	Synthetic	DBLP	TreeBank
Size(MB)	8.8	130	82
Elements(million)	1.0	3.3	2.4
Max/Avg depth	12/6.1	6/2.9	36/7.8

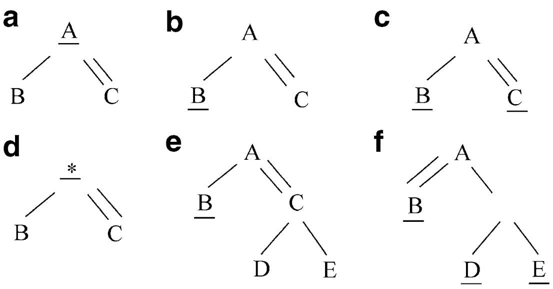


Fig. 5.24 Queries for random data. (**a**) Q_1 (optimal), (**b**) Q_2 (optimal), (**c**) Q_3 (optimal), (**d**) Q_4 (optimal), (**e**) Q_5 (optimal), (**f**) Q_6 (sub-optimal)

order-based XML tree pattern, but does not work for query with negative edge, and finally, (3) TwigStackListNot is proposed for queries with negative edge, but it does not work for ordered query. Only TreeMatch algorithm can efficiently process queries with order restriction, negative edge, and wildcards.

5.4.4.1 Experiment Settings and Dataset

All tested algorithms are implemented in JDK 1.4 using the file system as a simple storage engine, and all the experiments are conducted on a computer with Intel Pentium IV 1.7 GHz CPU and 768 M of RAM using both synthetic and real XML data. The synthetic dataset is generated randomly. There are totally 7 tags A, B, $\ldots$, F, G in the dataset, and tags are assigned uniformly from them. The real data are DBLP (highly regular) and Treebank (highly irregular), which are included to test the two extremes of the spectrum in terms of the structural complexity. The recursive structure in TreeBank is deep (average depth: 7.8, maximal depth: 36). We can easily find queries on this dataset to demonstrate the suboptimality for the tested algorithms. Table 5.6 summarizes the dataset characteristics.

5.4.4.2 Query Class $Q^{/,//,*}$

We show the experimental results for queries class $Q^{/,//,*}$. All queries tested for random datasets are shown in Fig. 5.24. All queries proposed for DBLP and TreeBank data are shown in Fig. 5.25. We also show the optimality of each tested query according to Theorem 5.6 in Fig. 5.24.

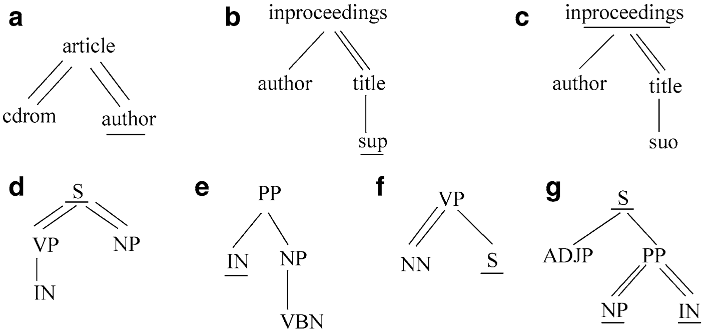


Fig. 5.25 Query for DBLP (Q_7–Q_9) and TreeBank (Q_{10}–Q_{13}). (**a**) Q_7 (optimal), (**b**) Q_8 (sub-optimal) (**c**) Q_9 (optimal), (**d**) Q_{10} (optimal), (**e**) Q_{11} (sub-optimal), (**f**) Q_{12} (optimal), (**g**) Q_{13} (optimal)

Table 5.7 Number of elements, output (O) versus useful (U)

	$D1$		$D2$		$D3$	
Query	O	U	O	U	O	U
Q_1	1,321	1,321	6,576	6,576	13,290	13,290
Q_2	3,558	3,558	17,757	17,757	35,649	35,649
Q_3	9,575	9,463	95,291	95,240	156,954	148,383
Q_4	6,635	6,635	33,055	33,055	65,691	65,691
Q_5	296	296	1,313	1,313	2,782	2,782
Q_6	7,506	7,506	94,132	94,132	127,478	127,420

$D1$:100 K nodes, $D2$:500 K nodes, $D3$:1 M nodes

Limited Main Memory Case

In the first experiment, the outputlist in TreeMatch is not allowed to buffer any elements in the main memory. That is, any element added to outputlist should be output to the hard disk. Then the requirement for main memory size is quite small. The purpose of this experiment is to simulate the application where the document is large but the available main memory is very small. Table 5.7 shows the number of output elements and the number of final query answers. The experiments are made by using three different sizes of random documents. In particular, $D1$ has 100 K nodes, $D2$ has 500 K nodes, and $D3$ has 1 M nodes. From Table 5.7, we observe that for most of queries, TreeMatch achieves the optimality in the sense that each of the output elements does belong to final results. The only exception is in Q_3 and Q_6, where according to Theorem 5.6, we cannot guarantee the optimality. Interestingly, Q_6 is optimal for $D1$ and $D2$ but only slightly suboptimal for $D3$. This can be explained that $D3$ is a larger document than $D1$ and $D2$ so that $D3$ demonstrates the suboptimality which is hidden in $D1$ and $D2$.

a

	D_1	D_2	D_3
Q1	5	6	6
Q2	9	10	11
Q3	528	27067	89779
Q4	6	7	8
Q5	7	8	10
Q6	520	26808	89627

b

	D_1	D_2	D_3
Q14	3926	20182	39796
Q15	9	9	10
Q16	4	5	6
Q17	3	5	6
Q18	6	8	9
Q19	9	11	11

Fig. 5.26 Number of required buffered elements for optimality (random data)

Large Main Memory Case

In the second experiment, the outputlist is allowed to buffer all elements in the main memory. The purpose of this experiment is to simulate the application where the available main memory is large so that a big portion of documents can be fit in the main memory. Figure 5.26a shows the maximal number of elements that is needed to be buffered to guarantee the optimality of TreeMatch. An obvious observation is that Q_3 and Q_6 need to buffer many elements, but all other queries only need to buffer very small number of elements. This can be perfectly explained that all queries except Q_3 and Q_6 belong to the optimal query class.

The benefit of this Algorithm TreeMatch is more apparent when we compare it with previous work TwigStack and TJFast. We compare the performance of three algorithms in Fig. 5.27a, b. Obviously, TreeMatch is superior to TwigStack and TJFast, reaching 20–95 % improvement in execution time for all queries.

Clearly, in most real application, the main memory size is not so large that the whole document can fit in memory, neither so limited that only the elements in a single path can load in memory. In order to check whether TreeMatch has the ability to fully exploit the available medium size of main memory, we show that performance of algorithm in terms of the number of output elements with varying the size of main memory in Fig. 5.28b, c. In this experiment, we choose Q_1 and Q_6 since Q_1 is an optimal query for TreeMatch but Q_6 is suboptimal. The experimental results show that the number of output elements in TreeMatch is always much less than that in TwigStack and TJFast for all sizes of main memory. In particular, for Q_1, with the increasing of the size of the available main memory, the number of output elements in TwigStack and TJFast decreases linearly. The reason is that TwigStack and TJFast buffer the intermediate results in the main memory and reduce the output of intermediate results. But the numbers of output elements in TreeMatch remain the same, which always equal the final result size. This experimental result confirms Theorem 5.6 about the optimality of TreeMatch for query Q_1. For query Q_6, all algorithms are not optimal. But TreeMatch still outputs much less elements than TwigStack and TJFast. Note that when the number of buffered elements reaches

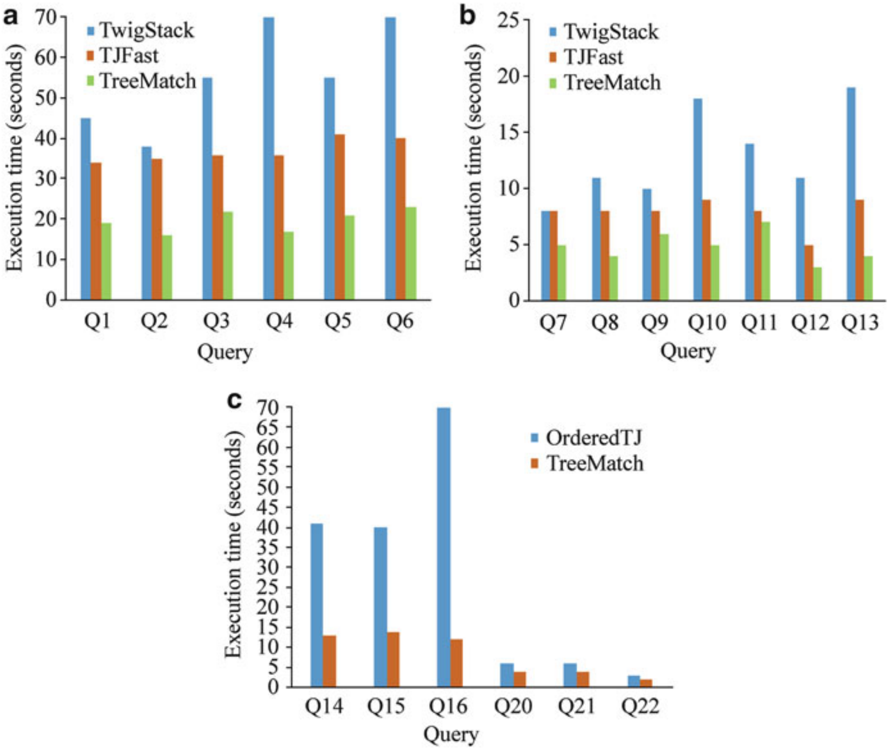


Fig. 5.27 Comparisons of different algorithms. (**a**) Time (Random data for $Q^{/,//,*}$), (**b**) Time (DBLP, tree bank for $Q^{/,//,*}$), (**c**) Time (Random data for $Q^{/,//,*,<,\neg}$)

40 K, TreeMatch does not output any useless elements for this data. It means that such main memory size is enough for TreeMatch to hold all the uncertain elements in the main memory.

5.4.4.3 Query Class $Q^{/,//,*,<,\neg}$

We show the experimental results for queries class $Q^{/,//,*,<,\neg}$, which may contain order restriction, negative edge, and wildcard. The tested queries are shown in Figs. 5.29 and 5.30.

The experiments are made by using three sizes of random documents. Table 5.8 shows the number of output elements and the number of final query answers in the case where we do not allow the outputlist in TreeMatch to buffer any elements in the main memory. For all optimal queries (Q_{15}–Q_{19}), the number of output elements is the same as that of final results. This result verifies the correctness of theorems about the optimality of TreeMatch algorithm.

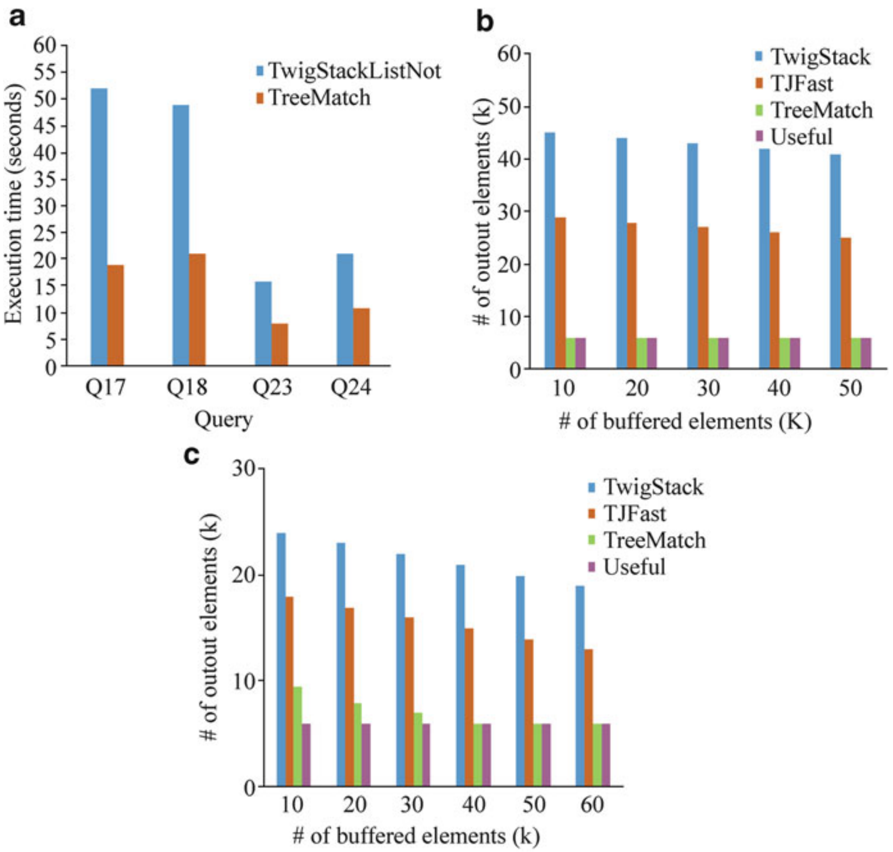


Fig. 5.28 Performance and I/O cost with varying memory. (**a**) Time (DBLP, treeBank for $Q^{/,//,*,<,\neg}$), (**b**) output with varying memory (Q_1), (**c**) output with varying memory (Q_6)

Table 5.8 Number of elements for ordered query, output (O) versus useful (U)

	*D*1		*D*2		*D*3	
Query	O	U	O	U	O	U
Q_{14}	3, 596	2, 451	17, 922	12, 505	35, 959	24, 716
Q_{15}	2, 481	2, 481	12, 367	12, 367	24, 575	24, 575
Q_{16}	1, 075	1, 075	5, 408	5, 408	10, 820	10, 820
Q_{17}	19, 792	19, 792	100, 008	100, 008	199, 727	199, 727
Q_{18}	3, 926	3, 926	20, 182	20, 182	39, 796	39, 796
Q_{19}	19, 565	19, 565	190, 789	190, 789	246, 783	246, 783

Finally, experiments on the DBLP and TreeBank are made with queries in Fig. 5.30. The execution time is shown in Figs. 5.27c and 5.28a. Since Q_{14}–Q_{16} and Q_{20}–Q_{22} are order-based queries, we compare TreeMatch with OrderedTJ. In addition, Q_{17}, Q_{18}, Q_{23}, and Q_{24} have negative edges, so we compare TreeMatch

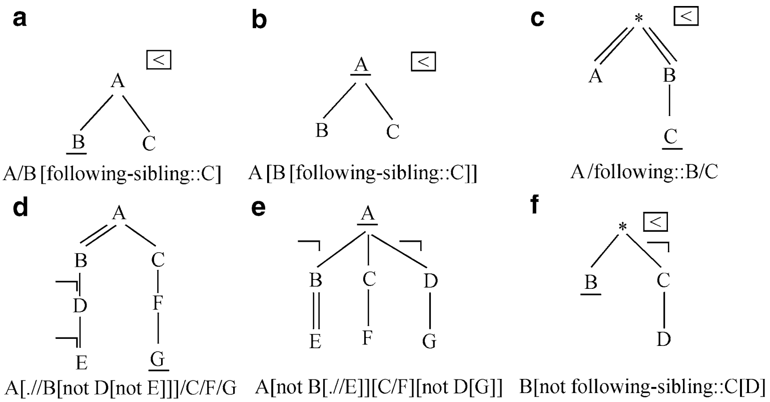
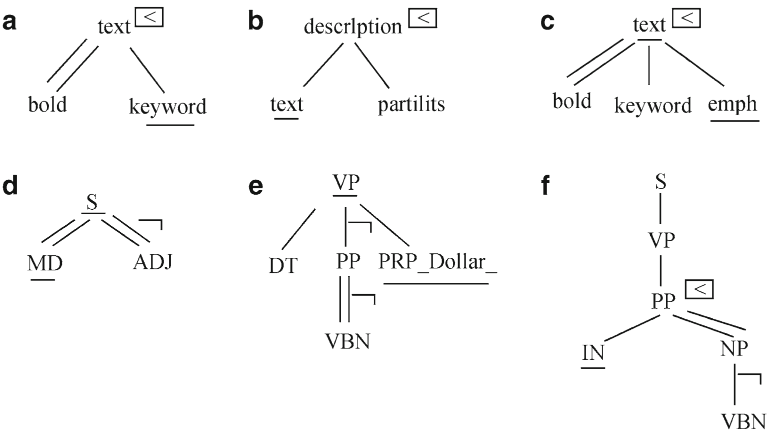

a
A
B C
A/B [following-sibling::C]
b
A
B C
A [B [following-sibling::C]]
c
*
A B
C
A/following::B/C
d
A
B C
D F
E G
A[.//B[not D[not E]]]/C/F/G
e
A
B C D
E F G
A[not B[.//E]][C/F][not D[G]]
f
*
B C
D
B[not following-sibling::C[D]

Fig. 5.29 Queries for class $Q^{/,//,*,<,\neg}$. (**a**) Q_{14} (sub-optimal), (**b**) Q_{15} (optimal), (**c**) Q_{16} (optimal), (**d**) Q_{17} (optimal), (**e**) Q_{18} (optimal), (**f**) Q_{19} (optimal)



Query	XPath experssions
Q20	text/bold/following-sibling::keyword
Q21	//description/partilist/preceding-sibling::text
Q22	//text/bold[following-sibling::keyword[following-sbling::exph]]
Q23	S[not.//ADJ]//MD
Q24	VP[DT][not PP[not.//VBN]]/PRP_DOLLAR_
Q25	S/VP/PP/IN[follwing-sibling::NP[not VBN]]

Fig. 5.30 Query for DBLP and TreeBank data. (**a**) Q_{20} (optimal), (**b**) Q_{21} (sub-optimal), (**c**) Q_{22} (optimal), (**d**) Q_{23} (optimal), (**e**) Q_{24} (sub-optimal), (**f**) Q_{25} (optimal)

with TwigStackListNot. From all tested queries, TreeMatch has better performance than the previous algorithms. We contribute this improvement to the larger optimal query class TreeMatch algorithm achieves. Finally, as for queries Q_{19} and Q_{25}, since two queries contain wildcards, negative edge, and order restriction, only TreeMatch can answer such complicated queries. Note that the above execution performance is achieved by using a relatively very small buffer size; we expect that this system can scale well for even gigabytes of XML data based on the current machine.

5.5 Summary

In this chapter, we mainly introduce algorithms to process ordered XML tree pattern, generalize tree pattern, and extend XML tree pattern. All of these approaches are proposed to reduce useless nodes to output and to identify a large query class to guarantee the I/O optimality.

Using ordered child extension (OCE) to determine whether an element possibly involves in query answers, OrderedTJ [LLY+05] gets the final results efficiently. Besides, a large query class is identified to guarantee I/O optimal for OrderedTJ. According to the experiment results, we find the effectiveness, scalability, and efficiency of this algorithm. To efficiently evaluate generalized XML tree patterns, GTJFast [LU06] was proposed to compactly represent the intermediate matching results and avoid the output of non-return nodes. In the end, we introduced a notion of matching cross [LLB+11] to address the problem of the suboptimality in holistic XML tree pattern matching algorithms. Matching cross reflects the situation in which holistic algorithms have to output some uncertain elements to avoid losing any useful solutions within the constraint of limited main memory. We identify a large optimal query classes for three kinds of queries, that is, $Q^{/,//,*}$, $Q^{/,//,*,<}$, and $Q^{/,//,*,<,\neg}$, respectively. Based on these results, the Algorithm TreeMatch [LLB+11] is shown to achieve such theoretical optimal query classes.

References

[BBC+02] Berglund, A., Boag, S., Chamberlin, D., Fernandez, M.F., Kay, M., Robie, J., Simeon, J.: XML path language (XPath) 2.0 W3C working draft 16, Technical Report WDxpath20-20020816, World Wide Web Consortium, August 2002. http://www.w3.org/TR/xpath20N.

[BCF+07] Boag, S., Chamberlin, D., Fernandez, M.F., Florescu, D., Robie, J., Simeon, J.: XQuery 1.0: an XML query language. W3C Recommendation 23 Jan 2007

[BKS02] Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. Technical Report, Columbia University (2002)

[CLL05] Chen, T., Lu, J., Ling, T.W.: On boosting holism in XML twig pattern matching using structural indexing techniques. In: Proceeding of the ACM SIGMOD International Conference on Management of Data, Baltimore (2005)

[CLT+06] Chen, S., Li, H., Tatemura, J., Hsiung, W., Agrawal, D., Candan, K.S.: Twig2Stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In: Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, pp. 283–294 (2006)

[CMW03] Choi, B., Mahoui, M., Wood, D.: On the optimality of holistic algorithms for twig queries. In: DEXA, pp. 28–37 (2003)

[CVZ+02] Chien, S., Vagena, Z., Zhang, D., Tsotras, V.J., Zaniolo, C.: Efficient structural joins on indexed XML documents. In: Proceeding of VLDB, Hong Kong, pp. 263–274 (2002)

[JLW04] Jiang, H., Lu, H., Wang, W.: Efficient processing of XML twig queries with OR-predicates. In: Proceeding of the SIGMOD, Paris, pp. 59–70 (2004)

[JWLY03] Jiang, H., Wang, W., Lu, H., Yu, J.X.: Holistic twig joins on indexed XML documents. In: Proceedings of 29th International Conference on Very Large Data Bases, Berlin, pp. 273–284 (2003)

[LCL04] Lu, J., Chen, T., Ling, T.W.: Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In: CIKM, Washington, DC, pp. 533–542 (2004)

[LCL05] Lu, J., Chen, T., Ling, T.W.: TJFast: effective processing of XML twig pattern matching. In: Proceedings of the WWW (Special interest tracks and posters), Chiba, pp. 1118–1119 (2005)

[LLB+11] Lu, J., Ling, T.W., Bao, Z., Wang, C.: Extended XML tree pattern matching: theories and algorithms. IEEE Trans. Knowl. Data Eng. (TKDE) **23**(3), 402–416 (2011)

[LLCC05] Lu, J., Ling, T.W., Chan, C., Chen, T.: From region encoding to extended Dewey: on efficient processing of XML twig pattern matching. In: Proceedings of 31th International Conference on Very Large Data Bases (VLDB), Trondheim, pp. 193–204 (2005)

[LLY+05] Lu, J., Ling, T.W., Yu, T., Li, C., Ni, W.: Efficient processing of ordered XML twig pattern matching. In: DEXA, Copenhagen, pp. 300–309 (2005)

[LU06] Lu, J.: Efficient processing of XML twig pattern matching. In: PhD thesis, National University of Singapore (2006)

[QYD07] Qin, L., Yu, J.X., Ding, B.: TwigList: make twig pattern matching fast. In: Proceedings of the 12th International Conference on Database Systems for Advanced Applications, DASFAA 2007, Bangkok, pp. 850–862 (2007)

[SY08] Shalem, M., Bar-Yossef, Z.: The space complexity of processing XML twig queries over indexed documents. In: ICDE, Cancun (2008)

[TB] Treebank.: http://www.cs.washington.edu/research/xmldatasets/www/repository.html

[WLH07] Wang, F., Li, J., Homayounfar, H.: A space efficient XML DOM parser. Data Knowl. Eng. (DKE) **60**(1), 185–207 (2007)

[YLL06] Yu, T., Ling, T.W., Lu, J.: Twigstacklistnot: a holistic twig join algorithm for twig query with not-predicates on XML data. In: DASFAA, Singapore, pp. 249–263 (2006)

[ZND+01] Zhang, C., Naughton, J.F., Dewitt, D.J., Luo, Q., Lohman, G.M.: On supporting containment queries in relational database management systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Santa Barbara, pp. 425–436 (2001)

Chapter 6
Effective XML Keyword Search

Abstract XML keyword search has emerged as one of the most effective paradigms for finding desired information in hierarchical XML documents. One of the key advantages of the keyword search is its simplicity, that is, users do not have to learn complex query languages and be familiar with the structures of the XML documents. In this chapter, the state-of-the-art meaningful keyword search semantics, such as SLCA, VLCA, and MLCA, are fully illustrated. Furthermore, both the inverted-lists-based and stack-based keyword search algorithms built upon the semantics are also studied. In addition, in order to address the new challenges in XML keyword search, that is, identifying the user search intention and resolving keyword ambiguity, we show an XML TF*IDF ranking strategy based on guidelines that a search engine should meet in both search intention identification and relevance-oriented ranking for search results.

Keywords Keyword search • LCA • SLCA • MLCEA • MCT • DMCT • GDMCT • ICA • IRA • ELCA • VLC • MCN • MEW • LCEA • MLCEA • DIL • BMS • IMS • Query refinement • TF*IDF • Data model • XML TF&DF • Search For • Search Via • IQD • SD • VTD

6.1 Introducing Effective XML Keyword Search

Keyword search is a proven user-friendly way of querying HTML documents in the World Wide Web. As XML is becoming a standard in data representation, it is desirable to support keyword search in XML database. It allows users to find the information they are interested in without having to learn complex query languages or needing prior knowledge of the structure of the underlying data.

Traditional query processing approaches on relational and XML databases are constrained by the query constructs imposed by the language such as SQL and XQuery. Firstly, the query language themselves are hard to comprehend for

J. Lu, *An Introduction to XML Query Processing and Keyword Search*,
DOI 10.1007/978-3-642-34555-5_6,

non-database users. For example, the XQuery is fairly complicated to grasp. Secondly, these query languages require the queries to be posed against the underlying, sometimes complex, database schemas. These traditional querying methods are powerful but unfriendly for day-to-day users. Keyword search is proposed as an alternative means of querying the database, which is simple and yet familiar to most internet users as it only requires the input of some keywords. Although keyword search has been proven to be effective for text documents (e.g., HTML documents), the problem of keyword search on the structured data (e.g., relational databases) and the semistructured data (e.g., XML databases) is neither straightforward nor well studied.

Keyword search in text documents takes the documents that are more relevant with the input keywords as the answers, while that on relational databases will take the correlative records of database that contain all the keywords as the answers. However, it still remains an open problem that, for XML database, what should be the answer for keyword search? The notion of lowest common ancestor (LCA) has been introduced to answer keyword queries on XML databases [LFW+07]. More recently, XRank, meaningful LCA (MLCA), smallest LCA (SLCA), grouped distance minimum connecting tree (GDMCT), and XSeek have been proposed to improve the efficiency and effectiveness of keyword search against LCA in [LFW+07, GSB+03, SKW01, LYJ04, HKP+06, LD07, ZBL+09], respectively.

However, existing proposals on keyword search over XML databases suffer from two problems: meaningfulness and ranking of answers. First, the existing approaches, such as SLCA and XRank, return some irrelevant results or false positives as answers and may also miss some results from answers. Second, the number of the answers for the input query is usually large; therefore, it is necessary to rank the result and choose the best K results to users rather than throwing all the answers to them.

In this chapter, we first present a survey on the existing XML keyword search semantics algorithms and ranking strategy. We begin with the introduction of XML keyword search semantics. Next we introduce XML keyword search algorithms. Finally we introduce the XML keyword search ranking strategy.

6.2 XML Keyword Search Semantics

Most existing works model XML data as a tree and propose matching semantics based on it. LCA (lowest common ancestor) [LFW+07] is proposed to find XML nodes, each of which is the lowest common ancestor containing all query keywords in its subtree. XSEarch [HXZ+08] introduces the concept of interconnection that is a variation of LCA, which claims two matches are interconnected if there are no two distinct nodes with the same tag name on the path between these two nodes (through their LCA), excluding themselves. Subsequently, SLCA (smallest LCA) is proposed to find all nodes, each of which contains matches to all keywords in its subtree and none of its descendants does. In particular, Li et al. incorporate meaningful LCA search in XQuery; XKSearch focuses on studying efficient algorithms to compute SLCAs by designing an index lookup algorithm and a scan eager algorithm. Multiway-SLCA is proposed to further optimize the performance of finding SLCAs by maximizing the skipping of redundant LCA computations that contribute to the same SLCA result. GDMCT (minimum connecting trees) excludes the subtrees rooted at the LCAs that do not contain any query keyword. XSeek infers the return nodes of a keyword query based on the characteristics of XML data and the pattern of input keywords. Recently, EASE proposes a unified graph index to handle keyword search on heterogenous data which includes unstructured, structured, and semistructured data.

6.2.1 LCA and the Meet Operator

Given m nodes $n_1, n_2, \ldots, n_m$, v is called LCA [GSB+03, SKW01] of these m nodes, iff $\forall 1 \le i \le m$, v is an ancestor of node n_i, and not $\exists$ the node u, $v < u$, u is also an ancestor of each n_i, denoted as $v = \mathrm{LCA}(n_1, n_2, \ldots, n_m)$. Schmidt et al. introduce the meet operator for XML; their approach is based on computing the lowest common ancestor (LCA) of nodes in the XML syntax tree: for example, given two strings, they look for nodes whose offspring contains these two strings. For two nodes in the syntax tree o_1 and o_2, the meet operator [SKW01] meet (o_1, o_2) simply returns the lowest ancestor of nodes o_1 and o_2. Node v is an LCA of keyword set $K = \{w_1, w_2, \ldots, w_k\}$ if the subtree rooted at v contains at least one occurrence of all keywords in K, after excluding the subelements that already contain all keywords in K.

