ABSTRACT

SEMANTICS AND EFFICIENT EVALUATION OF PARTIAL TREE-PATTERN QUERIES ON XML

by
Xiaoying Wu

Current applications export and exchange XML data on the web. Usually, XML data are queried using keyword queries or using the standard structured query language XQuery the core of which consists of the navigational query language XPath. In this context, one major challenge is the querying of the data when the structure of the data sources is complex or not fully known to the user. Another challenge is the integration of multiple data sources that export data with structural differences and irregularities. In this dissertation, a query language for XML called Partial Tree-Pattern Query (PTPQ) language is considered. PTPQs generalize and strictly contain Tree-Pattern Queries (TPQs) and can express a broad structural fragment of XPath. Because of their expressive power and flexibility, they are useful for querying XML documents the structure of which is complex or not fully known to the user, and for integrating XML data sources with different structures. The dissertation focuses on three issues. The first one is the design of efficient non-main-memory evaluation methods for PTPQs. The second one is the assignment of semantics to PTPQs so that they return meaningful answers. The third one is the development of techniques for answering TPQs using materialized views.

Non-main-memory XML query evaluation can be done in two modes (which also define two evaluation models). In the first mode, data is preprocessed and indexes, called inverted lists, are built for it. In the second mode, data are unindexed and arrives continuously in the form of a stream. Existing algorithms cannot be used directly or indirectly to

efficiently compute PTPQs in either mode. Initially, the problem of efficiently evaluating partial path queries in the inverted lists model has been addressed. Partial path queries form a subclass of PTPQs which is not contained in the class of TPQs. Three novel algorithms for evaluating partial path queries including a holistic one have been designed. The analytical and experimental results show that the holistic algorithm outperforms the other two. These results have been extended into holistic and non-holistic approaches for PTPQs in the inverted lists model. The experiments show again the superiority of the holistic approach. The dissertation has also addressed the problem of evaluating PTPQs in the streaming model, and two original efficient streaming algorithms for PTPQs have been designed. Compared to the only known streaming algorithm that supports an extension of TPQs, the experimental results show that the proposed algorithms perform better by orders of magnitude while consuming a much smaller fraction of memory space.

An original approach for assigning semantics to PTPQs has also been devised. The novel semantics seamlessly applies to keyword queries and to queries with structural restrictions. In contrast to previous approaches that operate locally on data, the proposed approach operates globally on structural summaries of data to extract tree patterns. Compared to previous approaches, an experimental evaluation shows that our approach has a perfect recall both for XML documents with complete and with incomplete data. It also shows better precision compared to approaches with similar recall.

Finally, the dissertation has addressed the problem of answering XML queries using exclusively materialized views. An original approach for materializing views in the context of the inverted lists model has been suggested. Necessary and sufficient conditions have been provided for tree-pattern query answerability in terms of view-to-query homomorphisms. A time and space efficient algorithm was designed for deciding query

answerability and a technique for computing queries over view materializations using stack-based holistic algorithms was developed. Further, optimizations were developed which (a) minimize the storage space and avoid redundancy by materializing views as bitmaps, and (b) optimize the evaluation of the queries over the views by applying bitwise operations on view materializations. The experimental results show that the proposed approach obtains largely higher hit rates than previous approaches, speeds up significantly the evaluation of queries without using views, and scales very smoothly in terms of storage space and computational overhead.

SEMANTICS AND EFFICIENT EVALUATION OF PARTIAL TREE-PATTERN QUERIES ON XML

by
Xiaoying Wu

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science

January 2010

APPROVAL PAGE

SEMANTICS AND EFFICIENT EVALUATION OF PARTIAL TREE-PATTERN QUERIES ON XML

Xiaoying Wu

12/8/2009

Dimitri Theodoratos, PhD, Dissertation Advisor — Date
Associate Professor, Computer Science, NJIT

12/8/2009

James Geller, PhD, Committee Member — Date
Professor, Computer Science, NJIT

12/8/2009

Alexandros V. Gerbessiotis, PhD, Committee Member — Date
Associate Professor, Computer Science, NJIT

12/8/2009

Vincent Oria, PhD, Committee Member — Date
Associate Professor, Computer Science, NJIT

12/8/09

Michael Halper, PhD, Committee Member — Date
Associate Professor, Computer Science, Kean University

BIOGRAPHICAL SKETCH

Author: Xiaoying Wu

Degree: Doctor of Philosophy

Date: January 2010

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, Newark, New Jersey, 2010
- Master of Science in Computer Science,
  National University of Singapore, Singapore, 2002
- Bachelor of Science in Computer Science,
  Central South University, China, 1995

Major: Computer Science

Publications:

Xiaoying Wu, Dimitri Theodoratos, Wendy Hui Wang, "Answering XML Queries Using Materialized Views Revisited", *Proc. of the 18th ACM Conference on Information and Knowledge Management (CIKM'09)*, Hong Kong, Nov. 2009, ACM Press, pages 475-484.

Xiaoying Wu, Dimitri Theodoratos, Stefanos Souldatos, Theodore Dalamagas, Timos Sellis, "Efficient Evaluation of Generalized Tree-Pattern Queries with Same-Path Constraints", *Proc. of the 21st Intl. Conference on Scientific and Statistical Database Management (SSDBM'09)*, New Orleans, LA, June 2009, Springer, Lecture Notes in Computer Science, pages 361-379.

Dimitri Theodoratos, Xiaoying Wu, "Eager Evaluation Generalized Tree-Pattern Queries on XML Streams", *Proc. of the 14th Intl. Conference on Database Systems for Advanced Applications (DASFAA'09)*, Brisbane, Australia, April 2009, Springer, Lecture Notes in Computer Science, pages 241-246.

Xiaoying Wu, Dimitri Theodoratos, "Evaluating Partial Tree-Pattern Queries on XML Streams", *Proc. of the 17th ACM Intl. Conference on Information and Knowledge Management (CIKM'08)*, Napa Valley, CA, Oct. 2008, ACM Press, pages 1409-1410.

Xiaoying Wu, Stefanos Souldatos, Dimitri Theodoratos, Theodore Dalamagas, Timos Sellis, "Efficient Evaluation of Generalized Path Pattern Queries on XML Data", *Proc. of the 17th Intl. World Wide Web Conference (WWW'08)*, Beijing, China, Apr. 2008, ACM Press, pages 835-844.

Dimitri Theodoratos, Xiaoying Wu, "Assigning Semantics to Partial Tree-Pattern Queries", *Data and Knowledge Engineering (DKE) Journal*, Vol. 64, Issue 1, Jan. 2008, Elsevier Science, pages 242 - 265.

Stefanos Souldatos, Xiaoying Wu, Dimitri Theodoratos, Theodore Dalamagas, Timos Sellis, "Evaluation of Partial Path Queries on XML Data", *Proc. of the 16th ACM Intl. Conference on Information and Knowledge Management (CIKM'07)*, Lisboa, Portugal, 2007, ACM Press, pages 21-30.

Wugang Xu, Dimitri Theodoratos, Calisto Zuzarte, Xiaoying Wu,Vincent Oria, "A Dynamic View Materialization Scheme for Sequences of Query and Update Statements", *Proc. of the 9th Intl. Conference on Data Warehousing and Knowledge Discovery (DaWaK'07)*, Regensburg, Germany, Sep. 2007, Springer, Lecture Notes in Computer Science, pages 55-65.

Dimitri Theodoratos, Xiaoying Wu, "An Original Semantics to Keyword Queries for XML Using Structural Patterns", *Proc. of the 12th Intl. Conference on Database Systems for Advanced Applications (DASFAA'07)*, Bangkok, Thailand, April 2007, Springer, Lecture Notes in Computer Science, pages 727-739.

Dimitri Theodoratos, Xiaoying Wu, "Integration of XML Data Sources with Structural Differences", *Proc. of the Intl. Workshop on Scalable Web Information Integration and Service (SWIIS'07)*, Bangkok, Thailand, April 2007, World Scientific, pages 95-104.

Xiaoying Wu, Tok Wang Ling, Sin Yeung Lee, Mong Li Lee, Gillian Dobbie, "NF-SS: A Normal Form for Semistructured Schemata", *Proc. of the Intl. Workshop on Data Semantics in Web Information Systems (DASWIS'01)*, Yokohama, Japan, Nov. 2001, Springer-Verlag, pages 292-305.

Xiaoying Wu, Tok Wang Ling, Mong Li Lee, Gillian Dobbie, "Designing Semistructured Databases Using the ORA-SS Model", *Proc. of the 2nd Intl. Conference on Web Information Systems Engineering (WISE'01)*, Kyoto, Japan, Dec. 2001, IEEE Computer Society, pages 171-180.

To my beloved mother.

ACKNOWLEDGMENT

I am deeply indebted to my supervisor Professor Dimitri Theodoratos. He made it possible for me to pursue my PhD at New Jersey Institute of Technology (NJIT) five years ago while I was a master student at San Jose State University. I thank him for the continued guidance and support throughout my years at NJIT. I am grateful for the numerous meetings that he had with me to discuss my research and to guide me through it with his profound knowledge in many diverse areas. I feel even more indebted to him for the countless hours that he spent on improving my writing. In addition to his advice on technical issues, he has helped me in many other areas, including giving presentations, working with people, balancing life and work, strategies for job search, and the list goes on. I can never thank him enough for the positive impact that he has had on my life.

I am grateful to Professor James Geller, Professor Alexandros Gerbessiotis, Professor Michael Halper, and Professor Vincent Oria, who served as my dissertation committee members and provided me with so much useful feedback. In particular, I want to thank Professor James Geller and Professor Vincent Oria who also provided me with information on my academic career and wrote letters of recommendation for me.

I would like to thank my collaborator Theodore Dalamagas with whom I worked on the project of flexible querying of XML data. I have also been very fortunate to work with Stefanos Souldatos in the last three years. Part of my thesis has benefited from our animated discussions, where different ways of thinking and different approaches to problems were shared. I also want to thank Professor Wendy Hui Wang with whom I worked on answering XML queries using materialized views and on archiving XML data.

Special thanks are given to Calisto Zuzarte at the IBM Toronto Laboratory, Canada. He was my supervisor when I was a visiting student at IBM Toronto Laboratory from July to August, 2007, and provided me with a wonderful lab space. His support made my stay in Canada an enjoyable and fruitful experience.

I would like to thank the administrative staff in the Computer Science department of NJIT for their collaboration and help since my first day there.

My appreciation also goes to other members of the Database and Data Integration lab at NJIT. I acknowledge Wugang Xu, Pawel Placek, Jichao Sun, Sheetal Rajgure for their feedback on my work, stimulating discussions and great companionship.

I remember my beloved father, who bestowed on me the deepest love, which I will recall forever. Although he has been gone for ten years, I can feel his presence and encouragement whenever I am in difficulty.

Last but not least, I am deeply indebted to my mother. Her tremendous sacrifice, love, support and guidance will accompany me as I journey to the end. I dedicate this dissertation to her.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Chapter **Page**

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

Chapter Page

LIST OF FIGURES

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

LIST OF FIGURES
(Continued)

Figure **Page**

CHAPTER 1

INTRODUCTION

Nowadays, massive amounts of data are published on the web on a daily basis from various sources. Inevitably, web data is becoming increasingly heterogeneous. Extensible Markup Language (XML) is by now the de facto standard for exporting and exchanging data on the web due to its *semi-structured* characteristics: its inherent self-describing capability, and its flexibility of organizing data [1]. As increasing amounts of information are stored, exchanged, and presented using XML, it becomes increasingly important to effectively and efficiently query XML data sources. The lack of precise yet flexible exploration tools to query XML data sources directly impacts the usability and maintainability of the information contained in the web data.

1.1 Tree-Pattern Queries for XML

In the XML model, data is represented in a tree structured form.Query languages for XML are mainly based on the specification of structural patterns to be matched against the data tree. In practice, these structural patterns are specified using XPath [2], a language that lies at the core of the standard XML query language XQuery [3]. Usually, the structural patterns are in the form of trees (Tree-Pattern Queries – TPQs).

1.1.1 The Problem of Query Dependency from Data Structures

The semi-structured XML data does not have to comply with a schema. Even if the data comply with some schema, its structure could be complex [4] or might not be fully known

to the user [5, 6, 7, 8]. Formulating a TPQ that will retrieve the desired results becomes complex. The reason is that, as what will be explained below, the user has to specify an order for the elements in every path of a tree-pattern query (TPQ) even though (a) the user might not know this order, and (b) the user might not be interested in imposing an order as a structural restriction in the query. Further, data sources usually export data on the web under different structures even if they export the same information or information from the same knowledge domain. Since elements may be ordered differently in these structures, querying all these data sources in an integrated way becomes an issue: usually, a single TPQ is not able to retrieve the desired information from all of them [9, 5, 6, 7]. Then, the user might have to specify a number of tree patterns, which in some cases can be exponential on the number of elements in the query [10, 11].

Example 1.1.1 *Consider an XML bibliography which contains several datasets on books. These datasets organize books differently, grouped either by* publisher, *or by* year, *or by* author, *or by* subject. *Suppose that the user wants to find the title of a book on the subject XML published by O'Reilly in 2008). In addition, the user would like to impose the following structural restrictions: (1)* author *is the child of* book*; (2)* year *and* publisher *are ancestors of* book*; and* subject *is either an ancestor or a descendant of* book. *It is not difficult to see that the requirements* cannot *be expressed by one TPQ, but they need a* set *of TPQs. The set of eight TPQs for the above requirements are shown in Figure 1.1. Node labels are abbreviated as shown in the figure. Double arrows indicate descendant relationships and single arrows indicate child relationships. Each query pattern in the figure represents two TPQs. If a node has a label of the form* $V(U)$, V *is the label of the*

node in the first TPQ and U is the label of the node in the second TPQ. For simplicity, value predicates ('XML', 'O'Reilly', and '2008') were omitted in the query patterns .

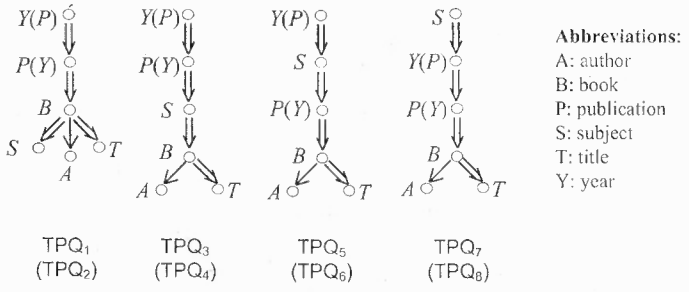


Figure 1.1 A set of eight TPQs for the query requirements in Example 1.1.1

TPQs constitute a very restricted fragment of XPath that involves forward axes (child and descendant), wildcards, and predicates. For instance, the TPQs shown in Figure 1.1 involve descendant relationships which denote a path of zero or more elements between two elements, and child relationships which indicate zero elements between two elements. A TPQ can also have wildcard nodes (labeled by a '*') which indicate the presence of some element in a path without specifying which one.

Even though TPQs provide some freedom in the specification of a tree structure (e.g., through the use of descendant relationships and wildcards), they all have a common restrictive structural requirement: *in every root-to-leaf path, there is a total order for the nodes*. It is not possible in a TPQ to indicate that two nodes n_1 and n_2 occur in a path without specifying a precedence relationship between them: node n_1 has to precede node n_2 or vice-versa.

1.1.2 Limitations of Previous Approaches

Two different solutions have been suggested to deal with the dependence of queries for XML from the structure of the data: one that adapts the unstructured keyword-search

techniques to query tree-structured data [12, 9, 13, 14], and one that extends structured TPQ languages with keyword search capabilities [15, 16, 5, 6].

The structureless keyword-based solution. The first solution modifies keyword-based techniques used by search engines for HTML to distinguish between text (data) and elements (metadata). It also modifies these techniques to return fragments of the documents that contain the keywords, as is appropriate for XML, instead of links to documents [12, 9, 13, 14]. This solution offers a very convenient way for specifying queries, even for a naive user. Nevertheless, its major limitation is that structural restrictions cannot be specified in the query. Structural restrictions are necessary when querying tree-strcutured data for two reasons: (a) they can express user requirements and therefore, refine the query answer, and (b) they can express structural constraints that are known to hold in order to speed up the evaluation of the queries.

The structural queries with keyword search extension solution. The second solution is applied to extend structured query languages for XML to enable keyword search [15, 16, 5, 6]. However, these languages cannot avoid having a syntax which is complex for the simple user [13, 9, 10, 11].

1.2 Partial Tree-Pattern Query Language

In this dissertation, a query language for XML, called Partial Tree-Pattern Query (PTPQ) language, was considered. PTPQs were initially introduced in [7]. This language addresses the problem of query dependence from the structure by allowing a *partial* specification of tree patterns in queries. PTPQs generalize and strictly contain TPQs. They are flexible enough to allow a large range of queries from keyword-style queries with no structure,

to keyword queries with arbitrary structural constraints, to fully specified TPQs. PTPQs are not restricted by a total order for the nodes in a path of the query pattern since they can constrain a number of (possibly unrelated) nodes to lie on the same path (*same-path* constraint). These nodes together form a *partial path*. PTPQs can express XPath queries with the reverse axes parent and ancestor, in addition to forward child and descendant axes and branching predicates. They can also express the identity equality (is) operator of XPath (sharing of a node by two partial paths) by employing *node sharing expressions*. Overall, PTPQs represent a broad fragment of XPath which is very useful in practice.

To provide some intuition, consider again the query requirements in Example 1.1.1. They can be easily specified by a PTPQ shown in Figure 1.2. The PTPQ has two (partial) paths surrounded by dotted lines. The nodes in a partial path are not necessarily related through a total order: *subject* can precede *book* in a path or vice versa. Undirected edges labeled by the symbol $\approx$ between two nodes from different partial paths indicate that these two nodes coincide (that is, these two nodes denote a single node shared by the two partial paths). For instance, in Figure 1.2, the two nodes labeled by *book* denote a common node of partial paths 1 and 2. Note that this same query can be used to retrieve results from different datasets which structure their data differently.

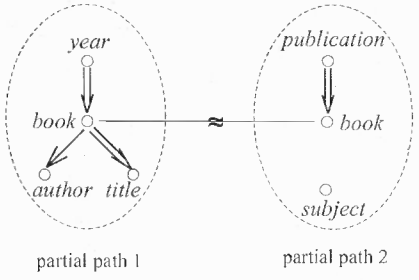


Figure 1.2 A PTPQ for the query requirements in Example 1.1.1

1.3 Focus of the Investigation

This doctoral investigation focuses on three issues. The first one is the efficient evaluation of PTPQs on XML data. The second one is on assigning semantics to PTPQs so that they return to the user meaningful answers (that is, the queries are not matched to unrelated parts of the XML document). The third one is on answering PTPQs on XML data using materialized views. The investigation on the third issue started with TPQs which is the restricted subclass of PTPQs.

1.3.1 Evaluation Issue

Finding all the matches of structural patterns in an XML tree is a key operation in XML query processing. A recent approach for evaluating queries on XML data assumes that the data is preprocessed and the position of every node in the XML tree is encoded [17, 18, 19, 20, 21]. Further, the nodes are partitioned, and an index of inverted lists called *streams* is built on this partition. In order to evaluate a query, the nodes of the relevant streams are read in the pre-order of their appearance in the XML tree. Every node in a stream can be read *only once*. In the dissertation, this evaluation model is referred to as *indexed streaming* model. Algorithms in this model [18, 19, 20, 22, 23, 21, 24, 25, 26, 27] are based on stacks that allow encoding an exponential number of pattern matches in a polynomial space. Another evaluation model is called *streaming model.* In the streaming context, data arrive continuously, are unindexed, and can potentially be infinite. Because of the limited storage space available, systems that query data streams require algorithms that process data in only one sequential scan and deliver query results as soon as they are available. Streaming processing is the only option in a number of applications such as publish-subscribe systems, data monitoring in sensor networks, and managing network

traffic information [28, 29, 30]. Further, many applications adopt streaming processing because of the advantages it presents: (a) no preprocessing of the XML data is required, (b) at every point in time, only the part of the data that is relevant to the evaluation of the query needs to be stored in memory, and (c) the data is read only once, thus avoiding multiple traversals of the XML document.

A broad fragment of XPath such as PTPQs can be useful only if it is complemented with efficient evaluation techniques. This task is complex because, in the general case, PTPQs are directed acyclic graphs (dags). Existing non-main-memory evaluation algorithms on XML data focus almost exclusively on path-pattern or tree-pattern queries. One motivation of this doctoral research is to fill the gap in the efficient non-main-memory evaluation of broad fragments of XPath that go beyond TPQs. We have dealt with this issue in stages.

Partial path query evaluation in the indexed streaming model. In the first stage of the investigation, the problem of efficiently evaluating generalized path-pattern queries called *partial path* queries in the indexed streaming model was addressed. Partial path queries are PTPQs with a single partial path. They cannot be expressed by path or even tree-pattern queries.

Example 1.3.1 *Consider querying the XML bibliography of Example 1.1.1. Suppose that the user wants to find authors of a book on the subject XML published by O'Reilly in 2008 with the following structural restrictions: (1)* author *is the child of* book*; and (2)* year, publisher *and* subject *are ancestors of* book. *It is not difficult to see that these requirements* cannot *be expressed by a TPQ. However, they can be easily specified by a partial path query. Such a query is shown as a directed graph in Figure 1.3. For simplicity, value predicates were omitted in the query.*

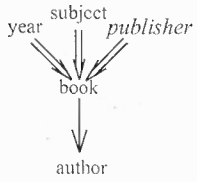


Figure 1.3 A partial path query

Note that Olteanu et al. [31, 11] showed for queries in a class that comprises partial path queries that they can be equivalently rewritten as *sets* of TPQs. However, such a rewriting may lead to a number of TPQs which is exponential on the size of the initial query [31, 11]. This result applies also to partial path queries. Clearly, it is inefficient to evaluate a partial path query by evaluating an exponential number of TPQs.