6.2.2 MLCA and MLCAS

6.2.2.1 MLCA

Let the set of nodes in an XML document be N. Given A, $B \subseteq N$, where A is comprised of nodes of type A, and B is comprised of nodes of type B, the meaningful lowest common ancestors set $C \subseteq N$ of A and B satisfies the following conditions:

1. $\forall C_k \in C$, $\exists a_i \in A$, $b_j \in B$, such that $C_k = \mathrm{LCA}(a_i, b_j) \cdot C_k$ is denoted as $\mathrm{MLCA}(a_i, b_j)$.
2. $\forall a_i \in A$, $b_j \in B$ if $d_{ij} = \mathrm{LCA}(a_i, b_j)$ and $d_{ij} \notin C$, then $\exists C_k \in C$, descendent(C_k,d_{ij}) = true.

The set C is denoted as MLCASET(A, B) [SCG07].

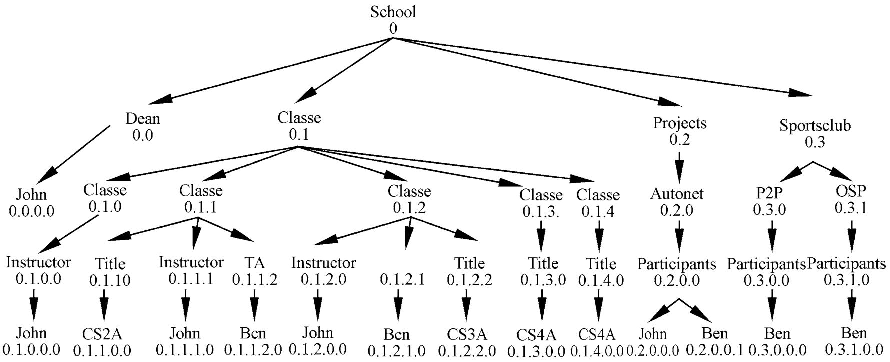


Fig. 6.1 SLCA

6.2.2.2 MLCA Set

Let the set of nodes in an XML document be N. Given $A_1, A_2, \ldots, A_n \subseteq N$, where $\forall i$, $a_{ij} \in A_j$ is of type A_j ($j \in [1, \ldots, m]$),the meaningful lowest common ancestor structure set $S = \{(r, a_1, \ldots, a_m) | r \in N, a_i \in A_j \ (j \in [1, \ldots, m], r = \text{MLCA}(a_1, \ldots, a_m))\}$. Each element of this set is denoted as MLCA($a_1, \ldots, a_m$) [LYJ04], with r as its root. Each MLCAS is a refined context for query evaluation and contains only the nodes that are meaningfully related to one another. If an MLCAS satisfies the search conditions, it is unlikely to contain a wrong answer.

6.2.3 SLCA

Given a list of keywords $w_1, \ldots, w_k$, and an input XML tree T, we assume S_i denotes the keyword list of w_i, that is, the list of nodes whose label directly contains w_i sorted by id. $V < v''$ denotes that node v is an ancestor of node v'; $v \leq v'$ denotes that $v < v'$ or $v = v'$. Given a node $v \in S$, v is called an ancestor node in S if there exists a node v' in S such that $v \leq v'$. When the set S is implied by the context, we simply say v is an ancestor node.

The function lca($v_1, \ldots, v_k$) computes the lowest common ancestor or LCA of nodes $v_1, \ldots, v_k$ and returns null if any of the arguments is null. Given two nodes v_1, v_2 and their Dewey numbers p_1, p_2, lca(v_1,v_2) is the node with the Dewey number that is the longest common prefix of p_1 and p_2. For example, the LCA of nodes 0.1.1.1.0 and 0.1.1.2.0 is the node 0.1.1 in Fig. 6.1.

Given sets of nodes $S_1, \ldots, S_n$, a node belongs to lca($v_1, \ldots, v_k$), if there exist $v_1 \in S_1, \ldots, v_k \in S_k$ such that $v = \text{lca}(v_1, \ldots, v_k)$. Then v is called an LCA of sets $S_1, \ldots, S_n$.

A node v belongs to the smallest lowest common ancestor (SLCA) [CMK03, XP05] slca($S_1, \ldots, S_k$) of $S_1, \ldots, S_n$ and $\forall u \in \text{lca}(S_1, \ldots, S_n)$. v is called an SLCA

Fig. 6.2 Input-labeled tree used in examples

```
[r,1,42,0] root
 [cl,2,41,1] conference
   [s1,3,16,2] session
     [p1,4,9,3] paper
       [a1,5,6,4] author (Harry Smith)
       [a2,7,8,4] author (Tom Jones)
     [p2,10,15,3] paper
       [a3,11,12,4] author (Tom Brown)
       [a3,13,14,4] author (Dick Smith)
    [a2,17,26,2] seesion
      [p3,18,25,3] paper
       [a5,19,20,4] author (Tom Green)
       [a6,21,22,4] author (Harry Brown)
       [a7,23,24,4] author (Dick Jones)
    [a3,27,40,2] session
     [p4,28,31,3] paper
       [a8,29,30,4] author (Harry Jones)
     [a5,32,35,3] paper
       [a9,33,34,4] author (Tome Smith)
     [a6,36,39,3] paper
       [a10,37,38,4] author (Dick Brown)
```

of sets $S_1, \ldots, S_n$, if $v \in \text{slca}(S_1, \ldots, S_k)$. Notice that the query result $\text{slca}(w_1, \ldots, w_k)$ is $\text{slca}(S_1, \ldots, S_k)$ and that $\text{slca}(S_1, \ldots, S_k) = \text{removeAncestor}(\text{lca}(S_1, \ldots, S_k))$ where removeAncestor removes ancestor nodes from its input.

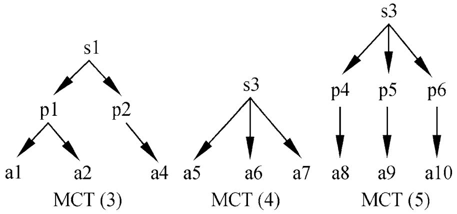
6.2.4 GDMCT

6.2.4.1 MCT

The minimum connecting tree(MCT) [HKP+06] of nodes $v_1, \ldots, v_m$ of the input-labeled tree T is the minimum size subtree TM of T that connects $v_1, \ldots, v_m$. The root of the tree is called the lowest common ancestor (LCA) of the nodes $v_1, \ldots, v_m$.

An MCT of keywords $k_1, \ldots, k_m$ is an MCT of nodes $v_1, \ldots, v_m$ that contain the keywords. For example, the MCTs (1) and (2) are two of the MCTs of the query “Tom, Harry” and the MCTs (3), (4), and (5) correspond to the query “Tom, Dick, Harry.” (Fig. 6.2).

$$a1 \leftarrow p2 \rightarrow a2 \qquad \text{MCT (1)}$$
$$a2 \leftarrow p4 \rightarrow s3 \rightarrow p5 \rightarrow a9 \qquad \text{MCT (2)}$$



6.2.4.2 DMCT

Consider nodes $v_1,\ldots, v_m$ of the input tree *T*. The Distance MCT(DMCT) TD $= d$(TM) of the MCT TM of nodes $v_1,\ldots, v_m$ is the minimum node-labeled and edge-labeled tree such that:

1. TD contains the nodes $v_1,\ldots, v_m$.
2. TD contains the LCAs $u_1,\ldots, u_k$ of any pair of nodes (v_i,v_j), where $v_i,v_j \in [v1,\ldots, v_m],i \neq j$.
3. There is an edge labeled with the number between any two distinct nodes n, $n' \in \{v_1,\ldots, v_m,u_1,\ldots, u_m\}$ if there is a path of length 1 from n' to n in TM and the path does not contain any node $n^n \in \{u_1,\ldots, u_m\}$ other than n and n'.

The DMCT (10) corresponds to the MCT (1), and the DMCTs (11)–(14) correspond to the MCTs (6)–(9).

$$a_2 \leftarrow p_1 \rightarrow s_1 \leftarrow c_1 \rightarrow s_2 \rightarrow p_3 \rightarrow a_6 \quad \text{MCT (6)}$$
$$a_3 \leftarrow p_2 \rightarrow s_1 \leftarrow c_1 \rightarrow s_3 \rightarrow p_4 \rightarrow a_g \quad \text{MCT (7)}$$
$$a_2 \leftarrow p_1 \rightarrow s_1 \leftarrow c_1 \rightarrow s_3 \rightarrow p_4 \rightarrow a_g \quad \text{MCT (8)}$$
$$a_3 \leftarrow p_2 \rightarrow s_1 \leftarrow c_1 \rightarrow s_2 \rightarrow p_3 \rightarrow a_6 \quad \text{MCT (9)}$$

6.2.4.3 GDMCT

A grouped DMCT of a tree *T* is a labeled tree where edges are labeled with numbers and nodes are labeled with lists of node ids from *T*. A DMCT *D* belongs to a GDMCT *G* if *D* and *G* are isomorphic. Assuming that *f* is the mapping of the nodes of *D* to the nodes of *G*, which induces a corresponding mapping, also called *f*, of the edges of *D* to the edges of *G*, the following must hold:

1. If *nD* is a node of *D*, *nG* is a node of *G*, and $F(nD) = nG$, then the label of *nG* contains the id of *nD*.
2. If *eD* is an edge of *D*, *eG* is an edge of *G*, and $F(eD) = eG$, then the label of *eD* and the label of *eG* are the same number.

$$a_1 \xleftarrow{1} p_1 \xrightarrow{1} a_2 \qquad \text{DMCT (10)}$$

$$a_2 \xleftarrow{3} c_1 \xrightarrow{3} a_6 \qquad \text{DMCT (11)}$$

$$a_3 \xleftarrow{3} c_1 \xrightarrow{3} a_g \qquad \text{DMCT (12)}$$

$$a_2 \xleftarrow{3} c_1 \xrightarrow{3} a_g \qquad \text{DMCT (13)}$$

$$a_3 \xleftarrow{3} c_1 \xrightarrow{3} a_6 \qquad \text{DMCT (14)}$$

$$u_1[a_2, a_3] \xleftarrow{3} u_0[c_1] \xrightarrow{3} u_2[a_6, a_g] \qquad \text{GDMCT (15)}$$

The GDMCT (15) captures DMCTs (11)–(14). The notation u_1 [a_2, a_3] indicates that the label of the node u_1 is [a_2, a_3]. Note that each tree that is an instance of a GDMCT and is also a subtree of the XML data tree T is a DMCT of an MCT of T.

6.2.5 ICA (Interested Common Ancestor) and IRA (Interested Related Ancestors)

6.2.5.1 ICA

Given a query Q of a list of keywords, the Interested Common Ancestors (ICAs) [LD07] of Q, denoted as ICA (Q), is a list L of interested objects such that each $o_i \in L$ contains all query keywords.

ICA semantics can be viewed as an extension from LCA semantics. However, as discussed, ICA offers more flexibility based on the interested object concept whose conceptual subtree may include ancestors or group-by information in the physical XML databases.

6.2.5.2 IRA

Given a query Q of a list of keywords, the Interested Related Ancestors (IRA) [LD07] of Q, denoted as IRA(Q), is a list L of conceptually related interested object pairs such that each object of a pair contains some query keywords and the pair together contain all query keywords. We also say that an interested object is in IRA results or is an IRA object for Q if it is a member of some IRA pair and it is not in ICA results for Q. Note that an ICA object o can also form an IRA pair with an object o containing some query keywords to include o as an IRA object. However, the ICA result o is not double counted as IRA object.

6.2.6 ELCA (Exclusive Lowest Common Ancestor)

A node v is called an exclusive lowest common ancestor (ELCA) of $S_1, \ldots, S_k$ if and only if there exist nodes $n_1 \in S_1, \ldots, n_k \in S_k$ such that $v = \text{lca}(n_1, \ldots, n_k)$ and for every $n_i (1 \leq i \leq k)$ the child of v in the path from v to n_i is not an LCA of $S_1, \ldots, S_k$ itself nor ancestor of any LCA of $S_1, \ldots, S_k$.

6.2.7 VLCA (Valuable Lowest Common Ancestor)

Given two nodes, u, v, and $w = \text{LCA}(u,v)$. uSet and vSet are two sets of nodes in the paths of $w \rightarrow u$ and $w \rightarrow v$, respectively. Let wSet=uSet ∪ v set.

u and v are heterogenous, iff $\exists u' \in \text{wSet}$, $v' \in \text{wSet}$, u' and v' are of the same elementary type, that is, $\lambda(u') = \lambda(v')$, except u and v may be having the same elementary type. On the contrary, u and v are homogenous, iff there are no two nodes of the same elementary type, except u and v.

6.2.7.1 Valuable LCA

Given m nodes $n_1, n_2, \ldots, n_m$, $v = \text{LCA}(n_1, n_2, \ldots, n_m)$. $\text{VLCA}(n_1, n_2, \ldots, n_m) = v$, iff these m nodes are homogenous, that is, $\forall 1 \leq i < j \leq m$, $n_i \neq n_j$ [LC07].

6.2.7.2 VLCASet

Given query $K = \{k_1,k_2, \ldots, k_m\}$ and an input XML document D, the set of VLCAs, w.r.t. K and D, is $\text{VLCASet} = \text{VLCA}(I_1, \ldots, I_m) = \{v | v = \text{VLCA}(v_1, \ldots, v_m), v_i \in I_i\}$. LCASet is the set of the nodes, which contain all of the input keywords. SLCASet is a subset of LCASet, which eliminates the LCAs that have LCA descendants. VLCASet can avoid the false positives and false negatives introduced by SLCA and is a more accurate subset of LCASet to answer keyword queries. To describe how each result matches a keyword query, we introduce a meaningful concept as defined as.

Given a query $K = \{k_1,k_2, \ldots, k_m\}$ and an XML document D. $K(D,(k_1,k_2, \ldots, k_m))$ is the answer of K on D, where $K(D,(k_1,k_2, \ldots, k_m)) = \{((r; \lambda(v1){:}c1, \lambda(v2){:}c2, \ldots, \lambda(v_\mathrm{m}){:}C_m)| \ \forall v_i \in I_i, r = \text{VLCA}(v_1,v_2, \ldots, v_m))$, where c_i denotes the test content that v_i contains.$\}$

Each result is represented as a connected subtree, which is rooted at a VLCA and contains the corresponding content nodes so that each result is self explained and can explain how it matches the keyword query.

6.2.8 *MCN*

6.2.8.1 Walk

A $v_0 - v_k$ walk W:v_0,e_1,v_1,e_1,v_2,...,v_{k-1},e_k,v_k of the undirected schema graph Su is a sequence of vertices of S_u beginning with v_0 and ending at v_k, such that each two consecutive vertices v_{i-1} and v_i are joined by an edge e_i of S_u. The number of edges of W is called the length of W, which is denoted as $L(W)$. For any two nodes u and v of S_u, if there exists at most one edge joining u and v in S_u, W can be written as W:v_0,v_1,v_2,...,v_k. Any $v_i - v_j$ walk W':v_i,e_{i+1},v_{i+1},...,v_{j-1},e_j,$v_j$$(0 \leq i \leq j \leq k)$ extracted from W is called a subwalk of W, which is denoted as $W' \subseteq W$.

6.2.8.2 Meaningful Entity Walk (MEW)

Let S be a schema graph; a $v_0 - v_k$ walk W of the undirected schema graph S_u [ZBL+09] is a meaningful entity walk of S if both v_0 and v_k are entity nodes and

1. $L(W) \leq 1$ or
2. W doesn't contain a subwalk W' that has the form $u \rightarrow v \leftarrow w$ in S, where "→" denotes a solid arrow from $u(w)$ to v in S. Moreover, if W' has the form $u \leftarrow v \rightarrow w$ in S, v must be an entity node.

6.2.8.3 Meaningful Connected Network (MCN)

Let $Q = \{k_1, k_2, \ldots, k_m\}$ be the given keyword query. A meaningful connected network of Q [ZBL+09] on the XML document D is a subgraph T of D, which holds all the following properties:

1. T contains $k_i (1 < i < m)$ at least once.
2. For any node u of T, if u is not an entity instance and joined by just one edge with other nodes of T, u contains at least one keyword.
3. For any two nodes u and v of T, if u and v are entity instances, there exists at least one $u - v$ MEW instance W on T.
4. No proper subgraph of T can hold for the above three properties.

6.2.9 *Meaningful SLCA*

XML data model: a rooted, labeled, unordered tree, denoted as $D = (V,E)$, where V is a set of nodes and E is a set of tree edges in D, such as the bibliographic data shown in Fig. 6.3.

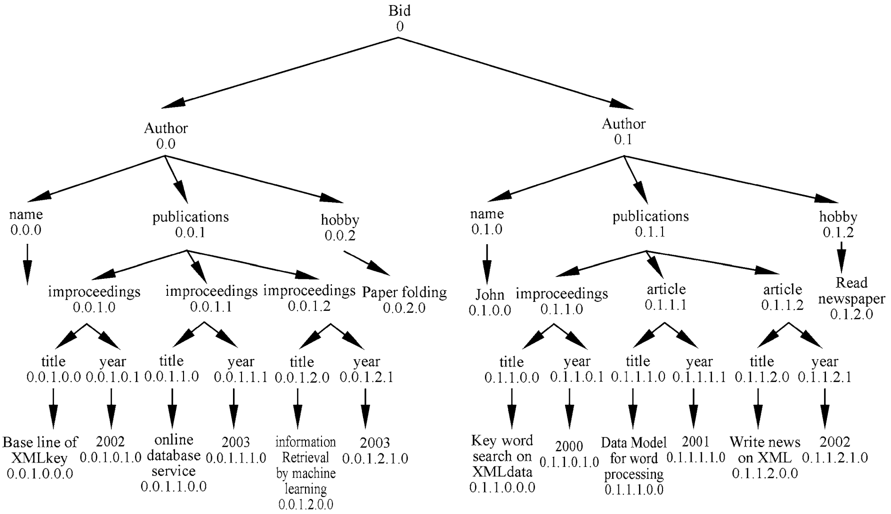


Fig. 6.3 Meaningful SLCA

Each node $n \in V$ is represented in form of tag: deweyID, where tag is the tag name of n, and deweyID is the Dewey label of n. For example, in Fig. 6.3, the first author element under bib is assigned a label 0.0, denoting it is the first child of node bib:0. A user keyword query $Q = \{k_1,k_2,\ldots, k_m\}$ is a set of keywords, where each k_i may match the label term and/or value term in XML data D, and the order of keywords is insensitive. We use S_i to denote the inverted list of keyword k_i. A set of nodes $M = \{v_1,v_2,\ldots, v_n\}$ is defined as a match for Q if $v_i \in S_i$ for each $i \in [1,n]$.

6.2.9.1 Meaningful SLCA

An SLCA result r is a meaningful SLCA [SCG07] of query Q on XML database D, iff:

1. $r \in$ SLCA (Q, D) by SLCA.
2. r is not the root node of D.

Besides the above two mandatory properties for meaningful SLCA, an optional property is that the element denoted by r is of either entity category or attribute category according to the entity inference rule defined. This property is designed to coincide with the fact that user's search target is normally at real entity level. For example, in Fig. 6.3, people's concern is on author, in proceedings and article which are of entity category, while publications are not. Given two keyword sets $S1$ and $S2$ and a database D, if $S1$ is a subset of $S2$ and there is at least one meaningful SLCA result in D based on $S2$, then there is also at least one meaningful SLCA result in D based on $S1$.

6.2.10 LCEA (Lowest Common Entity Ancestor)

Given sets of nodes n_1,n_2, . . . , n_m, node v is their Lowest Common Entity Ancestor (LCEA) [SSY+09] iff:

1. v is an entity node.
2. $\forall 1 \leq i \leq m$, v is an ancestor of node n_i.
3. not $\exists$ the entity node u, $v < u$, u is also an ancestor of each n_1, then $v =$ LCEA (n_1,n_2, . . . , n_m).

6.2.11 MLCEA (Meaningful LCEA)

The attribute of value node u is represented inform of Attr(u). If v is the attribute node of u, then Attr(u) = tag(u). If u is also an attribute node, then Attr(u) = Tag(u). Given sets of nodes n_1, n_2, . . . , n_m, $v =$ LCEA(n_1, n_2, . . . , n_m). v is a meaningful LCEA, that is, $v =$ MLCEA(n_1, n_2, . . . , n_m), iff not $\exists$ the node n_i,n_j($1 \leq i \leq j \leq m$) satisfy the expression Attr(n_i) $\neq$ Attr(n_j) $\wedge$ LCEA(n_i) $\neq$ LCEA(n_j).

6.3 XML Keyword Search Algorithms

6.3.1 DIL (Dewey Inverted List) Query Processing Algorithm

The inputs to the algorithm are n query keywords (k_1, . . . , k_n), and the desired number of top-ranked query results(m). The algorithm works for $n > 1$, and the case where $n = 1$ is handled as a (simple) special case. The algorithm maintains two data structures: the result heap and the Dewey stack. The result heap keeps track of the top m results seen so far. The Dewey stack [V09] stores the ID, rank, and position list of the current Dewey ID, and also keeps track of the longest common prefixes computed during the merge of the inverted lists.

The overall ranking of a result element v_1 for query $Q = (k_1, \ldots, k_n)$ is computed as

$$R(v_1, Q) = \left(\sum \bar{r}(v_1, k_i)\right) \times p\,(v_1, k_1, k_2, \ldots, k_n) \tag{6.1}$$

By the definition of R, we know that contains (v_1, k_i) is true for every k_i. Hence, there is a sequence of containment edges of the form (v_1, v_2) (v_2, v_3), . . . , (v_{n-1}, v_n) such that v_n directly contains k_i. We define

$$r(v_1, k_1) = \text{ElemRank}(v_n) \times \text{decay}^{n-1} \tag{6.2}$$

Intuitively, the rank of v_1 with respect to a keyword k_i is ElemRank(v_n), where v_n directly contains k_i, scaled appropriately to account for the specificity of the result. When the result element v_1 directly contains the keyword (i.e., $v_1 = v_n$), the rank is just the ElemRank of the result element. When the result element indirectly contains the keyword (i.e., $v_1 \neq v_n$), the rank is scaled down by a factor decay for each level. Decay is a parameter that can be set to a value in the range 0–1. The astute reader may have noticed that $\bar{r}(v_1, k_i)$ does not depend on the ElemRank of the result node v_1, except when $v_1 = v_n$. We chose to have $\bar{r}(v_1, k_i)$ depend on the ElemRank of v_n rather than the ElemRank of v_1 for the following two reasons: first, by scaling down the same quantity ElemRank(v_n), we ensure that less specific results indeed get lower ranks, and second, the ElemRank(v_n) is in fact related to ElemRank(v_1) due to certain properties of containment edges. In the above formula, we have implicitly assumed that the query keyword k_i occurs only once in the result element. In case there are multiple(say, m) occurrences of k_i, we first compute the rank for each occurrence using the above formula. Let the computed ranks be r_1, $r_2, \ldots, r_m$. The combined rank is

$$\bar{r}\,(v_1, k_i) = f\,(r_1, r_2, \ldots, r_m)$$

Here f is some aggregation function. $f = \max$, but other choices (such as $f = \text{sum}$) are also supported.

```
PROCEDURE EvaluateQuery($k_1$,$k_2$,...,$k_n$,m) returns idList
// $k_1$,$k_2$,...,$k_n$ are the query keywords, m is the desired number of query results
// invertedList[i] is the inverted list for keyword $k_i$
resultHeap = empty;    //Initialize the result heap of size m
deweyStack = empty;    //Initialize the Dewey stack
while(eof has not been reached on all inverted lists){
    //Read the next entry from the inverted list having the smallest DeweyID
    find ilIndex such that the next entry of invertedList[ilIndex]is the smallest DeweyID
    currentEntry = invertedList[ilIndex].nextEntry;
    //Find the longest common prefix between deweyStack and currentEntry.deweyId
    find largest lcp such that deweyStack[i] = currentEntry.deweyId[i],1 < =i < =lcp
    //Pop non-matching entries in the Dewey stack;add to result heap if appropriate
    while(deweyStack.size > lcp){
        stackEntry = deweyStack.pop();
        if(stackEntry.potentialResult and stackEntry.posList non-empty for all keywords)
        {
            compute overall rank using formula (6.1)
            if overall rank is among top m seen so far,add deweyStack ID to resultHeap
        }
    }
```

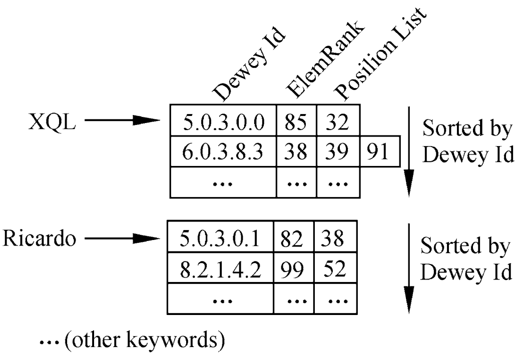


Fig. 6.4 Keyword search with query “XQL Ricardo”

```
    //Update the rank and position lists of the longest common prefix entries
15. for(all i such that 1 < =i < =lcp){
16. deweyStack[i].rank[ilIndex] = rank computed using formula (6.2)
17. deweyStack[i].posList[ilIndex] + =currentEntry.posList;
18. }
    //Add non-matching components of currentEntry.deweyId to deweyStack
19. for(all i such that mcp < i < =currDeweyIdLen){
20. stackEntry.rank[ilIndex] = rank computed using formula (6.2)
21. stackEntry.posList[ilIndex] = currentEntry.posList;
22. deweyStack.push(deweyStackEntry);
23. }
    //Set the longest common prefix entry to be a potential result
24.   deweyStack[lcp].potentialResult = true;
25. }//End of looping over all inverted lists
26. pop entries of deweyStack and add to result heap if appropriate(similar to lines
     8–15)
27. return ids in resultHeap;
```

The algorithm works by merging the inverted lists by the DeweyID (Lines 3–5) and computing the longest common prefix of the current entry and the previous entry stored in the Dewey stack (Line 6). It then pops all the Dewey stack components that are not part in the common prefix (Lines 7–14), and if any of the popped components are potential query results, they are added to the result heap (Lines 9–13). The current entry is then pushed onto the Dewey stack and the ranks and posLists are updated accordingly (Lines 15–23). The longest common prefix is set to be a potential result (Line 24). The longest common prefix will be added to the output heap in a later loop when it is popped from the Dewey stack (Lines 9–13). We now walk through the algorithm using an example. Consider the DIL shown as follows:

And consider the keyword search query “XQL Ricardo” (see Fig. 6.4). The algorithm first reads the entry with the smallest Dewey ID-5.0.3.0.0. Since the Dewey stack is initially empty, the longest common prefix is empty, and the Dewey ID components are simply pushed onto the stack, with the appropriate rank and posList fields (Lines 19–24). The state of the stack is shown as follows:

Fig. 6.5 Dewey stacks

a

Dewey	Rank[1]	Rank[2]	PosList[1]	PosList[2]	PotentialResult
0	85		32		1
0	77		32		0
3	68		32		0
0	61		32		0
5	56		32		0

b

Dewey	Rank[1]	Rank[2]	PosList[1]	PosList[2]	PotentialResult
1		82		38	0
0	77	74	32	38	1
3	68	66	32	38	0
0	61	60	32	38	0
5	56	54	32	38	0

Note that the ranks of the ancestors (prefixes) have been scaled down as per the ranking function (2). The algorithm then reads the next smallest entry, which is Dewey ID 5.0.3.0.1 in the "Ricardo" inverted list. The longest common prefix (5.0.3.0) of the current entry and the Dewey stack is often determined (Line 6), and nonmatching entries are popped from the stack (Lines 7–14). The ranks and position lists of the longest common prefix components are updated (Lines 15–18), and the longest common prefix is also marked as a potential result (Line 24). The current state of the Dewey stack is shown in Fig. 6.5b. Note that ancestors of the longest common prefix are not marked as potential results, thereby eliminating spurious results. The algorithm then reads the next smallest Dewey ID (6.0.3.8.3). Since the longest common prefix with the Dewey stack is empty, it pops the contents of the stack and adds the potential result (5.0.3.0) to the output heap. The algorithm then pushes 6.0.3.8.3 onto the stack and proceeds as before.

6.3.2 The Stack Algorithm

In the stack-based sort-merge DIL, each stack entry has a pair of components (id Keywords). Assume the id components from the bottom entry to a stack entry en are id$_1$,id$_2$,...,id$_m$, respectively. Then the stack entry en denotes the node with the Dewey number id$_1$,id$_2$,...,id$_m$. Keywords is an array of length k of Boolean values where Keywords$[i] = T$ means that the subtree rooted at the node denoted by the stack entry directly or indirectly contains the keyword w_i. For example, the top entry of the stack in Fig. 6.6b denotes the node 0.1.0, and the middle entry denotes the node 0.1. The Stack algorithm merges all keyword lists and computes the longest common prefix of the node with the smallest Dewey number from the input lists and the node denoted by the top entry of the stack. Then it pops out all top entries containing Dewey components that are not part of the common prefix. If a popped entry en contains all keywords, then the node denoted by en is an SLCA. Otherwise, the information about which keywords contain is used to update its parent entry's Keywords array. Also a stack entry is created for each Dewey component of the smallest node which is not part of the common prefix, effectively

Fig. 6.6 Stack states of stack, where J stands for "John," B stands for "Ben," and C stands for "Class"

	J	B	C	
0	T	F	F	
0	F	F	F	**a** node 0.0.0
0	F	F	F	
0	F	F	T	
1	F	F	F	**b** node 0.1.0
0	T	F	F	
0	T	F	F	
0	F	F	F	
0	F	F	T	**c** node 0.1.0.0.0
1	F	F	F	
0	T	F		
1	F	F	T	
1	T	F	T	**d** node 0.1.1
0	T	F	T	
0	T	F	F	
1	F	F	F	
1	F	F	T	**e** ndoe 0.1.1.1.0
1	T	F	T	
0	T	F	T	
0	F	T	F	
2	F	F	F	
1	T	F	T	**f** node 0.1.1.2.0
1	F	F	F	
0	T	F	T	
2	F	F	T	**g** node 0.1.2. report 0.1.1 as a SLCA
1	F	F	F	
0	F	F	F	

pushing the smallest node onto the stack. The above action is repeated for every node from the sort merged input lists.

Consider again the query "John, Ben, Class" applied on the data of Fig. 6.1 (in SLCA semantics). The keyword lists for "John, Ben, Class" are [0.0.0, 0.1.0.0.0, 0.1.1.1.0, 0.1.2.0.0, 0.2.0.0.0], [0.1.1.2.0, 0.1.2.1.0, 0.2.0.0.1, 0.3.0.0.0, 0.3.1.0.0], and [0.1.0, 0.1.1, 0.1.2, 0.1.3., 0.1.4], respectively. Initially, the smallest node is 0.0.0, and Fig. 6.6a shows the initial state of the stack where Keywords[1] = T in the top entry denotes that the node (0.0.0) represented by the top entry contains the first keyword "John." The next smallest node is the "Class" node 0.1.0. Since the longest common prefix of 0.0.0 and 0.1.0 is 0 (Line 4), the top two entries are popped out (Line 6). 0.0.0 contains "John" and this information is passed to the current top entry (Lines 12–13). Then two new entries from the two components of node 0.1.0 that are not among the longest common prefix are pushed into the stack (Line 16). Notice that in Fig. 6.6b, Keywords[1] = T in the bottom entry denotes that the node 0 (School) contains the keyword "John." Each figure in Fig. 6.6 shows the state of the stack after processing the node shown in the caption.

```
1. Stack = empty
2. while(has not reached the end of all keyword lists){
3.    V = getSmallestNode()
      //find the largest p such that stack[i] = v[i],1<=i<=p
4.  P = lca(stack,v)
5.  while(stack.size>p){
6.      stackEntry = stack,pop()
7.      if(isSLCA(stackEntry)){
     //Any other stack entry cannot represent a SLCA
8.          Output stackEntry as a SLCA
9.          Set all entried of the Keywords array of any stack entry all falses
10.       }
11.       else{//pass keyword witness information to the top entry
12.           for(j = 1 → k)
13.               if(stackEntry,Keywords[j] = true) Stack,top,Keywords[j] = true
14.       }
15.   }
     //add non-matching components of v to stack
16.       for(p < j ≤ v.length) stack.push(v[j],[])
17.               Stack.top, Keywords[i] = true
18. }
19. Check entried of the stack and return any SLCA if exists
```

```
FUNCTION isSLCA(stackEntry)
1. if (stackEntry. Keywords[i] = true for 1 ≤ i ≤ k)
2.      return true
3. else return false
```

FUNCTION getSmallestNode /*returns the node v with the smallest Dewey number from All keyword lists and advances the cursor of the list where v is from. Assume v is an array consisting of its Dewey number components. For example v is 0.1.3 if its Dewey number is 0.1.3*/

For example, when the algorithm processes node 0.1.2, the initial stack is shown in Fig. 6.6f and the stack after processing 0.1.2 is shown in Fig. 6.6g. The longest common prefix of 0.1.2 and the stack (0.1.1.2.0) is 0.1 (Line 4). Thus, the top three entries are popped out (Line 6). When popping out the third entry, the algorithm reports 0.1.1 as an SLCA since its Keywords array contains all *T* (Line 7). Notice that the *T* for "Ben" from the top entry in Fig. 6.6f is used in the decision that the third entry is an SLCA.

6.3.3 Basic Multiway-SLCA Algorithm (BMS)

The algorithm computes the SLCAs iteratively. At each iteration, an anchor node v_m is judiciously selected to compute the match anchored by v_m and its LCA (denoted by α). If α is potentially an SLCA, it is maintained in an intermediate SLCA result list given by $\alpha_1, \ldots, \alpha_n$, $n \geq 1$, where all the LCAs α_n in the list are definite SLCAs except for the most recently computed candidate α_n. To minimize the computation of LCAs that are not SLCAs, it is important to optimize the anchor node selected at each iteration. Initially, step 2 initializes the first candidate SLCA α_1 to be *droot* (the virtual root node of data tree); if α_1 remains as *droot* at the end of the algorithm (step 22), then it means that the SLCA result list is empty. The first anchor node v_m is selected in step 1. Instead of choosing the first node $v_1 \in S_1$ as the anchor (as is done in the BS approach), BMS selects the first node $v_1 \in S_1$, $m \in [1,k]$ that is the "furthest" node among all the first nodes in $S_1, \ldots, S_k$. In doing so, all the nodes in S_1 that precede $u_1 = \text{closet}(v_m,S_1)$ are skipped from consideration as anchor nodes.

```
1. **let** $\mathrm{v_m = last(\{first(S_i)|i{\in}[1,k]\}), where\ v_m{\in}S_m}$
2. Initialize $\mathrm{n = 1; \alpha_1 = d_{root}}$
1. **while**($\mathrm{v_m \neq null}$) **do**
2.   **if**($\mathrm{m \neq 1}$) **then**
3.       $\mathrm{v_1 = closet(v_m,S_1)}$
4.       **if**($\mathrm{v_m <_p v_1}$) **then**
5.           $\mathrm{v_m = v_1}$
6.       **end if**
7.   **end if**
8.   $\mathrm{v_i = closet(v_m,S_i)\ for\ each\ i{\in}[1,k], i \neq m}$
9.   $\mathrm{\alpha = lca(first(v_1,\ldots,v_k),last(v_1,\ldots,v_k))}$
10.   **if** ($\mathrm{\alpha_n \leq_\alpha \alpha}$) **then**
11.       $\mathrm{\alpha_n = \alpha}$
12.   **else if** ($\mathrm{\alpha_n \nless_\alpha \alpha}$) **then**
13.       $\mathrm{n = n + 1; \alpha_n = \alpha}$
14.   **end if**
15.   $\mathrm{v_m = last(\{next(v_m,S_i)|i{\in}[1,k]v_1 \leq_p v_m\})}$
16.   **if**($\mathrm{v_m \neq null}$) and ($\mathrm{\alpha_n \nless_\alpha v_m}$) then
17.       $\mathrm{v_m = last(\{v_m\} \cup \{out(a_n,S_i)|i{\in}[1,k], i \neq m\})}$
18.   **end if**
19. **end while**
20. **if**($\mathrm{\alpha_1 = d_{root}}$) **then return** $\emptyset$
21. **else return** $\mathrm{\{\alpha_1,\ldots,\alpha_n\}}$
```

The correctness of this optimization stems from the fact that for each $v \in S_1$ that precedes $\mathrm{u_1}$, closet(v,S_m) must be v_m; therefore, no SLCAs will be missed out by using v_m as the first anchor node.

Steps 4–9 further optimize the selection of the anchor node to ensure that the total number of candidate SLCAs computed is no more than $|S_1|$. Specifically, if the selected anchor node v_m precedes closet(v_m,S_1), then no SLCAs will be omitted by replacing the anchor node v_m with closet(v_m,S_1). After an anchor node v_m has been chosen, step 10 computes the match anchored by v_m, and step 11 computes the LCA α of this match in terms of only its first and last nodes. Steps 12–16 check whether the newly computed LCA α can be a candidate SLCA and, if so, whether α can be used to eliminate the previous candidate SLCA α_n. Steps 17–20 optimize the selection of the next anchor node by choosing the furthest possible node that maximizes the number of skipped nodes.

6.3.4 Incremental Multiway-SLCA Algorithm (IMS)

The IMS algorithm shares many similarities with the BMS algorithm. The key difference lies in steps 10–17 which determine lca(M) for a match M anchored by a node v_m without actually computing M. The repeat loop enumerates a sequence of matches M_1,M_2, ... to compute first(M) and last(M) and hence lca(M).

```
1. Let $v_m$= last({first($S_i$)|i∈[1,k]}),where $v_m \in S_m$
2. initialize n = 1; $\alpha_1 = d_{root}$
3. While($v_m \neq$ null)do
4.     if(m $\neq$ 1)then
5.         $v_1$= closet($v_m$,$S_1$)
6.     if($v_m <_p v_1$)then
7.         $v_m = v_1$
8.     end if
9.   end if
10.  P = {pred($v_m$,$S_i$)|i∈[1,k],i $\neq$ m}∪{$v_m$}
11.  N = {next($v_m$,$S_i$)|i∈[1,k],next($v_m$,$S_i$) $\neq$ null}
12. initialize $r_{max}$= last(N);r = $v_m$
13. repeat
14.       remove l from P, where l = first(P)
15.       $\alpha$ = lca(l,r)
16.       r = last(r,v) where v∈$N_{s,t}$v = next($v_m$,$S_j$)
17.    unitl(r = null)or($\alpha \not\prec_{\alpha}$r)or(r = $r_{max}$)
18.    if(r = null)or($\alpha \not\prec_{\alpha}$r) then
19.       if($\alpha \not\prec_{\alpha} \alpha$) then
20.       $\alpha_n = \alpha$
21.    else of ($\alpha \not\prec_{\alpha} \alpha_n$) then
22.          = n + 1; $\alpha_n = \alpha$
23.    end if
24.    $v_m$ = last(r,out($\alpha_n$,$S_1$), . . . ,out($\alpha_n$,$S_k$))
25.    else
26.    $v_m$ = r
27.    end if
28.    end while
29.    if($\alpha_1 = d_{root}$) then return Ø else return {$\alpha_1, \ldots, \alpha_n$}
```

In the ith iteration of the repeat loop (steps 13–17), first(M_i) is computed in step 14 as 1, and α_i is computed in step 15 (with r representing last(M_i)). Step 16 determines last($M_i + 1$) for the next iteration. The search for M is terminated when any one of the three conditions in step 17 is met. Firstly, if r = null, then it means that next(v_m,S_j = null and there are no further matches in the data; therefore, α = lca(M) and the next anchor node is correctly set to null by step 24. Secondly, if α a r, then α = lca(M) and Lemma 3.5 is applied to optimize the selection of the next anchor node in step 24. Finally, if r = last(N), then it means that all the matches

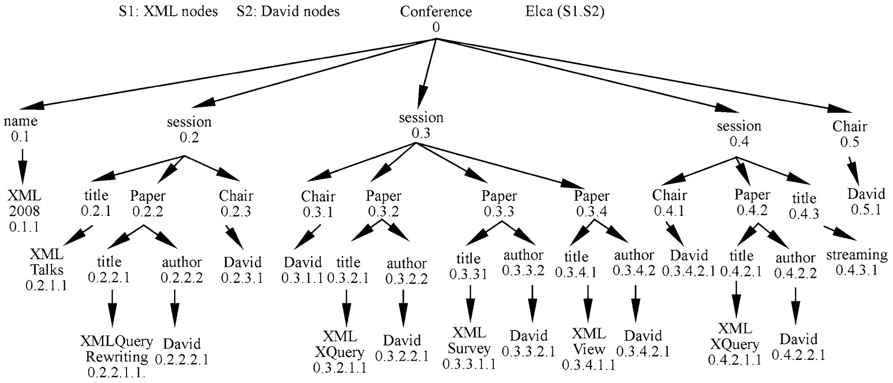


Fig. 6.7 ISA

enumerated within the repeat loop must have last(M_i) = last(N) as well; therefore, M must correspond to the very last match in the enumeration. To quickly skip to this last match without continuing with the enumeration, step 26 applies Lemma 3.1 to update the next anchor node to be r.

6.3.5 Indexed Stack Algorithm

The set elca_can (S_1, $S_2, \ldots, S_k$), whose members are called ELCA_CAN nodes (S_1 among $S_2, \ldots, S_k$):

$$\text{elca}_{\text{can}(S_1,S_2,\ldots,S_k)} = \cup_{v_i \in S_i} \text{slca}\left(\{v_1\}, S_2, \ldots, S_k\right)$$

For example, in Fig. 6.7 elca_can (S_1;S_2) = [0, 0.2, 0.2.2, 0.3, 0.3.2, 0.3.3, 0.3.4, 0.4.2].

$\forall i \in [1, \ldots, k]$ elca(S_1,$S_2, \ldots,$ S_k) = elca_can(S_i;S_1,$S_2, \ldots,$ S_k) of particular importance is the instantiation of the above property for $i = 1$ (i.e., elca(S_1,$S_2, \ldots,$ S_k) $\subseteq$ elca_can(S_i; S_1, $S_2, \ldots,$ S_k) since elca_can(S_1; S_1, $S_2, \ldots,$ S_k) has the most efficient computation (recall S_1 is the shortest inverted list). For presentation brevity, we define elca_can(v) for $v \in S_1$ to be the node l, where{l = elca_can($\{v\}$; $S_2, \ldots,$ S_k) = slca($\{v\}$, $S_2, \ldots,$ S_k). The node elca_can(v) is called the exclusive lowest common ancestor candidate or ELCA_CAN of v (in sets of $S_2, \ldots, S_k$). Note that each node in lca($\{v\}$, $S_2, \ldots,$ S_k) is either an ancestor node of v or v itself and elca_can(v) is the lowest among all nodes in lca($\{v\}$, $S_2, \ldots,$ S_k). For instance, consider S_1 and S_2 in Fig. 6.7.

The following shows elca_can(v) for each v in S_1: elca_can(0.1.1) = 0, elca_can(0.2.1.1) = 0.2, elca_can(0.2.2.1.1) = 0.2.2, elca_can(0.3.2.1.1) = 0.3.2, elca_can(0.3.3.1.1) = 0.3.3, elca_can(0.3.4.1.1) = 0.3.4, elca_can(0.3.5.1) = 0.3, and elca_can(0.4.2.1.1) = 0.4.2. Let child_elcacan(v) be the set of children of v that contain all keyword instances. Equivalently child_elcacan(v) is the set of child nodes u of v such that either u or one of u's descendant nodes is an ELCA_CAN node, that is,

$$\text{child}_{\text{elcacan}(v)} = \{u \mid u \in \text{child}(v) \wedge \exists x(u \leq x \wedge x \in \text{elca\_can}(S_1, S_2, \ldots, S_k))\}$$

where child(v) is the set of child nodes of v. Given the list ch which is the list of nodes in child_elcacan(v) sorted by id, the function isELCA(v, ch) returns true if v is an ELCA node by applying the operations described in the previous paragraph. As an example, consider the query "XML David" and the inverted lists S_1 and S_2 in Fig. 6.1. child_elcacan(0) = [0.2,0.3,0.4]. We will see how isELCA(0, [0.2,0.3,0.4]) works and returns true. In this example, the number of keywords is two. First the function isELCA searches and finds the existence of anELCA witness node (i.e., the node 0.1.1) for 0.2 in the first keyword list S_1 in the subtree rooted under 0 to the left of the path from 0 to 0.2(0.2 is the first child ELCA_CAN node of 0). Then the function searches the existences of an ELCA witness node in the second keyword list S_2 for 0 in the forest to the left of the path from 0 to 0.2, in the forest between the path from 0 to 0.2 and the path from 0 to 0.3, in the forest between the path from 0 to 0.3 and the path from 0 to 0.4, and in the forest to the right of the path from 0 to 0.4. All of the above searches fail except that the last search successfully finds a witness node(0.5.1) for 0.2 in S_2. Therefore, isELCA(0,[0.2,0.3,0.4]) returns true.

The Indexed Stack algorithm needs not keep all ELCA_CAN nodes in memory; it uses a stack whose depth is bounded by the depth of the tree. At any time during the computation, any node in the stack is a child or descendant node of the node below it (if present) in the stack. Therefore, the nodes from the top to the bottom of the stack at any time are from a single path in the input tree.

We go through every node v_1 in S_1 in order, compute $\text{elca}_{v1} = \text{elca\_can}(v_1)$, and create a stack entry stackEntry consisting of elca_{v1}. If the stack is empty, we simply push stackEntry to the stack to determine later whether elca_can_{v1} is an ELCA node or not. If the stack is not empty, what the algorithm does depends on the relationship between stackEntry and the top entry in the stack. The algorithm either discards stackEntry or pushes stackEntry to the stack (with or without first popping out some stack entries). The algorithm does not need to look at any other non-top entry in the stack at any time and only determines whether an ELCA_CAN node is an ELCA node at the time when a stack entry is popped out.

The challenging issue that the Indexed Stack algorithm has to deal with is illustrated with the running example "XML David." Before we compute elca_can(0.3.5.1) = 0.3, we have already computed 0.3.2, 0.3.3, 0.3.4 as ELCA_CAN nodes which are the child ELCA_CAN nodes of 0.3. We have to store these three ELCA_CAN nodes in order to determine whether 0.3 is an ELCA node or not before we see 0.3 in the processing. Note that if the node 0.3.1.1 was not in the tree in Fig. 6.7, we would still see 0.3 in the processing as an ELCA_CAN node and still see 0.3 after 0.3.2, 0.3.3, and 0.3.4 in the processing, but then 0.3 would not be an ELCA node, which could be determined only if we have kept the information that

0.3.2, 0.3.3, and 0.3.4 are ELCA_CAN nodes until we see 0.3 and know that 0.3 would not have any child or descendant ELCA_CAN nodes in the processing later after we see 0.3. It is possible that we would not see 0.3 at all in the processing (i.e., if the node 0.3.5.1 was not in the tree, 0.3 would not be an ELCA_CAN node), in which case we still need to keep 0.3.2, 0.3.3, and 0.3.4 until the point we are sure that those nodes cannot be child or descendant of any other ELCA_CAN nodes. Based on some key tree properties, the Indexed Stack algorithm knows when an ELCA_CAN node needs to be stored in the stack and when it can be discarded.

6.3.6 Stack-Based Query Refinement Algorithm

A user keyword query may often be an imperfect description of their real information need. Even when the information need is well described, a search engine may not be able to return the results matching the query as stated possibly due to term mismatch, keyword ambiguity, spelling error, etc. For example, a user issues a query "database, proceeding" on XML data in Fig. 6.3 (in meaningful SLCA semantics), but all the results in XML data use a mismatched term "inproceedings" or "article," which causes an empty result to return.

The question then becomes whether we can offer a solution during search which automatically refines queries, in order to better represent users' search needs and help users more easily find the relevant information. In the scenario of common Web search, there are often a plethora of documents to match query keywords, and query refinement is carried out to make the query result more specific. In contrast, the original query has no meaningful matching result over XML database and needs to be refined to a closely related query that has meaningful matching results.

STACK-BASED QUERY REFINEMENT ALGORITHM

```
input: Q = {k1,k2,...,kn}, refinement ruleset R, XML database D
output: result = {(RQmin,SLCA(RQmin))}
Boolean needRefine = true;
minCost = ∞, RQmin =; result← ; stack← ;
Let KS = getNewKeywords(Q) + Q;
{S_1,S_2,...,S_m}←getInvertedLists(KS);
while(!end(Si) or stack is not empty) do
vs = getSmallestNode();                 /*1 ≤ s ≤ |KS|*/
p = lca(vs,stack);
while(stack.size>p.length) do
entry e = stack.pop();KS←all keywords of witness T in e;
if(isSLCA(e,Q)) then
result.add(Q,e); needRefine = false;
 reset all entries in keywords to false;
if(needRefine) then
  < RQ,costRQ > = getRefineCost(Q,KS);
```

```
17. if(costRQ<minCost)∩(isSLCA(e,RQ)) then
18.   RQmin = RQ;minCost = costRQ;
19.   result.clear();result.add(RQ,e);
20.   reset terms only in RQ to false for all entries;
21.   pass the rest witness T to current top entry of stack;
22. for(j = p.length to vs.length) do
23.   stack.push(vs[p],vs[length]);
```

For each entry e popped from stack, KS is reset to be a sequence of keywords contained by *e* (Line 9). Then, we check whether the top entry e is an SLCA result of original query Q by Definition 3.1. If so, Q does not need to be refined, and all entries of keywords of any stack entry are set to be false (Lines 10–12); otherwise, getRefineCost is invoked to get the RQ($\subseteq$ KS) with minimal refinement cost for e (Line 14); Rqmin and minCost are updated accordingly (Lines 16–17). A critical difference with[25] is at the updating of witness information in *e*'s parent entry: only the keywords that are unique to RQ is set to false, while those that are in common with other RQ candidates or Q are kept as true (Lines 18–19). Lastly, a stack entry is created and pushed into stack for each Dewey component that is not part of the common prefix (Lines 20–21).

6.4 XML Keyword Search Ranking Strategy

Inspired by the great success of information retrieval (IR) style keyword search on the Web, keyword search on XML has emerged recently. The difference between text database and XML database results in three new challenges: (1) Identify the user search intention, that is, identify the XML node types that user wants to search for and search via. (2) Resolve keyword ambiguity problems: a keyword can appear as both a tag name and a text value of some node; a keyword can appear as the text values of different XML node types and carry different meanings. (3) As the search results are subtrees of the XML document, new scoring function is needed to estimate its relevance to a given query. However, existing methods cannot resolve these challenges, thus return low result quality in terms of query relevance.

Effectiveness in terms of result relevance is the most crucial part in keyword search, which can be summarized as the following three issues in XML field:

Issue 6.1 It should be able to effectively identify the type of target node(s) that a keyword query intends to search for. We call such target node as a search for node.

Issue 6.2 It should be able to effectively infer the types of condition nodes that a keyword query intends to search via. We call such condition nodes as *search via nod*es.

Issue 6.3 It should be able to rank each query result in consideration of the above two issues.

The first two issues address the search intention problem, while the third one addresses the relevance-based ranking problem w.r.t. the search intention. Regarding Issue 6.1 and Issue 6.2, XML keyword queries usually have ambiguities in interpreting the search for node(s) and search via node(s), due to two reasons below:

Ambiguity 6.1 A keyword can appear both as an XML tag name and as a text value of some other nodes.

Ambiguity 6.2 A keyword can appear as the text values of different types of XML nodes and carry different meanings.

*6.4.1 TF*IDF Cosine Similarity*

TF*IDF (Term Frequency*Inverse Document Frequency) similarity [SM84] is one of the most widely used approaches to measure the relevance of keywords and document in keyword search over flat documents. We first review its basic idea, then address its limitations for keyword search in XML. The main idea of TF*IDF is summarized in the following three rules:

Rule 6.1 A keyword appearing in many documents should not be regarded as being more important than a keyword appearing in a few.

Rule 6.2 A document with more occurrences of a query keyword should not be regarded as being less important for that keyword than a document that has less.

Rule 6.3 A normalization factor is needed to balance between long and short documents, as Rule 2 discriminates against short documents which may have less chance to contain more occurrences of keywords.

To combine the intuitions in the above three rules, the TF*IDF similarity is designed:

$$\rho(q,d) = \frac{\sum_{k \in q \cap d} W_{q,k} \cdot W_{d,k}}{W_q \cdot W_d} \tag{6.3}$$

where q represents a query, d represents a flat document, and k is a keyword appearing in both q and d. A larger value of $\rho(q, d)$ indicates q and d are more relevant to each other. $W_{q,k}$ and $W_{d,k}$ represent the weights of k in query q and document d, respectively, while W_q and W_d are the weights of query q and document d. Among several ways to express $W_{q,k}$, $W_{d,k}$, W_q, and W_d, the following are the conventional formulae:

$$\mathrm{W}_{q,k} = \ln(N/f_k + 1) \tag{6.4}$$

$$W_{d,k} = 1 + \ln(f_{d,k}) \qquad (6.5)$$

$$W_q = \sqrt{\sum\nolimits_{k \in q} W_{q,k}^2} \qquad (6.6)$$

$$W_d = \sqrt{\sum\nolimits_{k \in d} W_{q,d}^2} \qquad (6.7)$$

where N is the total number of documents and document frequency f_k in Formula 6.4 is the number of documents containing keyword k. Term frequency $f_{d,k}$ in Formula 6.5 is the number of occurrences of k in document d. $W_{q,k}$ is monotonical decreasing w.r.t. f_k (Inverse Document Frequency) to reflect Rule 6.1; while $W_{d,k}$ is monotonical increasing w.r.t. $f_{d,k}$ (term frequency) to reflect Rule 6.2. The logarithms used in Formulae 6.4 and 6.5 are designed to normalize the raw document frequency f_k and raw term frequency $f_{d,k}$. Finally, W_q and W_d are increasing w.r.t. the size of q and d, playing the role of normalization factors to reflect Rule 6.3.

6.4.2 Data Model

Node Type: The type of a node n in an XML document is the prefix path from root to n. Two nodes are of same node type if they share the same prefix path [BCL+09].

Value Node: The text values contained in the leaf node of XML data (i.e., #PCDATA) is defined as a value node.

Structural Node: An XML node labeled with a tag name is called a structural node. A structural node that contains other structural nodes as its children is called an internal node.

Single-Valued Type: A given node t is of single-valued type if each node of type t has at most one occurrence within its parent node.

Multivalued Type: A given node t is of multivalued type if each node of type t has more than one occurrence within its parent node.

Grouping Type: An internal node t is defined as a grouping type if each node of type t contains child nodes of only one multivalued type.

6.4.3 XML TF&DF

Intuition 6.1 The more XML nodes of a certain type T (and their subtrees) contain a query keyword k in either their text values or tag names, the more intuitive that nodes of type T are more closely related to the query w.r.t. keyword k.

XML DF $f^T{}_k$(Document Frequency): The number of T-typed nodes that contain keyword k in their subtrees in XML database.

XML TF $f_{a,k}$(Term Frequency): The number of occurrences of a keyword k in a given value node a in XML database.

6.4.4 Inferring the Node Type to Search For

The desired node type to search for is the first issue that a search engine needs to address in order to retrieve the relevant answers. Given a keyword query q, a node type T is considered as the desired node to search for only if the following three guidelines hold:

Guideline 6.1 T is intuitively related to every query keyword in q, that is, for each keyword k, there should be some (if not many) T-typed nodes containing k in their subtrees.

Guideline 6.2 XML nodes of type T should be informative enough to contain enough relevant information.

Guideline 6.3 XML nodes of type T should not be overwhelming to contain too much irrelevant information.

By incorporating the above guidelines, we define $C_{\text{for}}(T,q)$, which is the confidence of a node type T to be the desired search for node type w.r.t. a given keyword query q as follows:

$$C_{\text{for}}(T,q) = \log_{\varepsilon}\left(1 + \prod_{k \in q} f_k^T\right) * r^{\text{depth}(T)} \tag{6.8}$$

where k represents a keyword in query q, $f^T{}_k$ is the number of T-typed nodes that contain k as either values or tag names in their subtrees, r is some reduction factor with range(0,1] and normally chosen to be 0.8, and depth(T) represents the depth of T-typed nodes in document.

6.4.5 Inferring the Node Types to Search Via

Unlike the search for case which requires a node type to be related to all keywords, it is enough for a node type to have high confidence as the desired search via node if it is closely related to some (not necessarily all) keywords, because a query may intend to search via more than one node type. For example, we can search for customer(s) named "Smith" and interested in "fashion" with query "name smith interest fashion." In this case, the system should be able to infer with high confidence that name and interest are the node types to search via, even if keyword "interest" is probably not related to name nodes. Therefore, we define $C_{\text{via}}(T,q)$, which is the confidence of a node type T to be a desired type to search via below:

$$C_{\text{via}}(T,q) = \log_{\varepsilon}\left(1 + \sum_{k \in q} f_k^T\right) \tag{6.9}$$

where variables k, q and T have the same meaning as those in Formula 6.8.

6.4.6 Capturing Keyword Co-occurrence

Given a keyword query q and a certain value node v, if there are two keywords kt and k in q, such that kt matches the type of an ancestor node of v and k matches a keyword in v, then we define the following distances:

In-Query Distance(IQD): The In-Query Distance Dist*q*(q,v,kt,k) between keyword k and node type kt in query q with respect to a value node v is defined as the position distance between kt and k in q if kt appears before k in q; otherwise, Dist*q*(q,v,kt,k) $= \infty$.

Structural Distance(SD): The structural Distance Dist*s*(q,v,kt,k) between kt and k w.r.t. a value node v is defined as the depth distance between v and the nearest kt-typed ancestor node of v in XML document.