Query evaluation algorithms for path-pattern or tree-pattern queries usually apply either a top-down [18, 19, 20, 22, 23, 21, 24, 25, 26] or a bottom-up strategy [27]. When a path-pattern or tree-pattern query is evaluated with the top-down strategy, a match on a query node is decided based on the match of its parent query node. When a query is a dag, a node can have multiple parents. For instance, node *book* in Figure 1.3 has *year*, *subject*, and *publisher* as its parents. When a node has multiple parents, its match cannot be decided until all its parent nodes have been matched. This makes the evaluation of these queries more complex. With the bottom-up evaluation strategy for path-pattern or tree-pattern queries, a node x in an XML tree is a match for a query node X if each child node Y of X has a match y which satisfies with x the structural relationship between Y and X in the query. This is not necessarily true when a query is a dag, since the matches of the sub-dags rooted at the child nodes of X should coincide on the common nodes of the subdags in the query. For instance, with the query of Figure 1.3, the matches of the sub-dags rooted at *year*, *subject*, and *publisher* should coincide on *book* (and *author*). Such a verification

process can be expensive [32]. Therefore, existing algorithms cannot be directly used for partial path queries.

An indirect way of exploiting existing algorithms for dealing with query dags would be to produce for a given partial path query Q, a set of path queries that together compute Q. Such an attempt faces two obstacles: (a) as mentioned above the number of path queries may be exponential on the number of query nodes, and (b) the best known algorithm for evaluating path queries under the indexed streaming model (*PathStack* [20]) does not account for repeated labels in a path query.

Therefore, existing algorithms cannot be used directly or indirectly to compute partial path queries. To the best of our knowledge, there are no previous algorithms for evaluating this generalized class of queries in the indexed streaming model.

In Chapter 4, three novel approaches were presented for evaluating partial path queries with repeated labels. The first approach exploits a structural summary of the XML data to evaluate an equivalent set of path-pattern queries for a given partial path query dag. The second approach evaluates a given query dag by generating a spanning tree for the dag. The third approach is a holistic algorithm that evaluates a given query dag directly against the XML tree.

Partial tree-pattern query evaluation in the indexed streaming model. Based on the work on partial path queries, the problem of developing efficient algorithms for PTPQs in the indexed streaming model is addressed. In Chapter 5, an original polynomial time holistic algorithm for PTPQs is presented. In order to assess its performance, two other techniques are designed which evaluate PTPQs by exploiting the state-of-the-art existing

algorithms for smaller classes of queries. An extensive experimental evaluation shows that the holistic algorithm outperforms the other ones.

Partial tree-pattern query evaluation in the streaming model. The problem of developing efficient algorithms for PTPQs on XML streams is also addressed. Two streaming algorithms for PTPQs are designed and implemented. They are presented in Chapter 6. One algorithm, called PSX, works in a lazy fashion. It uses a stack-based technique to compactly encode query matches, thus avoiding query match enumeration. It also avoids processing matches of the query dag that do not contribute to new solutions (redundant matches). Further, it produces solutions incrementally instead of waiting until the whole XML document streams are processed. Compared to the only known streaming algorithm that supports an extension of TPQs, the experimental results show that the proposed algorithm performs better by orders of magnitude while consuming a much smaller fraction of memory space. Algorithm PSX is the first streaming algorithm that supports such a broad fragment of XPath.

Current streaming applications have stringent requirements on query response time and memory consumption because of the large (possibly unbounded) size of data they handle. In order to keep memory usage and CPU consumption low for the PTPQ streaming evaluation, another streaming algorithm called $EagerPSX$ for PTPQs is designed (Section 6.5). Its key feature is that it applies an eager evaluation strategy to quickly determine when node matches should be returned as solutions to the user and also to proactively detect redundant matches. It is theoretically analyzed and experimentally tested on its time and space performance as well as the scalability. It is compared with PSX. The results show that $EagerPSX$ not only achieves better space performance without compromising time

performance, but it also greatly improves query response time for both simple and complex queries, in many cases, by orders of magnitude.

1.3.2 Semantic Issue

To face the challenge of assigning semantics to XML queries so that they return meaningful answers, most existing approaches exploit directly or indirectly the notion of Lowest Common Ancestor (LCA) of a set of nodes in the XML tree. However, in most practical cases, the information in the XML tree is incomplete (e.g., optional elements/values in the schema of the document are missing), or irregular (e.g. different structural patterns coexist in the same document) [33]. For instance, in the DBLP data set (data collected in May 2006), almost 10% of the "book" entries and over 1% of "article" entries do not have an author, while almost all "proceedings" entries do not have authors (this latter one is reasonable and expected). In such cases, the approaches of the first solution (the structureless keyword-based solution), even if they succeed in retrieving all the meaningful answers, they comprise only a tiny percentage of meaningful answers in their answer set. Most of the answers are meaningless. In other words, these approaches have low precision. Consider, for instance, the XML bibliography shown in Figure 1.4. In the XML tree, "book" does not have an author. Suppose that we want to find the publications on XML authored by "Mary." Most existing approaches return "book" and "article" as answer, which is meaningless. Our experiments in Chapter 7 with DBLP-based data sets show that in some cases the precision falls below 1% for some approaches. Clearly, such a low precision is a serious limitation for those approaches.

A recent approach (MLCAS [5, 6]) of the second solution (the solution that extends TPQs with keyword capabilities) shows improved precision. However, the percentage

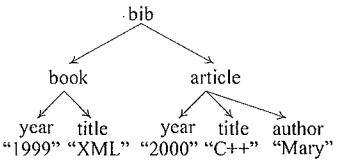


Figure 1.4 An XML tree

of meaningful answers returned (i.e. the recall) is low. In the experiments presented in Chapter 7, the recall of the MLCAS approach falls below 60% for several cases of XML data. Clearly, the poor recall cannot be improved by further imposing structural restrictions. This performance is not satisfactory for data integration environments for which this approach is intended. In addition, it employs different semantics for the keyword part (MLCAS) and the structured part of a query (XQuery). As a consequence, structural restrictions in a query cannot be used to recuperate answers that are not returned by the keyword search.

In Chapter 7, an original approach for assigning semantics to our PTPQ language is presented. The novel semantics seamlessly applies to keyword queries and to queries with structural restrictions. The originality of the proposed approach relies on the use of structural summaries of the XML document for identifying structural patterns (TPQs) for a given query. Meaningful TPQs that return meaningful answers were identified by using a partial order between TPQs. Previous approaches identify meaningful answers by operating *locally* on the *data* (usually computing Lowest Common Ancestors of nodes in the XML tree). In contrast, the proposed approach operates *globally* on *structural summaries* of data to compute meaningful TPQs. This overview of data gives an advantage to the proposed approach compared to previous ones.

1.3.3 Answering XML Queries Using Materialized Views

Answering queries using views is a well-established technique in databases. In this context, two outstanding problems can be formulated. The first one consists in deciding whether a query can be answered exclusively using one or multiple materialized views. Given the many alternative ways to compute the query from the materialized views, the second problem consists in finding the best way for computing the query from the materialized views. In the realm of XML, there is a restricted number of contributions in the direction of these problems due to the many limitations associated with the use of materialized views in traditional XML query evaluation models.

In Chapter 8, the previous two problems are addressed under the indexed streaming model. Together with holistic algorithms, the indexed streaming evaluation model has been established as the prominent technique for evaluating queries on large persistent XML data. This new context revises these problems since it requires new conditions for view usability and new techniques for computing queries from materialized views. An original approach for materializing views is suggested, which stores for every view node only the list of XML nodes necessary for computing the answer of the view. Necessary and sufficient conditions are specified for answering a TPQ using one or multiple materialized views in terms of homomorphisms from the views to the query. In order to efficiently answer queries using materialized views, a stack-based algorithm is designed which compactly encodes in polynomial time and space all the homomorphisms from a view to a query. Further, space and time optimizations are proposed, which use bitmaps to encode view materializations and employ bitwise operations to minimize the evaluation cost of the queries. Finally, an extensive experimentation is conducted which demonstrates that the proposed approach

yields impressive query hit rates in the view pool, achieves significant time and space savings and shows smooth scalability.

1.4 Organization

This dissertation is organized as follows. In Chapter 2, a review of the state-of-the-art is provided for query evaluation techniques on XML data, for the semantics of XML keyword queries, and for answering XML queries using materialized views. In Chapter 3, the partial tree-pattern query (PTPQ) language is formally defined. The same chapter comprises a discussion of the expressiveness and the generality of the PTPQ language for specifying queries on XML data. Three evaluation algorithms for evaluating partial path queries in the indexed streaming model are presented in Chapter 4. Next, in Chapter 5, an original polynomial time holistic algorithm for PTPQs in the indexed streaming model is designed. In Chapter 6, two efficient algorithms for PTPQs in the streaming model are developed. The novel semantics for the PTPQ language is discussed in Chapter 7. In Chapter 8, a novel approach for answering XML queries using materialized views is presented. Finally, Chapter 9 summarizes the obtained results, and provides a discussion of future work.

CHAPTER 2

STATE OF THE ART

This chapter provides a review of the state-of-the-art of evaluation techniques for queries on XML data, on the semantics for XML keyword queries, and answering XML queries using materialized views.

2.1 XML Query Evaluation

2.1.1 XML Streaming Evaluation

The majority of XPath streaming evaluation algorithms focus on tree-pattern queries (TPQs). These algorithms broadly fall in three categories: the *automata-based* approach [34, 28], the *tree-based* approach [35, 36], and the *stack-based* approach [37, 29, 30]. There is also a particular case of XPath streaming evaluation algorithms called filtering algorithms. These algorithms do not literally evaluate the input queries, but they task to determine which of them have a nonempty output on an incoming data stream.

Automata-based algorithms (e.g. *XSQ* [34]) suffer from the problem of exponential state blow-up. Tree-based algorithms (e.g. *TurboXPath* [35]) first build a parse tree for a given TPQ and then find matches of the parse tree nodes on the data streams. *TurboXPath*, in particular, uses smart matching arrays to avoid an exponential memory usage typical for automata-based algorithms. Nevertheless, both automata-based and tree-based algorithms have a worst case complexity which is exponential in the size of the query.

Stack-based approaches [37, 29, 30] exploit stack techniques [20] to compactly encode query pattern matches in stacks, thus avoiding their enumeration and explicit storage during

evaluation. They evaluate TPQs against XML streams in polynomial time and space, which is a significant improvement over automata-based and tree-based algorithms.

However, the problem of the efficient streaming evaluation of subclasses of XPath beyond TPQs has not been adequately addressed. Algorithm X_{aos} [32] is presently the only streaming algorithm that supports an XPath expression with child and descendant axes and their symmetrical reverse axes parent and ancestor. X_{aos} extends the tree-based streaming algorithm *TurboXPath*.

Unfortunately, X_{aos} has three limitations. First, X_{aos} explicitly enumerates and stores all pattern matches for a given query. When the data is *recursive* (more than one element in a path has the same tag), the number of pattern matches can be exponential in the size of query and data. Second, X_{aos} may store multiple copies of the same output, since a single match of an output query node can participate in multiple matches of the query. As a result, it needs an additional process to eliminate duplicate solutions at the final stage. Third, X_{aos} does not deliver query answers until the entire stream is processed. In the case of an infinite stream, the evaluation may be unnecessarily postponed infinitely. Because of these limitations, this type of processing is inefficient and not viable for applications that need to process infinite streams or require incremental outputs. In Chapter 6, we present a streaming algorithm which not only supports a much larger fragment of XPath, but it also does not have the limitations of X_{aos}. As we show later, it outperforms X_{aos} in terms of both time performance and memory usage.

2.1.2 XML Indexed Streaming Evaluation

The indexed streaming evaluation model uses indexes built over the input data to avoid: (1) preloading XML documents in memory, and (2) processing large portions of the XML

documents that are not relevant to the query evaluation. Because of these desirable properties, many query evaluation algorithms for XML have been developed in this model. These algorithms broadly fall in two categories: the *structural join* approach [19, 38, 22, 23], and the *holistic twig join* approach [20, 24, 39, 21, 40, 26, 39]. All these algorithms, however, focus almost exclusively on TPQs.

The *structural join* approach first decomposes a TPQ into a set of binary descendant or child relationships. Then, it evaluates the relationships using binary merge join. The solutions for the binary relationships are "stitched" together to form the answer of the query. This approach might not be efficient because it generates a large number of intermediate solutions (that is, solutions for the binary relationships that do not make it to the answer of the TPQ). Algorithms for structural join order optimization were introduced in [38]. Structural join techniques can be further improved using various types of indexes [22, 23].

The *holistic twig join* approach (e.g. $TwigStack$ [20]) represents the state of the art for evaluating TPQs. This approach evaluates TPQs by joining multiple input streams at a time to avoid producing large intermediate solutions. Algorithm $TwigStack$ was shown optimal for TPQs without child relationships.

Several papers focused on extending *TwigStack*. For example, in [24], algorithm *TwigStackList* evaluates efficiently TPQs in the presence of child relationships. Algorithm $iTwigJoin$ extended $TwigStack$ by utilizing structural indexes built on the input streams [39]. Chen et al. [26] proposed algorithms that handle queries over graph structured data. Evaluation methods of TPQs with OR predicates were developed in [40]. In [21], the XR-tree index [23] is used to skip XML data elements that do not participate in the query answer. Algorithm $Twig^2Stack$ was presented in [27] to avoid merge-joining path solutions needed by *TwigStack*.

All the above algorithms are developed for TPQs and cannot be used nor extended so that they evaluate PTPQs. The reason is that PTPQs are not mere tree patterns but dags augmented with same-path constraints.

PTPQs were initially introduced in [7]. Their containment problem was studied in [41] and PTPQ semantic issues were addressed in [8]. Relevant to our work are also the evaluation algorithms for partial path queries [42, 43]. Partial path queries are not a subclass of TPQs but they form a subclass of PTPQs.

2.2 Semantics for XML Keyword Queries

A number of papers deal with the assignment of meaningful semantics to keyword-based query languages for XML [12, 9, 13, 5, 10, 6]. All of them are based on some variation of the concept of Lowest Common Ancestor (LCA). Among them, the query language in [13] allows also some primitive structural restrictions to be expressed. [5, 6] provide an extension of XQuery to allow users to query an XML document without full knowledge of the structure. It uses the concept of Meaningful Lowest Common Ancestor Structure (MLCAS) of a set of nodes for assigning semantics to keyword queries. In Chapter 7, we present an approach for assigning semantics to keyword queries with structural restrictions. We analytically compare our approach with the three approaches in [12, 13, 5, 6] in Section 7.4 and experimentally in Section 7.5. Our approach shows better recall in all cases, including cases where the XML data are incomplete. Among approaches with similar recall, our approach shows better precision. In [10] the concept of Smallest Lowest Common Ancestor (SLCA) is used to assign semantics to keyword queries. SLCAs are defined to be LCAs that do not contain other LCAs. This semantics is similar to that of the MLCA approach. For this reason, we do not directly compare it to ours.

In order to cope with the low precision, some approaches extend the database techniques with information retrieval techniques. In this direction, they rank the answers of keyword search queries on XML documents according to their estimated relevance [13, 14]. Information retrieval systems using ranking functions may trade recall for precision. The PTPQ language is a database query language. Therefore, it does not employ any ranking functions. Its goal is to not miss any meaningful answer and to exclude as many meaningless answers as possible.

Some languages employ approximation techniques to search for answers when the initial query is too restricted to return any. They either relax the structure of the queries or the matchings of the queries to the data [44, 45]. In contrast to our language, these languages return approximate (not exact) answers.

Several papers focus on providing efficient algorithms for evaluating LCAs for keyword queries [12, 9, 13, 5, 10, 6, 46]. Our approach is different and does not have to explicitly compute LCAs of nodes in the XML tree. In contrast, it computes a number of meaningful TPQs for PTPQs that involve keywords and/or structural restrictions. Since TPQs can be evaluated using an XQuery engine, our approach can directly take advantage of the various optimization techniques developed so far for XQuery [47, 19, 20].

2.3 Answering XML Queries Using Views

Because of the increasing importance of XML, a number of papers have recently addressed the important problems of XML query rewriting using views and of XML view selection [48, 49, 50, 51, 52, 53, 54, 55, 56, ?, 57]. A common assumption made by most of these works is that a view materialization is a set of subtrees rooted at the images of the view

output nodes, or references to the base XML tree. In order to obtain the answer of the original query, downward navigation in the subtrees is needed.

Two types of XML query rewriting problems, namely, equivalent rewritings and contained rewritings have been considered. An equivalent rewriting produces all the answers to the original query using the given view materialization(s), whereas a contained rewriting may produce a subset of the answer to the query. The majority of the recent research efforts have been directed on rewriting XPath queries using materialized XPath views. Among them most works focus on the equivalent rewriting [49, 52, 53, 51, 55]. Balmin et al. [49] presented a framework for answering XPath queries using materialized XPath views. A view materialization may contain XML fragments, node references, full paths, and typed data values. A query rewriting is determined through a homomorphism from a view to the query and the view usability (or query answerability) depends on the availability of one or more of the four types of materializations. Mandhani and Suciu [52] presented results on equivalent TPQ rewritings when the TPQs are assumed to be minimized. Xu et al. [53] studied the equivalent rewriting existence problem for three subclasses of TPQs. Tang and Zhou [51] considered rewritings for TPQs with multiple output nodes. However, the rewritings are restricted to those obtained through a homomorphism from the view to the query which maps the query output nodes to the view output nodes (*output preserving* homomorphism).

The problem of maximally contained TPQ rewritings was studied in [58] both in the absence and presence of a schema. All contributions in [49, 52, 53, 51, 58, 55] are restricted to query rewritings using a *single* materialized view. A common constraining requirement for view usability is the existence of a homomorphism that satisfies two conditions: (a) it maps the view output node to an ancestor-or-self node of the query output node, and (b) it

is an isomorphism on query nodes that are not descendants of the image of the view output node.

The problem of equivalently answering XPath queries using multiple views has been studied in [55, 56, 59, 57]. Arion et al. [55] considered the problem in the presence of structural summaries and integrity constraints. As in [51], a query can have multiple output nodes, and a rewriting is obtained by finding output preserving homomorphisms from views to the query. Answers of views are tuples whose attributes include node ids of the original XML tree, XML subtrees, and/or nested tuple collections. The answer to a query is computed by combining the answers to the views through a number of algebraic operations. The materialization scheme of storing node ids together with XML subtrees is also adopted by [56, 59]. Both papers assumed that output preserving homomorphisms exist among views and they presented rewriting algorithms which use intersection of view answers on node ids.

Tang et al. [57] addressed the multiple view rewriting problem based on the assumptions that structural ids in the form of extended Dewey codes [60] are stored with view materializations. This way, the common ancestors of nodes in different view fragments can be derived for checking view usability. Also, structural joins on the view fragments can be performed based on Dewey codes to produce query answers. The paper also studied a view selection problem defined as finding a minimal view set that can answer a given query. In [50, 54] the equivalent rewriting problem has been addresses but for queries and views which are XQuery expressions.

Phillips et al. [61] consider materializing intermediate query results as sets of tuples in order to allow additional evaluation plans for structural joins. However, their context of

view usability is very restricted and they do not address query answerability from materialized views issues.

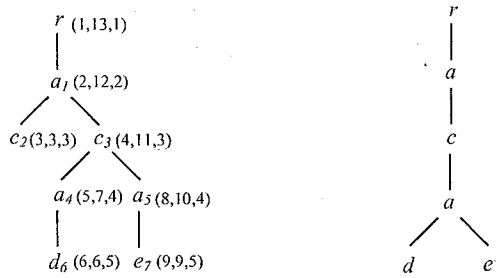
CHAPTER 3

XML DATA MODEL AND PARTIAL TREE-PATTERN QUERY LANGUAGE

In this chapter, we define the XML data model and the partial tree-pattern query (PTPQ) language. We also discuss the expressiveness and the generality of the PTPQ language for specifying queries on XML data.

3.1 XML Data Model

XML data is commonly modeled by a tree structure. Tree nodes are labeled and represent elements, attributes, or values. Let $\mathcal{L}$ be the set of node labels. Tree edges represent element-subelement, element-attribute, and element-value relationships. Without loss of generality, we assume that only the root node of every XML tree is labeled by $r \in \mathcal{L}$. We denote XML tree labels by lower case letters. To distinguish between nodes with the same label, every node in the XML tree has an identifier shown as a subscript of the node label. Figure 3.1 shows an XML tree. The triplets by the nodes will be explained below.



(a) (b)

Figure 3.1 (a) An XML tree T, (b) The index tree of T

For XML trees, we adopt the positional representation widely used for XML query processing [19, 20, 21]. The positional representation associates with every node a triplet (*start*,*end*,*level*) of values. The *start* and *end* values of a node are integers which can be determined through a depth-first traversal of the XML tree, by sequentially assigning numbers to the first and the last visit of the node. The *level* value represents the level of the node in the XML tree. Interestingly, similar positional representation scheme is used for processing class hierarchies in the area of Artificial Intelligence [62].

The positional representation allows efficiently checking structural relationships between two nodes in the XML tree. For instance, given two nodes n_1 and n_2, n_1 is an ancestor of n_2 iff $n_1.start < n_2.start$, and $n_2.end < n_1.end$. Node n_1 is the parent of n_2 iff $n_1.start < n_2.start$, $n_2.end < n_1.end$, and $n_1.level = n_2.level - 1$.

In this dissertation, we often need to check whether a number of nodes in an XML tree lie on the same path. This check can be performed efficiently using the following proposition.

Proposition 3.1.1 *Given a set of nodes $n_1, \ldots, n_k$ in an XML tree T, let* maxStart *and* minEnd *denote respectively the maximum start and the minimum end values in the positional representations of $n_1, \ldots, n_k$. Nodes $n_1, \ldots, n_k$ lie on the same path in T iff* maxStart $\leq$ minEnd.