Value-Type Distance(VTD): The Value-Type Distance Dist(q,v,kt,k) between kt and k w.r.t. a value node v is defined as max(Distq(q,v,kt,k), Dists(q,v,kt,k)).

In general, the smaller the value of Dist(q,v,kt,k) is, it is more likely that q intends to search via the node v with value matching keyword k. Therefore, we define the confidence of a value node v as the node to search via w.r.t. a keyword k appearing in both query q and v as follows:

$$C_{\text{via}}(q,v,k) = 1 + \sum\nolimits_{kt \in q \cap \text{ancType}(v)} \frac{1}{\text{Dist}(q,v,kt,k)} \tag{6.10}$$

6.4.7 Relevance-Oriented Ranking

6.4.7.1 Principles of Keyword Search in XML

Principle 6.1 When searching for D-typed nodes via a single-valued type V, ideally only the values and structures nested in V-typed nodes can affect the relevance, regardless of the size of other typed nodes nested in D-typed nodes. However, TF*IDF similarity in IR normalizes the relevance score of each document w.r.t. its size.

Principle 6.2 When searching for nodes of type *D* via a multivalued type V', the relevance of a *D*-typed node which contains a query-relevant V'-typed node should not be affected (i.e., normalized) too much by other query-irrelevant V'-typed nodes.

Principle 6.3 The order of keywords in a query is important to indicate the search intention. Incorporate the search via confidence C_{via} we defined before.

6.4.7.2 XML TF*IDF Similarity

We propose a recursive Formula 6.11, which captures XML's hierarchical structure, to compute XML TF*IDF similarity [BLL10] between an XML node of the desired type to search for and a keyword query. It first (base case) computes the similarities between leaf nodes l of XML document and the query, then (recursive case) it recursively computes the similarities between internal nodes n and the query, based on the similarity value of each child c of n and the confidence of c as the node type to search via, until we get the similarities of search for nodes:

$$\rho_s(q,a) = \begin{cases} \dfrac{\sum_{k \in q \cap a} W_{q,k}^{Ta} * W_{a,k}}{W_q^{Ta} * W_a} & \text{(a) a is value node (base case)} \\ \dfrac{\sum_{c \in chd(n)} \rho(q,c) * C_{via}(T_{c,q})}{W_n^q} & \text{(b) a is internal node (recursive case)} \end{cases} \tag{6.11}$$

6.4.7.3 Base Case

Similarity between value nodes a and q. Apply original TF*IDF directly since a contains keywords only without any structure.

In the base case for XML leaf nodes, each k represents a keyword appearing in both query q and value node a; T_a is the type of a's parent node; $\mathrm{W}_{q,k}^{T_a}$ represents the weight of keyword k in q w.r.t. node type T_a. $W_{a,k}$ represents the weight of k in leaf node a; $\mathrm{W}_q^{T_a}$ represents the weight of q w.r.t. node type T_a; and W_a represents the weight of a. Following the conventions of original TF*IDF,

$$\mathrm{W}_{q,k}^{T_a} = C_{via}(q,a,k) * \ln\left(1 + N_{T_a} \Big/ \left(1 + f_k^{T_a}\right)\right) \tag{6.12}$$

$$\mathrm{W}_{a,k} = 1 + \ln(f_{a,k}) \tag{6.13}$$

$$\mathrm{W}_q^{T_a} = \sqrt{\sum\nolimits_{k \in q} (W_{q,k}^{T_a})^2} \tag{6.14}$$

$$\mathrm{W}_a = \sqrt{\sum\nolimits_{k \in a} W_{a,k}^2} \tag{6.15}$$

6.4.7.4 Recursive Case

Similarity between structural nodes n and q. Based on similarities of its children c and the confidence level of c as the node type to search via.

Intuition 6.2 An internal node n is relevant to q, if n has a child c such that the type of c has high confidence to be a search via node w.r.t. q (i.e., large $C_{\text{via}}(T_c,q)$), and c is highly relevant to q (i.e., large sim(q, c)).

Intuition 6.3 An internal node n is more relevant to q if n has more query-relevant children when all others being equal.

In the recursive case of Formula 6.11, c represents one childnode of a; T_c is the node type of c; $C_{\text{via}}(T_c,q)$ is the W_q^{Ta} confidence of T_c to be a search via node type presented in Formula 6.9; $\rho_s(q,c)$ represents the similarity between node c and query q which is computed recursively; W_a^q is the overall weight of a for the given query q.

Next, we explain the similarity design of an internal node a in Formula 6.11: we first get a weighted sum of the similarity values of all its children, where the weight of each child c is the confidence of c to be a search via node w.r.t. query q. This weighted sum is exactly the numerator of Formula 6.11, which also follows Intuition 6.2 and 6.3 mentioned above. Besides, since Intuition 6.3 usually favors internal nodes with more children, we need to normalize the relevance of a to q. That naturally leads to the use of W_a^q (Formula 6.16) as the denominator.

6.4.7.5 Normalization Factor Design

$$W_a^q = \begin{cases} \dfrac{\sqrt{\sum_{c \in chd(a)} (C_{\text{via}}(T_c, q) * B + DW(C))^2}}{\sqrt{\sum_{T \in chdType(Ta)} C_{\text{via}}(T, q)^2}} & \text{(a) } a \text{ is grouping node} \\ & \text{(b) Otherwise} \end{cases} \tag{6.16}$$

Formula 6.16 presents the design of W_a^q, which is used as a normalization factor in the recursive case of XML TF*IDF similarity formula. W_a^q is designed based on Principle 6.1 and Principle 6.2 pointed out in section 6.4.7.1.

Formula 6.16(a) presents the case that internal node a is a grouping node; then for each child c of a (i.e., $c \in \text{chd}(a)$), B is considered as a Boolean flag: $B = 1$ if $\rho_s(q, c) > 0$ and $B = 0$ otherwise; DW(c) is a small value as the default weight of c which we choose $\text{DW}(c) = 1/\log_{\varepsilon}(e{-}1 + |\text{chd}(a)|)$.

If $B = 1$, where $|\text{chd}(a)|$ is the number of children of a, so that W_a^q for grouping node a grows with the number of query-irrelevant child nodes, but grows very slowly to reflect Principle 6.2. Note DW(c) is usually insignificant as compared to $C_{\text{via}}(T_c, q)$.

6.5 Summary

Traditional query processing approaches on relational and XML databases are constrained by the query constructs imposed by the language such as SQL and XQuery. Firstly, the query language themselves are hard to comprehend for non-database users. Secondly, these query languages require the queries to be posed against the underlying, sometimes complex, database schemas. These traditional querying methods are powerful but unfriendly for day-to-day users. Keyword search is proposed as an alternative means of querying the database, which is simple and yet familiar to most internet users as it only requires the input of some keywords. Keyword search is well suited to XML trees as well. It allows users to find the information they are interested in without having to learn a complex query language or needing prior knowledge of the structure of the underlying data.

In this chapter, we present existing works on XML Keyword Search. We present XML keyword search semantics such as SLCA, VLCA [HKP+06], MLCEA [LYJ04], which is useful and meaningful for keyword search; based on some of the semantics, we show XML keyword search algorithms [SCG07, ZBL+09, LC07].

In addition, we introduce the XML keyword search ranking strategy [BCL+09]. We show an IR-style approach which basically utilizes the statistics of underlying XML data to address these challenges. We first propose specific guidelines that a search engine should meet in both search intention identification and relevance-oriented ranking for search results. Then based on these guidelines, we design formulae to identify the search for nodes and search via nodes of a query and present a novel XML TF*IDF ranking strategy to rank the individual matches of all possible search intentions.

References

[BCL+09] Bao, Z., Chen, B., Ling, T.W., Lu, J.: Effective xml keyword search with relevance oriented ranking. In: ICDE, Shanghai, pp. 517–528 (2009)

[BLL10] Bao, Z., Lu, J., Ling, T.W., Chen, B.: Towards an effective XML keyword search. Knowl. Data Eng. (TKDE) **22**(8), 1077–1092 (2010)

[CMK03] Cohen, S., Mamou, J., Kanza, Y., Sagiv, Y.: XSEarch: A semantic search engine for XML. In: VLDB, pp. 45−56 (2003)

[GSB+03] Guo, L., Shao, F., Botev C., Shanmugasundaram, J.: XRANK: Ranked keyword search over XML documents. In: SIGMOD, San Diego, pp. 16−27 (2003)

[HKP+06] Hristidis, V., Koundas, N., Papakonstantinou, Y., Srivastava, D.: Keyword proximity search in XML trees. IEEE Trans. Knowl. Data Eng. (TKDE) **18**(4), 525–539 (2006)

[HXZ+08] Huang, J., Xu, J., Zhou, J., Meng, X.: MLCEA: An entity based semantics for XML keyword search. J Comput. Res. Dev. **45**(Suppl), 372–377 (2008). 10 (NDBC2008 Guilin)

[LC07] Liu, Z., Chen, Y.: Identifying meaningful return information for XML keyword search. In: SIGMOD, Beijing, pp. 329−340 (2007)

[LD07] Ling, T.W., Dobbie, G.: Using semantics in XML data management. In: WSPC, 31 March 2007

[LFW+07] Li, G., Feng, J., Wang, J., Zhou, L.: Effective keyword search for valuable LCAs over XML documents. In: CIKM, Lisbon, pp. 31–40 (2007)

[LYJ04] Li, Y., Yu, C., Jagadish, H.V.: Schema-free XQuery. In: VLDB, Toronto (2004)

[SCG07] Sun C., Chan C.Y., Goenka, A.K.: Multiway slca-based keyword search in xml data. In: WWW, Banff, pp. 1043–1052 (2007)

[SKW01] Schmidt, A., Kersten, M.L., Windhouwer, M.: Querying XML documents made easy: Nearest concept queries. In: ICDE, Heidelberg (2001)

[SM84] Salton, G., Mcgill, M.: Introduction to Modern Information Retrieval. McGraw-Hill Book Company, New York (1984)

[SSY+09] Supasitthimethee, U., Shimizu, T., Yoshikawa, M., Porkaew, K.: XSemantic: An extension of LCA based XML semantic search. IEICE Trans. (IEICET) **92-D**(5), 1079–1092 (2009)

[V09] Vesper, V.: Let's do Dewey. http://www.mtsu.edu/vvesper/dewey.html

[XP05] Xu, Y., Papakonstantinou, Y.: Efficient keyword search for smallest LCAs in XML databases. In: SIGMOD, Baltimore, pp. 537–538 (2005)

[ZBL+09] Zhou, J., Bao, Z., Ling, T.W., Meng, X.: MCN: A new semantics towards effective XML keyword search. In: DASFAA, Brisbane, pp. 511–526 (2009)

Chapter 7
XML Keyword Pattern Refinement

Abstract In XML keyword search, users' queries usually contain irrelevant or mismatched terms, typos, etc., which may easily lead to empty or meaningless results. In this chapter, we first introduce the problem of content-aware XML keyword query refinement, aiming to integrate the job of finding the desired refined queries and generating their matching results as a single problem. Furthermore, a statistics-based query ranking model, which takes into account of both keyword dependencies and the relevance, is proposed. The ranking model evaluates the quality of a refined query, which captures the morphological/semantic similarity between the original query and refined queries and the dependency of keywords of the refined queries over the XML data. In addition, two adaptive query refinement algorithms are also proposed. Finally, we experimentally demonstrate the efficiency and effectiveness of the approach presented in this chapter.

Keywords XML keyword query refinement • Keyword ambiguity • SLCA • Node type • LCA • MSLCA • Refinement operations • Similarity score • Dependency score • Short-list eager

7.1 Introducing XML Keyword Pattern Refinement

The great success of Web search engine has made keyword search a popular way for users to access information. As XML is becoming a standard in data exchange, there is a growing research interest to support keyword search on XML [GSBS03, XP05, LYJ04, LC07, BLC+09, XP08]. Among those proposals, SLCA (smallest lowest common ancestor) is widely adopted [XP05], where each SLCA result contains all query keywords but has no descendant whose subtree also contains all keywords. Unfortunately, all of them enforce a conjunctive search semantics, assuming each keyword in the query is intended as part of it. However, a user query may often be an imperfect description of their real information need, which may easily cause an empty matching result. Even when the information need is well described, a search

J. Lu, *An Introduction to XML Query Processing and Keyword Search*,
DOI 10.1007/978-3-642-34555-5_7,

Table 7.1 Query before and after refinement

Initial query	Refined query
Q_1: IR, 2003, Mike	RQ$_1$: Information retrieval, 2003, Mike
Q_2: Mike, publication	RQ$_2$: Mike, publications
Q_3: keyword, paper	RQ$_3$: keyword, inproceedings
Q_4:XML, John, 2003	RQ$_4$: XML, John
Q_5: mechin, learn	Q_5: machine, learning
Q_6: hobby, news, paper	Q_6: hobby, newspaper
Q_7: on, line, data, base	Q_7: online, database

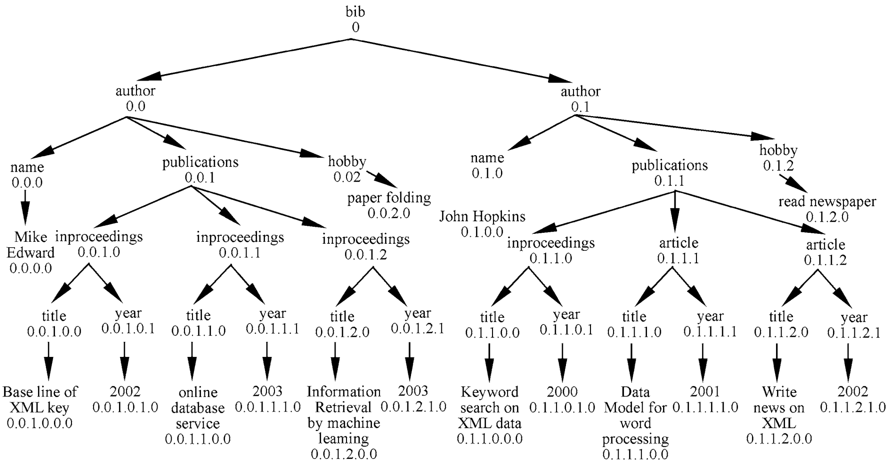


Fig. 7.1 Example XML document

engine may not be able to return the results matching the query as stated possibly due to term mismatch, keyword ambiguity or unintentional spelling error, etc., as shown in Example 7.1.

Example 7.1 Consider $Q_3 = \{$keyword, paper$\}$ (in Table 7.1) issued on a bibliographic XML document in Fig. 7.1, intending to find the paper about keyword search. By SLCA, the whole XML tree rooted at bib:0 is returned (as lca(0.0.2.0, 0.1.1.0.0.0)=0) due to the occurrence of the ambiguous "paper" at node 0.0.2.0, which contains "paper folding" as a hobby of an author. However, such result is meaningless and irrelevant to user's search intention; moreover, the result is overwhelmingly large for user to consume.

Now, the question becomes whether we can offer a solution during search, which peruses the content of XML data being queried and refines the queries that have no meaningful matching result, in order to better represent users' search needs and help users more easily find the relevant information, without any initial result retrieval or any intervention on user part. This is called content-aware XML keyword query refinement as addressed in this chapter. In particular, there are four critical issues to be addressed.

Issue 7.1 It should be able to adaptively and judiciously decide whether the initial query Q needs to be refined during the processing of Q, without an initial result retrieval.

Issue 7.2 It should find a list of RQ candidates, where each RQ is aware of the contents of the corresponding query answers and assured to have meaningful matching results over the XML data.

Issue 7.3 (Effectiveness) It should be able to provide a query ranking model that closely relate the query to the XML data being queried in evaluating the quality of the RQ candidates found in Issue 7.2.

Issue 7.4 (Efficiency) In addressing the above three issues especially Issue 7.2, it should be able to scan the corresponding keyword inverted lists as few times as possible (optimally only once).

Regarding Issues 7.1 and 7.2, the first challenge is to define what a meaningful query result should be for an XML keyword query, as it will be used to judge whether a query needs to be refined. Compared to the traditional IR-style keyword search whose search target is usually the flat document, the search target of an XML keyword query is usually implicit or unknown [BLC+09], which makes the problem more difficult to solve. The second challenge is that it is unknown whether the refinement is required or not before processing the initial query. A brute force approach needs to submit a query for an initial result retrieval, before deciding whether the refinement should be used [JRMG06]. However, it is a prohibitively expensive operation to answer one query by evaluating several potential RQs, as it has to scan the related keyword inverted lists multiple times, which defeats the primary efficiency goal as requested in Issue 7.4.

Regarding Issue 7.3, no previous work has touched on building a content-aware query ranking model to evaluate the quality of a refined query in a statistics-based way in XML keyword search yet. For example, consider a query $Q = \{\text{database, publication}\}$ issued on Fig. 7.1, at least two candidates $\text{RQ}_1 = \{\text{database, article}\}$ and $\text{RQ}_2 = \{\text{database, inproceedings}\}$ seem to be of equal rank as both are the synonym of "publication," but to know which one has the best match, w.r.t Q needs the exploration of more content-aware ranking factors, such as the keyword dependency in the XML data.

At first glance people may think there is no big difference for query refinement between Web search and XML search and consider extending the methods for Web search to XML. But when one actually attempts to implement such a facility, one is faced with myriad options and difficult decisions every step of the way.

Here, we would like to discuss the difference of query refinement between Web search and XML keyword search and analyze why it is unrealistic or difficult to extend the existing approaches designed for Web search to XML keyword search.

First, despite the limitations of existing IR methods (designed for Web search), a unique challenge for XML keyword search that limits the extension of IR methods is as follows: compared to Web search whose search target is flat document with no exception, an XML keyword search engine needs to identify the target node of an XML keyword query which is usually implicit or unknown [BLC+09], and it is becoming even more challenging for queries that do have no matching result and need to be refined.

Second, the primary purpose of the refinement is different in these two scenarios. In Web search, there are often a plethora of documents to match (part of) the query keywords, and the result is computed in an information retrieval way by adopting a scoring model to measure the similarity between the result and the query, which does not enforce each keyword to appear in the result. The query refinement is carried out to make the query result more specific. However, it may not be true in XML field. In XML keyword search, we usually search on a particular large XML document where only the most relevant fragments of the document are expected to return. Besides, the widely adopted conjunctive search semantics (which enforces at least one occurrence of each query keyword in the result such as LCA [GSBS03], SLCA [XP05]) usually implies few results. Therefore, to a certain extent, we aim to increase the query coverage for XML keyword query refinement.

Lastly, most refinement techniques designed for Web search adopt a *machine learning* way such as mining from user's search history to do the query refinement, which requires a thesaurus of user log data (which can only come from a widely used commercial/academic search engine) for model training. Unfortunately, the lack of such widely used commercial XML keyword search engine (with abundant user searching activity) prevents us from acquiring enough user log data for model training. Furthermore, the information in an XML document is usually domain specific, and building the training dataset which is domain specific is still a problem unaddressed yet.

Therefore, query refinement in XML keyword search is not just a trivial extension of its counterpart in Web search, which motivates us to start this work.

In order to address the above challenges, we integrate the job of looking for the desired RQs and finding the matching results of such RQs as a single problem, namely, as the content-aware solution. This integration is important, because from user's perspective, when judging the quality of a RQ, her concern is on checking the results of RQ over the XML data, rather than judging from the literal meaning of RQ itself, as even a good refinement may not have any meaningful matching result in the XML data being queried. In this chapter, the major contributions are summarized below.

1. We formally define the problem of keyword query refinement in XML keyword search, propose an enhanced notion of the widely adopted SLCA matching semantics to judge whether a query has meaningful matching result, and define four typical refinement operations in a rule-based way.
2. We build a query ranking model to evaluate the quality of RQ candidates from a statistical perspective based on tree structural data, by considering the semantical and morphological similarity between RQ and Q, together with the dependency of keywords of RQ in XML data.
3. We design a dynamic programming solution to efficiently find the optimal RQ.

4. We propose two solutions to achieve an efficient XML keyword query refinement and results generation: partition-based approach and short-list eager approach, both of which are orthogonal to any existing SLCA computation methods. Moreover, the partition-based approach needs only one-time scan of the corresponding keyword inverted lists.
5. We conduct extensive experiments to show the efficiency and effectiveness of our refinement framework, by using the real-life datasets and real-life user queries.

7.2 Related Work

Sponsored search [FP05], which aims to match enormous user queries to a much smaller corpus of advertising lists, is an application scenario of our refinement solution due to two reasons. First, the objective of sponsored search is to match enormous number of user queries to a much smaller advertising list. Even if a good refinement is found for the initial query, it may not necessarily have a matching result in the advertiser's listing. It is analogous to the objective of our content-aware XML keyword query refinement, as both aim to find the refinements that guarantee to have the matching results at last. Second, XML is now gradually becoming a standard template for publishing the advertising data in the Web by many commercial search engine companies [XML].

Regarding the keyword search on XML data, earlier works model the XML data as a labeled tree and focus on how to find the most relevant data fragments for a keyword query [GSBS03, LC08, LYJ04, XP05, XP08]. In particular, LCA (lowest common ancestor) [GSBS03] is first proposed to find XML nodes, each of which is the lowest common ancestor containing all query keywords in its subtree. Schema-Free [LYJ04] incorporates a constrained LCA search into XQuery. Subsequently, SLCA (smallest LCA [XP05]) is proposed to find all nodes, each of which contains matches to all keywords in its subtree and none of its descendants does. In particular, XKSearch [XP05] studies efficient algorithms to compute SLCA by designing an index lookup and scan-eager approach; XSeek [LC07] proposes an enhancement on SLCA, which infers the return nodes of a query by relying on the concept of entity. MaxMatch defines an axiomatic way to judge the quality of the matching semantics. XML TF*IDF [BLC+09] is proposed to perform effective search results ranking. Unlike all previous works, this chapter takes a novel perspective by studying keyword query refinement and refined query ranking (not results ranking) to improve the effectiveness on XML keyword search.

Query refinement in IR field can be divided into two spectrums: an automatic refinement [GXLC08, JRMG06, XC00] which modifies terms to queries according to a thesaurus or terms in documents, with no intervention on user part; and an interactive refinement [Rut03], such as relevance feedback which requires user to manually identify the relevant documents. In RDBMS field, [PY08] introduces a preprocessing stage to clean the raw text and extract a high-quality keyword query, in order to reduce the search space of a keyword query. This method, while pioneering, has some drawbacks: (1) the cleaned query is not guaranteed to have matching results in database, and (2) their methods to rank the cleaned queries do not consider their real matching results in database, thus significantly hampering effectiveness. In the rest of the chapter, our focus is to build a content-aware refinement solution that circumvents all the pitfalls encountered above by efficiently retrieving the real matching results of the refined queries.

In the field of XML retrieval, a dominant refinement strategy is query expansion, which adds new terms highly correlated with the initial query to enhance the result quality. In particular, [MM04] adopts a pseudo-feedback to choose the expanded terms from the top-ranked results of the initial query; [PTS04] expands the initial query based on the feedback of result relevance from user; TopX [TBM+08] generates the potential expanded terms by using a thesaurus database WordNet [Fel98]. All of them are complementary to our work, as they are only applicable on queries that have nonempty matching result.

7.3 Preliminaries

As shown in Fig. 7.1, we model XML document D as a rooted and labeled tree; each node n is denoted as tag:dewey, where tag is the tag name of n and dewey is the Dewey label of n. A keyword query $Q = \{k_1, k_2, \ldots, k_n\}$ is an ordered sequence of terms.

7.3.1 Meaningful SLCA

As addressed in our recent work [BLC+09], an XML search engine has to identify the target that a user desires to search for, namely, search for node. We first describe two concepts adopted from [BLC+09].

Definition 7.1 (Node Type) The type of a node n in XML document D is the prefix path from the root node of D to n.

To facilitate our discussion, we use its tag name instead of prefix path to represent T (if no ambiguity is caused) in the rest of this chapter.

Definition 7.2 XML DF f_k^T is the number of T-typed nodes containing keyword k in their subtrees in the XML document.

XML document frequency f_k^T is to distinguish statistics for subtrees rooted at different node types. For example, in Fig. 7.1, $\mathrm{f}^{\text{inproceedings}}_{\text{“XML”}} = 2$, as two inproceedings 0.0.1.0 and 0.1.1.0 contain “XML”.

Intuitively, a node type T is the desired search for node if (1) it is related to as many query keywords as possible, and (2) the subtrees (in the XML document) rooted at the nodes of type T should contain enough but non-overwhelming information. Following this intuition, we design Formula 7.1 to measure the confidence of a node type T to be the desired *search for node* w.r.t a given query Q.

$$C_{\text{for}}(T, Q) = \ln\left(1 + \sum_{k \in Q} f_k^T\right) * r^{\text{depth}(T)} \tag{7.1}$$

In Formula 7.1, r is a reduction factor ranging in (0, 1); depth(T) represents the depth of T-typed nodes in XML data. Here, we use *Sum* of f_k^T to combine the statistics of all keywords for each node type T, as some query keywords may not appear in the XML data.

Due to the keyword ambiguity problem [BLC+09], different people may issue the same query for different search intentions, so the search for node may not be unique. In the worst case, the number of search for node candidates can be as large as the number of distinct node types of an XML document; an appropriate threshold δ should be set to filter those that cannot be a promising candidate. We choose the desired search for node in two steps.

In the first step, we believe there should be at least one subtree (in the XML data tree) that contains at least one keyword of the initial query. Thus, the lower bound of Formula 7.1 is $\delta = \ln(1+1) * r^{\text{depth}(D)}$, where depth($D$) denotes the depth of the XML data tree D. As a result, those Ts whose search for confidence is above δ will be chosen as the search for node candidates.

In the second step, we try to get only the promising ones from those resulted in step 1. We first sort all the above candidates and pick the node type $T_{\max}$ with the highest confidence. Then we compute the relative difference percentage of the confidence scores of the remaining node types with $C_{\text{for}}(T_{\max}, Q)$ and choose those whose difference percentage is within a given threshold δ as one of the final desired search for node candidates. Note that there is no one-fit-all threshold value, and our empirical study demonstrates that a value of $\delta = 30\,\%$ has a reasonable and effective performance.

As a result, those *desired* search for node candidates will be used to further constrain the meaningfulness of an SLCA result of a query, which is shown in Definition 7.3. Regarding the tunable value of r in Formula 7.1, there is no one-fit-all choice, and through our empirical study, a choice of $r = 0.8$ works well in general.

Definition 7.3 A node n is a meaningful SLCA of query Q on XML document D, if all the following properties hold:

1. $n \in \text{SLCA}(Q, D)$[24] (i.e., n contains all the query keywords in either its labels or the labels of its descendants and has no descendant that also contains all the query keywords).
2. n is not the root node of XML document D.
3. n is a self or descendant of one of the search for node candidates T inferred by Formula 8.1 and above a given threshold.

Definition 7.4 A keyword query Q is said to need refinement if Q does not have any meaningful SLCA on XML document D.

In Definition 7.3, Property 2 is specified as users only expect the fragments of D to be returned; Property 3 is specified to ensure the result is related to one of the potential search targets. A detailed reasoning on Definition 7.3 is shown below.

First, we would like to argue the rationality of the properties (described in Definitions 7.3 and 7.4) to trigger a query refinement. For a given query Q, whether Q needs refinement may vary from users, as different people may have different search intentions even when they issue the same query. However, despite of the subjective search intention issue, we observe that no matter which user issues a keyword query on an XML data tree, he is interested in particular fragments, rather than the whole XML data tree, which is overwhelmingly large for user to consume. Therefore, if all the matching results of Q are the root node of the XML tree, it is certain that Q needs refinement. It naturally drives us to impose Property 2 in Definition 7.3. Moreover, when a user issues a query, she usually has her search target in mind ahead, though the search condition that is used to constrain the resulted instances of the search target may have various interpretations. Therefore, if all the potential search targets that a user query may intend to search for can be inferred, we can further constrain the condition to trigger the refinement. That explains why we define the search for node and specify Property 3 in Definition 7.3.

In other words, Definition 7.3 indeed describes an objective bottom line that necessitates a query refinement.

Second, we would like to discuss the flexibility of Definition 7.3. In this chapter, Definition 7.3 is defined to fit the SLCA matching semantics [LC07], but it is not confined to SLCA only. In fact it can be easily adapted to accommodate to any other matching semantics proposed for keyword search over XML data tree, such as LCA [GSBS03] and MSLCA [LYJ04], because only Property 1 in Definition 7.3 needs to be adjusted. Accordingly, our refinement solutions (i.e., partition-based approach and short-list eager approach proposed in Sect. 7.6) can also be easily adapted to any other matching semantics, because as described in Lemma 7.1, they are orthogonal to any method of finding the matching results of a query under a certain matching semantics.

As an example, consider Q_6 and RQ$_6$ in Table 7.1 issued on Fig. 7.1. Suppose author is one search for node candidate of Q_6 (by Formula 7.1). Q_6 does not have any meaningful SLCA, as its only SLCA result is the root node bib:0 which violates Property 2 in Definition 7.3. In contrast, the only SLCA of RQ$_6$ is hobby: 0.1.2 which is a descendant of author, so it is a meaningful SLCA.

7.3.2 Refinement Operations

As reported by the Web query logs tracing users' search modifications [RF03], a frequently used strategy by users is deleting terms, presumably to obtain greater coverage, while term substitution is the major strategy adopted by search engines.

Table 7.2 Sample refinement rules with dissimilarity scores

Rule#	Rule r	ds_r
$r1$	on, line → online	1
$r2$	data, abse → database	1
$r3$	article → inproceedings	1
$r4$	learn, ing → learning	1
$r5$	mechin → machine	2
$r6$	WWW → world, wide, web	1
$r7$	online → on, line	1

Besides, we observe that there are four potential sources that frequently cause ill-formed queries: (1) queries may contain misspelled or mismatched words (e.g., $Q_1 - Q_3$, Q_5 in Table 7.1), (2) mistakenly split words (e.g., Q_6, Q_7 in Table 7.1), (3) mistakenly merged words (such as queries in Table 7.5), and (4) queries that contain strong conjunctive constraints have no match against a small corpus (e.g., Q_4 in Table 7.1). These four ill-forms cause the results of the initial query either empty or nonsensical.

As a result, four refinement operations, that is, term substitution, term merging, term split, and term deletion, are defined to increase the query coverage. For example, a query {online, newspaper} may often be written as {on, line, news, paper} by user, which needs term merging. Term deletion is employed presumably to obtain greater coverage, as queries with no matches can have words deleted till a match is obtained, such as Q_4 in Table 7.1. Term substitution is wide-ranging, which mainly includes spelling error correction (Q_5 in Table 7.1), synonym substitution (Q_3), acronym expansion (Q_1), and word stemming (Q_2).

Definition 7.5 A refinement rule r associated with a refinement operation op is in form of $S_1 \rightarrow_{\mathrm{op}} S_2$, where S_1 and S_2 are two keyword sequences, and r has an associated dissimilarity score ds_r, which models the dissimilarity between S_1 and S_2.

Some typical refinement rules are listed in Table 7.2. In particular, for term merging/split and spelling error correction, ds_r can be a variant of the morphological metric such as string edit distance. For example, the dissimilarity ds_r of a one-time term merging/split is 1, as a single space is removed/added, such as $r1$, $r2$, $r4$, and $r7$ in Table 7.2. For $r5$, $\mathrm{ds}_{r5} = 2$ as two string edits are needed to correct the spelling error.

In this chapter, we mainly measure the dissimilarity of a refinement rule r by the string edit distance between the LHS and RHS of r. Since the term substitution is wide-ranging, which may include synonym substitution, typo correction, acronym expansion, etc., the dissimilarity varies correspondingly. For synonym substitution rule such as $r3$ in Table 7.2, ds_r can be the similarity score provided by checking against a corpus of the known database tokens or the semantic lexical database such as WordNet [Fel98]. For example, in WordNet [Fel98] the dissimilarity score between two words a and b is the length of the path from a's belonging synset to b's belonging synset; two synonyms in the same synset has a dissimilarity score of 1, such as $r3$. For acronym expansion such as $r6$ in Table 7.2, a score of 1 is designated.

Since it is out of the scope of this chapter to study a normalized measurement scheme that well handle the dissimilarity caused by either the semantic similarity or the literal similarity, and in order to minimize the effect of the dissimilarity assignment in evaluating the effectiveness of the query ranking model (as proposed in Sect. 7.4), we designate a uniform dissimilarity score for all refinement rules (that invoke a single refinement operation) except for term deletion. Besides, in this chapter we do not consider the recursive refinement which further applies refinement rule(s) on the newly generated keywords.

Since term deletion has the greatest potential in changing the meaning of initial query, we adopt the principle that its dissimilarity score is greater than any other three rules throughout this chapter.[1] The refinement rules can be obtained from data mining, query log analysis [RF03], or manual annotation [GXLC08]. However, how to generate these rules is orthogonal here.

Lastly, we define the dissimilarity between a RQ and initial Q.

Definition 7.6 Given a set R of refinement rules, the dissimilarity between Q and a RQ, denoted as dSim(Q, RQ), is the minimum of the sum of the cumulated ds_r among all possible sequences of application of the rules in R to transform Q into RQ.

7.4 Ranking of Refined Queries

In Sect. 7.3.2 $\mathrm{dSim}(Q, \mathrm{RQ})$ is defined as a preliminary quality metric for a RQ, mainly based on its lexical and morphological similarity w.r.t Q. However, it is inadequate without considering the local context (i.e., the XML data) being queried, especially for those RQs that have the same dissimilarity. Motivated by the ability of statistics in modeling patterns or drawing inferences about the underlying data, we aim to utilize the statistic knowledge of underlying XML data to build an in-depth content-aware query ranking model. In general, the overall quality of a RQ can be evaluated in two complementary aspects: (1) the similarity score of RQ which captures the relevance of RQ w.r.t the initial search intention, and (2) the dependency score of RQ which captures the keyword dependencies of RQ in XML data D.

We begin with introducing some statistic notations used in later discussion. tf(k, T) denotes the term count of k in all the subtrees rooted at node type T; F_T denotes the number of distinct terms contained in either the values or tags of all the subtrees rooted at node type T. For example, in Fig. 7.1, tf (“XML,” author) = 3, as “XML” appears three times within the subtrees rooted at author; $F_{\text{article}} = 14$ as there are 14 distinct keywords within the subtrees rooted at article.

[1] To facilitate our discussion, the dissimilarity score of a single term deletion rule is 2 throughout all examples in this chapter.

7.4.1 Similarity Score of a RQ

Given a query Q issued on an XML document D and a RQ candidate, we propose four intuitive guidelines in an incremental way to compute the similarity of RQ w.r.t the initial search intention.

Guideline 7.1 The more frequently the keyword in RQ appears within a search for node type T, the more important RQ is.

Following Guideline 7.1, Formula 7.2 is designed to accumulate the term frequencies of all keywords in RQ, and F_T is chosen as a normalization factor to prevent a bias to a search for node type T whose subtrees are of large size.

$$\mathrm{Imp}(\mathrm{RQ}, T) = \sum_{k \in \mathrm{RQ}} \frac{\mathrm{tf}(k, T)}{F_T} \tag{7.2}$$

In addition, we notice that each keyword in a query has its own ability to discriminate the query results, that is, each $k \in Q$ as a constraint of Q actually has different importance. Take term deletion as an example (Example 7.2); k_i is one of the keywords that are deleted from Q to form a RQ. Thus, the less frequent k_i appears in the subtrees rooted at T-typed nodes, the more discriminative k_i is to Q, that is, the RQ resulted from deleting this k_i from Q is less favored.

Example 7.2 Consider $Q = \{\text{XML, twig, pattern, join}\}$ issued on DBLP, where no matching result is found. Suppose inproceedings is a search for node candidate. Then, by deleting "join" and "pattern" respectively, we get two candidates $\mathrm{RQ}_1 = \{\text{XML, twig, pattern}\}$ and $\mathrm{RQ}_2 = \{\text{XML, twig, join}\}$, where $\mathrm{dSim}(\mathrm{RQ}_1, Q) = \mathrm{dSim}(\mathrm{RQ}_2, Q)$ as only one term deletion is adopted. Though, $f_{\text{pattern}}^{\text{inproceeding}} = 17{,}297$ is much larger than $f_{\text{pattern}}^{\text{inproceeding}} = 946$, that is, "join" and "pattern" have different importance as a constraint of Q, which also should affect the similarity of the resulted RQ.

Guideline 7.2 The more discriminative a keyword k_i that is either deleted from the initial query Q or newly generated by the term merging/split/substitution rules is w.r.t the initial query Q, the lower the rank of RQ, which is resulted from the corresponding refinement involving k_i, should be assigned.

By Definition 7.2, the XML DF f_k^T provides an effective way to measure the discriminative power of a keyword k, as Guideline 7.2 is in line with the design intuition of document frequency. In other words, the less XML DF of a keyword k_i (i.e., $f_{k_i}^T$) is, the more discriminative this k_i is w.r.t. Q. As a result, Formula 7.3 is designed to address Guideline 7.2 alone.

$$\mathrm{Imp}_{k_i}(Q, T) = \log \frac{N_T}{1 + f_{k_i}^T} \tag{7.3}$$

where N_T is the total number of nodes of type T in XML document D. The log function is applied to normalize the raw ratio.

Guideline 7.1 favors the RQ whose keywords have large term frequencies, which is analogous to the intuition of TF part in TF*IDF definition, while Guideline 7.2 favors the RQ whose deleted or newly generated keyword has the largest importance in the initial query, which is analogous to the intuition of *IDF* part. Therefore, we define the similarity ρ(RQ, Q|T) of a RQ w.r.t Q, for a given search for node type T in Formula 7.4, where the first multiplier addresses Guideline 7.1 and the second multiplier addresses Guideline 7.2 by accumulating the importance of k_i involved in refining Q to RQ.

$$\rho\,(\mathrm{RQ}, Q|T) = \mathrm{Im}\, p(\mathrm{RQ}, T)* \sum_{k_i \in (RQ \Delta Q)} \mathrm{Im}\, p_{k_i}(Q, T) \tag{7.4}$$

Here, RQΔQ denotes a set of keywords that are either deleted from Q or newly generated by term merging/split/substitution rules to produce RQ.

So far, we have only considered the case that the search for node candidate T of a query is unique. However, the keyword ambiguity problem [BLC+09] may cause more than one T to have comparable and promising search for confidence by Formula 7.1. Motivated by this fact, Guideline 7.3 is proposed.

Guideline 7.3 For an initial query Q that has multiple desired search for node candidates T, the confidence $C_{\mathrm{for}}(T, Q)$ of each such T should be taken into account. The higher the confidence of T as a desired search for node is, the more important its associated similarity $\rho(\mathrm{RQ}, Q|T)$ is.

Therefore, we incorporate the confidence of T as the desired search for node (i.e., Formula 7.1) and its similarity below:

$$\rho(\mathrm{RQ}, Q) = \sum_{T \in T_{\mathrm{for}}} C_{\mathrm{for}}(T, Q)^* \rho(\mathrm{RQ}, Q|T) \tag{7.5}$$

where T_{for} denotes a set of candidates of the desired search for node. Note that, Guideline 7.3 holds based on the principle that both the initial and refined query share the same search for node(s).

Lastly, by taking the semantic and morphological dissimilarity dSim(Q, RQ) between Q and RQ into account, whose intuition is mentioned in Guideline 7.4, the similarity of a RQ w.r.t the initial query Q is presented in Formula 7.6.

Guideline 7.4 The smaller the dissimilarity between Q and RQ is, the closer RQ is to Q in terms of the search intention.

$$\rho(\mathrm{RQ}, Q) = W^{(\mathrm{dSim}(Q,\mathrm{RQ}))^*} \sum_{T \in T_{\mathrm{for}}} C_{\mathrm{for}}(T, Q)^* \rho(\mathrm{RQ}, Q|T) \tag{7.6}$$

where dSim(Q, RQ) denotes the dissimilarity between Q and RQ as described in Definition 7.6. w is a decay factor ranging in (0,1) to enforce Guideline 7.4, and $w = 0.7$ is a good choice as evident by our empirical study.

7.4.2 Dependence Score of a RQ

In evaluating the quality of a RQ, the above similarity function emphasizes the relevance between Q and RQ, which has a limitation: the query terms are assumed to be mutually independent. As a complementation of the similarity score, if RQ contains more than one keyword, the dependency between the keywords in RQ over the XML data being queried should also be captured.

Guideline 7.5 A refined query candidate RQ is effective for a certain search for node T, if RQ has as many keywords as possible that co-occur frequently in the subtrees of type T.

Since the desired search for node candidate T may not be unique, for convenience we first discuss the case that T is unique. In order to quantify the dependency of keywords in a refined query RQ, we novelly utilize a variant of association rule [AIS93]. For each keyword $k_i \in \text{RQ}$, we measure how often another keyword $k \in \text{RQ}$ appears in the subtrees of type T that contain k_i, as shown in Formula 7.7:

$$C(k_i \Rightarrow k) = \frac{f^T_{k,k_i}}{f^T_{k_i}} \tag{7.7}$$

where $f^T_{k_i}$ represents the number of subtrees (rooted at node type T) that contain keyword k_i, and f^T_{k,k_i} denotes the number of subtrees (rooted at node type T) that contain both k_i and k.

The dependency score of a RQ is shown in Formula 7.8. The inner sum is a cumulation of how often each other keyword $k_i \in \text{RQ}$ appears together with k, while the outer sum cumulates such score for each keyword in RQ. In addition, as Guideline 7.5 usually favors the RQ with large size, a normalization factor |RQ| is introduced to prevent such bias.

$$\text{Dep}(\text{RQ}, Q|T) = \sum_{k \in \text{RQ}} \frac{\sum_{k_i \in \text{RQ}, k_i != k} C(k_i \Rightarrow k)}{|\text{RQ}|} \tag{7.8}$$

Once again, when Q is inferred to have multiple desired search for node candidates, we adopt Guideline 7.3 again to get the overall dependency score Dep(RQ, Q) below.

$$\mathrm{Dep}(\mathrm{RQ},Q|T) = \sum_{T \in T_{\mathrm{for}}} \left(C_{\mathrm{for}}(T,Q)^{*}\mathrm{Dep}(\mathrm{RQ},Q|T)\right) \tag{7.9}$$

At last, the overall rank of a refined query RQ (w.r.t the initial query Q) is completed by a weighted sum of its similarity score and dependency score in Formula 7.10.

$$\mathrm{Rank}(\mathrm{RQ},Q) = \alpha^{*}\rho(\mathrm{RQ},Q) + \beta^{*}\mathrm{Dep}(\mathrm{RQ},Q) \tag{7.10}$$

where α and β are tunable weights that reflect the importance of each metric. $\alpha = \beta = 1$ is the default choice, and the effectiveness impact of different choices will be evaluated in Sect. 7.7.3.

The index construction on collecting all the above statistic data for efficient ranking computation is discussed below.

The first index built is the traditional keyword inverted list. For each keyword k, it stores a list of nodes that directly contain k in document order. Such order is in accordance with the query processing order in both Algorithms 7.1 and 7.2. A B-tree index (on the keyword) is built over all the inverted lists to accelerate the lookup.

As another core part of our query refinement framework, how to efficiently calculate the ranking score of a RQ is also important. Among all the statistics needed in the whole ranking model, F_T (in Formula 7.2) and N_T (in Formula 7.3) can be easily collected when parsing the input XML document and stored in two separate tables, where each entry of the table corresponds to a unique node type T.

Furthermore, some statistics such as the XML term frequency tf(k,T) (in Formula 7.2) and document frequency f_k^T (in Formula 7.3) involve both the node type and the keyword. Therefore, we build another index called *keyword stats table*, which stores both tf (k,T) and f_k^T for each combination of keyword k and node type T in the XML document D. In the worst case, each node type may directly contain all the distinct keywords in D; so the space is $O(m^*t)$, where m denotes the number of distinct keywords in D and t denotes the number of node types in D (i.e., the number of distinct prefix paths of the XML data tree). Similar to the above keyword inverted list, a B-tree index is built on the keyword stats table for an efficient support on the following two operations:

1. getTF(T, k), which returns the XML TF tf(k,T).
2. getDF(T, k), which return the XML DF f_k^T

Lastly, an index called keyword dependency table is built to store the statistics f_{k,k_i}^T (in Formula 7.7). For each combination of any two distinct keywords k and k_i and any node type T, we have an entry storing f_{k,k_i}^T.

7.5 Exploring the Refined Query

Since the RQs that have both minimum dSim(Q, RQ) and meaningful matching result over XML data D is unknown ahead of processing Q, we should adaptively explore those RQs during the processing of Q. As can be seen in any refinement algorithm in Sect. 7.6 later, a fundamental problem encountered is as follows: we can obtain a set T of keywords, each of which is from a rule set R or the initial query Q and does exist in XML data D (see Line 9 of Algorithm 7.1). However, it remains a challenge to efficiently materialize a RQ from T, such that RQ has the minimum dSim(Q, RQ). This is what we mean the exploration of optimal RQ, as defined below:

7.5.1 *Problem Formulation*

Given a keyword sequence $S = \{k_1, k_2, \ldots, k_s\}$ (S denotes the initial query Q), a set $T = \{{k_1}', {k_2}', \ldots, {k_t}'\}$ of keywords and a set R of refinement rules. We aim to find a RQ, which is a subset of T and $\forall \mathrm{RQ}' \subseteq T$,dSim$(Q, \mathrm{RQ}) \le$ dSim(Q, RQ') (by Definition 7.6). We develop a bottom-up dynamic programming method, namely, getOptimalRQ(S, T), to resolve this problem.

7.5.2 *Subproblems*

We create subproblems below. Let $0 < i \le s$ be an integer. Let $S[1, i] = \{k_1, k_2, \ldots, k_i\}$ be a subsequence of S. Let C be an array of length $(|S| + 1)$, where $C[i]$ is the minimum dissimilarity between $S[1, i]$ and some RQ $\subseteq T$. Our final goal is to compute a value for $C[|S|]$, which is the minimum dissimilarity between S and some RQ $\subseteq T$.

7.5.3 *Notations*

Each $k_i \in S$ is associated with a set of refinement rules, denoted as R(k_i), R(k_i) $=$ $\{r | r \rightarrow <{*}k_i \rightarrow {k_\mathrm{m}}', \ldots {k_\mathrm{n}}'>\}$, where ${*}k_i = \{k_j, k_{j+1}, \ldots, k_{i-1}, k_i\}$ is a subsequence of S ended with k_i and $\{{k_\mathrm{m}}', \ldots {k_\mathrm{n}}'\} \subseteq T$. For each rule r, its left- and right-hand sides are denoted as LHS(r) and RHS(r), respectively.

7.5.4 Initialization

$C[0] = 0$, which means the dissimilarity between an empty query and any other query is 0.

7.5.5 Recurrence Function

Regarding the subproblem of computing $C[i]$ for $0 < i \leq |S|$, we have three options to consider:

- Option 1: When the ith keyword $k_i \in S$ also appears in T, then the dissimilarity score remains unchanged.
- Option 2: When k_i does not appear in T and term deletion is applied to delete k_i.
- Option 3: For a refinement rule r, if LHS(r) = *k and RHS(r) $\subseteq T$, $C[i]$ should be equal to a sum of $C[i\text{-}|\text{LHS}(r)|]$ and ds$_r$. If more than one rule can be applied here, the one with the minimum sum is selected. This case is used to handle term merging, split, and/or substitution.

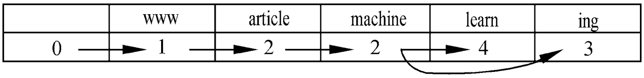
Among these three options, the one with the minimum value is assigned to $C[i]$, as summarized in Formula 7.11.

$$C[i] = \min \begin{cases} C[i-1] & \text{if } k_i \in T \\ C[i-1] + \text{cost of deleting } k_i & \text{if } k_i \notin T \\ C[i - |\text{LHS}(r)|] + \min\{ds_r\} & \text{if } k_i \notin T \\ \quad \text{and LHS}(r)={}^*k_i \text{ and RHS}(r) \subseteq T & \\ \quad \text{for each } r \in R(k_i) & \end{cases} \tag{7.11}$$

As we can see, getOptimalRQ is insensitive to the order of keywords in T, but sensitive to the order of keywords in S, because S denotes the original query which is a sequence of keywords (as defined in Sect. 7.3). However, this property does not limit its applicability and correctness w.r.t the four refinement operations defined. For instance, if a user mistakenly splits a term to two terms in his query S, then these two neighboring terms must also appear next to each other in the same order in the LHS of a term merging rule.

7.5.6 Time Complexity

getOptimalRQ runs in $|Q|$ loops, where in the worst case, each subsequence of Q is related to a certain refinement rule r. In addition, suppose a B-tree index is built upon the refinement rule set R, so the cost of locating such r is O(log$|R|$). As a result, the worst-case time complexity of getOptimalRQ is



	www	article	machine	learn	ing
0	1	2	2	4	3

Fig. 7.2 A running example of finding the optimal RQ

$$O\left(\sum_{i=1}^{|Q|}\sum_{j=1}^{i}\log|R|\right) = O\left(\left(\frac{|Q|(|Q|+1)(2|Q|+1)}{12}+\frac{|Q|(|Q|+1)}{4}\right)\log|R|\right)$$
$$= O(|Q|^3\log||R||)$$

As a summary, *getOptimalRQ* serves two purposes. First, it generates the optimal RQ in term of dSim(Q, RQ). Second, as a side product, a ranked list of some (but not all) nonoptimal RQ candidates by dSim(Q, RQ) can also be obtained, as they are indeed the intermediate results kept during executing *getOptimalRQ*. They will be used as the candidates for Top-K RQs later in Sect. 7.6.

A running example of getOptimalRQ is shown below.

Example 7.3 Given a query $Q = \{$WWW, article, machine, learn, ing$\}$ and a keyword set $T = \{$machine, inproceedings, learning, worldwide, Web, World, Wide$\}$, three relevant rules $r3$, $r4$, and $r6$ in Table 7.2 are identified. Figure 7.2 shows how array C is filled during the process of getOptimalRQ(Q, T). To compute $C[1]$, option 2 offers a cost of $C[0] +$ cost of deleting "WWW" $= 0 + 2 = 2$, while option 3 offers a cost of $C[0] + \text{ds}_{r6} = 1$. So $C[1] = \min(2,1) = 1$. Similarly, $C[2] = \text{C}[1] + 1 = 2$, $C[3] = 2$ as "machine" exists in T; $C[4] = 2 + 2 = 4$, and $C[5] = \min(C[3] + 1, C[4] + 2) = 3$. Finally, the optimal RQ $= \{$World, Wide, Web, inproceedings, learning$\}$, and dSim(RQ, Q) $= 5$.

7.6 Content-Aware Query Refinement

The main challenge toward an effective query refinement is that it is unknown whether any refinement is needed ahead of processing the initial query, as each RQ must have meaningful SLCA results over the XML data by Definition 7.3. A straightforward solution is to try the initial query first, and if no matching result is found, we go to infer all potential RQ candidates based on the given refinement rule set and try them one by one until the desired RQ is found. However, it may involve the evaluation of multiple queries, which has to scan the corresponding keyword inverted lists multiple times; even worse, many top-ranked RQs may not have any matching result.

Therefore, we propose to integrate the job of looking for the refined queries of Q and generating their matching results together to guarantee the existence of meaningful SLCA result for each RQ found, and meanwhile accompany the refinement job with the job of processing the initial query Q, in order to scan the related keyword inverted lists as few times as possible (optimally only once). This is what we call *content-aware XML keyword query refinement*.

As a result, two separate solutions, namely, partition-based approach and shortlist eager approach, are designed to find the approximate Top-K RQs and their matching results in a flow of document order. The main procedure of these two algorithms is as follows: we first maintain a ranked list to store the approximate Top-2K RQ candidates in term of dSim(Q;RQ) during answering Q. In the end, we apply the complete query ranking model (proposed in Sect. 7.4) to generate the final Top-K RQs from the 2K candidates.

7.6.1 Partition-Based Algorithm

As evident by Definition 7.3, the root node of an XML data tree is a typical meaningless SLCA, because users are only interested in the fragments of XML data. Therefore, we can partition the XML data tree into a list of ordered partitions as defined below:

Definition 7.7 (Document Partition) Given an XML data tree D, a subtree D_i is a document partition of D if the root node R_{Di} of D_i is the ith child of D's root node.

Document partition is a virtual view of the XML data tree D by ignoring its root node without modifying the structure and order of the nodes in D. In this way, our algorithm proceeds from one partition to another partition in document order, thus avoids all SLCA computations leading to the meaningless root node of D. In Fig. 7.1, there are two document partitions of D: D_1 rooted at author:0.0 and D_2 rooted at author:0.1, and all the meaningful SLCA nodes are either the self or descendants of the root node of D_1 or D_2.

Algorithm 7.1: Partition-based Top-K query refinement

```
input: Q={k_1,...,k_n}, refine rule set R, XML document D, K
output: result={(RQ_1,SLCA(RQ_1)),..., (RQ_k,SLCA(RQ_k))}
let result←Ø; let KS=getNewKeywords(Q)+Q;
let RQSortedList=a list of RQs sorted by dSim(Q, RQ);
{S_1,S_2,...,S_m}←getInvertedList(KS);
while (!end(S_i) for each i Ø [1,m]) do
V_s=getSmallestNode();/* 1≤s≤m*/
D_pid=getDocPartition(V_s);
  {S_1′,S_2′,...,S_m′}←getKLPartition(pid);
  move cursor of S_i to the node next to the end of each S_i′
  let T={k_i|S_i′ is not empty};
  {<RQ_i,dSim(Q,RQ)>|i∈[1,2k]}=getOptimalRQ(Q,T,2K);
  foreach RQ_i do
      if (dSim(Q,RQ)<RQSortedList.max) then
```

```
13.             if (!RQSortedList.hasRQ(RQ_i)) then
14.                RQSortedList.remove(2K);
15.                RQSortedList.insert(<RQ_i,dSim(Q;RQ_i)>);
16.                slca_RQi=computeSLCA({S_1′,S_2′,...,S_m′},RQ_i);
17.                result.add(RQ_i,slca_RQi);
18.             reset S_1′,S_2′,...,S_m′ to empty
19. Apply Formula 7.10 on result to get final Top-K RQs;
```

In Algorithm 7.1, the input is an initial user query Q, a value of K, an XML document D, and a given refinement rule set R. The output is a list of Top-K refined queries and their corresponding matching results over the XML data. RQSortedList is developed to store the up-to-date Top-2K RQs during the procedure of query refinement. It is implemented as a sorted list with a B-tree index built on the dissimilarity of RQ, where method insert and remove can be done in $O(\log 2\mathrm{K})$ time. Besides, method hasRQ, which is used to check whether a RQ to be inserted is already in the list, can be done in O(1) time by maintaining a separate hash table whose key is RQ itself.

7.6.1.1 Time Complexity

If indexed lookup in [XP05] is adopted for SLCA computation, Algorithm 7.1 costs $O(F^*K\log K^*|S_1'|\mathrm{mdlog}|S'|)$, where S' $\left(S_1'\right)$ is the max (min) size throughout lists S_1' to S_m'; F is the fanout of document root node; m is number of keywords involved and d is the document depth. The total cost by getOptimalRQ is $O(F^*K\log K^*|S_1'|\mathrm{mdlog}|S'| + m^3)$.

Algorithm 7.1 presents the details of partition-based approach. Initially, it finds a set KS of keywords that appear in either R or Q via a consultation on a given pertinent refinement rule set R (Line 1). A ranked list called RQSortedList is developed to store the up-to-date Top-2K RQs during the procedure of query refinement (Line 2). It supports three major operations: insert a RQ into the list, remove the lowest-ranked RQ from the list, and hasRQ checking whether a RQ to be inserted is already in the list.

A cursor is maintained for each keyword inverted list S_i. The algorithm runs in an iterative way: as long as the end of all the related keyword lists haven't been reached, the smallest node V_s in document order is selected (Line 5), and the document partition that contains V_s is located by Definition 7.7, denoted as D_{pid}, where pid is the label of this partition's root node (Line 6). Function getKLPartition is responsible for identifying the corresponding sublist S_i' of each keyword list S_i within partition D_{pid}, based on the property that pid is the prefix of the Dewey label of each node in each S_i' (Line 7). Accordingly, the cursor of each S_i is moved to the node next to the end of S_i' (Line 8).

An extension of function getOptimalRQ(Q, KS, 2K) (proposed in Sect. 7.5) is invoked to find the Top-2K RQ candidates (if they do exist) within partition D_{pid} (Lines 9–10); this extension is easy to achieve, as those RQ candidates are in fact preserved as the intermediate results during the exploration of optimal RQ. For each RQ_i in the Top-2K RQs found, if the dissimilarity of RQ_i is smaller than that of the lowest-ranked query in RQSortedList and RQ_i has not been inserted before, then RQ_i is inserted into RQSortedList (Lines 13–15), and any existing SLCA computation method (such as [XP05, SCG07]) can be employed to find the SLCAs of RQ_i within partition D_{pid} (Line 16) and add them into result (Line 17). Lastly, the overall query ranking model (i.e., Formula 7.10) is applied on the 2K RQ candidates to get the final Top-K RQs (Line 19). A running example of Algorithm 7.1 is shown below.

Example 7.4 Consider a query $Q = \{\text{article, online, data, base}\}$ issued on the XML data in Fig. 7.1, and the Top-1 RQ is expected if Q needs to be refined. Rules $r2$, $r3$, and $r7$ in Table 7.2 are found to be relevant to Q. For illustrative purpose, we list only five typical RQ candidates in an ascending order of its dissimilarity w.r.t Q.

RQ_1: {article, online, database} (2 merges)
RQ_2: {article, on, line, database} (1 merge)
RQ_3: {inproceedings, online, database} (1 merge, 1 substitution).
RQ_4: {inproceedings, on, line, data, base} (1 split, 1 substitution).
RQ_5: {inproceedings, online, base} (1 deletion, 1 substitution) ......