3.2 Partial Tree-Pattern Query Language

Syntax. A partial tree-pattern query (PTPQ) specifies a pattern which partially determines a tree. PTPQs comprise nodes and child and descendant relationships between nodes. The nodes are grouped into disjoint sets called *partial paths*. PTPQs are embedded to XML

trees. The nodes of a partial path are embedded to nodes on the *same* XML tree path. However, unlike paths in TPQs the child and descendant relationships in partial paths do not necessarily form a total order. This is the reason for qualifying these paths as partial. PTPQs also comprise node sharing expressions. A node sharing expression indicates that two nodes from different partial paths are to be embedded to the same XML tree node. That is, the image of these two nodes is the same – *shared* – node in the XML tree.

The formal definition of a PTPQ follows.

Definition 3.2.1 (PTPQ) *Let $\mathcal{N}$ be an infinite set of labeled nodes. Nodes in $\mathcal{N}$ are labeled by a label in $\mathcal{L}$. Let X and Y denote distinct nodes in $\mathcal{N}$. A* partial tree-pattern query *is a pair* (S, N) *where:*

- S *is a list of n named sets $p_1, \ldots, p_n$ called* partial paths *(*PPs*). Each PP p_i is a finite set of expressions of the form X/Y (*child relationship*) or $X//Y$ (*descendant relationship*). We write $X[p_i]/Y[p_i]$ (resp. $X[p_i]//Y[p_i]$) to indicate that $X[p_i]/Y[p_i]$ (resp. $X[p_i]//Y[p_i]$) is a relationship in PP p_i. Child and descendant relationships are collectively called* structural *relationships.*
- N *is a set of* node sharing *expressions $X[p_i] \approx Y[p_j]$, where p_i and p_j are distinct PPs, and X and Y are nodes in PPs p_i and p_j respectively such that both of them are labeled by the same label in $\mathcal{L}$.*

Figure 3.2(a) shows a PTPQ Q_1 and Figure 3.2(b) shows the visual representation of Q_1. We use this representation later on in Section 5.3 to design a comparison algorithm for evaluating PTPQs. Unless otherwise indicated, in the following, "query" refers to a PTPQ. Note that the labels of the query nodes are denoted by capital letters to distinguish them

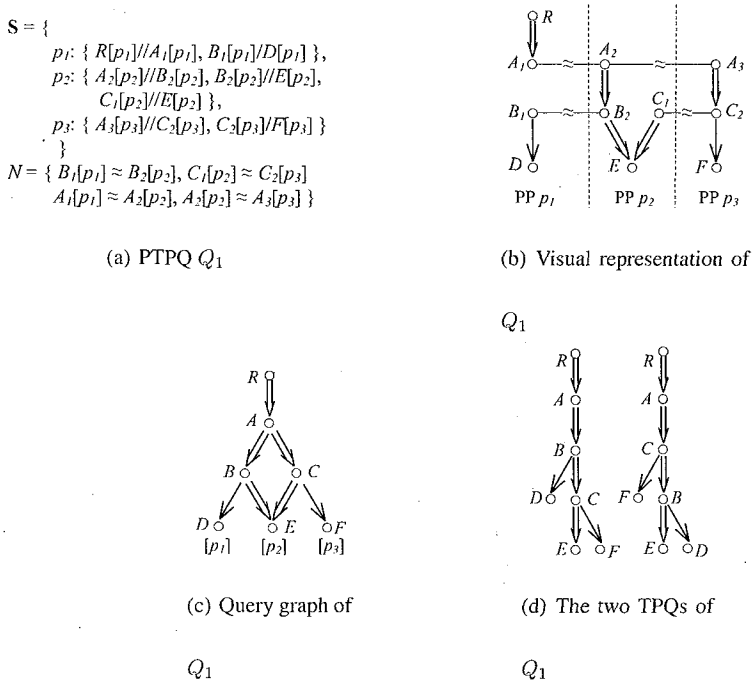


Figure 3.2 A PTPQ and its three representations

from the labels of the XML tree nodes. In this sense, label l in an XML tree and label L in a query represent the same label.

Semantics. The answer of a PTPQ on an XML tree is a set of tuples of nodes from the XML tree that satisfy the structural relationships and the same path constraints of the PTPQ. Formally:

Definition 3.2.2 (Query Embedding) *An* embedding *of a query Q into an XML tree T is a mapping M from the nodes of Q to nodes of T such that: (a) a node $A[p_j]$ in Q is mapped by M to a node of T labeled by a; (b) the nodes of Q in the same PP are mapped by M to nodes that lie on the same path in T; (c) $\forall\ X[p_i]/Y[p_i]$ (resp. $X[p_i]//Y[p_i]$) in Q, $M(Y[p_i])$ is a child (resp. descendant) of $M(X[p_i])$ in T; (d) $\forall\ X[p_i] \approx Y[p_j]$ in Q, $M(X[p_i])$ and $M(Y[p_j])$ coincide in T.*

We call *image* of Q under an embedding M a tuple that contains one field per node in Q, and the value of the field is the image of the node under M. Such a tuple is also called *solution* of Q on T. The *answer* of Q on T is the set of solutions of Q under all possible embeddings of Q to T.

Graph representation for PTPQs. For our evaluation algorithm, we represent queries as node labeled annotated directed graphs: a query Q is represented by a graph Q_G. Every node X in Q corresponds to a node X_G in Q_G, and vice versa. Node X_G is labeled by the label of X. Two nodes in Q participating in a node sharing expression correspond to the same node in Q_G. Otherwise, they correspond to distinct nodes in Q_G. For every structural relationship $X//Y$ (resp. X/Y) in Q there is a single (resp. double) edge in Q_G. In addition, each node in Q_G is annotated by the set of PPs of the nodes in Q it corresponds to. Note that these annotations allow us to express same-path constraints. That is, all the nodes annotated by the same partial path have to be embedded to nodes in an XML tree that lie on the same path.

Figure 3.2(c) shows the query graph of query Q_1 of Figure 3.2(a). Note that a node in the graph inherits all the annotating PPs of its descendant nodes. Because of this inheritance property of partial path annotations we can omit in the figures the annotation of internal nodes in queries when no ambiguity arises. For example, in the graph of Figure 3.2(c), node A is annotated by the PPs p_1, p_2, and p_3 inherited from its descendant nodes D, E, and F.

Clearly, a query that has a cycle is unsatisfiable (i.e., its answer is empty on any XML tree). Therefore, in the following, we assume a query is a dag and we identify a query with its dag representation.

3.3 Generality of Partial Tree-Pattern Query Language

Clearly, the class of PTPQs cannot be expressed by TPQs. For instance, PTPQs can constrain a number of nodes in a query pattern to belong to the same path even if there is no precedence relationship between these nodes in the PTPQ. Such a query cannot be expressed by a TPQ. TPQs correspond to the fragment $XP^{\{[],/,//\}}$ of XPath that involves predicates([]), and child (/) and descendant (//) axes. In fact, it is not difficult to see that PTPQs cannot be expressed either by the larger fragment $XP^{\{[],/,//,\backslash,\backslash\backslash\}}$ of XPath that involves, in addition, the reverse axes parent ($\backslash$) and ancestor ($\backslash\backslash$). On the other hand, PTPQs represent a very broad fragment $XP^{\{[],/,//,\backslash,\backslash\backslash,\approx\}}$ of XPath that corresponds to $XP^{\{[],/,//,\backslash,\backslash\backslash\}}$ augmented with the is operation ($\approx$) of XPath2 [1]. The is operator is a node identity equality operator. The conversion of an expression in $XP^{\{[],/,//,\backslash,\backslash\backslash,\approx\}}$ to an equivalent PTPQ is straightforward. There is no previous indexed streaming evaluation algorithm that directly supports such a broad fragment of XPath.

Note that as the next proposition shows, a PTPQ is equivalent to a *set* of TPQs.

Proposition 3.3.1 *Given a PTPQ Q there is a set of TPQs $Q_1, \ldots, Q_n$ in $XP^{\{[],/,//\}}$ such that for every XML tree T, the answer of Q on T is the union of the answers of the Q_is on T.* □

As an example, Figure 3.2(d) shows the two TPQs for query Q_1 of Figure 3.2(a), which together are equivalent to Q_1. Based on the previous proposition, one can consider evaluating PTPQs using existing algorithms for TPQs. In Section 5.3.1, we present such an algorithm. However, the number of TPQs that need to be evaluated can grow to be large (in the worst case, it can be *exponential* on the number of nodes of the PTPQ). Therefore, the performance of such an algorithm is not expected to be satisfactory.

CHAPTER 4

EVALUATING PARTIAL PATH QUERIES ON INDEXED XML STREAMS

In this chapter, we present our three evaluation algorithms for evaluating partial path queries in the indexed streaming model. The chapter is organized as follows. Section 4.1 defines the partial path query language and its properties. We describe data structures for the indexed streaming evaluation in Section 4.2. We present our three evaluation algorithms in Section 4.3, 4.4, and 4.4 respectively. Section 4.6 presents and analyses our experimental results.

4.1 Partial Path Query Language

A partial path query specifies a path pattern where the structure (an order among the nodes) may not be fully defined.

Syntax. In order to specify these queries, paths or even trees are not sufficient, and we need to employ directed graphs.

Definition 4.1.1 *A* partial path query *is a directed graph whose nodes are labeled by labels in L, and every node is incident to at least one edge. There is at most one node labeled by r and this node does not have incoming edges. Edges between nodes can be of two types:* child *and* descendant. □

In the rest of the paper, unless stated differently, "query" refers to "partial path query." Query nodes denote XML tree nodes but we use capital letters for their labels. Therefore, a query node labeled by A denotes XML tree nodes labeled by a. In order to distinguish

between distinct query nodes with the same label, we use subscripts. For instance, A_3 and A_4 denote two distinct nodes labeled by A. If Q is a query, and X and Y are nodes in Q, the expressions X/Y and $X//Y$ are called *structural relationships* and denote respectively a child and descendant edge from X to Y in Q.

Figure 4.1 shows four queries. Child (resp. descendant) edges are shown with single (resp. double) arrows. Query Q_1 is a partial path query which is also a path query since the structural relationships in the query induce a total order for the query nodes. Notice that a query graph can be disconnected, e.g. query Q_4 in Figure 4.1(d). Notice also that no order may be defined between two nodes in a query, e.g. between nodes A and C in Q_3, or between nodes A_1 and A_2 in Q_4.

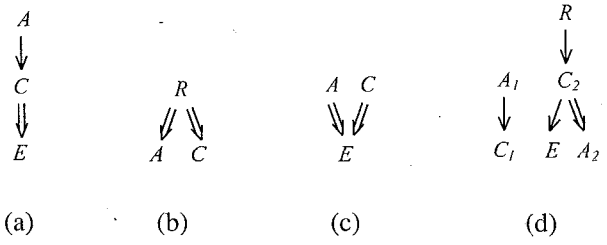


Figure 4.1 Queries (a) Q_1, (b) Q_2, (c) Q_3, (d) Q_4

Semantics. The answer of a partial path query on an XML tree is a set of tuples. Each tuple consists of XML tree nodes that lie *on the same path* and preserve the child and descendant relationships of the query. More formally:

An *embedding* of a partial path query Q into an XML tree T is a mapping M from the nodes of Q to nodes of T such that: (a) a node in Q labeled by A is mapped by M to a node of T labeled by a; (b) the nodes of Q are mapped by M to nodes that lie on the same path in T; (c) $\forall\ X/Y$ (resp. $X//Y$) in Q, $M(Y)$ is a child (resp. descendant) of $M(X)$ in T.

We call *image* of Q under an embedding M a tuple that contains one field per node in Q, and the value of the field is the image of the node under M. Such a tuple is also called *solution* of Q on T and the value of each field is called *solution* of the corresponding node in Q on T. The *answer* of Q on T is the set of solutions of Q under all possible embeddings of Q to T.

Consider query Q_2 of Figure 4.1. Notice that Q_2 is syntactically similar to a tree-pattern query (twig). However, the semantics of partial path queries is different: when query Q_2 is a partial path query, the images of the query nodes R, A and C should lie on the same path on the XML tree.

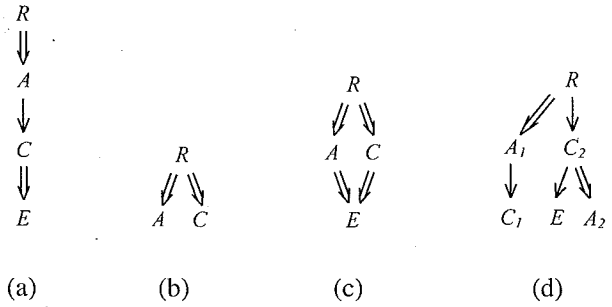


Figure 4.2 Queries of Figure 4.1 with the root R (a) Q_1, (b) Q_2, (c) Q_3, (d) Q_4

Clearly, we can add a descendant edge from node R to every node that does not have incoming edges in a query without altering its meaning. Therefore, without loss of generality, we assume that a query is a connected directed graph rooted at R. Figure 4.2 shows the queries of Figure 4.1 in that form.

Obviously, if a query has a cycle, it is unsatisfiable (that is, it does not have a non-empty answer on any database). Detecting the existence of cycle in a directed graph can be done in linear time on the size of the graph. In the following, we assume that a query is a directed acyclic graph (dag) rooted at node R.

4.2 Data Structures for Indexed Streaming Evaluation

In the indexed streaming evaluation model, the data is preprocessed and the position of every node in the XML tree is encoded. Usually, for every label in the XML tree an inverted list of the nodes with this label is produced. These lists are called *streams*. In order to evaluate a query, the nodes of the relevant streams are read in the pre-order of their appearance in the XML tree. Every node in a stream can be accessed *only once*. We present in this section the data structures and operations we use for the evaluation of the queries.

Let Q be a query. For simplicity, we assume for now that Q is a tree pattern rooted at R. We show in Section 4.5 how to handle queries that are dags. Let X be a node and L be a label in Q. Function *nodes(Q)* returns all nodes of Q; *label(X)* returns the label of X in Q; *label(Q)* returns the set of node labels in Q; *occur(L)* returns all nodes in Q labeled by L. Boolean function *isLeaf(X)* returns true iff X is a leaf node in Q. Function *parent(X)* returns the parent of X in Q; *children(X)* returns the set of child nodes of X in Q.

With every distinct node label L in Q, we associate a stream T_L of the positional representation (see Section 3.1) of the nodes labeled by L in the XML tree. The nodes in the stream are ordered by their *begin* field. To access sequentially the nodes in T_L, we maintain a cursor C_L. For simplicity, we may alternatively use C_L to denote the node pointed by pointer C_L in T_L. Operation *advance(C_L)* moves C_L to the next node in T_L. Function *eos(C_L)* returns true if C_L has reached the end of T_L.

With every query node X in Q, we associate a stack S_X. A stack entry in S_X consists of a pair: (positional representation of node from $T_{label(X)}$, pointer to an entry in stack $S_{parent(X)}$). A pointer denotes a position in a stack. The expression $S_X.k$ denotes the entry at position k of stack S_X. The position of the bottom entry in a stack is 1. We use the following stack operations: *push(S_X,entry)* which pushes *entry* on the stack S_X, *pop(S_X)*

which pops out the top entry from stack S_X, and *top*(S_X) which returns the position of the top entry of stack S_X.

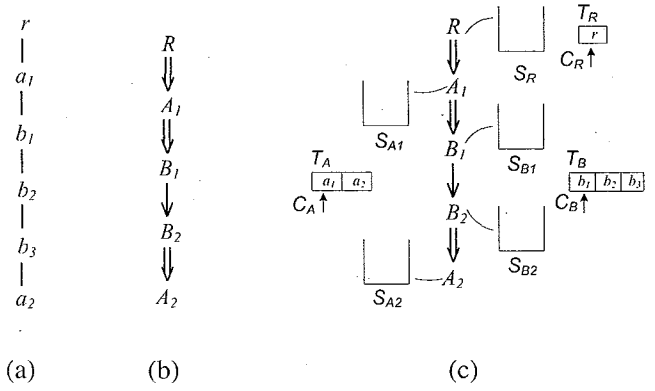


Figure 4.3 (a) Data path, (b) Query Q_5, (c) Initial state of cursors and stacks

Initially, all stacks are empty and every cursor C_L points to the first node in T_L. Figure 4.3(c) shows the initial state of cursors and stacks associated with the query Q_5 of Figure 4.3(b) on the data path of Figure 4.3(a). Stream nodes are accessed through cursors and they are possibly stored in stacks. In Figure 4.3(c), cursor C_A feeds stacks S_{A1} and S_{A2} and cursor C_B feeds stacks S_{B1} and S_{B2}. During execution of the algorithm, the entries that stack S_X can contain correspond to stream nodes in $T_{label(X)}$ before $C_{label(X)}$.

The entries in a stack below an entry e correspond to nodes in the XML tree that are ancestors of the node corresponding to e. The pointer of an entry e in a stack S_X points to the highest among the entries in stack $S_{parent(X)}$ that correspond to ancestors of e in the XML tree. At any point in time, stack entries represent partial solutions of the query that can be extended to the solutions as the algorithm goes on. An important feature of such a stack-based organization is that it encodes a potentially exponential number of solutions in a linear space.

4.3 IndexPaths-R: Leveraging Structural Indexes and Path Query Algorithms

Our first approach, called *IndexPaths-R*, endeavors to leverage existing algorithms for *path* queries [20]. Given a partial path query Q, *IndexPaths-R* exploits a structural summary of data, called *index tree*, to generate a set of path queries that together are equivalent to Q. In order to evaluate these queries, it extends the algorithm in [20] for path queries so that it can work on path queries with repeated labels.

4.3.1 Generating Path Queries from Index Trees

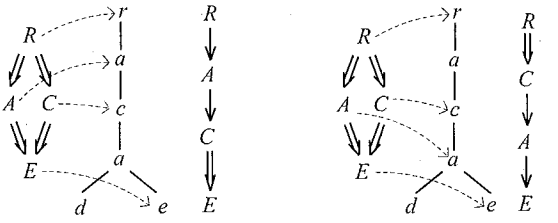
Given a partitioning of the nodes of an XML tree T, an index graph for T is a graph G such that: (a) every node in G is associated with a distinct equivalence class of element nodes in T, and (b) there is an edge in G from the node associated with the equivalence class $\mathcal{A}$ to the node associated with the equivalence class $\mathcal{B}$, iff there is an edge in T from a node in $\mathcal{A}$ to a node in $\mathcal{B}$. The equivalence class of nodes in T associated with each node in G is called *extent* of this node. Index graphs have been referred to with different names in the literature and they differ in the equivalence relations they employ to partition the nodes of the XML tree. An *1-index* [63, 64] considers as equivalent nodes in T that have the same incoming path from the root of T. A 1-index is a tree[1]. We define the *index tree* of T to be a 1-index of T without extents. The index tree can be built by a single depth-first traversal of T in time proportional to the size of T. Figure 3.1(b) shows the index tree for the XML tree of Figure 3.1(a).

1-indexes are usually much smaller than the corresponding XML data. According to the measurements of [55] on XML documents from different repositories, a 1-index is

[1] 1-indexes are similar to *strong DataGuides* [65] when the data is a tree.

three to five orders of magnitude smaller than the corresponding XML data. Since index trees do not have extents, their size is insignificant compared to the size of the XML data.

Given a query Q and an index tree I, we can generate a set $\mathcal{P}$ of path queries that is equivalent to Q by finding all the embeddings of Q into I. Any of the two algorithms presented later in this paper can be used to find the embeddings of a query to an index tree. However, even a naive approach would be satisfactory given the size of an index tree.



(a) Embedding 1 (b) Embedding 2

Figure 4.4 Two embeddings of query Q_3 of Figure 4.2(c) on the index tree of Figure 3.1(b) and the corresponding path queries

Figure 4.4 shows all the possible embeddings of the query Q_3 of Figure 4.2(c) on the index tree of Figure 3.1(b) (there are two of them) and the corresponding path queries. There is an one-to-one correspondence between the nodes of a path query and the nodes of Q. Two consecutive nodes in a path query are linked through a child relationship if the corresponding nodes in the index tree are linked through a child relationship. Otherwise, consecutive nodes in a path query are linked through a descendant relationship.

The next proposition shows that the answer of a partial path query can be correctly computed by the path queries generated. Its proof is straightforward.

Proposition 4.3.1 *Let T be an XML tree and I be its index tree. Let also Q be a partial path query and $\mathcal{P}=\{P_1, \ldots, P_n\}$ be the set of path queries generated for Q on I. Then, the answer of Q on T is the union of the answers of all the P_is on T.*

In practice, the number of the path queries for the query Q is expected to be small. However, in extreme cases, it can be exponential on the number of nodes in Q. This is, for instance, the case when the query does not specify an order for its non-root nodes and every ordering of these nodes has an embedding on the index tree. Nevertheless, even in this case, any one of the path queries generated represents a pattern that occurs in T. Therefore, it will return a non-empty answer when evaluated on T.

4.3.2 An Algorithm for Path Queries with Repeated Labels

Algorithm *PathStack* [20] optimally computes answers for path pattern queries under the indexed streaming model. However, it operates on a restricted class of path queries where a label cannot appear more than once. In this section, we extend *PathStack* so that it works on path queries with repeated labels. *PathStack* associates every query node with one stack and one stream. Attempting to associate multiple streams per query label (one for each query node with this label) would violate the indexed streaming model requirements since stream nodes would be accessed multiple times during the evaluation. Therefore, we designed Algorithm *PathStack-R*, which extends *PathStack* by allowing nodes with the same label to share the same stream. Query nodes with the same label are associated with distinct stacks but the same stream node might appear in multiple stacks.

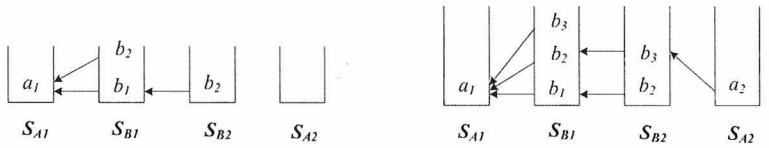
Algorithm *PathStack-R* is presented in Listing 1. *PathStack-R* gradually constructs solutions to a path query Q and compactly encodes them in stacks, by iterating through stream nodes in ascending order of their *begin* values. Thus, the query nodes are matched from the query root to the query leaf.

In line 2, *PathStack-R* calls function *getNextQueryLabel*. Function *getNextQueryLabel* identifies the stream node with the minimal *begin* value among the nodes pointed to by the

Listing 1 Algorithm PathStack-R

```
1 while ¬end() do
2     L = getNextQueryLabel()
3     cleanStacks(C_L)
4     for every X ∈ occur(L) in leaf-to-root order in Q do
5         moveStreamToStack(L, X)
6         if isLeaf(X) then
7             showSolutions(S_X)
8             pop(S_X)
9     advance(C_L)

Function end()
1 return ∀ X ∈ nodes(Q): isLeaf(X) ⇒ eos(C_label(X))

Function getNextQueryLabel()
1 return L ∈ labels(Q) such that C_L.begin is minimal

Procedure cleanStacks(C_L)
1 for (X in nodes(Q)) do
2     while (¬ empty(S_X) and S_X.top(S_X).end < C_L.begin) do {pop out all entries in S_X whose nodes are not ancestors of
      C_L}
3         pop(S_X)

Procedure moveStreamToStack(L, X)
1 P = parent(X)
2 if (P is not the query root and empty(S_P)) then
3     return
4 if (X is the query root) or (P//X ∈ Q or S_P.top(S_P).level = C_L.level-1) then
5     push(S_X, (C_L, pointer to S_P.top(S_P)))
```

cursors. Line 3 calls procedure *cleanStacks* to remove from all stacks the nodes that are not ancestors of the node under consideration in the XML tree. This way, partial solutions encoded in stacks that cannot become solutions are excluded from further consideration.



(a) States of stacks after reading b_2

(b) States of stacks after reading a_2

Figure 4.5 Running PathStack-R on query Q_5 of Figure 4.3(b) and the path of Figure 4.3(a)

Lines 4 and 5 call Procedure *moveStreamToStack* on all the occurrences of L in Q. Procedure *moveStreamToStack* is central to *PathStack-R*. It determines if the stream node C_L under consideration qualifies for being pushed on a stack S_X, where $label(X) = L$. Node C_L can be pushed on stack S_X if (1) X is the root, or (2) the structural relationship between C_L and the top stack entry of X's parent P satisfies the structural relationship between X and P in the query. This ensures that stream nodes that do not contribute to solutions will not be stored in stacks and processed. If multiple nodes in Q are labeled by L, we need to check if C_L can be pushed on the stack of each occurrence of L in Q. The order of pushing C_L on stacks is crucial. In order to prevent C_L from 'seeing' its own copy in a parent stack, *moveStreamToStack* is called on the occurrences of L in Q in their leaf-to-root order (line 4). We illustrate this with in Example 4.3.1 below.

Whenever the incoming stream node C_L is pushed onto the stack of the leaf node, we know the stacks contain at least one solution to the query. At that time, Procedure *showSolutions* is invoked to output them (lines 6-8). Procedure *showSolutions* iteratively outputs encoded solutions sorted on the nodes of the query in a leaf to root order. The details are omitted here and can be found in [20].

Example 4.3.1 *Consider the path query Q_5 in Figure 4.3(b) on the data path shown in Figure 4.3(a). When a_1 is read, it is not pushed on S_{A_2}, since the push condition is not satisfied: the parent stack S_{B_2} is empty. When b_2 is read, it is pushed first on the stack S_{B_2} and then on stack S_{B_1}. The state of the stacks at this moment is shown in Figure 4.5(a). For simplicity, the stack for the query root R is omitted. Note that if we do not check whether b_2 can be pushed onto stacks in this order, then we won't be able to push b_2 on stack S_{B_2}. The reason is that b_2 and the top entry of stack S_{B_1} (which would also be b_2) would not satisfy the child relationship between B_2 and B_1 in Q_5. This would result in missing one solution for Q_5. Figure 4.5(b) shows the state of the stacks after a_2 is pushed on stack S_{A_2}. At that time, Procedure* showSolutions *is invoked to output the answer of Q_5 which is* $\{ra_1b_1b_2a_2, ra_1b_2b_3a_2\}$.

4.3.3 Analysis of IndexPath-R

Given a node X in a path query Q, we call the path from the root of Q to X *ancestor* path of X. For example, the ancestor path of B_1 in the query Q_5 in Figure 4.3(b) is $R//A_1//B_1$. Given a stream node x of an XML tree T, we say that x *matches* X iff x is the image of X under an embedding of the ancestor path of X to T.

Proposition 4.3.2 *Let X be a query node in Q and x be a stream node with the same label. Node x is pushed on stack S_X iff x matches X.*

Proof. We prove the proposition by induction on the level of the query node X in Q. If the level of X is 1, X is the root R of Q. The proposition holds trivially because x is the root node r and it is always pushed onto S_R. Let's assume now that the level of X is > 1 and Y is the parent of X in Q.

Only if part: Procedure *moveStreamToStack* pushes stream node x on stack S_X only if x and the top entry y of the parent stack S_Y satisfy the relationship between X and Y in Q. By the induction hypothesis y matches Y. Therefore, if x is pushed on stack S_X, it matches X. Note that this is true even if X and Y have the same label since in this case, *moveStreamToStack* attempts to push x first to S_X then to S_Y.

If part: Since x matches X, there must exist one stream node that matches the parent Y of X and that node and x satisfy the structural relationship between Y and X in Q. Assume y is such a stream node with the largest level in T above x. By the induction hypothesis, when x is considered, y is the top entry of S_Y. Since x and y satisfy the structural relationship between X and Y, Procedure *moveStreamToStack* will push x on S_X. Note that this is true even if x and y have the same label since in this case, *moveStreamToStack* attempts to push x first to S_X then to S_Y. □

As a result of Proposition 4.3.2, Algorithm *PathStack-R* will find and encode in stacks all the partial (if X is a non-leaf node in Q) or complete (if X is a leaf node in Q) solutions involving x. When at least one complete solution is encoded in the stacks, procedure $showSolutions$ is invoked to output them. Therefore, Algorithm *PathStack-R* correcly finds all the solutions to Q.

We next provide time and space complexity results. Given a path query Q and an XML tree T, let $input$ denote the sum of sizes of the input streams, $output$ denote the size of the answer of Q on T, and $|Q|$ denote the number of nodes in Q. The *recursion depth* of a query node X in T is the maximum number of nodes in a path of T that match X [36]. We define the recursion depth of Q in T, denoted $recurDepth$, as the maximum of

the recursion depths of the query nodes of Q in T. Clearly, $recurDepth$ is bounded by the maximum number of occurrences of a query label in a path of T.

Theorem 4.3.1 Algorithm *PathStack-R* correctly evaluates a path query Q with repeated labels on an XML tree T. The algorithm uses $O(recurDepth \times |Q|)$ space. It has CPU time complexity $O((input + output) \times |Q|$ and disk I/O complexity $O(input + output)$.

Proof. The space complexity of *PathStack-R* depends mainly on the number stack entries at any point during execution. Since the worst-case size of any stack during execution is bounded by $recurDepth$, *PathStack-R* has space complexity $O(recurDepth \times |Q|)$.

We assume that query stacks fit in memory and all stack operations are conducted in memory. Thus the disk I/O complexity of *PathStack-R* consists of two parts: the I/O of accessing stream nodes, and the I/O of outputting query solutions. Since we always advance the cursors (using *advance*) and never backtrack, it takes $O(input)$ to access the stream nodes. As no any intermediate solutions are produced during execution, outputting query solutions takes $O(output)$. Therefore, the disk I/O complexity of *PathStack-R* is $O(input + output)$.

The CPU time complexity of *PathStack-R* depends mainly on the time spent on *getNextQuerylabel*, the time spent on *cleanStacks*, the number of calls to *moveStreamToStack*, and the time to produce solutions. Assuming a priority queue is used for getting the cursor with the minimum $begin$ value, the total time spent on calls to function *getNextQuerylabel* is $O(input \times log|label(Q)|) = O(input \times log|Q|)$. For each new stream node under consideration, procedure *cleanStacks* checks the top stack entry of every query node of Q. Thus, the total time spent on calls to *cleanStacks* is $O(input \times |Q|)$. For each stream node, procedure *moveStreamToStack* is invoked at most $maxOccur$ times, and each invocation

takes constant time. Procedure *showSolutions* takes $|Q|$ time on producing each solution. As Algorithm *PathStack-R* does not generate any intermediate solutions, the time it spends on producing all the solutions is $O(output \times |Q|)$. Therefore, *PathStack-R* has time complexity $O((input + output) \times |Q|)$. □

Clearly, assuming that the size of the query is insignificant compared to the size of data, *PathStack-R* is asymptotically optimal for path queries with repeated labels.

The correctness of the approach *IndexPaths-R* follows from Theorem 4.3.1 and Proposition 4.3.1. One advantage of this approach is that if a partial path query Q does not have any path queries on the index tree of T, we know that Q has empty answer on T without explicitly evaluating Q on T.

4.4 PartialMJ-R: a Partial Path Merge Join Algorithm

Algorithm *PartialMJ-R* is a stack-based algorithm. Given a partial path query Q, it extracts a spanning tree Q_s of Q. Then, it evaluates each root-to-leaf path of Q_s concurrently. Solutions for each root-to-leaf path of Q_s are merge-joined by guaranteeing that (a) they lie on the same path in the XML tree, and (b) they satisfy the structural relationships that appear in Q but not in Q_s.

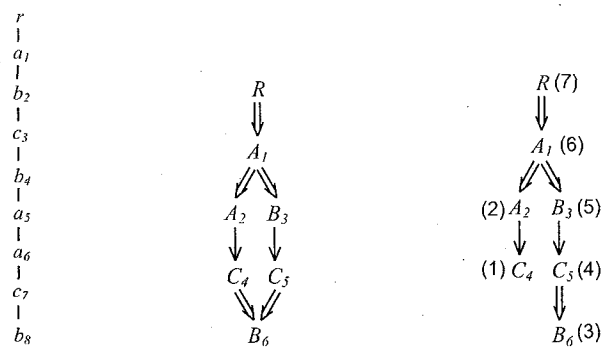
Figure 4.6(b) shows the graph of a query Q_6 and Figure 4.6(c) shows a spanning tree Q_{6s} of Q_6. Edge $C_4//B_6$ of Q_6 is missing from Q_{6s}. Solutions for each of the two root-to-leaf paths of Q_{6s} lying on the same path of the XML tree can be merged to produce a solution for Q_6, if they coincide on R and A_1 and satisfy the structural constraint $C_4//B_6$.

PartialMJ-R is shown in Listing 2. Compared to *PathStack-R*, *PartialMJ-R* has two important differences: (1) in line 1, *PartialMJ-R* produces a spanning tree Q_s for the given

Listing 2 Algorithm PartialMJ-R

```
1  create a spanning tree Q_s of Q. E denotes the set of edges in Q which do not appear in Q_s
2  while ¬end() do
3     L = getNextQueryLabel()
4     clean(C_L)
5     for every X ∈ occur(L) in the post-order of its appearance in Q_s do
6        moveStreamToStack(L, X)
7        if isLeaf(X) then
8           solns = showSolutionsWithBlocking(S_X, 1)
9           if (∀Y ∈ nodes(Q_s): isLeaf(Y) and Y ≠ X ⇒ pathSolns[Y] ≠ ∅) then
10             joinPathSolutions(solns, X)
11          add solns to pathSolns[X]
12          pop(S_X)
13    advance(C_L)

Procedure joinPathSolutions(solns, X)
1  merge-join the solutions in solns and in each one of the pathSoln_Y of each leaf node Y (Y ≠ X) of Q_s and return only the
   results that satisfy the structural relationships in E. {Because of procedure clean, all the solutions in solns and pathSoln_Y are
   guaranteed to lie on the same path of the XML tree}

Procedure clean(C_L)
1  for (X ∈ nodes(Q_s)) do
2     while (¬ empty(S_X) and S_X.top(S_X).end < C_L.begin) do {pop all entries in S_X whose nodes are not ancestors of C_L}
3        pop(S_X)
4     if isLeaf(X) then
5        remove solutions in pathSoln_X whose leaf nodes are not ancestors of C_L
```

partial path query Q and records the set of structural relationships that are present in Q but are missing in Q_s; (2) in line 10, *PartialMJ-R* calls procedure *joinPathSolutions* to merge-join solutions for a root-to-leaf path in Q_s with solutions for other root-to-leaf paths produced earlier. Each leaf node X of Q_s is associated with a list $pathSoln[X]$ which stores



(a) (b) (c)

Figure 4.6 (a) Data path (b) Query Q_6 (c) Q_6's spanning tree Q_{6s}

solutions of the root-to-leaf path in Q_s that ends in X. To facilitate the merge-join process, solutions stored in $pathSoln[X]$ are sorted on the nodes of the query path in root-to-leaf order. At any point in time, all the solutions in $pathSoln[X]$ lie on the same path in the XML tree.

For each stream node C_L under consideration, *PartialMJ-R* calls procedure *clean* (line 4). Procedure *clean* not only removes from all the stacks the nodes that are not ancestors of C_L in the XML tree (lines 2-3), but also removes from each of the $pathSoln[X]$ the solutions whose nodes are not ancestors of C_L (lines 4-5). For the latter one, it suffices to compare C_L with the node in each solution which is a match of the leaf node of the query path.

Similarly to *PathStack-R*, *PartialMJ-R* calls procedure *moveStreamToStack* on each occurrence of L in the spanning tree Q_s in a bottom-up way, that is, in the post-order of its appearance in Q_s (line 6).

When an occurrence X of L is a leaf node of Q_S, the stacks for the corresponding root-to-leaf path in Q_s contain at least one solution to the path. At that time, *PartialMJ-R* calls procedure *showSolutionsWithBlocking* to produce them and then stores them in $solns$

(line 8). Procedure *showSolutionsWithBlocking* iteratively produces encoded solutions sorted on the nodes of the path in root-to-leaf order. The details are omitted here and can be found in [20]. If at this time for every other leaf node Y of Q_s, $pathSoln[Y]$ is not empty (line 9), *PartialMJ-R* calls procedure *joinPathSolutions* to merge-join the newly produced solutions in $solns$ and the previously produced solutions stored in each $pathSoln[Y]$ and return only the results that satisfy the structural relationships in $\mathcal{E}$ (line 10). Note that because of the execution of procedure *clean*, all the solutions in $solns$ and $pathSoln[Y]$ are guaranteed to lie on the same path of the XML tree. Further, since every time solutions to Q are produced, they involve the newly pushed node C_L, *PartialMJ-R* is guaranteed not to generate duplicate solutions.

Compared to the approach *IndexPaths-R*, Algorithm *PartialMJ-R* evaluates the query by populating query stacks in one single pass of input streams. Nevertheless, this approach may generate many intermediate solutions. A solution of a root-to-leaf path in Q_s is called *intermediate*, if it does not participate in any final solution of Q. There are two reasons for a path solution to be intermediate: (1) it cannot be merged with other path solutions on a same data path, or (2) it can be merged but the result does not satisfy the structural constraints in Q that are not present in Q_s.

Example 4.4.1 *Consider evaluating the query Q_6 of Figure 4.6(b) on the data of Figure 4.6(a). Figure 4.7 shows the partial solutions of each query path of Q_{6s} and the merge-join results when the stream nodes b_4, c_7, and b_8 are processed. When c_7 is read, the result of merge-joining the partial solutions $soln = \{ra_1a_6c_7, ra_5a_6c_7\}$ and $pathSoln[B_6] = \{ra_1b_2c_3b_4\}$ is $\{ra_1a_6c_7b_2c_3b_4\}$. This result is not returned as a solution of Q_6, since c_7 and b_4 does not satisfy $C_4//B_6$. When b_8 is read, the result of merge-joining the partial*

solutions $soln = \{ra_1b_2c_3b_8\}$ *and* $pathSoln[C_4] = \{ra_1a_6c_7, ra_5a_6c_7\}$ *is* $\{ra_1a_6c_7b_2c_3b_8\}$. *This result is returned as the final answer of* Q_6. *The partial solutions* $\{ra_5a_6c_7\}$ *and* $\{ra_1b_2c_3b_4\}$, *which are outputs for the query path* $R//A_1//A_2/C_4$ *and* $R//A_1//B_3/C_5//B_6$ *in* Q_{6s} *respectively, do not participate in the answer of* Q_6. *Therefore they are intermediate solutions.*

Data node processed	Solutions added to $pathSoln[C_4]$	Solutions added to $pathSoln[B_6]$	Join Results	
b_4		$\{ra_1b_2c_3b_4\}$		
c_7	$\{ra_1a_6c_7, ra_5a_6c_7\}$		$\{ra_1a_6c_7b_2c_3b_4\}$	discarded
b_8		$\{ra_1b_2c_3b_8\}$	$\{ra_1a_6c_7b_2c_3b_8\}$	the answer of Q_6

Figure 4.7 Outputs of *PartialMJ-R* on Q_6 and data in Figure 4.6

Clearly, the intermediate solutions affect negatively the time and space worst case complexity of *PartialMJ-R*. When Q is a path, *PartialMJ-R* reduces to *PathStack-R* and does not produce intermediate solutions. Nevertheless, despite possible intermediate solutions, *PartialMJ-R* is sound and complete for evaluating partial path queries as the following theorem states.

Theorem 4.4.1 Algorithm *PartialMJ-R* correctly evaluates partial path queries with repeated labels on XML trees.

The proof of the theorem follows directly from the description of the algorithm and Proposition 4.3.2.

4.5 PartialPathStack-R: a Holistic Algorithm

To overcome the problem of intermediate solutions of Algorithm *PartialMJ-R*, we developed a holistic stack-based algorithm called *PartialPathStack-R* for the evaluation of partial path queries. In contrast to *PartialMJ-R*, *PartialPathStack-R* does not decompose a query into root-to-leaf paths. Instead, it matches the query graph to an XML tree as a whole. In this way, it avoids merge-joining path solutions. Also, unlike *PathStack-R*, *PartialPathStack-R* exploits multiple pointers per stack entry to avoid redundantly storing the same stream nodes in different stacks.

4.5.1 Preliminaries

As concluded in Section 4.1, a partial path query Q can be represented as a dag rooted at R. Let X denote a node and L denote a label in Q. We use for queries the functions defined in Section 5.1 with the following difference: Boolean function *isSink(X)* replaces function *isLeaf(X)* and returns $true$ if X is a sink node (i.e., it does not have outgoing edges in Q). Also, function *parents(X)* replaces function *parent(X)*, and returns the parent nodes of X in Q (X can have multiple parent nodes when Q is a dag).

As before, we associate every distinct node label L with a stream T_L and maintain a cursor C_L for that stream. However, we now associate a stack S_L with *every distinct node label* L (and not with every node in Q labeled by L). Initially, all stacks are empty and every cursor C_L points to the first node in T_L. During execution of the algorithm, the entries that stack S_L might contain correspond to stream nodes in T_L before C_L. The structure of a stack entry is now more complex in order to record additional information.

Before describing the structure of stack entries, we define an important concept, which is key to understanding *PartialPathStack-R*.

Definition 4.5.1 *Let Q be a partial path query, X be a node in Q, and T be an XML tree. The sub-dag of Q that comprises X and all its ancestor nodes is called* ancestor *query of X and is denoted as Q_X. We say that a node x_i in T* plays the role *of X if x_i is the image of X under an embedding of Q_X to T.*

A node in T_L can play multiple roles, each of which corresponds to a node in $occur(L)$. An entry e in stack S_L corresponds to a node in T_L and has the following three fields:

1. $(begin,\ end,\ level)$: the positional representation of the corresponding node in T_L.

2. $prevPos$: an array of size $|occur(L)|$ whose fields are indexed by the nodes in $occur(L)$. Given a node $X \in occur(L)$, $prevPos[X]$ is a pointer to the highest entry in S_L below e that plays the role of X. Following these pointers, we can access from e all the entries below e in S_L that play the role of X in leaf-to-root order in the XML tree. If X is the only node labeled by L (in which case, all the entries in S_l play a single role), $prevPos[X]$ refers to the entry just below e.

3. $ptrs$: an array of size k whose fields are indexed by the parent nodes $P_1, \ldots, P_k$ of all nodes labeled by L in Q. $ptrs[P_i]$ points to the highest among the entries in stack $S_{label(P_i)}$ that (a) play the role of P_i, and (b) correspond to ancestors of e in the XML tree. It is possible that for some P_i, $ptrs[P_i]$ is *null*. However, if e plays the role of a node $X \in occur(L)$, then $ptrs[P_i]$ is not null for every $P_i \in parents(X)$. Further, it is possible that some or all of $P_i \in parents(X)$, *label(P_i)* = L. In this case, $ptrs[P_i]$ points to an entry below e in the same stack S_L.

The expression $S_L.k$ denotes the entry at position k of stack S_L. The expression $S_L.k.ptrs[P_i]$ denotes the position of the entry in stack $S_{label(P_i)}$ recorded in the field $ptrs[P_i]$ of the entry $S_L.k$.

With every stack S_L, we associate an array $lastPos_L$ whose fields are indexed by the nodes in $occur(L)$. For a node $X \in occur(L)$, $lastPos_L[X]$ records the position of the highest entry in stack S_L that plays the role of node X. Therefore, starting from the position $lastPos_L[X]$, we can access all the entries in S_L that play the role of X in leaf-to-root order in the XML tree. Clearly, if X is the only node labeled by L, $lastPos_L[X]$ refers to the top entry in stack S_L.