Partition D_1 in Fig. 7.1 is identified to contain part of the related keywords, and the partitioned keyword lists are as follows: $S'_{\text{online}} = S'_{\text{database}} = \{0.0.1.1.0.0\}$, $S'_{\text{on}} = S'_{\text{data}} = S'_{\text{article}} = \{\}$, $S'_{\text{inproceedings}} = \{0.0.1.0,\ 0.0.1.1,\ 0.0.1.2\}$, $S'_{\text{line}} = S'_{\text{base}} = \{0.0.1.0.0.0\}$. getOptimalRQ returns RQ_3 (with dSim(Q, RQ_3) = 2) and RQ_5 (with dSim(Q, RQ_5) = 3) as Top-2 RQs. As RQSortedList is empty, both RQ_3 and RQ_5 are inserted and their SLCA results are computed. Then, we move to next partition D_2, where $S'_{\text{on}} = \{0.1.1.0.0.0\}$, $S'_{\text{data}} = \{0.1.1.0.0.0,\ 0.1.1.1.0.0\}$, $S'_{\text{line}} = S'_{\text{base}} = S'_{\text{online}} = S'_{\text{database}} = \{\}$, $S'_{\text{article}} = \{0.1.1.1, 0.1.1.2\}$, $S'_{\text{inproceedings}} = \{0.1.1.0\}$. Now, the optimal RQ found by getOptimalRQ is RQ = {article, data} with dSim(Q, RQ) = 4 (as two term deletions are applied on Q), which is even larger than the dissimilarity of the current 2-nd RQ (in RQSortedList), that is, 3. Therefore, we can skip computing the SLCA results for any new RQ (other than those in RQSortedList) found in partition D_2. Lastly, result = {<RQ_3, inproceedings:0.0.1.1>} is returned as the Top-1 RQ.

In summary, AlgorithmC.1 reveals two major advantages: (1) within each partition, it is able to decide the current Top-2K RQ candidates before computing their SLCA results. As evident in Lines 12–17, for a partition D_j whose associated RQ candidates have larger dissimilarity than that of the lowest-ranked RQ in RQSortedList, we can skip computing the SLCA results of such RQ candidates on D_j (as they never can be the Top-K RQ), which is an important optimization. (2) It follows in a flow of the document order, so for a query Q that needs no refinement, the refinement will immediately stop once the first meaningful SLCA result of Q in XML data is found; thus, the extra cost spent on finding its RQs is minimized. Lastly, Lemma 7.1 and Theorem 7.1 show the exclusive features of Algorithm 7.1.

Lemma 7.1 *Algorithm 7.1's query refinement is orthogonal to any existing method of computing the SLCA results of a query on a certain XML document.*

Proof Algorithm 7.1 proceeds from one partition to another in document order, where in each partition P, the RQ candidates are determined before finding their SLCA results within P, as evident in Lines 10–15. Thus, it is orthogonal to the concrete methods of computing the SLCA results of these RQ candidates.

Note that, without loss of generality, Algorithm 7.1 is orthogonal to any LCA computation methods, where the only modification is to relax the criteria of triggering a query refinement as defined in Property 1 of Definition 7.3.

Theorem 7.1 *Given a query Q issued on an XML document D, Algorithm 7.1 is able to return the Top-K RQs according to their dissimilarity* dSim(Q, RQ) *and meanwhile generate their matching results within a one-time scan of related keyword inverted lists.*

Proof In Algorithm 7.1, Line 4 guarantees a one-time scan of the related keyword inverted lists. Besides, for each partition visited, function getOptimalRQ is able to find the Top-K RQs; the RQSortedList can guarantee to store the up-to-date top-ranked RQs. Lastly, by Lemma 7.1 the correctness and completeness of the matching results of each RQ are guaranteed.

7.6.2 Short-List Eager Algorithm

As we can see, Algorithm 7.1 requires a full scan of the related keyword inverted lists, though it needs only one-time scan. In practice, however, the frequencies of query keywords typically vary significantly [XP05]. Therefore, during the exploration of Top-K RQs, if we can start from the RQ candidates that contain the keyword of the shortest inverted list first, it is possible to skip the full scan of all the other inverted lists involved, as shown in Example 7.5.

Example 7.5 Consider the Top-1 query refinement of $Q = \{$XML, database, 2002$\}$ issued on Fig. 7.1. $S_{\text{database}} = \{0.0.1.1.0.0\}, S_{\text{xml}} = \{0.0.1.0.0.0, 0.1.1.0.0.0, 0.1.1.2.0.0\}$, $S_{2002} = \{0.0.1.0.1.0, 0.1.1.2.1.0\}$ If we start from the shortest inverted list S_{database}, partition D_1 with pid $= 0.0$ is found to cover the first occurrence of database. Since D_1 contains all the keywords of Q, there is no need to find any refinement for Q in D_1 and all the subsequent partitions. Therefore, the sequential scan of S_{xml} and S_{2002} can be avoided.

This idea is presented in Algorithm 7.2. The input and output in Algorithm 7.2 are same as those in Algorithm 7.1.

Algorithm 7.2: Short-List Eager Algorithm
input: Q=$\{k_1, \ldots, k_n\}$, refinement rule set R, XML document D, K
output: result=$\{(RQ_1, SLCA(RQ_1)), \ldots, (RQ_x, SLCA(RQ_x))\}$

```
1. letRQSortedList be a list of RQs sorted by dissimilarity;
2. let KS=getNewKeywords(Q)+Q; let $C_{\text{potential}}$=0;
3. {$S_1$,$S_2$,...,$S_m$}←getInvertedLists(allKeywords);
4. let $KS_{\text{pid}}$ denote a set of keywords appearing in partition pid;
5. while (Cpotential≤RQSortedList.max) do
6.    $k_i$=the keyword in KS with the shortest inverted list $S_i$;
7.    foreach partition $D_{\text{pid}}$ in $S_i$do
8.       foreach keyword k∈KS other than $k_i$do
9.          if (k appears in Partition $D_{\text{pid}}$) then
10.             insert k into $KS_{\text{pid}}$;
11.   {<$RQ_i$,dSim(Q,$RQ_i$)>|i∈[1,2k]}=getOptimalRQ(Q,$KS_{\text{pid}}$,2k);
12.   foreach $RQ_i$do
13.   if (dSim(Q, $RQ_i$) < RQSortedList.max) then
14.      if (!RQSortedList.hasRQ($RQ_i$)) then
15.         RQSortedList.remove(2k);
16.         RQSortedList.insert(<$RQ_i$,dSim(Q,$RQ_i$)>);
17.         KS=KS-$k_i$; remove $S_i$ from {$S_1$,$S_2$,...,$S_m$};
18.         Compute $C_{\text{potential}}$=getOptimalRQ(Q,KS,2K);
19. Apply Formula 7.10 on RQs in RQSortedList to get final Top-K RQs;
20. foreach $RQ_i$∈RQSortedList do
21.    result.add($RQ_i$,computeSLCAs($RQ_i$));
```

7.6.2.1 Time Complexity

In the worst case, each keyword in KS is involved in Top-K RQ exploration, and let $m = |\mathrm{KS}|$. In each loop, the cost of finding all partitions covering k_j is $|S|$ (line 7), so let P_{k_j} denote the number of partitions containing k_j; random accesses to other keyword lists cost $\sum_{i=j+1}^{|\mathrm{KS}|} \log|S_i|$ (line 8–10) (assuming keyword lists are sorted by length ahead, i.e., $|S_j| \le |S_i|, \forall j < i$); *getOptimalRQ* costs $2|Q|^3$ (line $11, 17$); all the operations supported by RQSortedList is $O(1)$. Thus, its time complexity is $\sum_{j=1}^{m}\left(|S_j| + R_{kj} * \left(\sum_{i=j+1}^{m}(|Q|^3 + \log|S_i|) + T_{\text{slca}}\right)\right)$, where T_{slca} denotes SLCA computation time for Top-K RQs, depending on the concrete algorithm adopted.

Algorithm 7.2 runs in two main steps. In step 1, the Top-K RQs are found (line 5–19). In step 2, any existing method is employed to compute the SLCA matching results for each RQ found in step 1 (Lines 20–21).

The core part of Algorithm 7.2 is how to set the stop condition for the Top-2K RQ exploration, that is, whether the potentially minimum dissimilarity is larger than the dissimilarity of the 2Kth query in current RQSortedList when RQSortedList is already full (Line 5). $C_{\text{potential}}$ denotes the potentially minimum dissimilarity for those RQ candidates unexplored yet. If it is greater than the dissimilarity of the 2Kth RQ in current RQSortedList, then any RQ candidate found later can never be one of the final Top-K RQs, and we can safely stop step 1. Otherwise, the current shortest list S_i is selected, and for each partition D_{pid} containing k_i, keyword sequence KS_{pid} will collect all keywords covered in D_{pid} by random accessing the inverted list of each other related keyword (Lines 8–10). Then *getOptimalRQ* is invoked to find Top-2K RQs within D_{pid}, and qualified RQs are put into RQSortedList (Lines 11–16).

A salient feature of short-list eager approach is Lines 17–18: at the end of each iteration, all refined queries that contain k_i have been identified, so the shortest list S_i is removed, and k_i is removed from KS accordingly; lastly, the potentially minimum dissimilarity $C_{\text{potential}}$ between Q and some RQ (which is a subset of the updated KS) is computed, which will be used in the stop condition checking of next iteration.

In order to better understand Algorithm 7.2, a running example is shown below.

Example 7.6 Consider a query $Q_4 = \{\text{XML, Join, 2003}\}$ (in Table 7.1) issued on the XML data in Fig. 7.1, and the user expects the Top-2 refined query to be returned if Q_4 needs refinement. Initially, the inverted list for each keywords are as follows: $S_{\text{xml}} = <0.0.1.0.0.0, 0.1.1.0.0.0, 0.1.1.2.0.0>$, $S_{2003} = <0.0.1.1.1.0, 0.0.1.2.1.0>$.

In the first iteration, the shortest inverted list is S_{John}, which is contained in the partition with pid 0.1, denoted as $D_{0:1}$. Then we access S_{xml} and S_{2003} to find whether any occurrence of these two keywords is within $D_{0:1}$. Then getOptimalRQ computes the Top-4 RQ candidates (if any), each of which should contain keyword "John". As a result, $\text{RQ}_1 = \{\text{XML, John}\}$ (where dSim(Q, RQ_1) =2 as a term deletion is enforced) and $\text{RQ}_2 = \{\text{John}\}$ (where dSim(Q, RQ_2) = 4) are found and inserted into RQSortedList.

In the second iteration, keyword "2003" has the shortest inverted list and is contained in partition $D_{0:0}$, where keywords "2003" and "XML" are found to exist in partition 0.0. Thus, $\text{RQ}_3 = \{\text{2004, XML}\}$ and $\text{RQ}_4 = \{2003\}$ are the candidates in $D_{0:0}$ and are inserted into RQSortedList.

Finally, assuming the query ranking model gives equal rank to each RQ, RQ_1 and RQ_3 are returned as the Top-2 refined queries for Q_4, as both of them have the smallest dissimilarity.

7.6.2.2 Discussion

First, Lemma 7.1 and Theorem 7.1 also hold for Algorithm 7.2. Second, the performance of Algorithm 7.2 depends on two factors:

1. Whether the RQs that cover the keyword with the shortest inverted list are among the final Top-K RQs
2. How early the first match of each RQ in the final Top-K RQs appears in XML data

Based on this analysis, we can have a smarter choice of k_i and S_i in each iteration (Line 6): the k_i which either appears in the RHS of the refinement rules related to Q or never appears in the LHS of any rule related to Q (i.e., the keyword that does not need any refinement), and also has the shortest inverted list should be chosen first.

In this way, the RQ containing such k_i should have a high probability to be one of the final Top-K RQs, and thus, the exploration of Top-K RQs can finish earlier.

As a summary, Algorithm 7.1 achieves a one-time scan of the related keyword lists at the expense of a full scan for each related keyword list, while Algorithm 7.2 avoids the full scan at the expense of scanning the related keyword lists multiple times. The practical performance of these two approaches is query and data dependent.

7.7 Experiments

We use both real datasets and real user queries in order to test the empirical effect of our approach.

In the experimental study, we investigate the efficiency and scalability of the two refinement algorithms (i.e., the partition-based approach and short-list eager approach) proposed in Sect. 7.6, and the effectiveness of our query ranking model proposed in Sect. 7.4. Note that, we do not include the evaluation of the most related work [PY08] which is designed for keyword query cleaning in relational database, because it is nearly infeasible to extend it to fit into XML document, and it cannot guarantee the existence of the matching result of the cleaned keyword query.

7.7.1 Equipment

All experiments are performed on a 1.9-GHz AMD DualCore PC running Windows XP with 3-GB memory. All codes are implemented in Java, and Berkeley DB Java Edition [BDB] is used to store the keyword inverted lists.

7.7.2 Dataset and Query Set

Since our work is an empirical study closely related to user's real search experience, we used real dataset and real-world user queries instead of the synthetic datasets and queries. To our best knowledge, a common problem that all existing works in the field of XML keyword search have encountered in studying the practicability of their approaches is the lack of real-world datasets and user queries.

Due to the lack of real-world datasets, only two real datasets DBLP [Ley] (420 MB, depth = 2, up to 2007/12/10) and Baseball[2] (1 MB, depth = 5) are used in our experiments. DBLP contains publications in computer science; Baseball

[2] http://www.ibiblio.org/xml/books/biblegold/examples/baseball/

contains information on teams and players of North American baseball league. These two real datasets differ from each other in terms of the data organization and data application: DBLP is shallow and wide, while Baseball is deep and narrow. Our goal in choosing these diverse data sources is to understand the usefulness of our refinement strategies in different real-world environments.

Regarding the real-world user queries, the most recent 1,000 queries are selected from the query log of an DBLP online demo[3] of our previous work [BCLL09], out of which 219 queries (with an average length of 3.92 keywords) that have empty result are selected to form a pool of queries that need to be refined, which coincides with the primary motivation. Besides, we randomly pick 100 queries that have meaningful matching results and add them into the query pool, in order to increase the variety of queries. The refinement rules come from either WordNet [Fel98] or human annotation, and we adopt the same metrics for measuring the dissimilarity score of each rule (except for term deletion) as described in Sect. 7.3.2.

7.7.2.1 Notations

(1) SLCA refers to the scan-eager approach proposed in [XP05] for SLCA computation. (2) The short-list eager and partition-based algorithm proposed in Sect. 7.6 are called SLE and partition, respectively; both partition and SLE employ the scan-eager approach [XP05] in computing the SLCA results of a query.

7.7.3 Efficiency

In this section, we evaluate the efficiency of SLE and partition by measuring the latency between a query is issued and its Top-K RQs with their matching SLCA results are returned.

We first evaluate SLE and partition for Top-1 query refinement on all sample queries in Tables 7.3, 7.4, 7.5, and 7.6 plus Q_{x1}–Q_{x6}. Sample queries with a refinement using a typical operation are shown in Tables 7.3, 7.4, 7.5, and 7.6, respectively, where in each table the third column shows the refinements returned by our method, and the fourth column shows the cardinality of results based on the corresponding RQ. Besides, queries involving multiple mixed refinements, that is, $\mathrm{Q}_{\mathrm{x}1} - \mathrm{Q}_{\mathrm{x}6}$, are shown below.

Q_{x1}:{efficient, key, word, search}, which can be refined by substituting "efficient" for "eficient", followed by a merging of "key" and "word".

Q_{x2}:{efficient, sky, line, computation}, where a desired refinement is {efficient, skyline, computation}.

[3] http://xmldb.ddns.comp.nus.edu.sg

Table 7.3 Sample query sets for term deletion

ID	Initial query	Suggested refinements	Results
$Q_{\mathrm{D}1}$	Ling, Tok, Wang, twig, pattern, join	delete "pattern" of "join"	2 or 5
$Q_{\mathrm{D}2}$	Yufei, Tao, skyline, 2000	delete "2000"	5
$Q_{\mathrm{D}3}$	Tan, Kian, Lee, keyword, search	delete "keyword"	8
$Q_{\mathrm{D}4}$	XML, view, model, 1995	delete "XML" or "1995"	4 or 8
$Q_{\mathrm{D}5}$	XML, graph, keyword, search	delete "XML" or "graph"	1 or 22
$Q_{\mathrm{D}6}$	Ooi, Beng, Chin, Jagadish, index	delete "Jagadish" or "index"	8 or 11
$Q_{\mathrm{D}7}$	Yannis, graph, keyword, search	delete "Yannis" or "graph" or "keyword"	1 or 10 or 1

Table 7.4 Sample query for term merging

ID	Initial query	Suggested refinements	Results
Q_{M1}	Jia, wei, han, 2006	Jiawei	35
Q_{M2}	Xiao, fang, zhou, 2005	Xiaofang	16
Q_{M3}	on, line, news, paper	online, newspaper	6
Q_{M4}	electronic, text, book	textbook	6
Q_{M5}	xml, key, word, search	keyword	21
Q_{M6}	online, hand, writing	handwriting	47
Q_{M7}	work, shop, data, management, korea	workshop	2
Q_{M8}	net, work, routing, protocol	network	59
Q_{M9}	micro, array, gene, classification, selection	microarray	21
Q_{M10}	over, lay, routing, cost	overlay	3

Table 7.5 Sample query sets for term split

ID	Initial query	Suggested refinements	Results
$Q_{\mathrm{P}1}$	adhoc, search	ad, hoc	14
$Q_{\mathrm{P}2}$	webpage, filtering, 2006	web, page	2
$Q_{\mathrm{P}3}$	fulltext, search, networds	full, text	3
$Q_{\mathrm{P}4}$	floatingpoint, function	floating, point, function	10
$Q_{\mathrm{P}5}$	multiquery, processing	multi, query	24
$Q_{\mathrm{P}6}$	realtime, application, analysis	real, time	11
$Q_{\mathrm{P}7}$	hengtao, shen, video, 2007	heng, tao	5

Q_{x3}:{worldwide, web, search, engine} can be refined by either adopting worldwide → world, wide or www → worldwide web.

Q_{x4}:{inproceeding, xml, twig, match} can be refined by substituting "inproceedings" for "inproceeding", "matching" for "match".

Q_{x5}:{suficient, bundary, values} can be refined by a series of substitutions: sufficient → sufficient, bundary → boundary, values → value.

Q_{x6}:{private, data, preserve} can be refined by a series of substitutions: private → privacy, preserve → preservation.

We also compare them with a naive approach, where we first process the initial query and then enumerate all the RQ candidates if necessary, and try them one by one (in a descending order of its query rank) until a user is satisfied with the query

Table 7.6 Sample query sets for term substitution

ID	Initial query	Suggested refinements	Results
Q_{S1}	Jagadish, VLBD	VLDB	41
Q_{S2}	machin, learning, technique	machine	9
Q_{S3}	Jim, Gary, VLDB	Gray	8
Q_{S4}	principle, component, neural, network	principal	18
Q_{S5}	xml, document, object, model	DOM	11
Q_{S6}	extensible, markup, language, application	XML	71
Q_{S7}	privacy, preserving, cluster	clustering	24
Q_{S8}	fuzy, database, search	fuzzy	4
Q_{S9}	DASFA, 2007, XML	DASFAA	11
Q_{S10}	distributed, allocation, chanel	channel	42
Q_{S11}	search, bundary, constraints	boundary	2

results, while the time spent on user judgment is not counted. Besides, we record the time spent on processing the initial query by SLCA [XP05] to understand the extra cost brought by the exploration of RQs.

Figure 7.3 show the elapsed time for all sample queries that need refinement (on hot cache), where we have four observations.

1. Both partition and SLE outperform the naive approach for all sample queries; partition is about twice faster than SLE.
2. SLCA spends the least evaluation time, as it is only responsible for processing the initial query which even has no meaningful matching result. In contrast, partition brings a very small extra cost (about 30 % in average), but serves both the purpose of producing the Top-1 RQ and finding its matching result in XML data tree.
3. SLE outperforms partition for Q_{D2} and Q_{x3}, because the keyword with the shortest inverted list is also in the final Top-1 RQ, so that the full scan of corresponding inverted lists is avoided.
4. Interestingly, we find for Q_{M10}, Q_{S3}, Q_{S11}, and Q_{D7}, partition is even more efficient than SLCA which does not perform any refinement operation. This can be explained that the extra cost spent by Partition on computing the ranking scores of the RQ candidates is even smaller than the cost by SLCA in computing the meaningless LCAs (i.e., the document root node) for those queries.

Lastly, we randomly pick ten queries that do not need any refinement, and investigate the elapsed time by SLE, partition, and SLCA. As shown in Fig. 7.3, in average both SLE and partition spend about 20 % extra time as compared to SLCA, which is acceptable.

7.7.4 Scalability

In order to test the scalability of SLE and partition in Top-K query refinement, we design two experiments.

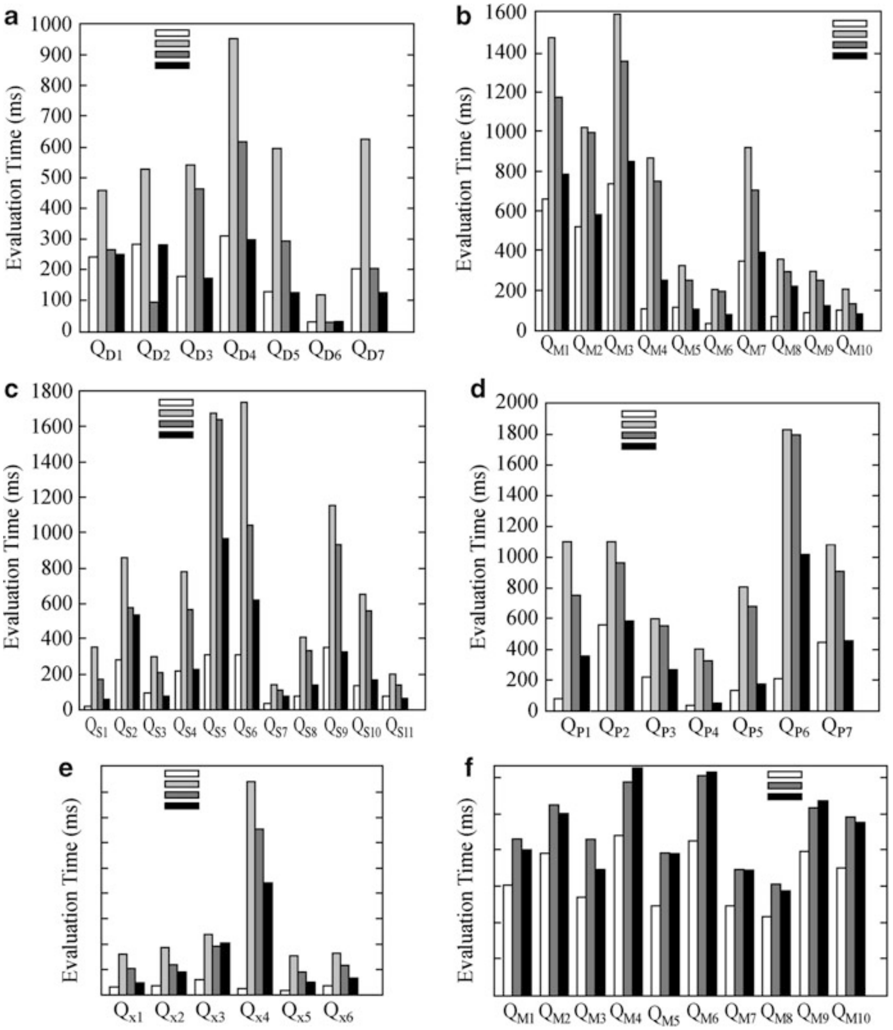


Fig. 7.3 Top-1 sample query refinement on DBLP. (**a**) Delete, (**b**) merge, (**c**) substitution (**d**) split (**e**) mix, (**f**) no refinement

Firstly, we measure the effects of different choices of K on the evaluation time of Top-K query refinement, where $k \in [1,6]$. A batch of 40 random queries with an average length of 3.71 for DBLP and 20 random queries with an average length of 3.18 for Baseball are tried, and the average time of those queries in five executions are shown in Fig. 7.4. As evident from Fig. 7.4, partition scales well all the way, while SLE's time increases much faster when $K > 3$. Since SLE has to find all Top-K RQs before evaluating them, the larger the K is, the more extra time on dissimilarity computation is, and more times of keyword lists scan are needed in employing

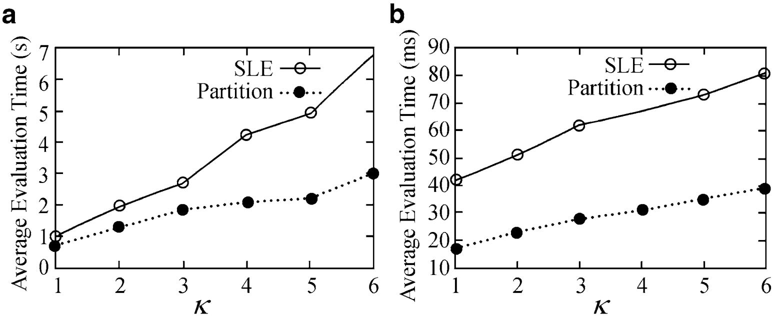
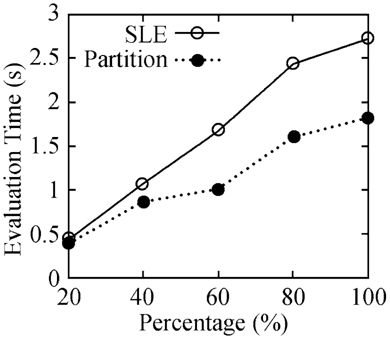


Fig. 7.4 Effects of K on Top-K query refinement. (**a**) DBLP, (**b**) baseball

Fig. 7.5 Effects of data size on Top-3 RQ computation



existing methods to find SLCA results. In contrast, for partition approach, the larger the K is, the higher possibility that the lower-ranked RQs and their SLCA results (that are detected before the higher-ranked queries) are preserved (rather than pruned away), so less extra cost is introduced. For Baseball data, both algorithms scale equally well, as shown in Fig. 7.4.

Secondly, we measure the response time of Top-3 query refinement by SLE and partition over the datasets of different size, which are obtained from DBLP (420 MB), and a batch of 40 random queries are used again. As shown in Fig. 7.5, both approaches have a good scalability over the data size. Note that SLE has a significant increase from 60 to 80 %, as SLE's efficiency relies heavily on how early the Top-K RQs are detected, which consequently affects the number of random accesses to the keyword inverted lists.

7.7.5 *Effectiveness of Query Refinement*

Having verified the efficiency of our algorithms, in the sequel we assess the effectiveness of our query ranking model.

7.7.5.1 Evaluation Method

Traditional IR evaluation methods include precision, recall, F-measure [LC07], reciprocal rank [BLC+09], etc.; however, all of them are based on a binary judgment (which judges a result to be either relevant or irrelevant). In contrast, by taking into account the fact that all results are not of equal relevance to users, cumulated gain-based evaluation (CG) [JK02] is proposed to combine the degree of relevance of the results and their ranks (affected by their possibility of relevance) in a coherent way, no matter what the recall base size is. In particular, given a ranked result list, [JK02] turns the list to a gained value vector $G[i]$, which denotes the relevance score of the ith result retrieved; then a cumulated gain vector CG is defined recursively as shown in Formula 7.12, where CG[i] is computed by summing G[BDB] up to $G[i]$. Therefore, we choose to adopt CG as a more precise way to evaluate the effectiveness of our query ranking model.

$$\mathrm{CG}[i] = \begin{cases} \mathrm{G}[i] & \text{if } i = 1 \\ \mathrm{CG}[i-1] + \mathrm{G}[i] & \text{otherwise} \end{cases} \tag{7.12}$$

7.7.5.2 Effectiveness Study

In order to study the empirical effects of our query ranking model, all queries tested are real-world user queries as logged in our XML keyword search engine [BCLL09]. For each query, we extract its Top-4 RQs. Six researchers are invited for relevance judgment of query refinement on DBLP, as DBLP is one of the few large real XML datasets, and the six researchers use DBLP to find papers frequently, which helps make their judgment more reliable. They are asked to look into each RQ and its matching results carefully, and judgments are done on a four-point scale as follows: (1) irrelevant, (2) marginally relevant, (3) fairly relevant, (4) highly relevant. As mentioned in [JK02], a proper choice of relevance score depends on the evaluation context. Thus, we use moderate relevance scores (say, 0-1-2-3) for the above four-point scale, as we assume that our users are patient enough to dig down the results of low-ranked RQs.

Table 7.7 shows the Top-4 RQs and their result numbers (separated by semicolon) for some queries in Table 7.10. For simplicity each keyword is denoted by its first letter if no ambiguity is caused. For each query in Table 7.7, all six users have an agreement that its Top-1 refined query RQ_1 is the most appropriate refinement.