As with previous two algorithms, during the execution of Algorithm *PartialPathStack-R*, the following properties hold: (1) The entries in *all* the stacks correspond to nodes located on the same path in the XML tree, and (2) Stack entries represent partial solutions of the query that can be extended to final solutions as the algorithm goes on. In what follows, we might not distinguish between an entry in a stack and its corresponding stream node.

4.5.2 The Algorithm

Algorithm *PartialPathStack-R* is presented in Listing 3. Given a partial path query Q, *PartialPathStack-R* processes stream nodes in ascending order of their $begin$ values and constructs partial and final solutions to Q. It exploits a topological order of the query nodes (i.e., a linear order of the query nodes which respects the partial order induced by the structural relationships of the query). Given a topological order, the nodes in Q are identified by their position in the topological order with 1 denoting the root node of Q.

Algorithm *PartialPathStack-R* calls procedure *cleanStacks* (line 4) introduced in Algorithm *PathStack-R* (Listing 1). For an incoming stream node C_L, *cleanStacks* pops out from *all* the stacks the nodes that are not ancestors of C_L in the XML tree.

Listing 3 Algorithm PartialPathStack-R

```
1  create a topological order 1, ..., n of the query nodes in Q, where n = |Q|, and identify each node by its position in the topological
   order.
2  while ¬end() do
3      L = getNextQueryLabel()
4      cleanStacks(C_L)
5      entry = constructCandEntry(L)          { entry has the structure of an entry in stack S_L }
6      if (R == L or ∃X ∈ occur(L) ∀P ∈ parents(X): entry.ptrs[P] ≠ null) then
7          push(S_L, entry)
8          sinkNodes = ∅
9          for (X ∈ occur(L)) do
10             if (isSink(X) and lastPos_L[X] == top(S_L))          { X is a sink node and the newly pushed entry in S_L plays
               the role of X }          then
11                 sinkNodes = sinkNodes ∪ {X}
           { Next the algorithm tests if the stacks contain solutions of Q and, if so, outputs them }
12         if (sinkNodes ≠ ∅ and ∀X ∈ nodes(Q): isSink(X) ⇒ lastPos_label(X)[X] ≠ null)    { entry plays the role of a sink
           node in the query and every sink node in the query has a stack entry that plays its role } then
13             if (n ∈ sinkNodes)      { entry plays the role of n, the last node in the topological order }      then
14                 outputSolutions(sinkNodes, n, lastPos_label(n)[n])
15             else
16                 i = lastPos_label(n)[n]
17                 repeat
18                     outputSolutions(sinkNodes, n, i)
19                     i = S_label(n).i.prevPos[n]
20                 until (i==null)
21     advance(C_L)
```

In line 5, Algorithm *PartialPathStack-R* calls function *constructCandEntry(L)* shown in Listing 4. For the stream node C_L under consideration, *constructCandEntry(L)* finds all the roles that C_L can play and records the information in a variable $candEntry$. Variable

Listing 4 Function constructCandEntry(L)

```
1  let candEntry be a variable that has the structure of an entry for stack S_L        { candEntry represents a candidate entry
   for stack S_L }
2  initialize candEntry so that its (begin, end, level) field is equal to C_L and each field in prevPos and ptrs is equal to null
3  if (R == L)      { we are at the root of the query } then
4     lastPos_L[1]=1
5  else
6     for (X ∈ occur(L)) do
7        create an array pptrs indexed by parents(X)
8        hasRole = true
9        for ( P ∈ parents(X)) do
10          if (lastPos_label(P)[P] ≠ null) then
11             entry = S_label(P).lastPos_label(P)[P]
12             if (P/X ∈ Q) and (entry.level ≠ C_L.level-1)      { child relationship between P and X is not satisfied by entry
               and C_L }      then
13                hasRole = false      { C_L does not play the role of X }
14          else
15             hasRole = false      { C_L does not play the role of X }
16          if (hasRole) then
17             pptrs[P] = lastPos_label(P)[P]
18       if (hasRole) then
19          for every P ∈ parent(X) do
20             candEntry.ptrs[P] = pptrs[P]
21          candEntry.prevPos[X] = lastPos_L[X]
22          lastPos_L[X] = top(S_L) + 1      { lastPos_L[X] is updated to reflect the position of candEntry after candEntry
            has been pushed on S_L }
23 return candEntry
```

$candEntry$ has the structure of an entry in stack S_L and represents a candidate entry for S_L. For a query node $X \in occur(L)$, node C_L plays the role of X iff for *each* parent

Listing 5 Procedure outputSolutions($outputSinkNodes$, $curNode$, $stackPos$)

```
1  solution[curNode] = stackPos
2  m = curNode-1
3  if (curNode = 1)   { curNode is the root query node } then
4     output(S_label(1).solution[1] ,..., S_label(n).solution[n])
5  else if (m ∈ outputSinkNodes)   { m is a sink node and the lastly pushed entry plays the role of m } then
6     outputSolutions(outputSinkNodes, m, lastPos_label(m)[m])
7  else if (isSink(m))   { m is a sink node and the lastly pushed entry does not play the role of m } then
8     i = lastPos_label(m)[m]
9     repeat
10       outputSolutions(outputSinkNodes, m, i)
11       i = S_label(m).i.prevPos[m]
12    until (i==null)
13 else {m is not a sink node}
14    i = min{S_label(c).solution[c].ptrs[m]}, c ∈ children(m)
15    repeat
16       outputSolutions(outputSinkNodes, m, i)
17       i = S_label(m).i.prevPos[m]
18    until (i==null)
```

P of X, there exists an entry $entry$ in stack $S_{label(P)}$ such that the structural relationship between $entry$ and C_L satisfies the structural relationship between P and X in the query (lines 9-15 in $constructCandEntry$). Note that it is not necessary to exhaustively visit the entries in $S_{label(P)}$ to find $entry$. The existence of $entry$ can be determined in constant time by just using the value of $lastPos_{label(P)}[P]$ (lines 10-12). If node C_L plays the role of X, then for every parent P of X, a pointer to the position $lastPos_{label(P)}[P]$ of stack $S_{label(P)}$ is generated and recorded first in a temporary array $pptrs$ (line 17) and then in the corresponding fields of $candEntry.ptrs$ (lines 19-20). The current value of $lastPos_L[X]$

is recorded in $candEntry.prevPos[X]$ (line 21). Finally, $lastPos_L[X]$ is updated to reflect the position of $candEntry$ in stack S_L after it is pushed there (line 22).

PartialPathStack-R uses the information returned by $constructCandEntry(L)$ to determine if node C_L is qualified for being pushed on stack S_L. Node C_L can be pushed on S_L iff it plays a role of at least one query node in $occur(L)$ (lines 6-7 in *PartialPathStack-R*). This way, only stream nodes that could eventually be part of solutions are pushed on stacks.

The timing for producing solutions is important in order to avoid generating duplicate solutions. Whenever node C_L that plays the role of a sink node in the query (lines 9-11) is pushed on a stack, and for every sink node in the query there is an entry in the stacks that plays this role (line 12), it is guaranteed that the stacks contain at least one solution to the query. Subsequently, procedure *outputSolutions* (Listing 5) is invoked to output all the solutions that involve C_L (lines 14 and 18). Note that since every time solutions are produced, they involve the newly pushed node C_L, *PartialPathStack-R* does not generate duplicate solutions.

A solution of a query is a tuple of nodes in the XML tree which are images of the query nodes under an embedding of the query to the XML tree. Procedure *outputSolutions* gradually produces the nodes in each solution in an order that corresponds to the reverse topological order of the query nodes. This way, the image of a query node is produced in a solution after the images of all its descendant nodes in the query are produced. Procedure *outputSolutions* takes three parameters: *outputSinkNodes*, *curNode*, and *stackPos*. Parameter *outputSinkNodes* denotes the set of those sink nodes of the query that are roles of the newly pushed node C_L. Parameter *curNode* denotes the query node currently under consideration. Parameter *stackPos* denotes the position in stack $S_{label(curNode)}$ currently under consideration. A solution under construction by *outputSolutions* is recorded in an array $solution$ indexed

by the query nodes. The image of *curNode* recorded in $solution[curNode]$ is the position of an entry in stack $S_{label(curNode)}$. Procedure *outputSolutions* calls itself recursively on the query nodes. When called on query node *curNode-1* (denoted as m), the following three cases are distinguished:

1. If node m is in *outputSinkNodes* (which implies that m is a sink node), only the entry in stack $S_{label(m)}$ pointed to by $lastPos_{label(m)}[m]$ is used as image of m for constructing solutions (line 6). As mentioned previously, this guarantees no duplicate solutions will be generated.

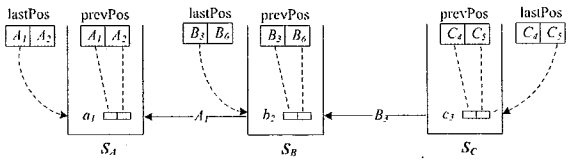
2. If node m is a sink node not in *outputSinkNodes*, the chain of entries that play the role of m in stack $S_{label(m)}$, starting with the entry pointed to by $lastPos_{label(m)}[m]$, are used as images of m for constructing solutions (lines 8-12).

3. If node m is an internal query node, the highest entry e in stack $S_{label(m)}$ that can be used in a solution as an image of m is the lowest ancestor in the XML tree of the images of the child nodes of m in the query. Since the child nodes of m have already been processed, their images are recorded in the array $solution$. Entry e is identified by the lowest position in stack $S_{label(m)}$ pointed to by pointers from stack entries that are images of the child nodes of m in the solution under construction (line 14). The chain of entries that play the role of m in stack $S_{label(m)}$ starting with e are used as images of m for constructing solutions (lines 15-18).

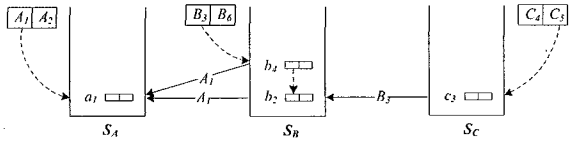
Procedure *outputSolutions* shown in Listing 5 deals with the case where no child edges are present in Q. When child edges are present in Q, we need to check the existence of outgoing child edges from each internal query node m, and modify the recursive calls

of *outputSolutions* on m (lines 15-18): only a single recursive call of *outputSolutions* ($outputSinkNodes$, m, i) is needed, where i is the position computed by line 14.

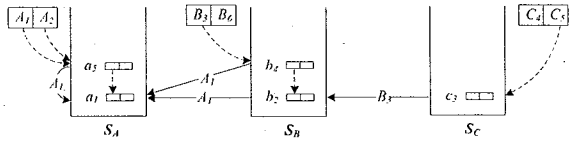
Example 4.5.1 *Figure 4.8 shows a running example for* PartialPathStack-R, *where, for simplicity, the stack for the query root R is omitted. We also do not show the $lastPos$ and $prevPos$ of query nodes with a single occurrence in the query. We use for Q_6 the following topological order: R, A_1, A_2, B_3, C_4, C_5, B_6. The input streams are T_A: a_1, a_5, a_6, T_B: b_2, b_4, b_8, and T_C: c_3, c_7. The initial value for the input stream cursors C_A, C_B, and C_C in that order is a_1, b_2, c_3. The state of the stacks after c_3, b_4, a_5, a_6, c_7, and b_8 are read is shown respectively in the Figures 4.8(a)-(f). After c_3 is read (Figure 4.8(a)), there is no entry in stack S_A that plays the role of A_2. Therefore $lastPos_A[A_2]$ is 0. Similarly, $lastPos_B[B_6]$ for stack S_B and $lastPos_C[C_4]$ for stack S_C are both 0. After a_5 is read (Figure 4.8(c)), given that a_5 plays the role of A_2 in addition to the role of A_1, its entry has an outgoing pointer that points to a_1 (A_1 is the parent of A_2), which is the last entry in S_A playing the role of A_1. The position of a_5 in S_A is recorded in both $lastPos_A[A_1]$ and $lastPos_A[A_2]$. Before $lastPos_A[A_1]$ is updated, its value (the position of a_1 in S_A) is recorded in $prevPos[A_1]$ of a_5. This indicates that a_1 is the highest entry in S_A below a_5 that plays the role of A_1. Finally, when b_8 is read (Figure 4.8(f)), given that b_8 plays the role of B_6, the entry for b_8 has two outgoing pointers which respectively point to c_3 and c_7 in stack S_C (C_4 and C_5 are the parents of B_6). The stack position of b_8 is recorded in $lastPos_B[B_6]$. Since B_6 is a sink node of Q_6, b_8 triggers the generation of solutions. Note that when A_1 is processed by* outputSolutions, *a_1 is chosen as a value for A_1 in the solution under construction since a_1 is below a_5 in stack S_A (line 14 in* outputSolutions*). The final answer for Q_6 is $\{ra_1a_6c_7b_2c_3b_8\}$.*



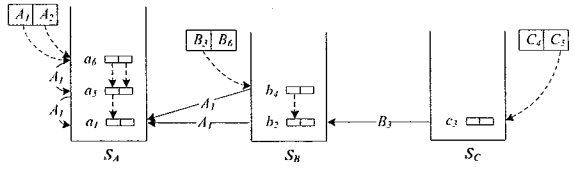
(a) state of stacks after reading c_3



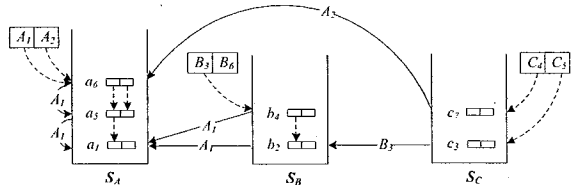
(b) state of stacks after reading b_4



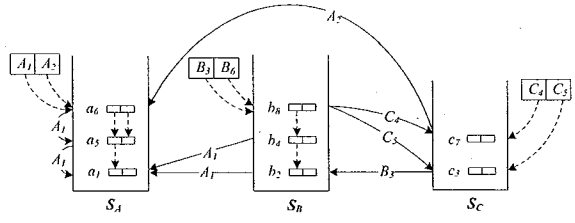
(c) state of stacks after reading a_5



(d) state of stacks after reading a_6



(e) state of stacks after reading c_7



(f) state of stacks after reading b_8

Figure 4.8 PartialPathStack-R on Q_6 and data in Figure 4.6

4.5.3 Analysis of PartialPathStack-R

Proposition 4.5.1 *A stream node x is pushed on stack S_L iff x plays the role of a query node X labeled by L.*

Proof. The *only if part* is straightforward given that Algorithm *PartialPathStack-R* pushes x to stack S_L only if there exists a query node X labeled by L such that, for *each* parent Y of X, the highest entry in the stack for Y that plays the role of Y satisfies the corresponding relationships between Y and X in the query.

The *if part*: If x is the node r, then x plays the role of the query node R and will be pushed on S_R. The proposition is trivially true. For a non-root stream node x, we prove the proposition by contradiction.

Let's assume x plays the role of a query node X labeled by L but is not in stack S_L. Then, for at least one parent Y of X, procedure *constructCandEntry* did not find a stream node that plays the role of Y. Since x plays the role of X, there must exist one stream node that plays the role of Y and the structural relationship between that node and x satisfies the structural relationship between Y and X in the query. Let y be the closet to x stream node above x in T, and let Y be labeled by M. There can be two reasons for procedure *constructCandEntry* not finding a stream node that plays the role of Y: (a) y is in stack S_M, but its position is not recorded in $lastPos_M[Y]$. However, since y plays the role of Y and it is the latest node pushed on stack S_M, its position is recorded in $lastPos_M[Y]$ by Procedure *constructCandEntry*, a contradiction. (b) y is not in S_M. But then, by applying the same reasoning recursively, we can conclude that the stream node r that plays the role

of R but not in stack S_R contradicting our assumption. Therefore, if x plays the role of a query node X labeled by L, it must be pushed on stack S_L. □

As a result of Proposition 4.5.1, Algorithm *PartialPathStack-R* will find and encode in stacks all the partial or complete (x plays the role of a sink node in Q) solutions involving x. When at least one complete solution is encoded in the stacks, procedure $ouputSolutions$ is invoked to output them. Therefore, Algorithm *PartialPathStack-R* correcly finds all the solutions to Q.

Following we provide the time and space complexity of Algorithm *PartialPathStack-R*. Given a partial path query dag Q and an XML tree T, let $height$ denote the hight of T, $indegree$ denote the maximum number of incoming edges to a query node, and $|Q|$ denote the size of Q. Other parameters are the same for the analysis of *PathStack-R*.

Theorem 4.5.1 Algorithm *PartialPathStack-R* correctly evaluates a partial path query Q on an XML tree T. The algorithm uses $O(height \times |Q|)$ space. It has the CPU complexity $O((input + output) \times |Q|)$ and the disk I/O complexity $O(input + output)$.

Proof. The disk I/O complexity of *PartialPathStack-R* is the same as *PathStack-R*, which is $O(input + output)$.

The space complexity depends mainly on how many stack entries are stored at a given point in time and the number of pointers associated with these entries. Note that for each stream node, it has at most one physical copy stored in a stack at any time, even that stream node plays multiple roles. The total number of stack entries at any time is thus $O(height)$. For each stack entry, the maximum number of outgoing pointers is $O(|Q|)$. Therefore, the worst case number of pointers in stacks is bounded by $height \times |Q|$.

Since *PartialPathStack-R* does not generate any intermediate solutions, the CPU time complexity of *PartialPathStack-R* depends mainly on the time spent on *getNextQuerylabel*, the time spent on *cleanStacks*, the number of calls to *constructCandEntry*, and the number of calls to *outputSolutions* to output solutions. As Algorithm *PathStack-R*, *PartialPathStack-R* spends $O(input \times log|label(Q)|) = O(input \times log|Q|)$ on calls to function *getNextQuerylabel* and $O(input \times |Q|)$ total time on calls to *cleanStacks*. For each stream node, Procedure *constructCandEntry* takes time in $O(|Q|)$. Procedure *outputSolutions* spends O($outdegree$) on each query node, due to the line 13 that finds the lowest stack position among pointers from its child nodes, thus it takes $O(output \times |Q|)$ to produce all the outputs. Therefore, *PartialPathStack-R* has the CPU time complexity $O((input + output) \times |Q|)$. □

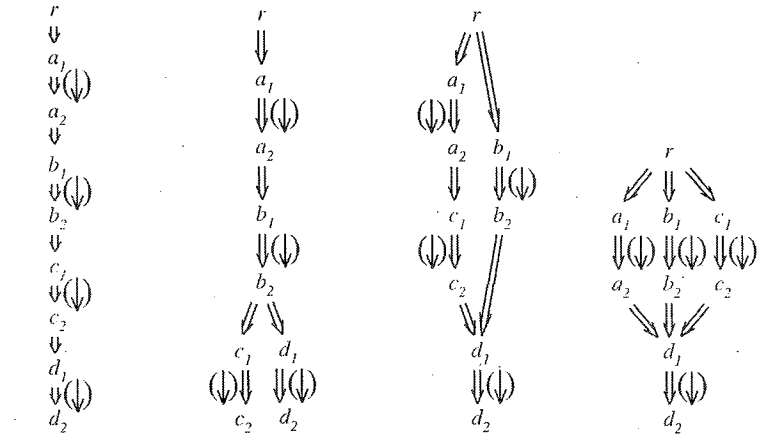
Clearly, if the size of the query is insignificant compared to the size of data, *PartialPathStack-R* is asymptotically optimal for partial path queries with repeated labels. Note that when a partial path query is a path, *PartialPathStack-R* uses less space than Algorithm *PathStack-R* does. The reason is that, in Algorithm *PathStack-R*, some stream nodes might have multiple copies in different stacks. Thus the number of entries of every stack may reach $height$ at a time. For *PartialPathStack-R*, the total number of entries of all the stacks is bound by $height$ at any time.

4.6 Experimental Evaluation

We ran a comprehensive set of experiments to measure the performance of *IndexPaths-R*, *PartialMJ-R* and *PartialPathStack-R*. In this section, we report on their experimental evaluation.

```
<!ELEMENT r (a+)>
<!ELEMENT a (a*, b+)>
<!ELEMENT b (b*, c*)>
<!ELEMENT c (c*, d+)>
<!ELEMENT d (d*, a*)>
```

Figure 4.9 DTD for the synthetic datasets SD_1 and SD_2



(a) $Q_1(Q_5)$ (b) $Q_2(Q_6)$ (c) $Q_3(Q_7)$ (d) $Q_4(Q_8)$

Figure 4.10 Partial path queries.

Setup. We evaluated the performance of the algorithms on both benchmark and synthetic datasets. For the benchmark dataset, we used the *Treebank* [66] XML document. This dataset consists of around 2.5 million nodes having 250 distinct element tags and its maximum depth is 36. This dataset includes multiple recursive elements. We used two synthetic datasets (SD_1 and SD_2). They are random XML trees generated by IBM's XML Generator [67], based on the DTD shown in Figure 4.9. The parameter $MaxRepeats$ (that determines the maximum number a node appears as a child of its parent node) was set to 4. The parameter $numLevels$ (that determines the maximum number of tree levels) was set to 12 for SD_1, and 20 for SD_2. The XML trees used in both SD_1 and SD_2 consist

of 1.5 million nodes. By construction, the two synthetic datasets include highly recursive structures. For each measurement on the synthetic datasets, $five$ different XML trees of the same number of nodes were used. Each displayed value in the plots is the average over these $five$ measurements.

On each of the three datasets, we tested the eight queries shown in Figure 4.10. Queries Q_1 to Q_4 include only descendant relationships, while queries Q_5 to Q_8 include child relationships as well. Our query set comprises a full spectrum of partial path queries, from simple path-pattern queries to complex dags. The queries are appropriately modified for the $Treebank$ dataset, so that they can all produce results. Thus, node D_2 is removed, and node labels R, A, B, C and D correspond to $FILE$, S, VP, NP and NN, respectively, on $Treebank$.

We implemented all algorithms in C++, and ran our experiments on a dedicated Linux PC (AMD Sempron 2600+) with $2GB$ of RAM.

Execution time on fixed datasets. We compared the execution time of *IndexPaths-R*, *PartialMJ-R* and *PartialPathStack-R* for evaluating the queries in Figure 5.7 over the three datasets. For queries Q_1 and Q_5, which are path-pattern queries, we also measured the execution time of algorithm *PathStack-R*.

Figure 4.11(a), 4.11(b) and 4.11(c) present the evaluation results. Figure 5.3.2 shows the number of solutions obtained per query in each dataset.

As we can see, *PartialPathStack-R* has the best time performance, and in many cases it outperforms either *IndexPaths-R* or *PartialMJ-R* by a factor almost 3. Its performance is stable, and does not degrade on more complex queries and on data with highly recursive structures.

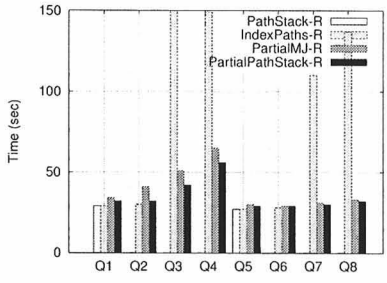
As expected, all algorithms perform almost as fast as *PathStack-R* in the case of the path-pattern queries Q_1 and Q_5. The execution time of *IndexPaths-R* is high for queries with a large number of path queries generated from the index tree, that is, for queries Q_3, Q_4, Q_7 and Q_8.

The performance of *PartialMJ-R* is affected by the existence of intermediate solutions. For example, when evaluating Q_2 on the synthetic dataset SD_2, *PartialMJ-R* shows the worst performance (Figure 4.11(c)), due to the large amount of intermediate solutions generated.

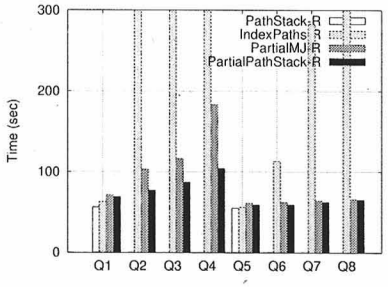
The performance of both *PartialMJ-R* and *PartialPathStack-R* in all datasets is affected by the number of solutions. This confirms our complexity results that show dependency of the execution time on the input and output size (number of solutions). In the case of queries Q_2 and Q_4 on $SD2$ (Figure 4.11(c)), where the number of solutions is high (Figure 5.3.2), the execution time of *PartialMJ-R* strongly increases.

Execution time varying the input size. We compared the execution time of *IndexPaths-R*, *PartialMJ-R*, and *PartialPathStack-R* as the size of the input dataset increases. Figures 4.12(a), 4.12(c), and 4.12(e) report on the execution time of the algorithms increasing the size of the synthetic dataset SD_2 for queries Q_2, Q_4 and Q_8, respectively, of Figure 4.10. *PartialPathStack-R* consistenly has the best performance.

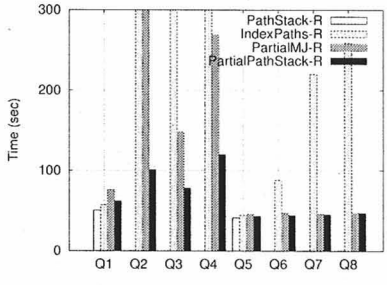
Figures 4.12(b), 4.12(d), and 4.12(f) present the number of solutions of Q_2, Q_4 and Q_8, respectively, increasing the size of the dataset. As we can see, an increase in the input size results in an increase in the output size (number of solutions). When the input and the output size go up, the execution time of the algorithms increases. This confirms the



(a) Execution time ($Treebank$)



(b) Execution time (SD_1)



(c) Execution time (SD_2)

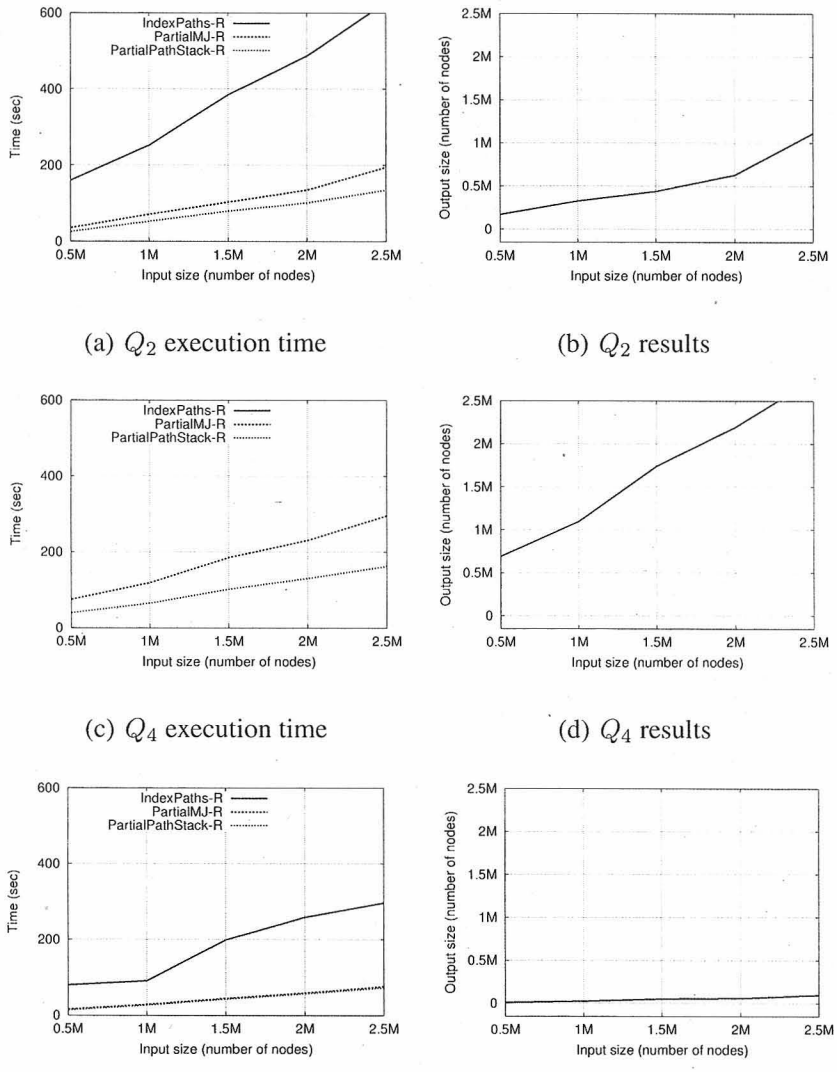
Query	$Treebank$	SD_1	SD_2
Q_1	94859	99576	884899
Q_2	94859	439725	2991754
Q_3	678011	855090	1542299
Q_4	1565355	1742980	3779766
Q_5	1136	3527	39392
Q_6	1136	3715	51058
Q_7	1239	5510	45944
Q_8	1258	6081	51583

(d) Number of solutions

Figure 4.11 Evaluation of queries on the three datasets.

complexity results that show dependency of the execution time on the input and output size.

We also observe that as the input and the output size increase, the execution time of *PartialPathStack-R* increases very slowly. In the experimental evaluation of query Q_4, the output size (Figure 4.12(d)) increases sharper than in the evaluation of query Q_2 (Figure 4.12(b)). The execution time of *PartialPathStack-R* is only slightly higher in the evaluation of Q_4 (Figure 4.12(c)) than in the evaluation of Q_2 (Figure 4.12(a)). In contrast, the execution time of *PartialMJ-R* is strongly affected. The reason is that, for *PartialMJ-R*,



(a) Q_2 execution time

(b) Q_2 results

(c) Q_4 execution time

(d) Q_4 results

(e) Q_8 execution time

(f) Q_8 results

Figure 4.12 *PartialMJ-R* vs *PartialPathStack-R*, varying the size of the XML tree.

an increase in the output size is accompanied by an increase in the number of intermediate solutions produced during evaluation. Notice also that *IndexPaths-R* is extremely slow in Q_4 as it includes the evaluation of a large number of equivalent path queries.

Query Q_8 is more "restrict" than Q_4 due to the child relationships (Figure 5.6(d)). It produces only a small number of solutions (Figure 4.12(f)). Given the small number

of solutions, and consequently a small number of intermediate solutions in *PartialMJ-R*, *PartialMJ-R* and *PartialPathStack-R* have the similar performance (Figure 4.12(e)).

CHAPTER 5

EVALUATING PARTIAL TREE-PATTERN QUERIES ON XML INVERTED LISTS

In this chapter, we present an original polynomial time holistic algorithm for PTPQs in the indexed streaming model. The chapter is organized as follows. Section 5.1 describes data structures for PTPQ indexed streaming evaluation. We present our evaluation algorithms in Section 5.2. Section 5.3 presents and analyses our experimental results.

5.1 Data Structures and Functions for PTPQ Evaluation

We present in this section the data structures and operations we use for PTPQ evaluation in the inverted lists model.

Query functions. Let Q be a query, X be a node in Q, and p_i be a partial path in Q. Node X is called *sink node of* p_i, if p_i annotates X but no any descendant nodes of X in Q. We make use of the following functions in the evaluation algorithm. Function *sinkNodes*(p_i) returns the set of sink nodes of p_i. Function *partialPaths*(X) returns the set of partial paths that annotate X in Q and *PPsSink*(X) returns the set of partial paths where X is a sink node. Boolean function $isSink(X)$ returns $true$ iff X is a sink node in Q (i.e., it does not have outgoing edges in Q). Function *parents*(X) returns the set of parent nodes of X in Q. Function *children*(X) returns the set of child nodes of X in Q.

Operations on inverted lists. With every query node X in Q, we associate an inverted list T_X of the positional representation of the nodes labeled by x in the XML tree. The nodes in T_X are ordered by the their *start* field (see Section 3.1). To access sequentially the nodes

in T_X, we maintain a cursor. We use C_X to denote the node currently pointed by the cursor in T_X and call it the *current match* of X. Operation *advance*(X) moves the cursor to the next node in T_X. Function *eos*(X) returns true if the cursor has reached the end of T_X.

Stacks. With every query node X in Q, we associate a stack S_X. An entry e in stack S_X corresponds to a node in T_X and has the following two fields:

1. A field consisting of the triplet $(start, end, level)$ which is the positional representation of the corresponding node in T_X.
2. A field $ptrs$ which is an array of pointers indexed by $parents(X)$. Given $P \in parents(X)$, $ptrs[P]$ points to the highest among the entries in stack S_P that correspond to ancestors of e in the XML tree.

Stack operations. We use the following stack operations: *push*(S_X,*entry*) which pushes *entry* on the stack S_X, *top*(S_X) which returns the top entry of stack S_X, and *bottom*(S_X) which returns the bottom entry of stack S_X. Boolean function *empty*(S_X) returns *true* iff S_X is empty.

Initially, all stacks are empty, and for every query node X, its cursor points to the first node in T_X. At any point during the execution of the algorithm, the entries that stack S_X can contain correspond to nodes in T_X before the current match C_X. The entries in a stack below an entry e are ancestors of e in the XML tree. Stack entries form partial solutions of the query that can be extended to become the solutions as the algorithm goes on.

Matching query subdags. Recall that C_X denotes the current match of the query node X. Below, we define a concept which is important for understanding the query evaluation algorithm.

Definition 5.1.1 (Current Binding) *Given a query Q, let X be a node in Q and Q_X be the subdag (subquery) of Q rooted at X. The* current binding of Q *is the tuple β of current matches of the nodes in Q. Node X is said to* have a solution in β, *if the matches of the nodes of Q_X in β form a solution for Q_X.*

If node X has a solution in β, then the following two properties hold: (1) C_X is the ancestor of all the other current matches of the nodes in Q_X, and (2) current matches of the query nodes in Q_X in the same partial path lie on the same path in the XML tree.

When all the structural relationships in Q are regarded as descendant relationships, we can show the following proposition.

Proposition 5.1.1 *Let X be a node in a query Q where all the structural relationships are regarded as descendant relationships, $\{Y_1, \ldots, Y_k\}$ be the set of child nodes of X in Q, and $\{p_1, \ldots, p_n\}$ be the set of partial paths annotating X in Q. Let also β denote the current binding of Q. Node X has a solution in β if and only if the following three conditions are met:*

1. *All Y_is have a solution in β.*
2. *C_X is a common ancestor of all C_{Y_i}s in the XML tree.*
3. *For each partial path p_j, the current matches of all the sink nodes of p_j that are descendants of X lie on the same path in the XML tree.*

The proof follows directly from Definition 5.1.1. Clearly, if X is a sink node, it satisfies the conditions of Proposition 5.1.1, and therefore, it has a solution in β.

As an example for Proposition 5.1.1, consider evaluating query Q_3 of Figure 6.18(b) on the XML tree of Figure 6.8(a). Suppose the cursors of R, A, B, D, C, E, G, and F are at r, a_1, b_1, d_1, c_1, e_1, g_1, and f_1, respectively. By Proposition 5.1.1, node D has a solution

in the the current binding β of Q_3, since (1) child nodes E and F both have a solution in β; (2) b_1 is a common ancestor of e_1 and f_1; and (3) E and F are the only descendant sink nodes of D in partial paths p_1 and p_2, respectively. However, node B does not have a solution in β because the condition 3 of Proposition 5.1.1 is violated: g_1 and f_1, which respectively are the current matches of the descendant sink nodes G and F in partial path p_2, are not on the same path in the XML tree.

5.2 PTPQ Evaluation Algorithm

The flexibility of the PTPQ language in specifying queries and its increased expressive power makes the design of an evaluation algorithm challenging. Two outstanding reasons of additional difficulty are: (1) a query is a dag (which in the general case is not merely a tree) augmented with constraints, and (2) the same-path constraints should be enforced for all the nodes in a partial path in addition to enforcing structural relationships. In this section, we present our holistic evaluation algorithm $PartialTreeStack$, which efficiently resolves these issues. The presentation of the algorithm is followed by an analysis of its correctness and complexity.

5.2.1 Algorithm PartialTreeStack

Algorithm $PartialTreeStack$ operates in two phases. In the first phase, it iteratively calls a function called $getNext$ to identify the next query node to be processed. Solutions to individual partial paths of the query are also computed in this phase. In the second phase, the partial path solutions are merge-joined to compute the answer of the query.

Function $getNext$ Function $getNext$ is shown in Listing 6. It is called on a query node and returns a query node (or $null$). Starting with the root R of the query dag Q, function $getNext$ traverses the dag in left-right and depth-first search mode. For every node under consideration, $getNext$ recursively calls itself on each child of that node. This way, $getNext$ first reaches the left-most sink node of Q. Starting from that sink node, it tries to find a query node X with the following three properties:

1. X has a solution in the current binding β of Q but none of X's parents has a solution in β.
2. Let P be a parent of X in the invocation path of $getNext$. The current match of X, i.e., C_X, has the smallest $start$ value among the current matches of all the child nodes of P that have a solution in β.
3. For each partial path p_i annotating X, C_X has the smallest $start$ value among the current matches of all the nodes annotated by p_i that have a solution in β.

Node X is the node returned by $getNext(R)$ to the main algorithm for processing. The first property guarantees that: (1) C_X is in a solution of Q_X, and (2) a query node match in a solution of Q is always returned before other query node matches in the same solution that are descendants of it in the XML tree. The third property guarantees that matches of query nodes annotated by the same partial path are returned in the order of their $start$ value (i.e., according to the pre-order traversal of the XML tree).

During the traversal of the dag, function $getNext$ discards node matches that are guaranteed not to be part of any solution of the query by advancing the corresponding cursors. This happens when a structural constraint of the dag or a same-path constraint is violated.

Listing 6 Function getNext(X)

```
1  if (isSink(X) ∨ knownSoln[X] ) then
2      return X
3  for (Y_i ∈ children(X)) do
4      invPath[Y_i] ← invPath[X] + 'Y_i'
5      Y ← getNext(Y_i)
6      if (Y ≠ Y_i ∧ Y ≠ X ) then
7          return Y
8  Y_min ← minarg_{Y_i}{C_{Y_i}.start}, Y_max ← maxarg_{Y_i}{C_{Y_i}.start}, where Y_i ∈ children(X) ∧ knownSoln[Y_i]
9  while (C_X.end < C_{Y_max}.start) do
10     advance(X)
11 if (C_X.start < C_{Y_min}.start) then
12     updateSPStatus(X)
13     if (∀p_i ∈ partialPaths(X): SP[p_i]) then
14         knownSoln[X] ← true
15         return X
16     else
17         return null
18 if (bottom(S_X) is an ancestor of C_{Y_min}) then
19     if (∃P ∈ parents(Y_min): C_P is an ancestor of C_{Y_min}) then
20         return the lowest ancestor of P among the nodes in invPath[X]
21     if (∃ sink node Z ∈ Q: partialPaths(Z)⊆ partialPaths(Y_min)∧ C_Z.start < C_{Y_min}.start) then
22         return the lowest ancestor of Z among the nodes in invPath[X]
23 if (∀p_i ∈ partialPaths(Y_min): SP_{Y_min}[X, p_i] ≠ null) then
24     return Y_min
25 updateSPStatus(X)
26 if (∀p_i ∈ partialPaths(X): SP[p_i] )) then
27     return Y_min
28 else
29     return null
```

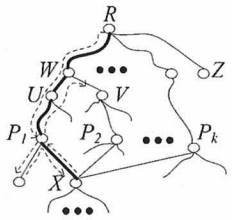


Figure 5.1 Traversal of a query dag by $getNext$

Dealing with the query dag. Since Q is a dag, some nodes of Q along with their subdags could be visited multiple times by $getNext$ during its traversal of Q. This happens when a node has multiple parents in Q. Figure 5.1 shows a scenario of the traversal of a query dag by $getNext$, where node X has parents $P_1, \ldots, P_k$. Function $getNext$ will be called on X from each one of the k parents of X. To prevent redundant computations, a boolean array, called $knownSoln$, is used. Array $knownSoln$ is indexed by the nodes of Q. Given a node X of Q, if $knownSoln[X]$ is $true$, $getNext$ has already processed the subdag Q_X rooted at X, and X has a solution in the current binding β of Q. In this case, subsequent calls of $getNext$ on X from other parents of X are not processed on the subdag Q_X since they are known to return X itself.

The traversal of the query nodes is not necessarily in accordance with the pre-order traversal of the query node matches in the XML tree. It is likely that the current match of a node X already visited by $getNext$ has larger $start$ value than that of a node that has not been visited yet. If this latter node is an ancestor of X and has a match that participates in a solution of Q, this match should be returned by $getNext$ before the match of X in the same solution is returned. In order to enforce this returning order, we let $getNext$ "jump" to and continue its traversal from an ancestor of X before X is returned (lines 19-20 in $getNext$).

The target ancestor node of X is chosen as shown in the example below: consider again the dag of Figure 5.1. The path from the root R to X in bold denotes the invocation path of $getNext$ from R to X. The invocation path is recorded in an array $invPath$ associated with each query node (line 4). Assume P_1 is the node under consideration by $getNext$, and P_1 has no solution in β. Assume also that P_2 has not yet been returned by $getNext$ but has a solution in β. Function $getNext$ on P_1 will return the *lowest ancestor* of P_2 among the nodes of $invPath[P_1]$ (which is node W). This enforces $getNext$ to go upwards along the invocation path of P_1 until it reaches W. From there, $getNext$ continues its traversal on the next child V of W.

The same technique is also used when there is an unvisited node Z annotated by a partial path that also annotates X, but the current match C_Z of Z has a smaller $start$ value than C_X. The existence of such a node is detected using the sink nodes of Q (lines 21-22). This technique ensures that the matches of nodes in a same partial path are returned by $getNext$ in the order of their $start$ value.

Dealing with the same-path constraint. Let X denote the node currently under consideration by $getNext$. After $getNext$ finishes its traversal of the subdag Q_X and comes back to X, it invokes procedure $updateSPStatus$ (lines 12 and 25). Procedure $updateSPStatus$ (shown in Listing 7) checks the satisfaction of the same-path constraints for the subdag Q_X, and updates the data structures SP and SP_Y (described below) accordingly to reflect the result of the check.

Data structure SP is a boolean array indexed by the set of partial paths annotating X in the query Q. For each partial path p_i, $SP[p_i]$ indicates whether the same-path constraint for p_i in Q_X is satisfied by the matches of nodes in Q_X (i.e., whether the matches of the

Listing 7 Procedure updateSPStatus(X)

```
1  for (p_i ∈ partialPaths(X)) do
2      let nodes denote the set of sink nodes of p_i that are descendants of X in Q
3      let node denote the node in nodes whose current match has the smallest end value
4      matches1 ← {C_Y | Y ∈ nodes}
5      SP[p_i] ← false
6      if (onSamePath(matches1)) then
7          SP[p_i] ← true
8      else
9          if (∃Y ∈ nodes : ¬ empty(S_Y)) then
10             matches2 ← {empty(S_Y)?C_Y : top(S_Y) | Y ∈ nodes}
11             if (onSamePath(matches2)) then
12                 SP[p_i] ← true
13     if (¬SP[p_i]) then
14         advanceUntilSP(nodes)
15         if (noMoreSolns) then
16             return
17     for (every child node Y of X annotated by p_i) do
18         SP_Y[X, p_i] ← SP[p_i]? C_node : null
```

Function onSamePath($matches$)

```
1  minEnd ← min_{m∈matches}{m.end}
2  maxStart ← max_{m∈matches}{m.start}
3  return (maxStart ≤ minEnd)
```

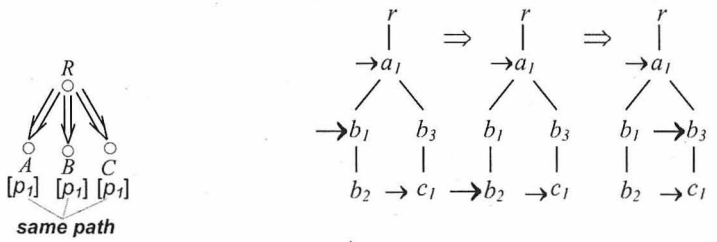
Procedure advanceUntilSP($nodes$)

```
1  repeat
2      minENode ← minarg_{Y∈nodes}{C_Y.end}
3      advance(minENode)
4      if (eos(minENode)) then
5          noMoreSolns ← true
6      matches ← {C_Y | Y ∈ nodes}
7  until (noMoreSolns ∨ onSamePath(matches))
```

nodes that are below X and are annotated by p_i in Q lie on the same path in the XML tree). Let $nodes$ denote the sink nodes of p_i in Q_X (line 2 in $updateSPStatus$). In order to check the same-path constraint for p_i, it is sufficient to check whether the matches of sink nodes in $nodes$ lie on the same path in the XML tree. Note that the match of a sink node can be its current match or the one that has already been returned by $getNext$ and is now in its stack.