Next, we make an in-depth analysis of the query ranking model. As the overall rank of a RQ consists of two complementary parts, that is, similarity score and dependency score, we conduct two sets of experiments to test their respective effects individually.

In the first experiment, we investigate the query ranking model that takes the similarity score into account alone. As Guidelines 7.1–7.4 (in Sect. 7.4.1) contribute to the similarity score of a RQ, we test how each of them contributes to the overall quality. Let RS_0 denote the original ranking scheme and RS_i denote a variant of RS_0

Table 7.7 Top-4 ranked RQs with their result number

Q	RQ$_1$	RQ$_2$	RQ$_3$	RQ$_4$
Q_{M1}	jiawei, h, 2006; 35	h, w, 2006; 45	j, w, 2006; 29	h, j, 2006; 9
Q_{M2}	xiaofang, z, 2005; 16	xiaofang, z; 91	x, z, 2005; 27	f, z, 2005; 7
Q_{M9}	microarray, g, c, s; 21	microarray, g, s; 60	array, g, c, s; 2	m, a, c, s; 1
Q_{S3}	J, Gary, VLDB; 8	J, Gary; 21	J, VLDB; 11	G, VLDB; 4
Q_{S5}	XML, DOM; 11	d, o, m; 9	XML, o, m; 5	XML, d, m; 8
Q_{S6}	XML, a; 71	m, l, a; 6	e, m, l; 22	l, a; 189
Q_{P3}	full, text, s, n; 3	t, s, n; 7	f, t, n; 5	f, s, n; 3
Q_{P6}	real, time, ap, an; 11	ap, an; 1187	realtime, ap; 5	realtime, an; 2
Q_{X1}	efficient, keyword, s; 19	efficient, k, s; 4	word, s; 21	key, w, s; 1
Q_{X2}	efficient, skyline, c; 8	skyline, c; 13	eff, skyline; 17	efficient, l, c; 4
Q_{X3}	world, wide, w, s, e; 9	www, s, e; 39	web, s, e; 156	w, w, w, s; 43

Table 7.8 Query statistics

Refinement	Num
Term merging	18
Term split	10
Term substitution	31
Term deletion	4

by removing Guideline 7–i from consideration for $i \in [1,4]$. In our experiment, we adopt the above CG evaluation, but the input now is a ranked list of RQs (associated with their matching results). Fifty queries that have no meaningful result on DBLP, involve various refinement(s), and have at least four possible RQ candidates are chosen from our query pool. Table 7.8 shows a summary of the number of queries that involve the four refinement operations respectively. The decay factor w in Formula 7.6 is set to 0.7 here.

Table 7.9 shows the results of the average CG values judged by the above six users for Top-K RQs, for $K = 1$–4. (1) From column 2 of Table 7.9, we find the original ranking model, that is, RS$_0$ is the most effective one that can capture the most relevant result as Top-1 RQ, compared to all its four variants. (2) By comparing CG[i] of each model for $i \in [0,4]$, we find the original ranking model outperforms all its four variants in finding the Top-K RQs for any $k \in [0,4]$. (3) In finding the Top-1 RQ, Guideline 7.4 plays a much more important role than other guidelines, as CG[1] of RS$_4$ has the smallest value. (4) From the last column of Table 7.9, we find both the original ranking model and its four variants have similar value for CG[4], which means all of them are able to find the desired Top-4 RQs, although the relative ranks of these RQs vary in each variant.

In the second experiment, we test a combined effect of the similarity score and the dependency score of a RQ. The importance of these two factors is investigated by varying the choice of the tunable parameters α and β in Formula 7.10. From the result as shown in Table 7.10, we have the following observations: (1) by comparing variants [1, 1] and [1, 0], we find the consideration of the dependency score does improve the overall effectiveness of our query ranking model. (2) By comparing the

Table 7.9 CG@4 by different ranking models

Variants	CG[1]	CG[2]	CG[3]	CG[4]
RS_0	2.631	3.562	4.233	4.539
RS_1	2.343	3.491	4.127	4.516
RS_2	2.416	3.525	4.161	4.525
RS_3	2.427	3.509	4.058	4.497
RS_4	2.305	3.456	4.16	4.521

Table 7.10 CG@4 by different weights

$[\alpha, \beta]$	CG[1]	CG[2]	CG[3]	CG[4]
[1, 2]	2.626	3.56	4.217	4.532
[2,1]	2.64	3.565	4.241	4.537
[1,1]	2.675	3.569	4.236	4.543
[1,0]	2.631	3.562	4.233	4.539

CG[1] for all the variants, we find the similarity score is more effective than the dependency score in contributing to infer the Top-1 RQ.

In summary, the naive query refinement approach is not adequate due to its costly query time; the original SLCA algorithm without refinement functionality is not reliable because it fails to report meaningful answers for many queries. In contrast, SLE and partition can detect and produce high-quality refined queries and their matching results in an efficient way. Overall the best solution is the partition algorithm, which offers the best-fit refinement and scales better than SLE. Furthermore, the comprehensive CG evaluation demonstrates the effectiveness of our query ranking model.

7.8 Summary

In this chapter, we introduce the problem of content-aware XML keyword query refinement, aiming to integrate the job of finding the desired refined queries and generating their matching results as a single problem, with no intervention on user part. As a core part of this work, we propose a statistics-based query ranking model which takes into account of both the keyword dependencies in RQ and the relevance of RQ w.r.t original search intention. We further propose two adaptive query refinement algorithms. Lastly, experiments show the efficiency and effectiveness of our approach.

References

[AIS93] Agrawal, R., Imielinski, T., Swami, A.: Mining association rules between sets of items in large databases. In: SIGMOD 1993, Washington, DC (1993)

[BCLL09] Bao, Z., Chen, B., Ling, T.W., Lu, J.: Demonstrating effective ranked XML keyword search with meaningful result display. In: DASFAA 2009, Brisbane (2009)

[BLC+09] Bao, Z., Ling, T.W., Chen, B., Lu, J.: Effective XML keyword search with relevance oriented ranking. In: ICDE, Shanghai (2009)

[Fel98] Fellbaum, F.C.: WordNet: a electronic lexical database. Cambridge, MA: MIT Press (1998)

[FP05] Fain, D.C., Pedersen, J.O.: Sponsored search. In: Bulletin of the American Society for Information Science and Technology (2005)

[GSBS03] Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: ranked keyword search over XML documents. In: SIGMOD, San Diego (2003)

[GXLC08] Guo, J., Xu, G., Li, H., Cheng, X.: A unified and discriminative model for query refinement. In: SIGIR, Singapore (2008)

[JK02] Jarvelin, K., Kekalainen, J.: Cumulated gain-based evaluation of IR techniques. ACM Trans. Inf. Syst. **20**(4), 422 (2002)

[JRMG06] Jones, R., Rey, B., Madani, O., Greiner, W.: Generating query substitutions. In: WWW (2006)

[LC07] Liu, Z., Chen, Y.: Identifying meaningful return information for XML keyword search. In: SIGMOD, Beijing (2007)

[LC08] Liu, Z., Chen, Y.: Reasoning and identifying relevant matches for XML keyword search. PVLDB **1**(1), 921–932 (2008)

[LYJ04] Li, Y., Yu, C., Jagadish, H.V.: Schema-free XQuery. In: VLDB, Toronto, pp. 72–83 (2004)

[MM04] Mass, Y., Mandelbrod, M.: Component ranking and automatic query refinement for XML retrieval. In: INEX, Dagstuhl (2004)

[PTS04] Pan, H., Theobald, A., Schenkel, R.: Query refinement by relevance feedback in an XML retrieval system. In: ER, Shanghai (2004)

[PY08] Pu, K.Q., Yu, X.: Keyword query cleaning. In: VLDB, Auckland (2008)

[RF03] Jones, R., Fain, D.: Query word deletion prediction. In: SIGIR, Toronto (2003)

[Rut03] Ruthven, I.: Re-examining the potential effectiveness of interactive query expansion. In: SIGIR, Toronto (2003)

[SCG07] Sun, C., Chan, C.Y., Goenka, A.K.: Multiway SLCA-based keyword search in XML data. In: WWW, 2007, Banff (2007)

[TBM+08] Theobald, M., Bast, H., Majumdar, D., Schenkel, R., Weikum, G.: Topx: efficient and versatile top-k query processing for semistructured data. VLDB J. **17**(1), 81–115 (2008)

[XC00] Xu, J., Croft, W.B.: Improving the effectiveness of information retrieval with local context analysis. ACM Trans. Inf. Syst. **18**(1), 79–112 (2000)

[XP05] Xu, Y., Papakonstantinou, Y.: Efficient keyword search for smallest LCAs in XML databases. In: SIGMOD, Baltimore (2005)

[XP08] Xu, Y., Papakonstantinou, Y.: Efficient LCA based keyword search in XML data. In: EDBT, Nantes (2008)

Chapter 8
LCRA, XML Keyword Search System, and LotusX, Graphical Query Processing System

Abstract Keyword search is a convenient way to query XML document and auto-completion is a user-friendly type-as-you-search feature to make queries more simple. In this chapter we propose two XML search systems LCRA and LotusX where LCRA is a system supporting keyword search and returning meaningful searching results while LotusX provides an auto-completion feature in a graphical search interface to simplify the processing of twig search. In addition the system architectures and the features of the two systems are also presented. The result ranking strategies for the keyword search system and the twig pattern query system are also proposed respectively. Furthermore we introduce the query rewriting approaches for twig pattern search in LotusX.

Keywords Keyword search • SLCA • LRA pairs • ELRA pairs • ELRA group • LCRA • XQuery • Textual information • Structural information • Position-aware • Search-as-you-type • Auto-completion • Static tag-trie • Dynamic value-trie • Order-sensitive • Result ranking • Query rewriting

8.1 Introduction of LCRA and LotusX

Keyword search is a convenient way to query XML documents since it allows users to easily issue keyword queries without the knowledge of complex query languages and/or the structure of underlying data. And the goal of XML keyword search is to find only the meaningful and relevant data fragments corresponding to interested objects that users really concern on. Majority of the research efforts in XML keyword search focus on keyword proximity search in either tree model or general digraph model. Both approaches generally assume that a smaller substructure of the XML document that includes all query keywords indicates a better result. Therefore, we propose a system to support XML keyword search, called *LCRA*,

J. Lu, *An Introduction to XML Query Processing and Keyword Search*,
DOI 10.1007/978-3-642-34555-5_8,

which offers various query flexibility: users can issue pure keyword queries that can be any combinations of words in full or partial specification of author names, topics, conference/journal, and/or year.

In addition, we observe that the existing query languages for XML (e.g., XQuery) require professional programming skills to be formulated; however, learning such complex query languages is a tedious and a time-consuming process that can be very challenging especially to novice users. In addition, when issuing an XML query, users are required to be familiar with the content (including the structural and textual information) of the hierarchical XML, which is difficult for common users. The need for designing user-friendly interfaces to reduce the burden of query formulation is fundamental to the spreading of XML community. Therefore, we present a twig-based XML graphical search system, called *LotusX*, that provides a graphical interface to simplify the query processing without the need of learning query languages, data schemas, nor the knowledge of the content of the XML document. The basic idea is that LotusX proposes *position-aware* and *auto-completion* features to help users to create tree-modeled queries (twig pattern queries) by providing the reasonable candidates on-the-fly. In addition, complex twig queries (including order-sensitive queries) are supported in LotusX. Furthermore, a new ranking strategy and a query rewriting solution are implemented to rank the results and automatically rewrite queries, respectively.

8.2 LCRA: Search Semantics

8.2.1 SLCA and LRA

In tree model, SLCA (smallest lowest common ancestor) [GSBS03, LYJ04, SKW01, XP05] is a simple and effective semantics for XML keyword proximity search. Each SLCA result of a keyword query is a smallest XML node[1] that (1) covers all keywords in its descendants and (2) has no single proper descendant to cover all query keywords. However, the SLCA semantics based on tree model does not capture ID reference information which is usually present and important in XML databases. As a result, it may return a large tree including irrelevant information.

On the other hand, XML documents can be modeled as digraphs to take into account ID reference edges. The key concept in digraph model is called reduced subtrees [CKKS05, KS05], which searches for minimal connected subtrees in graphs. However, the problem of finding all reduced subtrees and enumerating results by increasing sizes of reduced subtrees is NP-hard. Thus, most previous algorithms on XML digraph model are intrinsically expensive and heuristics-based.

[1] When we say an XML node in this chapter, we refer to the subtree rooted at the node.

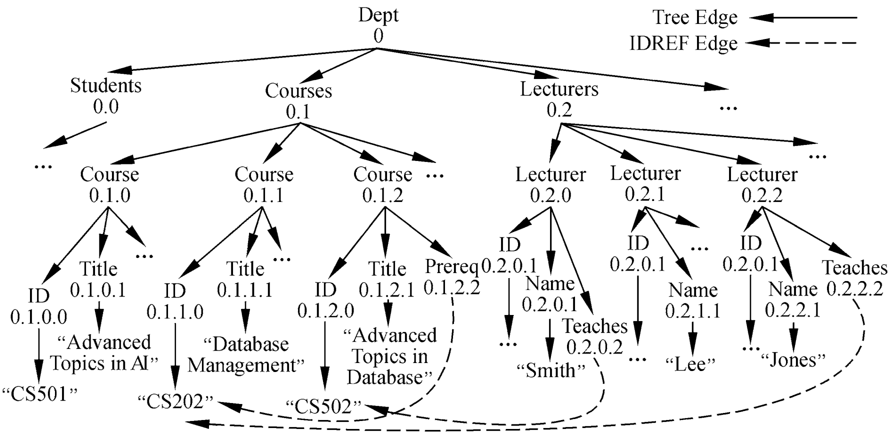


Fig. 8.1 Example XML document of computer science department (with Dewey IDs)

In this section, we will introduce a special graph, Tree + IDREF model, to capture the same ID references in digraph model which are missed in tree model and meanwhile to achieve better efficiency than general digraph model by distinguishing reference edges from tree edges in XML.

We first propose LRA pairs (lowest referred ancestor pairs) semantics. Informally, LRA pair semantics returns a set of lowest ancestor pairs such that each pair in the set is connected by ID references and the pair together covers all keywords in their subtrees. Then, we extend LRA pairs that are directly connected by ID references to node pairs that are connected via intermediate node hops by a chain of ID references, which we call ELRA pairs (extended lowest referred ancestor pairs). Finally, we further extend ELRA pairs to ELRA groups to define the relationships among two or more nodes which together cover all keywords and are connected with ID references.

8.2.2 Background and Data Model

8.2.2.1 ID References in XML

In many XML databases, ID references are present and play an important role in eliminating redundancies and representing relationships between XML elements. For example, in Fig. 8.1, references indicate important teach relationships between Lecturer and Course elements. Without ID references, the relationships has to be expressed in further nested structures (e.g., each lecturer is nested and duplicated in each course she/he teaches or vice versa), potentially introducing harmful redundancies.

8.2.2.2 Data Model

We model XML as special digraphs, Tree + IDREF, $G = (N, E, E_{\text{ref}})$, where N is a set of nodes, E is a set of tree edges, and E_{ref} is a set of ID reference edges between two nodes. Each node $n \in N$ corresponds to an XML element, attribute, or text value. Each tree edge denotes a parent–child relationship. We denote a reference edge from u to v as $(u,v) \in E_{\text{ref}}$. In this way, we distinguish the tree edges from reference edges in XML. The subgraph $T = (N, E)$ of G without ID reference edges, E_{ref}, is a tree. When we talk about parent–child (P-C) and ancestor–descendant (A-D) relationships between two nodes in N, we only consider tree edges in E of T.

8.2.3 Search Semantics

8.2.3.1 LRA Pair Semantics

Definition 8.1 (Reference Connection) Two nodes u and v with no A-D relationship in an XML database have a reference connection (or are reference-connected) if there is an ID reference between u or u' descendant and v or v' descendant.

Definition 8.2 (LRA Pairs) In an XML database, LRA pair semantics of a list of keywords K returns a set of unordered node pairs $\{(u_1, v_1),(u_2, v_2), \ldots,(u_m, v_m)\}$ such that for any (u_i, v_i) in the set:

1. u_i and v_i each covers some and together cover all keywords in K.
2. There is a reference connection between u_i and v_i.
3. There is no proper descendant u' of u_i (or v' of v_i) such that u' forms a pair with v_i (or v' forms a pair with u_i resp.) to satisfy conditions (1) and (2).

For example, there is a reference connection between nodes Lecturer: 0.2.0 and Course: 0.1.2 in Fig. 8.1 since there is an ID reference edge between their descendants (Teaches: 0.2.0.2 and ID: 0.1.2.0). Therefore, Lecturer: 0.2.0 and Course: 0.1.2 form an LRA pair for query "Smith Advanced Database," whose SLCA is the overwhelming root. Note reference-connected Lecturers: 0.2 and Courses: 0.1 do not form an LRA pair for "Smith Advanced Database" since their descendants already form a lower connected pair to cover all keywords.

8.2.3.2 ELRA Pair Semantics

Next we extend the direct reference connection in LRA pairs to a chain of connections as n-hop-connection in extended LRA (ELRA) pair semantics.

Definition 8.3 (N-Hop-Connection) Two nodes u and v with no A-D relationship in an XML database have an n-hop-connection (or are n-hop-connected) if there are n–1 distinct intermediate nodes $w_1, \ldots, w_{n-1}$ with no A-D pairs in $w_1, \ldots, w_{n-1}$ such that $u, w_1, \ldots, w_{n-1}, v$ form a chain of connected nodes by reference connection.

Definition 8.4 (ELRA Pairs) In an XML database, ELRA pair semantics of a list of keywords K returns a set of unordered node pairs $\{(u_1, v_1), (u_2, v_2), \ldots, (u_m, v_m)\}$ such that for any (u_i, v_i) in the set:

1. u_i and v_i each covers some and together cover all keywords in K.
2. There is an n-hop-connection between u_i and v_i for an upper limit L of the number of intermediate hops.
3. There is no proper descendant u' of u_i (or v' of v_i) such that u' forms a pair with v_i (or v' forms a pair with u_i resp.) to satisfy conditions (1) and (2).

ELRA pair semantics returns a set of pairs such that two nodes in each pair are lowest n-hop-connected to together cover all keywords. The upper limit (L) of the length of n-hop-connection can be determined at system tuning phase. Then, if users are interested in more results, the system can progressively increase L.

For example, in Fig. 8.1, Lecturer: 0.2.0 and Course: 0.1.1 are connected by a 2-hop-connection via node Course: 0.1.2. Thus, with L set as 2 or more, this pair will form an ELRA pair for query "Smith Database Management," which can be understood as Database Management is a prerequisite of the course that Smith teaches. We can see from this example that ELRA pair semantics has better chance to find smaller results than LRA pair semantics since LRA pairs are the lowest pairs with direct reference connection, while ELRA pairs are the lowest pairs with connections up to L intermediate hops including reference connection.

8.2.3.3 ELRA Group Semantics

We extend ELRA pair semantics to ELRA group semantics to define relationships among two or more connected nodes that cover all keywords.

Definition 8.5 (ELRA Groups) In an XML database, ELRA group semantics of K keywords returns a set of node groups $\{G_1, G_2, \ldots, G_m\}$s.t. $\forall G_i (1 \leq i \leq m)$:

1. All nodes in G_i each covers some and together cover all K keywords.
2. There is a node h_i to connect all nodes in G_i by n-hop-connection with up to L' as the number of intermediate hops; we call h_i as the hub for G_i.
3. There is no proper descendants of any node u in G_i to replace u to cover the same set of keywords as u and are n-hop-connected ($n \leq L'$) to the hub.
4. There is no proper descendant d of G_i's hub h_i such that d is the hub of another ELRA group G_d and $(G_d \cup \{h_i\}) \supseteq G_i$.

Similar to ELRA pairs, we can set the value for L' at system tuning phase for ELRA group semantics, and L' can be increased upon users' requests.

Compared to SLCA and ELRA pairs, ELRA groups can potentially find more and smaller nodes in the result. For example, in Fig. 8.1, with L' set as two, for query

"Jones Smith Database," the node group Course: 0.1.1, Course: 0.1.2, Lecturer: 0.2.0, and Lecturer: 0.2.2 form an ELRA group result. Both nodes Course: 0.1.1 and Course: 0.1.2 can be considered as the hubs in this ELRA group result.

8.3 LCRA, System Architecture, and Ranking Techniques

8.3.1 Tree Model

In tree model, smallest lowest common ancestor (SLCA) is an effective semantics. However, it cannot capture the ID references in XML data which reflect the relevance among objects, while digraph model can. In digraph model, a widely adopted semantics is to find the reduced subtrees (i.e., the smallest subtrees in a graph containing all keywords). However, enumerating results by increasing the sizes of reduced subtrees is an NP-hard problem, leading to intrinsically expensive solutions. Moreover, it neither distinguishes the containment and reference edge in XML data nor utilizes the database schema in defining matching semantics.

Besides, the existing approaches in both models have two common problems: first, regarding the design of matching semantics, they fail to effectively identify an appropriate information unit for result display to users. Neither SLCA (and its variants) nor reduced subtree is an appropriate choice, as neither of them is able to capture user's search target, as shown in Example 8.1. Second, the existing ranking strategies in both models are built at XML node level, which cannot meet user's search concern more precisely at object level.

Example 8.1
Query 1: "Suciu" is issued on the XML data in Fig. 8.2, intending to find papers written by "Suciu." Both SLCA and reduced subtree return the author nodes with value "Suciu," which is not informative enough to user.
Query 2: "Suciu XML" is issued on Fig. 8.2 to find XML papers written by Suciu. As there is no Suciu's paper containing "XML," the SLCA result is the whole subtree under the root node, which contains too much irrelevant information.

Next, we will introduce an XML keyword search system LCRA. Generally speaking, the technical challenges of the demo lie in as below:

1. The results need to be semantically meaningful to the user to precisely meet user's search needs and meanwhile avoid overwhelming the user with a huge number of trivial results. However, methods on graph model suffer from producing large number of trees containing the same pieces of information many times.
2. How to define appropriate matching semantics to find more relevant results by capturing ID references in XML data while optimizing the search efficiency.
3. How to design a general-purpose and effective ranking scheme.

By modeling XML data as a set of interconnected object trees, LCRA first automatically recognizes a set of objects of interest and the connections between them. Meanwhile, object trees as query results contain enough but non-overwhelming

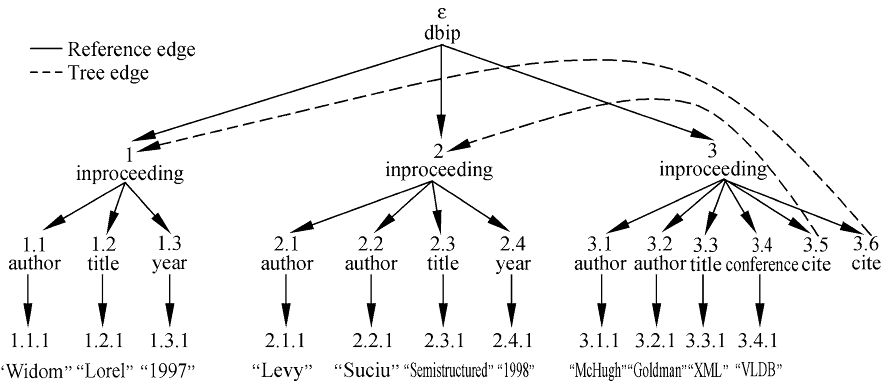


Fig. 8.2 Example XML document (with Dewey numbers)

information to represent a real world entity, so the problem of proper result display is solved. To capture user's search concern on a single object, LCA is proposed; to capture user's search concern on multiple objects, LRA pair (group) is proposed to find a pair (group) of object trees that are related via direct or indirect citation/reference relationships and together contain all keywords. That is, LRA helps find more relevant results. For Query 1 in Example 8.1, LCA returns inproceeding: 2 rather than its author subelement, which is both informative and relevant. For Query 2, LCA cannot find any qualified single inproceeding, while LRA finds a pair of inproceedings (inproceeding: 2, inproceeding: 3), where inproceeding: 2 written by "Suciu" is cited by inproceeding: 3 containing "XML."

8.3.2 Ranking Techniques

As LCA and LRA correspond to different user search needs, different ranking schemes are designed. To rank the LCA results, traditional TF*IDF similarity is extended to measure the similarity of an object tree's content w.r.t. the query. Besides considering the content, the structural information within the results is also considered: (1) Weight of matching elements in object tree. (2) Pattern of keyword co-occurrence. Intuitively, an object tree o is ranked higher if a nested element in o directly contains all query keywords, as they co-occur closely. (3) Specificity of matching element. Intuitively, an object tree *o* is ranked higher if an element nested in *o* exactly contains (all or some of) the keywords in *Q*, as *o* fully specifies *Q*. For example, for query "Won Kim," Won Kim's publications should be ranked before Dae-Won Kim's publications. To rank the LRA results, we combine the self-similarity of an LRA object o and the bonus score contributed from its LRA counterparts.

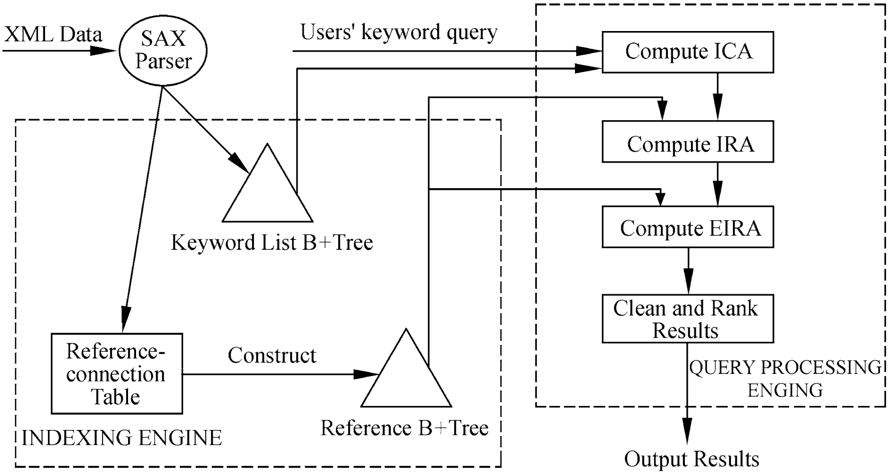


Fig. 8.3 System architecture

8.3.3 System Architecture

System architecture is shown in Fig. 8.3. During data preprocessing, the indexing engine parses the XML data, identifies the object trees, and builds the keyword inverted list storing for each keyword *k* a list of object trees containing *k*; it also captures ID references in XML data and stores them into reference connection table. A B^+ tree is built on top of these two indexes respectively. During the query processing stage, it retrieves the object trees containing the specified keywords to compute LCA results; then it computes LRA and extended LRA (ELRA) results with the help of reference connection table. Lastly, it cleans and ranks the results.

8.3.4 Overview of Online Demo Features

LCRA provides a concise interface user can explicitly specify their search concern—publications (default) or authors. LCRA offers various query flexibility: users can issue pure keyword queries that can be any combinations of words in full or partial specification of author names, topics, conference/journal, and/or year.

8.3.4.1 Search for Publications

Users can search for publications with various types of queries as below.

Author Name. For example, we can query "Jiawei Han" for his publications. LCRA will rank Jiawei Han's papers higher than papers coauthored by Jiawei and Han.

Multiple Author Names. To search for coauthored papers.

Topic. For example, we can query "XML query processing."

Topic by an Author. For example, we can query "Jim Gray transaction" for his publications related to transaction. Jim Gray's papers containing "transaction" are ranked before his papers citing or cited by "transaction" papers.

Topic of a Year. For example, we can query "keyword search 2006."

Conference and Author. For example, we can query "VLDB Raghu Ramakrishnan."

8.3.4.2 Search for Authors

Users can also search for authors with various types of queries as below:

Author Name. By typing an author name, LCRA returns this author followed by a ranked list of all his/her coauthors (e.g., try "Tova Milo").

Topic. We can search for authors who have the most contributions to a research topic (e.g., try "XML keyword search").

Conference/Journal Name. We can find active authors in a particular conference or journal (e.g., try "DASFAA").

Author Name and Topic/Year/Conference/Journal. Besides the author himself/herself, we can also search for his/her coauthors in a particular topic or year or conference/journal (e.g., we can search for Ling Tok Wang and his coauthors in DASFAA 2006 with a query "Ling Tok Wang DASFAA 2006").

8.3.4.3 Browsing

Besides searching, LCRA also supports browsing from search results to improve its practical usability. For example, users can click an author (or conference/journal) name in a result to see all publications of this author (or the proceeding/journal). Link is provided to find the references and citations of a paper. When searching for authors, we output both the number of publications containing all keywords and the number of publications that may be relevant via the reference connections.

8.3.4.4 Result Display

LCRA displays the result for LCA and LRA semantics separately in "AND" and "OR" part. In addition, since a same paper may appear in more than one LRA pair/group, it will annoy the user in result consumption if such paper appears many times. Therefore, LCRA only outputs one LRA object o for each LRA pair/group and provides links to the objects that form LRA pair/group with *o*.