Procedure $updateSPStatus$ uses function $onSamePath$ to check if the matches of a set of query nodes lie on the same path in the XML tree (lines 6 and 11). This check is based on Proposition 3.1.1. If the same-path constraint is not satisfied, procedure $advanceUntilSP$ is invoked to advance the cursors of the nodes in $nodes$ until the current matches of the nodes lie on the same path in the XML tree or one of the cursors reaches the end of its list. In the latter case, it is guaranteed that there are no new solutions for Q. Hence, a boolean flag $noMoreSolns$ is set to $false$ in order for $PartialTreeStack$ to end the evaluation (line 5 in $advanceUntilSP$). During each iteration in $advanceUntilSP$, the node in $nodes$ whose current match has the smallest end value is chosen and its cursor is advanced (lines 2-3). This way of advancing the cursors guarantees that all the matches of the nodes in $nodes$ that satisfy the same-path constraint will be eventually detected. Figure 5.2 shows an example of cursor movement during evaluation that results in the current matches of the sink nodes of a query to lie on the same path.

Every non-root query node Y in Q is associated with a two-dimensional array SP_Y. The first dimension of SP_Y is indexed by the parents of Y in Q, while the second one is indexed by the partial paths annotating Y in Q. For every parent X of Y and partial path p_i, if the same-path constraint for p_i in Q_X is satisfied, $SP_Y[X, p_i]$ stores the current match of $node$ (line 18 in $updateSPStatus$). $node$ denotes the sink node of p_i in the subdag Q_X



(a) Query Q_2 (b) XML tree and positions of cursors of Q_2

Figure 5.2 A sequence of cursor movements resulting in the current matches of sink nodes A, B and C of Q_2 to lie on the same path

whose current match has the smallest *end* value (line 3). Otherwise, $SP_Y[X, p_i]$ is set to *null* (line 18). Note that *node* is not necessarily a node in Q_Y but can be a node in the subdag rooted at a sibling of Y under the common parent X. Array SP_Y is updated by procedure $updateSPStatus$ when the parent X of Y is under consideration by $getNext$, and Y has a solution in the current binding of Q. $SP_Y[X, p_i]$ is possibly reset to *null* when the cursor of node Y is advanced and the current match C_Y is on a different path than the match stored in $SP_Y[X, p_i]$ (Procedure $resetSPFlags$ shown in Listing 8).

Array SP_Y records the execution states that are needed to prevent redundant computations of $getNext$. For a selected node Y, the non-*null* values of SP_Y indicate that node Y has a solution in the current binding of Q and should be returned by $getNext$ (lines 23-24 in $getNext$). In this case, no call to procedure $updateSPStatus$ is needed.

Main Algorithm Listing 8 shows the main part of $PartialTreeStack$. The main part repeatedly calls $getNext(R)$ to identify the next candidate node for processing (line 4). For a selected node X, $PartialTreeStack$ removes from some stacks entries that are not ancestors of C_X in the XML tree (line 9). The cleaned stacks are: (1) the stack of X, (2)

Listing 8 Algorithm PartialTreeStack

```
1  noMoreSolns ← false
2  while (¬end() ∧ ¬noMoreSolns ) do
3      Initialize the fields of knownSoln to true for all the sink nodes of Q, and to false for all the non-sink nodes.
4      X ← getNext(R)
5      if (X ≠ null) then
6          nodes ← parents(X) ∪ {X}
7          for (p_i ∈ PPsSink(X)) do
8              nodes ← nodes ∪ sinkNodes(p_i)
9          for every Y ∈ nodes, pop out entries that are not ancestors of C_X from stack S_Y
10         if ((X = R) ∨ (∀P ∈ parents(X): top(S_P) and C_X satisfy the structural relationship between P and X in Q)) then
11             push(S_X, (C_X, pointers to the top entry of every parent stack of X))
12             for (p_i ∈ PPsSink(X)) do
13                 if (∀Y ∈ sinkNodes(p_i): ¬empty(S_Y)) then
14                     outputPPSolutions(p_i, X)
15         advance(X)
16         resetSPFlags(X)
17 mergeAllPPSolutions()

Function end()
1  return ∀ node X ∈ Q: isSink(X) ⇒ eos(X)

Procedure resetSPFlags(X)
1  for (Y ∈ parents(X)p_i ∈ partialPaths(X)) do
2      if (¬onSamePath({C_X, SP_X[Y, p_i]})) then
3          SP_X[Y, p_i] ← null
```

the parent stacks of X, and (3) the stacks of sink nodes of every partial path of which X is a sink node (lines 6-8).

Subsequently, $PartialTreeStack$ checks if for every parent P of X, the top entry of stack S_P and C_X satisfy the structural relationship between P and X in the query (line

10). If this is the case, we say that C_X has *ancestor extensions*. Then, $PartialTreeStack$ creates a new entry for C_X and pushes it on S_X (line 11).

If X is a sink node of a partial path p_i and the stacks of all the sink nodes of p_i are non-empty (lines 12-13), it is guaranteed that the stacks contain at least one solution of p_i. Subsequently, procedure *outputPPSolutions* is invoked to output all the solutions of p_i that involve C_X (line 14). Procedure *outputPPSolutions* iteratively generates the solutions for p_i which are encoded in the stacks. Such a procedure can be found in [43].

Finally, procedure $mergeAllPPSolutions$ is called to merge-join all the partial path solutions in order to form the answer of the query (line 17). The details are simple and are omitted here in the interest of space.

5.2.2 An Example

We evaluate query Q_3 of Figure 6.18(b) on the XML tree of Figure 6.8(a) using Algorithm $PartialTreeStack$. The answer is shown in Figure 5.3(c). In Figure 5.5 and Figure 5.4, we show respectively, different snapshots of the query stacks and the contents of arrays SP_A, SP_B, and SP_D, during the execution of the algorithm. Initially, the cursors of R, A, B, D, C, E, G, and F are at r, a_1, b_1, d_1, c_1, e_1, g_1, and f_1, respectively. Before the first call of $getNext(R)$ returns r, g_1 is discarded by $advanceUntilSP$ because g_1 and f_1 are not on the same path. Right after the eighth call returns e_1, the stacks contain solutions for the partial path p_1, and are produced by $outputPPSolutions$ (Figure 5.5(a)). At this time, the cursors of R, A, B, D, C, E, G, and F are at ∞, ∞, b_2, d_2, c_2, e_2, g_2, and f_2 respectively. In the next call, $getNext$ first goes up from D to R, then continues on B because b_2 is the ancestor of d_2. This call finally returns g_2 since $g_2.start < d_2.start$. Subsequently, the solutions for the partial path p_2 are produced (Figure 5.5(b)). The eleventh call returns g_3

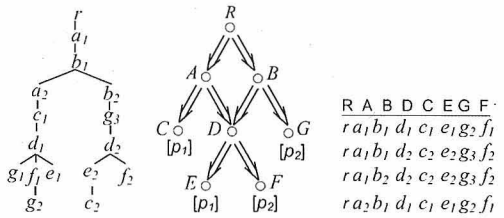


Figure 5.3 (a) An XML tree T, (b) Query Q_3, (c) the answer of Q_3 on T

SP	Calls of getNext(R)								
	1	2	3	4	5	9	10	11	12
$SP_A[R, p_1]$	e_1	e_1	e_1	e_1					
$SP_A[R, p_2]$	g_2	g_2	g_2	g_2					
$SP_B[R, p_1]$	e_1	e_1	e_1						
$SP_B[R, p_2]$	g_2	g_2	g_2						
$SP_D[A, p_1]$	e_1	e_1	e_1	e_1	e_1	c_2	c_2	c_2	c_2
$SP_D[A, p_2]$	f_1	f_1	f_1	f_1	f_1	f_2	f_2	f_2	f_2
$SP_D[B, p_1]$	e_1	e_1	e_1	e_1	e_1		e_2	e_2	e_2
$SP_D[B, p_2]$	g_2	g_2	g_2	g_2	g_2		f_2	f_2	f_2

Figure 5.4 The contents of SP_A, SP_B, and SP_D for Q_3 during execution of $PartialTreeStack$

instead of d_2 because $g_3.start < d_2.start$. After f_2 and c_2 are returned, the solutions for p_2 and p_1 are generated respectively in that order (Figure 5.5(c)). Finally, these partial path solutions are merge-joined to form the answer of Q_3 (Figure 5.3(c)).

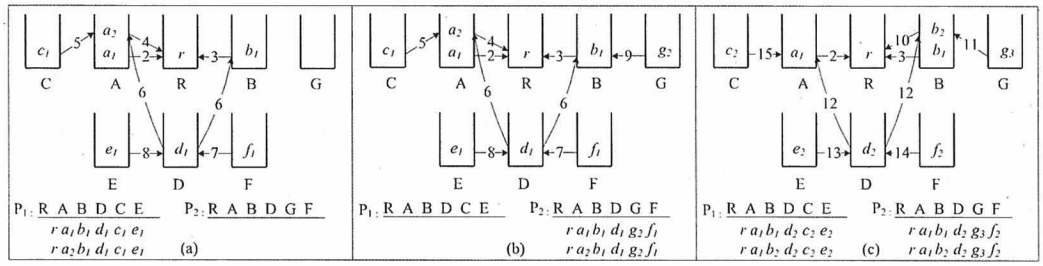


Figure 5.5 Three snapshots of the execution of $PartialTreeStack$ on query Q_3 and the XML tree T of Figure 5.3 (the numbers labeling the pointers denote the call to $getNext(R)$ as a result of which these pointers were created)

5.2.3 Analysis of PartialTreeStack

Correctness. Assuming that all the structural relationships in a PTPQ Q are regarded as descendant, whenever a node X is returned by $getNext(R)$, it is guaranteed that the current match C_X of X participates in a solution of subdag Q_X. These solutions of Q_X constitute of a superset of its solutions appearing in the answer of Q. Moreover, $getNext(R)$ always returns a match before other descendant matches of it in a solution of Q. In the main part of $PartialTreeStack$, C_X is pushed on S_X iff C_X has ancestor extensions. Whenever C_X is popped out of its stack, all the solutions involving C_X have been produced. Based on these observations, we can show the following proposition.

Proposition 5.2.1 *Given a PTPQ Q and an XML tree T, algorithm $PartialTreeStack$ correctly computes the answer of Q on T.*

Complexity. Given a PTPQ Q and an XML tree T, let $|Q|$ denote the size of the query dag, N denote the number of query nodes of Q, P denote the number of partial paths of Q, IN denote the total size of the input lists, and OUT denote the size of the answer of Q on T. The *ancestor dag* of a node X in Q is the subdag of Q consisting of X and its ancestor nodes. In [36], the *recursion depth* of X of Q in T is defined as the maximum

number of nodes in a path of T that are images of X under an embedding of the ancestor dag of X to T. We define the recursion depth of Q in T, denoted by D, as the maximum of the recursion depths of the query nodes of Q in T.

Theorem 5.2.1 *The space usage of Algorithm $PartialTreeStack$ is $O(|Q| \times D)$.*

The proof follows from the fact that: (1) the number of entries in each stack at any time is bounded by D, and (2) for each stack entry, the size of $ptrs$ is bounded by the out-degree of the corresponding query node.

When Q has no child structural relationships, Algorithm $PartialTreeStack$ ensures that each solution produced for a partial path is guaranteed to participate in the answer of Q. Therefore, no intermediate solutions are produced. Consequently, the CPU time of $PartialTreeStack$ is independent of the size of solutions of any partial path in a descendant-only PTPQ query.

The CPU time of $PartialTreeStack$ consists of two parts: one for processing input lists, and another for producing the query answer. Since each node in an input list is accessed only once, the CPU time for processing the input is calculated by bounding the time interval between two consecutive cursor movements. The time interval is dominated by updating array SP_X for every node X and is $O(|Q| \times P)$. The CPU time on generating partial path solutions and merge-joining them to produce the query answer is $O((IN + OUT) \times N)$.

Theorem 5.2.2 *Given a PTPQ Q without child structural relationships and an XML tree T, the CPU time of algorithm $PartialTreeStack$ is $O(IN \times |Q| \times P + OUT \times N)$.*

Clearly, if the size of the query is insignificant compared to the size of data, $PartialTreeStack$ is asymptotically optimal for queries without child structural relationships.

5.3 Experimental Evaluation

We ran a comprehensive set of experiments to assess the performance of *PartialTreeStack*. In this section, we report on its experimental evaluation.

5.3.1 Comparison Algorithms

As mentioned ealier, no previous algorithms exist in the inverted list model for the class of PTPQs. In order to assess the performance of *PartialTreeStack*, we designed, for comparison, two approaches that exploit existing techniques for more restricted classes of queries. The first approach, called $TPQGen$, is based on Proposition 3.3.1. Given a PTPQ Q, $TPQGen$: (1) generates a set of TPQs which is equivalent to Q, (2) uses the state-of-the-art algorithm [20] to evaluate them, and (3) unions the results to produce the answer of Q.

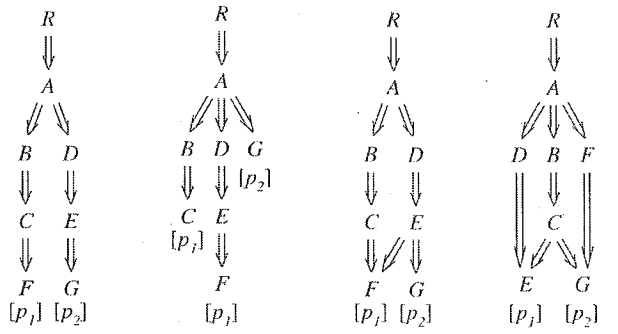
The second approach, called $PartialPathJoin$, is based on decomposing the given PTPQ into a set of queries corresponding to the partial paths of the PTPQ (*partial path queries*). For instance, for the PTPQ Q_1 of Figure 3.2(a), the partial path queries corresponding to the partial paths p_1, p_2, and p_3 of Figure 3.2(b) are produced. Given a PTPQ Q, $PartialPathJoin$: (1) uses the state-of-the-art algorithm [43] to evaluate the corresponding partial path queries, and (2) merge-joins the results on the common nodes (nodes participating in the node sharing expressions) to produce the answer of the PTPQ.

5.3.2 Experimental Results

Setup. We ran our experiments on both real and synthetic datasets. As a real dataset, we used the $Treebank$ [66] XML document. This dataset consists of around 2.5 million nodes and its maximum depth is 36. It includes deep recursive structures. The synthetic dataset is a set of random XML trees generated by IBM's XML Generator [68]. This dataset consists

of 1.5 million nodes and its maximum depth is 16. For each measurement on the synthetic dataset, 10 different XML trees were used. Each value displayed in the plots is averaged over these 10 measurements.

On each of the two datasets, we tested the 4 PTPQs shown in Figure 5.6. Our query set comprises a full spectrum of PTPQs, from a simple TPQ to complex dags. The query labels are appropriately selected for the $Treebank$ dataset, so that they can all produce results. Thus, node labels R, A, B, C, D, E, F and G correspond to $FILE$, $EMPTY$, S, VP, $SBAR$, PP, NP and PRP, respectively, on $Treebank$.

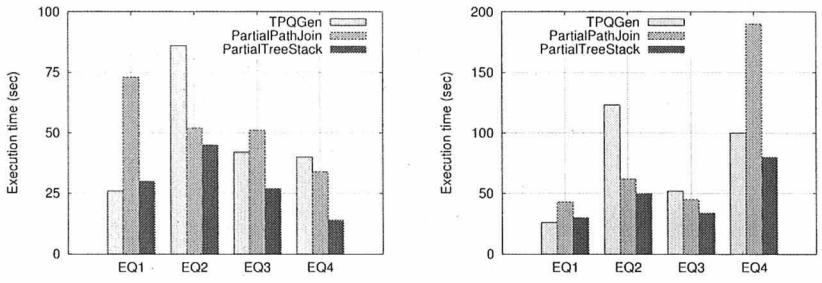


(a) EQ_1 (b) EQ_2 (c) EQ_3 (d) EQ_4

Figure 5.6 Queries used in the experiments.

We implemented all algorithms in C++, and ran our experiments on a dedicated Linux PC (Core 2 Duo $3GHz$) with $2GB$ of RAM.

Query execution time. We compare the execution time of *TPQGen*, *PartialPathJoin* and *PartialTreeStack* for evaluating the queries in Figure 5.6 over the two datasets. Figures 5.7(a) and 5.7(b) present the evaluation results. As we can see, *PartialTreeStack* has the best time performance, and in most cases it outperforms either *TPQGen* or *PartialPathJoin*



(a) Execution time (Treebank) (b) Execution time (Synthetic data)

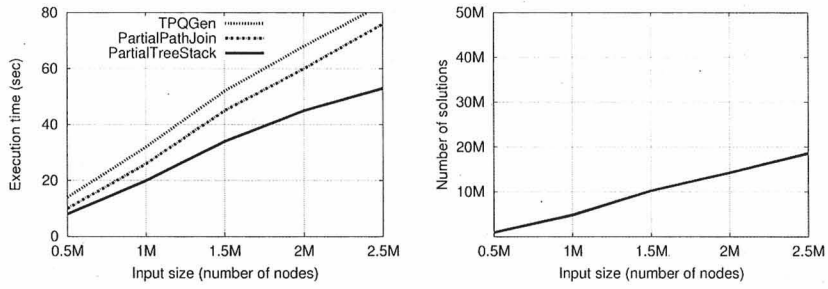
Figure 5.7 Evaluation of PTPQs on the two datasets.

by a factor almost 2. Its performance is stable, and does not degrade on more complex queries and on data with highly recursive structures.

The execution time of *TPQGen* is high for queries with a large number of TPQs, for example, EQ_2. Query EQ_2 is equivalent to 10 TPQs. *TPQGen* shows the worst performance when evaluating EQ_2 on both datasets (Figure 5.7(a) and 5.7(b)).

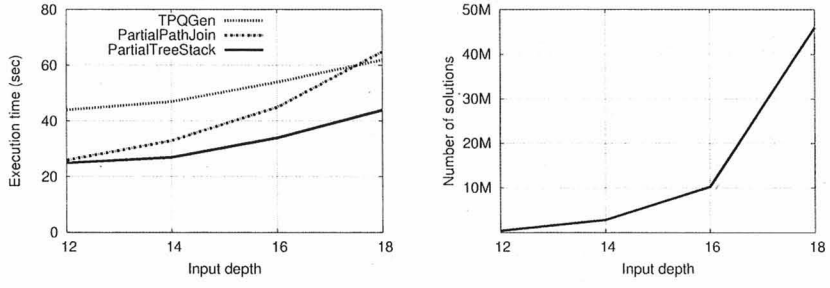
PartialPathJoin finds solutions for each partial path of the query independently. It is likely that some of the partial path solutions do not participate in the final query answer (intermediate solutions). The existence of intermediate solutions affects negatively the performance of *PartialPathJoin*. For example, when evaluating EQ_4 on the synthetic data, *PartialPathJoin* shows the worst performance (Figure 5.7(b)), due to the large amount of intermediate solutions generated.

Execution time varying the input size. We compare the execution time of the three algorithms as the size of the input dataset increases. Figure 5.8(a) reports on the execution time of the algorithms increasing the size of synthetic dataset for query EQ_3. *PartialTreeStack* consistenly has the best performance. Figure 5.8(b) presents the number of solutions of EQ_3 increasing the size of the dataset. As we can see, an increase in the input size results



(a) Execution time (b) Number of solutions

Figure 5.8 Evaluation of EQ_3 on synthetic data with increasing size.



(a) Execution time (b) Number of solutions

Figure 5.9 Evaluation of EQ_3 on synthetic data with increasing depth.

in an increase in the output size (number of solutions). When the input and the output size go up, the execution time of the algorithms increases. This confirms the complexity results that show dependency of the execution time on the input and output size. However, the increase in the execution time of *TPQGen* and *PartialPathJoin* is sharper than that of *PartialTreeStack*. The reason is that *PartialPathJoin* is also affected by the increase in the number of the intermediate solutions, while the performance of *TPQGen* is affected by the evaluation of 6 TPQs equivalent to EQ_3.

Execution time varying the input depth. We also compare the execution time of the three algorithms as the depth of the input dataset increases. Figure 5.9(a) reports on the

execution time of the algorithms increasing the input depth of synthetic dataset (its size is fixed to 1.5 million nodes) for query EQ_3. In all the cases, *PartialTreeStack* outperforms the other two algorithms. Figure 5.9(b) presents the number of solutions of EQ_3 increasing the input depth. As we can see, with the input depth increasing from 12 to 18, the output size increases from $0.4M$ to $46M$. When the output size goes up, the execution time of the algorithms increases. This again confirms our previous theoretical complexity results. We also observe that as the input depth increases, the execution time of *PartialTreeStack* increases very slowly. In contrast, the increase of the execution time of *PartialPathJoin* is sharper than that of the other two algorithms. The reason is that, for *PartialPathJoin*, an increase in the output size is accompanied by an increase in the number of intermediate solutions produced during evaluation. *TPQGen* does not increase sharper than *PartialPathJoin*. However, the execution time of *TPQGen* is strongly affected by the number of TPQs equivalent to the PTPQ, which in the worst case is exponential in the size of the PTPQ.