8.3.4.5 Effectiveness of LCRA

In the demo, we will compare the result quality of LCRA with typical academic demo systems for DBLP, such as BANKS [KPC+05], ObjectRank [HHP06], and FacetedDBLP [CKKS05]. We will also compare LCRA with commercial systems such as Microsoft Libra and Google Scholar. LCRA has a good overall performance in terms of both result quality and query response time.

8.3.4.6 Feature Comparisons

A comparison of the features in existing demos is given from a user's perspective. BANKS produces results in form of reduced trees which is difficult for novice users to consume. Query types supported by ObjectRank and FacetedDBLP are not as flexible as LCRA. For example, they cannot handle searching papers of coauthors, or a topic by author. ObjectRank doesn't support browsing. DBLP CompleteSearch doesn't employ any relevance oriented ranking functions.

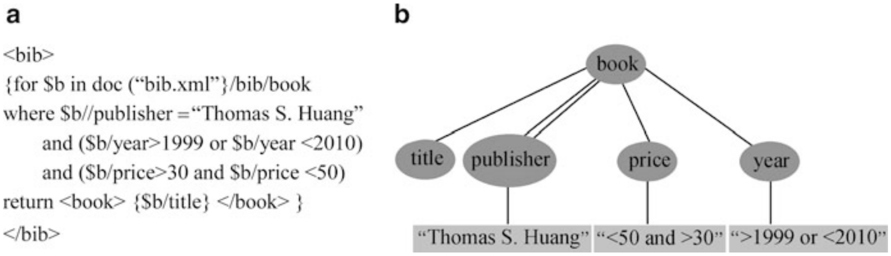
8.4 A Position-Aware XML Graphical Search System with Auto-completion

XML plays an important role in information exchange nowadays. As a result, a wide spectrum of users, including those with minimal or no computer programming skill at all, have the need to query hierarchical XML. Therefore, designing effective and efficient systems to simplify the query processing over XML documents attracts lots of research interests. In this section, we would propose an efficient XML search system to simplify the query processing and allow the users to search XML without any professional knowledge.

8.4.1 System Features

The well-known XML query languages (e.g., XQuery) are provided to process XML queries. However, these languages are far too complicated for unskilled users, who might only be aware of the basics of the XML data model or even lack the knowledge of the content (i.e., structural and textual information) of the XML documents.

For example, assume that a user wants to issue the following query: "List the title of books written by Thomas S. Huang and published before 1999 or later than 2010, and the price should be distributed in $30 \sim 50$ dollars". This query can be formulated as the XQuery expression in Fig. 8.4a. Unfortunately, formulating such query often demands considerable cognitive effort from the end users and requires



```
<bib>
{for $b in doc ("bib.xml"}/bib/book
where $b//publisher = "Thomas S. Huang"
      and ($b/year>1999 or $b/year <2010)
      and ($b/price>30 and $b/price <50)
return <book> {$b/title} </book> }
</bib>
```

Fig. 8.4 The XQuery and twig pattern expression of the query. (**a**) Xquery expression, (**b**) twig pattern query

programming skills that is at least comparable to SQL, which can be both time-consuming and error-prone. In order to deal with the problem, XML graphical languages are developed (e.g., XQBE [BC10], GLASS [NL05], XQE [BCK+10], XML-GL [CCD+00]) to allow the users, who do not know the professional query languages, to express queries. They allow users to create queries through simple graphical languages and then map the queries directly to XQuery in the background. However, (1) users are required to learn the syntax of the graphical languages; furthermore, (2) users need to have the knowledge of the structural (i.e., the parent–child (P-C) and ancestor–descendant (A-D) relation) and textual (i.e., node names and values) information of the XML documents, since the content of each node in the query should be input by users instead of the systems. For example, when issuing the query in Fig. 8.4a, the user needs to know the name of the publisher is "Thomas" rather than "Thomason" (i.e., textual information) and the *price* is a child of the *book* (i.e., structural information).

In order to simplify the query processing, (1) XML keyword search systems are proposed (e.g., XReal [BLL+10], XSeek [LWC+07]), which return the subtrees containing all the keywords. However, keywords can only express simple textual information but cannot describe the structural information and complex content. For example, these systems cannot answer the query in Fig. 8.4a, since keywords cannot describe the structures (e.g., *year* is a child of *book*) and the content conditions (e.g., "*year* > 1999 or *year* < 2010"). (2) Visual search systems are implemented (e.g., Xing[X03]). They present the structural and textual information of the document in visual interfaces, which allows the users to exploit the relationships of the elements and update the values directly. However, they need to load the whole document into memory and cannot answer the complex queries.

Reviewing the existing XML search systems, we can now derive some design goals for a user-friendly XML query system. First of all, we should not define another textual query language. Second, concrete XML syntax should be avoided. Third, since twig pattern query is a powerful concept that greatly supports the examination of structured data, pattern matching should be employed in the query system. Forth, twig patterns need to be extended to support complex queries. Finally, all the users including the novice users should be able to issue queries in the system without learning the query languages and the content of the XML documents.

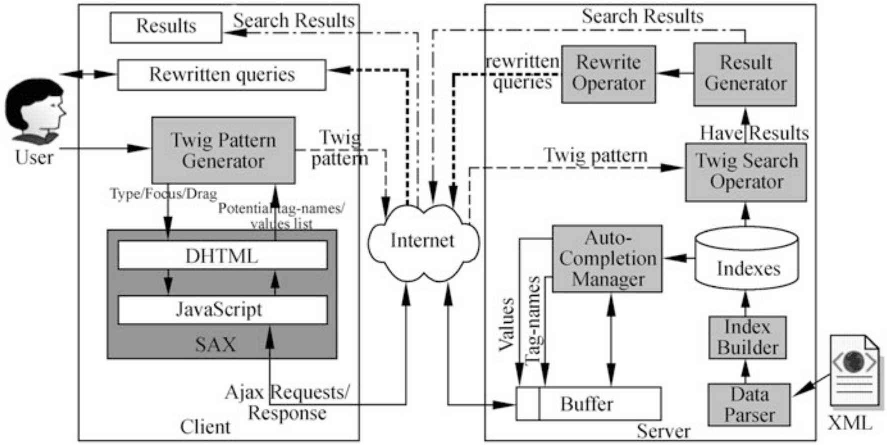


Fig. 8.5 Architecture of LotusX

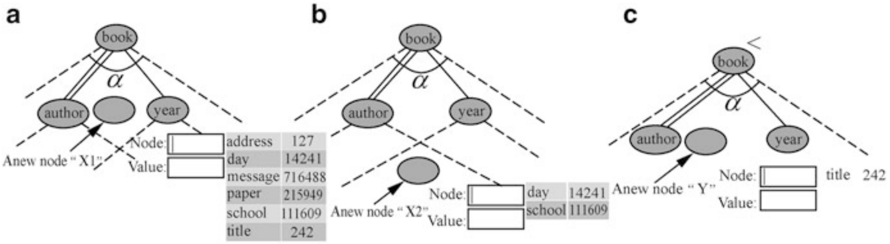
We propose a position-aware XML graphical search system with auto-completion (called LotusX). To our best knowledge, LotusX is the first system that applies position-aware and auto-completion features on XML query processing. LotusX has the following novel features compared to the existing XML search systems:

- It designs a *position-aware* graphical interactive interface to guide users to create twig pattern queries.
- It develops *auto-completion* feature based on two kinds of trie indexes to support search-as-you-type for both element tags and values.
- It applies a holistic twig pattern algorithm to answer twig pattern queries efficiently. In addition, it supports complex queries, including order-sensitive queries and queries with complex content predicates.
- It provides a novel ranking model to rank answers and a *relaxation similarity* to rewrite queries, which have no answer results.

8.4.2 LotusX: Architecture and Algorithms

8.4.2.1 Architecture

The system architecture of LotusX is presented in Fig. 8.5. The *data parser* parses the input XML data and schemas and labels the inherent elements, attributes, and values. Here we use region labeling [LLB+11], that is, (*start: end, level*), to present the position of a tree node in the data tree. The *index builder* constructs inverted indexes for efficiently answering the queries and two kinds of tries for providing



'X1' in the scope of book

'X2' in the scopes of book, author and year

order-sensitive query

Fig. 8.6 Position-aware feature. In (**a**), the number of the candidate tag names for the new node $X1$ is greater than that of $X2$ in (**b**), since $X2$ is affected by three nodes, while $X1$ is only affected by one. In (**c**) the candidate tag name is *title*, since author≪title≪year

an *auto-completion* feature. The *twig pattern generator* provides an interface for users to generate XML twig patterns graphically. During the generation of twig queries, the client issues AJAX requests on-the-fly to the server. Then the position-aware manager searches the candidates which satisfy the position information, and the *auto-completion manager* supports search-as-you-type by searching in the trie indexes to return the instant feedback. When users submit queries (including complex queries), the *twig search operator* employs a twig pattern algorithm to get the answers. Then *result generator* ranks the results according to their relevance and popularity and then demonstrates the results graphically as a form of trees. Finally, if there is no answer to match the user query, the *query rewriter* rewrites the original twig to generate new query candidates to help users easily reformulate queries.

8.4.2.2 Position-Aware Interactive Interface

A query is modeled as a small tree (i.e., twig pattern) in LotusX, since tree structure can well express both the structural and textual information. The twig pattern node labels include element tags, attribute values, and string values, and the query twig pattern edges are either parent–child edges (depicted using a single red line) or ancestor–descendant edges (depicted using a double green line). See an example twig pattern in Fig. 8.4b, which corresponds to the XQuery expression in Fig. 8.4a. LotusX allows novice users to generate a meaningful twig pattern through a graphically interactive interface, which can provide reasonable candidate tag names for a newly created node in various positions. See the query in Fig. 8.4b; when a new node is presented below the node "book", then the system automatically shows the candidate tag names, for example, "publisher" and "year." Note that the candidates must be children or descendants of "book" in the XML data. Otherwise, the query is meaningless. In order to specify such *position relationship*, we define the sector of α degree below each node as its scope (see the dotted lines in Fig. 8.6a). If a node A is in the scope of node $B \in q$, then we say A is affected by B, that is, the candidates of A should be all the potential descendant tag names of node B. In practice, a new node would be affected by multiple nodes in the query, and the nodes distribute in different levels, since the query is a twig. Therefore, we carefully design the strategy to generate reasonable candidates satisfying the above conditions. Assume a new node X is in the scope of node $n_{ij} \in q$, which is the jth node from left to right in the ith level of q; then the candidates of X can be computed as follows:

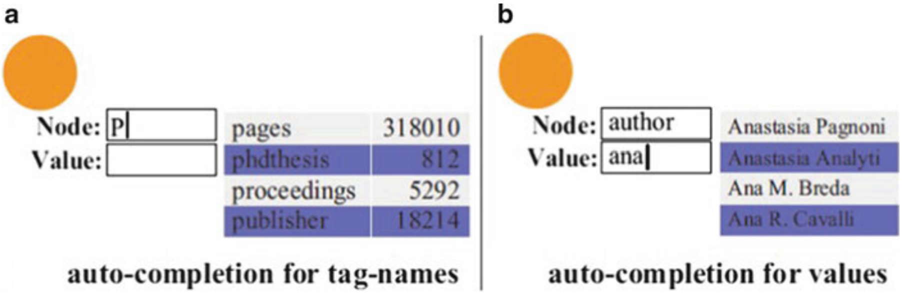
$$\cap_i \{\cup_j \mathrm{Desc}\left(n_{ij}\right)\}$$

where Desc(n) is a set containing all the children and descendant tag names of node n. To better understand this, let us see examples in Fig. 8.6a and b. In Fig. 8.6a, the candidate tag names of $X1$ are listed nearby the input box and each tuple in the list composed of a tag name and the total number of the occurrences of the tag name in the XML data set. The candidates are the children or descendants of *book*, since $X1$ is in the scope of *book*. However, in Fig. 8.6b, $X2$ is in the scope of three nodes (i.e., *book*, *author*, and *year*), and *book* has higher level than *author* and *year*. Thus the candidates of $X2$ is a set calculated by Desc(book)∩(Desc(author)∪Desc(year)).

8.4.2.3 Search-as-You-Type with Auto-completion

In information retrieval, query auto-completion is proposed to support search-as-you-type. As we discussed previously, LotusX suggests the reasonable candidates to users. However, the number of candidates might be large, and it is not elegant to list all the candidates to users. So we provide auto-completion to support search-as-you-type for both tag names and values from the candidates, which returns candidates on-the-fly as a user types letter by letter and gives the user instant feedback. Note the fact that tag names and values have different data characteristics, that is, the number of tag names is limited, while the values are usually too large to be completely loaded into the memory. Therefore, we design two trie indexes for them, respectively; they are *static tag-trie* for small tag data and *dynamic value-trie* for large value data. (1) We build a static tag-trie tree for all the tag names, which is not large and can be kept in the memory. Each tag name in the tree corresponds to a unique path from the root to a leaf node. We first find the corresponding trie node of the prefix, which is input by users, and then traverse the subtree to get the values with a prefix of the input message. For example, in Fig. 8.7a, user input a letter p; then the returned candidate tag names are all starting with p. (2) Due to the large size of the values and the values are related to different tag names, for example, "Anastasia Pagnoni" in Fig. 8.7b relates to the *author* rather than other tag names. Thus, we display a representative subset of values for each tag name, which can be loaded in the memory, to construct a dynamic value-trie. After selecting the tag name, the user focuses on the value input box. At this time, the client sends the tag name to the value-trie to locate the subtree rooted at the tag name.

In Fig. 8.7b, the value-trie locates the subtree of *author*. Once the users type any new letter in the value input box, value-trie returns the values belonging to the



Node: P
Value:

pages	318010
phdthesis	812
proceedings	5292
publisher	18214

Node: author
Value: ana

Anastasia Pagnoni
Anastasia Analyti
Ana M. Breda
Ana R. Cavalli

Fig. 8.7 The auto-completion for tag names and values. In (**a**), the candidate tag names starting with *p* are listed, and in (**b**), the candidate values of node *author* and starting with *ana* are chosen

tag name and starting with the letters by searching the corresponding path in the subtree. However, if there is no such value satisfying the requirement, we read the corresponding disk-resident file through the indexes and display in a recursive way, and each time we only read the representative subset of the rest data to build a value-trie.

8.4.2.4 Order-Sensitive Queries

To capture the semantics of XPath expressions with order axes, such as the following-sibling and preceding-sibling, we extend the common twig pattern by adding order constraint, in addition to P-C and A-D edges. Our paper [LLB+11] has the detailed discussion about the meaning of the symbol "<" (see Fig. 8.6c) and how to answer order-sensitive queries. In this chapter, we focus on how to automatically generate an order-sensitive query. Given two node types A and B in an XML database D, we say that type A precedes B written $A \ll B$, if there are two instance nodes a and b such that a precedes b in the depth first traverse of D. This kind of order information about node types can be obtained in *index builder* in preprocessing. See the example in Fig. 8.6b, c. Since b does not have the order constraint, the number of the candidate tag names for $X2$ in b is significantly larger than that for Y in c.

8.4.2.5 Result Ranking and Query Rewriting

We propose a ranking model to rank the results according to their importance and popularity. In addition, if there is no result based on users' initial query, we invoke a rewriting mechanism to provide refined queries. To rank a result r for a query q, we need to consider the structural and content factors to compute an overall relevance score. The followings are important factors: (1) weights of different tags, (2) length of each path, and (3) number of irrelative nodes. The following is a scoring function that reflects the above three factors. The score of a result *r* for a query *q* is defined as:

$$S_T(r,q) = \sum\nolimits_{(pq,pr)\in P} \left\{ \left(\frac{1+|pq|}{1+|pr|} \right) * \sum\nolimits_{pq\in Pq} \frac{\text{wf-idf}(t_{pq},r)}{\text{wf-idf}(t_{pq},q)} \right\} - \frac{\sum_{t_T\in T}\text{wf-idf}(t_T,r)}{\sqrt{\sum_{t_T\in r}\text{wf-idf}(t_T,r)^2}} \tag{8.1}$$

where wf-idf is similar to "XML tf-idf" in [BLC+09] and {*pq*, *pr*} is a mapping pair of paths from the query *q* and result *r*, respectively. Let *T* be the set of node types only in result r but not in query *q*, and *tpq*, *tr*, and *tT* are node types in path *pq*, result *r*, and set *T*, respectively. To understand the ST score, the first multiplier (i.e., (1 + |*pq*|)/(1 + |*pr*|)) actually addresses factor 2, while the second multiplier addresses factor 1 by computing the weights of the nodes. Finally, the third component reduces the score of the irrelative node types to address factor 3.

In order to automatically rewrite queries that have no results, our main strategy is to relax the conditions of the original query from the aspects of structural and/or value conditions. We define four basic relaxation operations from the lowest penalty to the highest, as follows: (1) remove the *order* constraint; (2) change P-C edge to A-D edges; (3) remove the value constraint; (4) prune the node with the smallest weight. Note that more relaxation operations result in less similarity with the original query.

Given an original query *Q* and a relaxation query *Q*′, we define the relaxation similarity (RS) to measure the similarity between *Q* and *Q*′:

$$\text{RS}(Q,Q') = \frac{1+\log|Q'(\text{D})|}{\sum_{i=1}^{|RT(QQ')|} 2^{w(i)}} \tag{8.2}$$

where $|Q'(D)|$ denotes the number of the answers to Q' in document *D*, |RT(*Q*, *Q*′)| represents the times of calling five basic operations to transform *Q* into *Q*′, and *w*(*i*) (>1) is the penalty of the relaxation operation *i*. In this way, a relaxation query *w*(*i*) is ranked higher if RS(*Q*, *Q*′) is greater, meaning that *Q*′ can return more answers and are more similar to the original query *Q*.

8.5 Summary

In this chapter, we design two XML systems, that is, LCRA [BLL+10] and LotusX [LLL+12]. In particular, LCRA is an XML keyword search system, which takes advantage of the schema knowledge to define the matching semantics and facilitate the result display and also performance optimization in terms of result quality and efficiency. LotusX is a graphical XML search system with auto-completion and rewriting features. LotusX provides a position-aware graphical interface to guide users to create queries and supports auto-completion feature to support search-as-you-type for both element tags and values. Also, it provides a novel ranking model to rank answers and a relaxation similarity to rewrite queries, which have no answer results.

References

[BC10] Braga, D., Campi, A.: a graphical environment to query xml data with XQuery. In: WISE, Roma, pp. 31–40 (2003)

[BCK+10] Borkar, V.R., Carey, M.J., Koleth, S., Kotopoulis, A., Mehta, K., Spiegel, J., Thatte, S., Westmann, T.: Graphical XQuery in the aqualogic data services platform. In: SIGMOD, Indianapolis, pp. 1069–1080 (2010)

[BLC+09] Bao, Z., Ling, T.W., Chen, B., Lu, J.: Effective XML keyword search with relevance oriented ranking. In: ICDE, Shanghai (2009)

[BLL+10] Bao, Z., Lu, J., Ling, T.W.: Xreal: an interactive xml keyword searching. In: CIKM, Toronto, pp. 1933–1934 (2010)

[CCD+00] Ceri, S., Comai, S., Damiani, E., Fraternali, P., Tanca, L.: Complex queries in XML-GL. In: SAC, Villa Olmo, pp. 888–893 (2000)

[CKKS05] Cohen, S., Kanza, Y., Kimelfeld, B., Sagiv, Y.: Interconnection semantics for keyword search in XML. In: Proceedings of CIKM Conference, Bremen, pp. 389–396 (2005)

[GSBS03] Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRANK: ranked keyword search over XML documents. In: SIGMOD, San Diego (2003)

[HHP06] Hwang, H., Hristidis, V., Papakonstantinou, Y.: Objectrank: a system for authority-based search on databases. In: VLDB, Seoul, Korea (2006)

[KPC+05] Kacholia, V., Pandit, S., Chakrabarti, S., Sudarshan, S., Desai, R.: Bidirectional expansion for keyword search on graph databases. In: VLDB, Trondheim (2005)

[KS05] Kimelfeld, B., Sagiv, Y.: Efficiently enumerating results of keyword search. In: Proceedings of DBPL conference, Trondheim, pp. 58–73 (2005)

[LLB+11] Lu, J., Ling, T.W., Bao, Z., Wang, C.: Extended xml tree pattern matching: theories and algorithms. IEEE Trans. Knowl. Data Eng. **23**(3), 402–416 (2011)

[LLL+12] Lin, C., Lu, J., Ling, T.W., Bogdan, C.: LotusX: a position-aware XML graphical search system with auto-completion. In: ICDE, Washington, DC (2012)

[LWC+07] Liu, Z., Walker, J., Chen, Y.: XSeek: a semantic XML search engine using keywords. In: VLDB, Vienna, pp. 1330–1333 (2007)

[LYJ04] Yunyao Li, Cong Yu, Jagadish, H.V.: Schema-free XQuery. In: VLDB, Toronto, pp. 72–83 (2004)

[NL05] Ni, W., Ling, T.: Translate graphical XML query language to SQLX. In: Proceedings of DASFAA, Beijing, pp. 907–913 (2005)

[SKW01] Schmidt, A., Kersten, M.L, Windhouwer, M.: Querying XML documents made easy: Nearest concept queries. In: ICDE, Heidelberg, pp. 321–329 (2001)

[X03] Erwig, M.: Xing: a visual xml query language. J. Vis. Lang. Comput. **14**(1), 5–45 (2003)

[XP05] Xu, Y., Papakonstantinou, Y.: Efficient keyword search for smallest LCAs in XML databases. In: SIGMOD, Baltimore, pp. 537–538 (2005)

Chapter 9
Summary and the Road Ahead

Abstract In this chapter, we summarize this book and present some issues which need to be further investigated: including full-fledged XML query engine, directed graph XML model, extended Dewey labeling scheme for ordered query, index structure based on TJFast, and MapReduce-based XML twig pattern matching.

Keywords Full-fledged • MapReduce

9.1 Summary of This Book

According to the content and the relevance of the nine chapters, we can make a summary for this book as follows:

Chapter 1 is employed as an introduction of basic XML, including the emergence of XML database, XML data model, and searching and querying XML data, to give a general view of XML for the readers who don't have idea of XML.

Chapters 2, 3, 4, and 5 focus on XML tree pattern query processing, which is a core operation in searching XML. In particular, we propose a number of labeling schemes for XML data in Chap. 2 to facilitate query process over XML data, which conforms to an ordered tree-structure data model efficiently. We propose several path indexes in Chap. 3 to overcome the inefficiency caused by exhaustive traversal on XML data. In Chap. 4, we introduce two kinds of join algorithms, both of which play significant roles in XML twig pattern matching, and present the solutions to speed up query processing and reduce the intermediate results. In Chap. 5, we propose a set of holistic algorithms to efficiently process the extended XML tree patterns, which contain wildcards, negation function, and order restriction, all of which are frequently used in XML query languages.

We study the XML keyword search in Chaps. 6, 7, and 8, which is a proven user-friendly way of querying XML documents for the users with no professional knowledge. In particular, we present a survey on the existing XML keyword search semantics algorithms and ranking strategy in Chap. 6. In order to address the

J. Lu, *An Introduction to XML Query Processing and Keyword Search*,
DOI 10.1007/978-3-642-34555-5_9,

problem of empty or meaningless results in keyword search, which is caused by irrelevant or mismatched terms or typos in user queries, we introduce the problem of content-aware XML keyword query refinement and design a novel content-aware XML keyword query refinement framework in Chap. 7. In Chap. 8, we present two native XML search systems, that is, LCRA and LotusX, where LCRA is a system supporting keyword search and returning meaningful searching results, while LotusX provides an auto-completion feature in a graphical search interface to simplify the processing of twig search.

Finally, in this chapter (Chap. 9), we summarize this book and present several future works, such as graphical XML data processing, complex XML pattern matching, and MapReduce-based XML query processing.

9.2 Future Work

While we have presented efficient algorithms for XML twig pattern matching, a number of issues need to be further investigated.

9.2.1 Full-Fledged XML Query Engine

Although there have been some encouraging results (i.e., efficient twig query matching algorithms) in XML query processing reported in this book, there is still much work to be done toward a full-fledged XML query engine. For example, the first step can be done to extend our algorithms to optimally process the XPath [BBC04] with positional predicates (e.g., book/author [AJP+02]). The existing method solves this problem by decomposing such a query to different axes, which may cause large useless intermediate results. We are considering to extend our TJFast [LLC+05] based on extended Dewey labeling scheme to design a new holistic algorithm. Specifically, we plan to extend the set structure in TJFast to contain the positional information for elements. Thus any element can be output from set only if it satisfied the respective positional predicate. More research can be done on concise and efficient presentation of positional information in sets.

9.2.2 Directed Graph XML Model

While XML data can be modeled as tree structure, further study can be conducted to handle labeled directed graph models [CCD+99, CLO03]. This is because the labeled directed graph model of XML data contains more information (i.e., ID references) than tree models do. We expect to use ID reference lists to record the reference information for each element in XML documents. By the combination of ID reference lists and extended Dewey labels, we can know both the reference and path information for each element in XML documents. But the main problem is that we need extra computation cost on ID reference lists to answer an XML query based on graph model. How to efficiently organize data in ID reference list for efficient query processing is the main concern of future research.

9.2.3 Extended Dewey Labeling Scheme for Ordered Query

In this book, based on the containment labeling scheme, we have proposed a holistic algorithm OrderedTJ [LLY+05] for ordered twig query. The future research could be done on algorithms based on the extended Dewey labeling scheme for ordered twig query. This is a promising research direction because (1) similar to the containment labeling scheme, extended Dewey also can be used to compare the element order. For example, "1.5" is a left sibling of "1.8", and (2) analysis and experiment results in Chap. 6 have showed that algorithms based on extended Dewey have much better performance in terms of execution time than algorithms on the containment labeling scheme. Our research plan includes two steps.

- Firstly, we need to extend the set structure of TJFast to contain the order information of elements. Before any element is inserted to a set in TJFast, we need to check its order restriction to see whether we can find the proper following and preceding elements.
- Secondly, we need to modify the output condition of TJFast. In the original algorithm, any element can be output if it satisfies the parent–child or ancestor–descendant relationship in query, but, to handle ordered query, we additionally check its order relationship according to respective set contents.

9.2.4 Index Structure Based on TJFast

We expect to accelerate our algorithm by using some index structures such as B+-tree on the individual data streams. We believe the extension of TJFast can make the use of B+ indexes to skip elements and improve the algorithm performance. An immediate thought is that we can arrange the data in each stream by their lexicography order of extended Dewey labels and skip elements by using similar strategy in TwigStackXB [BKS02]. On the other hand, more research also can be done on the indexes for the contents data (i.e., character data) in XML documents. This book mainly focuses on the structural part of XML queries. But an XML query normally includes both structural and content searches (i.e., text search). For example, in query "book[author='Jimmy']/title," "author='Jimmy'" is a content search, while "book[author]/title" is a structural search. To answer a content search, the existing papers use the inverted lists to map words in document to their positions and create B+ tree on words for efficient search. In this book, we have proposed tag + level and prefix path streaming scheme for efficient partitioning nodes to streams (i.e., inverted lists). Thus, a direction of future work is to investigate the index structure which can be used for both contents and structural parts of queries by integrating different types of indexes.

9.2.5 MapReduce-Based XML Twig Pattern Matching

While we can process XML twig pattern query quite efficiently in the centralized environment, the evaluation of big XML data in the magnitudes of TB and PB is still a challenge. In the future work, we will propose a MapReduce-based twig pattern matching algorithm with the XML document indexed by extended Dewey encoding [LLC+05], which is discussed in Chap. 2.

References

[AJP+02] Al-khalifa, S., Jagadish, H.V., Patel, J.M., Wu, Y., Koudas, N., Srivastava, D.: Structural joins: a primitive for efficient XML query pattern matching. In: Proceedings of the 20th International Conference on Data Engineering, pp. 141–152 (2002)

[BBC04] Berglund, L., Boag, S., Chamberlin, D.: XML path language (XPath) 2.0, W3C Working Draft 23 July 2004

[BKS02] Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: Optimal XML pattern matching. Technical Report, Columbia University, Columbia (2002)

[CCD+99] Ceri, S., Comai, S., Damiani, E., Fraternali, P., Paraboschi, S., Tanca, L.: XML-GL: a graphical language for querying and restructuring XML documents. In: Proceedings of the Eighth International World Wide Web Conference, May 1999

[CLO03] Chen, Q., Lim, A., Ong, K.W.: D(k)-index: an adaptive structural summary for graph-structured data. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 134–144 (2003)

[LLC+05] Lu, J., Ling, T.W., Chan, C., Chen, T.: From region encoding to extended Dewey: on efficient processing of XML twig pattern matching. In: Proceedings of 31th International Conference on Very Large Data Bases (VLDB), pp. 193–204 (2005)

[LLY+05] Lu, J., Ling, T.W., Yu, T., Li, C., Ni, W.: Efficient processing of ordered XML twig pattern matching. In: DEXA, pp. 300–309 (2005)

Index

DOI 10.1007/978-3-642-34555-5,