CHAPTER 6

EVALUATING PTPQS ON XML STREAMS

In this chapter, we present an efficient algorithm for PTPQs in the streaming model. The chapter is organized as follows. In Section 6.1, we extend the PTPQ definition in Section 3.2 with output nodes and wildcard nodes. Section 6.2 introduces data structures used for our streaming evaluation algorithms. Algorithm PSX is shown and analyzed in Section 6.3. Section 6.4 presents and discusses experimental results for PSX. Section 6.5 introduces and analyzes Algorithm $EagerPSX$. Experimental and comparison results for $EagerPSX$ are presented in Section 6.6.

6.1 Data Model and Partial Tree Pattern Query Language

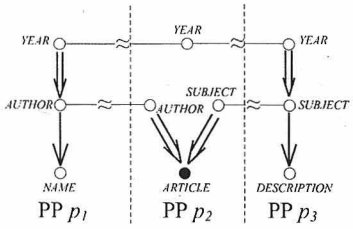
XML data is commonly modeled by a tree structure. Tree nodes are labeled by labels and represent elements, attributes, or values. Tree edges represent element-subelement, element-attribute, and element-value relationships. Let $\mathcal{L}$ be the set of node labels. Without loss of generality, we assume that only the root node of every XML tree is labeled by $r \in \mathcal{L}$. We denote XML tree labels by lower case letters. To distinguish between nodes with the same label, nodes in the XML tree may have a numeric identifier shown as a subscript of the node label.

6.1.1 Query Language

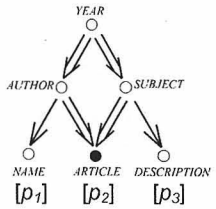
Syntax. A partial tree-pattern query (PTPQ) specifies a pattern which partially determines a tree. PTPQs comprise nodes and child and descendant relationships among them. Their

P = [p_1, p_2, p_3]
S = [{YEAR[p_1]//AUTHOR[p_1],
AUTHOR[p_1]/NAME[p_1]},
{AUTHOR[p_2]//ARTICLE[p_2],
SUBJECT[p_2]//ARTICLE[p_2]},
{ YEAR[p_3]//SUBJECT[p3],
SUBJECT[p3]/DESCRIPTION[p3]}]
N = {YEAR[p_1] ≈ YEAR[p_2],
YEAR[p_2] ≈ YEAR[p_3],
AUTHOR[p_1] ≈ AUTHOR[p_2],
SUBJECT[p_2] ≈ SUBJECT[p_3] }
o = ARTICLE

(a) PTPQ Q_1



(b) Visual representation of Q_1



(c) Q_1's query graph

Figure 6.1 A PTPQ, its visual representation and its query graph

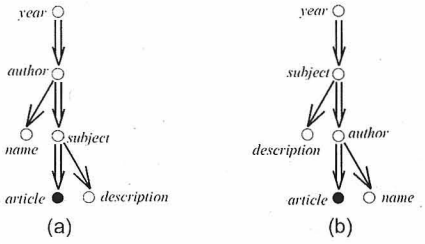


Figure 6.2 Two TPQs

nodes are grouped into disjoint sets called *partial paths*. PTPQs are embedded to XML trees. The nodes of a partial path are embedded to nodes on the same XML tree path. However, unlike paths in TPQs the child and descendant relationships in partial paths do not necessarily form a total order. This is the reason for qualifying these paths as partial. PTPQs also comprise node sharing expressions. A node sharing expression indicates that two nodes from different partial paths are to be embedded to the same XML tree node. That is, the image of these two nodes is the same (*shared*) node in the XML tree. The formal definition of a PTPQ follows.

Definition 6.1.1 (Partial Tree-Pattern Query) *We assume an infinite set of labeled nodes. The nodes in this set can be labeled by a wildcard (*) or by a label in $\mathcal{L}$. Let X and Y denote distinct nodes. A* partial tree-pattern query *is a quadruple $(P, \mathbf{S}, N, o)$ where:*

P *is a list of* n *names* $p_1, \ldots, p_n$ *called* partial path names.

$\mathbf{S}$ *is a list of* n *sets* $S_1, \ldots, S_n$ *where set* S_i *is called* partial path (PP) *and is named by* p_i. *Since their names are distinct, we identify PPs with their names. Each PP* p_i *is a finite set of expressions of the form* X/Y (child relationship) *or* $X//Y$ (descendant relationship). *No node occurs in two different PPs. We write* $X[p_i]/Y[p_i]$ *(resp.* $X[p_i]//Y[p_i]$*) to indicate that* $X[p_i]/Y[p_i]$ *(resp.* $X[p_i]//Y[p_i]$*) is a relationship in PP* p_i. *Child and descendant relationships are collectively called* structural *relationships.*

N *is a set of expressions* $X[p_i] \approx Y[p_j]$ *where* p_i *and* p_j *are distinct PPs and* X *and* Y *are nodes from* p_i *and* p_j *respectively such that: (a) at least one of them is labeled by a wildcard, or (b) both of them are labeled by the same label in* $\mathcal{L}$.

o *is a distinguished node in one of the PPs called* output node.

Figure 6.1(a) shows a PTPQ Q_1. Value predicates are omitted for simplicity. Figure 6.1(b) shows the visual representation of Q_1. Note that the labels of the query nodes are denoted by capital letters to distinguish them from the labels of the XML tree nodes. In this sense, label l in an XML tree and label L in a query represent the same label. Unless otherwise indicated, in the following, "query" refers to a PTPQ.

Semantics. The answer of a query on an XML tree is a set of solutions, where each solution is the image of the output node in a match of the query on the XML tree. A formal definition follows.

We say that an XML tree node labeled by *a matches* a query X if X is labeled by a wildcard ('*') or by A.

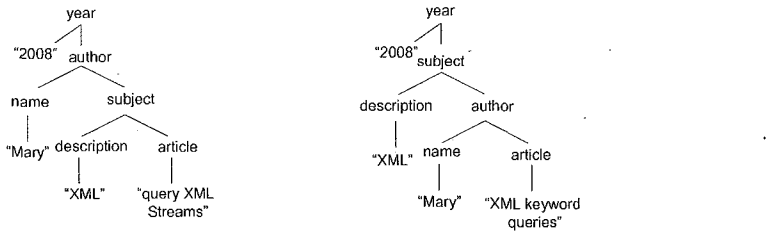


Figure 6.3 Fragments of two XML bibliography documents

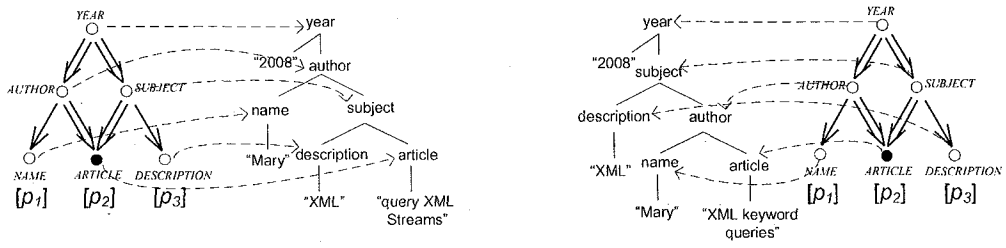


Figure 6.4 Embeddings of PTPQ Q_1 of Figure 6.1 on the two XML trees of Figure 6.3

Definition 6.1.2 (Query Embedding) *An* embedding *of a query Q into an XML tree T is a mapping M from the nodes of Q to nodes of T such that: (a) a node in Q is mapped by M to a matching node of T; (b) the nodes of Q in the same PP are mapped by M to nodes that lie on the same path in T; (c) $\forall\ X[p_i]/Y[p_i]$ (resp. $X[p_i]//Y[p_i]$) in Q, $M(Y[p_i])$ is a child (resp. descendant) of $M(X[p_i])$ in T; (d) $\forall\ X[p_i] \approx Y[p_j]$ in Q, $M(X[p_i])$ and $M(Y[p_j])$ coincide in T.*

The image of the output node of Q under an embeddings of Q to T is a *solution* of Q on T. The *answer* of Q on T is the set of all the solutions of Q on T.

We represent queries as node and edge labeled directed graphs: a query Q is represented by a graph Q_G. Every node X in Q corresponds to a node X_G in Q_G. Node X_G is labeled by the label of X, if this label belongs to $\mathcal{L}$. Every node X_G in Q_G corresponds to one or more nodes in Q which have the same label or are labeled by '*'. Node X_G is labeled by '*' if all the nodes in Q it corresponds to are labeled by '*'. Otherwise, it is

labeled by the label in $\mathcal{L}$ of one of the nodes in Q it corresponds to. In addition, node X_G in G is annotated by the set of PPs of the nodes in Q it corresponds to. Two nodes in Q correspond to distinct nodes in Q_G, unless they participate in the same node sharing expression in Q. For every structural relationship X/Y or $X//Y$ in Q there is an edge e in Q_G from X_G to Y_G. Edge e is labeled by '/' if there is a child relationship from a node X' to a node Y' in Q and X' and Y' correspond to X_G and Y_G in Q_G, respectively. Otherwise, e is labeled by '//'.

Figure 6.1(c) shows the query graph of query Q_1 of Figure 6.1(a). In the figures, edges labeled by '/' ('//') are shown as single (double) line edges. For simplicity of presentation, the annotations of some nodes might be omitted and it is assumed that a node inherits all the annotating PPs of its descendant nodes. For example, in the graph of Figure 6.1(c), node $YEAR$ is assumed to be annotated by the PPs p_1, p_2, and p_3 inherited from its descendant nodes $NAME$, $ARTICLE$, and $DESCRIPTION$. Figure 6.4 shows the two embeddings of the query graph of Q_1 on the two XML trees of Figure 6.3. The answer of query Q_1 on the two XML trees consists of the two $article$ nodes. Note that as this figure illustrates, the same query can be used to retrieve results from two XML trees that have different structures.

Clearly, a query that has a cycle is unsatisfiable (that is, its answer is empty on any XML tree). Therefore, in the following, we assume a query is a directed acyclic graph (dag) and we identify a query with its dag representation.

6.1.2 Generality of Partial Tree Pattern Query Language

Clearly, the class of PTPQs cannot be expressed by TPQs. For instance, PTPQs can constraint a number of nodes in a query pattern to belong to the same path (*same-path*

constraint) even if there is no precedence relationship between these nodes in the PTPQ. Such a query cannot be expressed by a TPQ. TPQs correspond to the fragment $XP^{\{[],/,//,*\}}$ of XPath that involves predicates([]), child (/) and descendant (//) axes, and wildcards (*). In fact, it is not difficult to see that PTPQs cannot be expressed either by the larger fragment $XP^{\{[],/,//,\backslash,\backslash\backslash,*\}}$ of XPath that involves, in addition, the reverse axes parent (\) and ancestor (\\). On the other side, PTPQs represent a very broad fragment $XP^{\{[],/,//,\backslash,\backslash\backslash,*,\approx\}}$ of XPath that corresponds to $XP^{\{[],/,//,\backslash,\backslash\backslash,*\}}$ augmented with the $is-same-node$ function ($\approx$) of XPath2 [1]. The $is-same-node$ function is a node identity equality operator. The conversion of an expression in $XP^{\{[],/,//,\backslash,\backslash\backslash,*,\approx\}}$ to an equivalent PTPQ is straightforward. There is no previous streaming evaluation algorithm that directly supports such a broad fragment of XPath.

Note that as the next proposition shows, a PTPQ is equivalent to a *set* of TPQs. These TPQs can be obtained by considering all the allowable orderings of the nodes in the partial paths of the PTPQ.

Proposition 6.1.1 *Given a PTPQ Q there is a set of TPQs Q_1, ..., Q_n in $XP^{\{[],/,//,*\}}$ such that for every XML tree T, the answer of Q on T is the union of the answers of the Q_is on T.*

Proof sketch. The proof is easy if we observe that the TPQs $Q_1, \ldots, Q_n$ are those that can be produced by adding descendant relationships to Q in all possible ways.

As an example, Figure 6.2 shows the two TPQs for query Q_1 of Figure 6.1, which together are equivalent to Q_1. The TPQ of Figure 6.2(a) is obtained by adding $AUTHOR//SUBJECT$ to Q_1, while the TPQ of Figure 6.2(b) is obtained by adding $SUBJECT//AUTHOR$ to Q_1. Based on the previous proposition, one could consider evaluating PTPQs using existing streaming algorithms for TPQs. Unfortunately, the number of TPQs that

need to be evaluated can be exponential on the number of nodes of the PTPQ. Therefore, previous streaming algorithms cannot be used for efficiently evaluating PTPQs.

6.2 Data Structures for PTPQ Streaming Evaluation

Open and close events. In a streaming evaluation, the XML document tree T flows in as a stream of *open* and *close* events. The appearance of events corresponds to the preorder traversal of the XML document tree. For each element node in the tree, an *open* event is produced when the open tag of the node is encountered and the node is called *open* from then on until it closes. After the subtree rooted at that node is processed, a *close* event is produced when the close element tag of that node is encountered. At this time the node *closes*. Each event carries the name and level of the corresponding element node in the tree. For example, suppose the incoming XML is a path with three elements: $/a_1/b_1/b_2$. The sequence of events for this XML path is: $\langle a_1\rangle$, $\langle b_1\rangle$, $\langle b_2\rangle$, $\langle /b_2\rangle$, $\langle /b_1\rangle$, $\langle /a_1\rangle$. The first three denote open events and the last three denote close events. For simplicity, the level information is omitted here. An XML node is *current* if it is *open* but none of its descendant nodes is *open*. The path in T from the root to the *current* node is called *current* path. Clearly, a *current* path consists of all the *open* nodes in T at that time.

Query functions. Let X be a node in a PTPQ Q and R be the root of Q. When X is the output node of Q, the ancestor nodes of X are called *backbone* nodes of Q, and the rest of the nodes of Q are called *predicate* nodes. Note that because of the generality of the class of queries considered, the backbone nodes of Q do not necessarily lie on the same path of the query dag. The backbone nodes of Q form a dag whose single root is R and whose

single sink node is the output node. Given a partial path p_i in Q, we call a node in Q a *sink node of* p_i, if p_i annotates X but does not annotate any descendant nodes of X in Q.

We make use of the following functions in the evaluation algorithm. Function *PPsSink*(X) returns the set of partial paths where X is a sink node. Function *parents*(X) returns the set of parent nodes of X in Q. Function *PChildren*(X) returns the set of predicate child nodes of X in Q, and function *BSiblings*(X) returns the set of backbone sibling nodes of X in Q. By removing the *descendant* edges from the dag of the backbone nodes of Q, we can logically partition it into a set of paths, each path involving only *child* edges. A path can be trivially be a single node. Let X be a node in a path p of the partition. Function *host*(X) returns the leaf node of p, if p does not contain the output node of Q, and *null* otherwise.

Example 6.2.1 *Consider the PTPQ Q_1 shown in Figure 6.1(c). Its backbone nodes are: YEAR, AUTHOR, SUBJECT, ARTICLE. The predicate nodes are: NAME, DESCRIPTION. Nodes NAME, ARTICLE, and DESCRIPTION are the only sink nodes of partial paths p_1, p_2, and p_3, respectively. Some instances of functions* PPsSink *and* host*:*

PPsSink(NAME) = $\{p_1\}$, *PPsSink(ARTICLE)* = $\{p_2\}$,

PPsSink(AUTHOR) = $\emptyset$, *host(YEAR)* = *YEAR*,

host(AUTHOR) = *AUTHOR*, *host(ARTICLE)* = *null*

Query matches. We use the notion of *candidate match* of a query node which is based on the notion of *ancestor match* of a query node. These notions are useful for describing and understanding the algorithm and for showing its correctness. We define them here and provide a proposition that relates solutions to candidates matches of nodes.

Definition 6.2.1 (Ancestor Match) *Let X be a node in a PTPQ Q and x be a node in an XML tree T that matches X. Node x is called an* ancestor match *of X, if either X is the root node of Q, or for every parent Y of X in Q, Y has an ancestor match y in T such that: (a) If $Y/X \in Q$, y is the parent of x in T, and (b) If $Y//X \in Q$, y is an ancestor of x in T.*

We say that a node x of an XML tree T *sustains* the partial path p_i of a PTPQ Q, if there exists an embedding of the nodes of Q to T that maps all the nodes in Q annotated by p_i to the path from the root of T to x. The concept of sustainability relates to the same-path constraint since if node x sustains a partial path p_i there are nodes in the path of x that satisfy the same-path constraint for p_i.

Definition 6.2.2 (Candidate Match) *Let x be a node in an XML tree T, and X be a node in a PTPQ Q. Node x is a* candidate match *of X, iff the following conditions are satisfied: (a) x is an ancestor match of X, (b) $\forall p_i \in PPsSink(X)$, x sustains p_i, and (c) $\forall Y \in PChildren(X)$, x has a descendant in T which is a candidate match of Y.*

A candidate match x of X is a *candidate output* if X is the output node of Q. Let Q_B denote the dag of the backbone nodes of Q, and x be a candidate output of Q in T. Then, there is an embedding of Q_B to the path from the root of T to x. The path formed by the images of the nodes of Q_B under such an embedding is called *output path* for x. The following proposition provides conditions for a candidate output to be a solution of Q.

Proposition 6.2.1 *Let x be a candidate output of Q in T. Node x is a solution, iff there is an output path for x in T such that every node on the path is a candidate match of the corresponding backbone node(s).*

The proof of Proposition 6.2.1 follows from Definitions 6.1.2 and 6.2.2.

Example 6.2.2 *Consider the PTPQ Q_1 of Figure 6.1(c) and the left XML tree of Figure 6.3. It is easy to see that the XML tree nodes $year$, $author$, and $subject$ are ancestor matches of query nodes $YEAR$, $AUTHOR$, and $SUBJECT$, respectively. Tree node $article$ is an ancestor match of query node $ARTICLE$, since $ARTICLE$'s parents $AUTHOR$ and $SUBJECT$ have ancestor matches $author$ and $subject$, respectively, both of which are ancestors of $article$ in the XML tree. Further, node $article$ sustains partial path p_2, since there is an embedding of the nodes of Q_1 to the XML tree that maps all the nodes annotated by p_2, that is, $YEAR$, $AUTHOR$, $SUBJECT$, and $ARTICLE$, to the XML path p : $/year/author/$ $subject/article$ (see the embedding 1 of Figure 6.4(a)). Therefore, node $article$ is a candidate match of query node $ARTICLE$ (the condition (c) of Definition 6.2.2 is trivially satisfied). Similarly, the XML tree nodes $year$, $author$, and $subject$ are candidate matches of their corresponding query nodes. Also, node $article$ is a candidate output, and the XML path p is the output path for $article$. Finally, node $article$ is a solution of Q_1 since every node on path p is a candidate match of the corresponding backbone node (Proposition 6.2.1).*

Stacks. With every query node X in Q, we associate a stack S_X. Each entry in stack S_X corresponds to an open node x in T and is a 3-tuple *(XMLNode, SPFlags, PCFlags)*. For an entry e of S_X, field *e.XMLNode* is the tree node x. Field *e.SPFlags* is a boolean array indexed by the partial paths in *PPsSink*(X). Given $p_i \in PPsSink(X)$, $e.SPFlags[p_i]$ indicates whether x sustains p_i. Field *e.PCFlags* is a boolean array indexed by the nodes in *PChildren*(X). Given $Y \in PChildren(X)$, $e.PCFlags[Y]$ indicates whether x has a descendant in T that is a candidate match of Y.

If X is a backbone node, we associate with e an additional field *candList* which stores a list of candidate outputs (these are closed nodes) that are descendants of x in T. Let

$MaxBChild$ denote the maximum number of backbone child nodes of a node in Q. When $MaxBChild > 1$ (in which case the backbone nodes of Q form a dag), each candidate output c in $e.candList$ is a 2-tuple *(XMLNode, BFlags)*. Field $c.XMLNode$ is a candidate match of the output node. Field $c.BFlags$ is a boolean array indexed by the backbone nodes of Q. Given a backbone node Y, $c.BFlags[Y]$ indicates whether $c.XMLNode$ has an ancestor in T that is a candidate match of Y. When $MaxBChild = 1$, each candidate output c in $e.candList$ is a 1-tuple *(XMLNode)*.

Stack operations. We use the following stack operations: boolean function *empty*(S_X) which returns true iff stack S_X is empty, $push(S_X, e)$ which pushes e on stack S_X, *pop*(S_X) which pops the top entry from S_X and returns it, and *top*(S_X) which returns the top entry of stack S_X. In what follows, we might not distinguish between an entry in a stack and its corresponding node in T.

6.3 Evaluation Algorithm

The flexibility of the PTPQ language in specifying queries and its increased expressive power compared to TPQs makes the design of an evaluation algorithm challenging. Two outstanding reasons of additional difficulty are: (1) a query is a dag (which in the general case is not merely a tree) augmented with constraints, and (2) the same-path constraints should be enforced for all the nodes in a partial path in addition to enforcing structural relationships. In this section, we present our evaluation algorithm which efficiently resolves these issues. The presentation of the algorithm is followed by an analysis of its correctness and complexity.

6.3.1 Overview

Listing 9 Algorithm PSX

```
1 while (event stream generates more events e) do
2   let x denote the tree node corresponding to e
3   compute the list nodesList of query nodes in Q that match x sorted so that they form a topological ordering of Q
4   if (e is an open event) then
5     for (every X ∈ nodesList in reverse topological order) do
6       startEval(X, x)
7   else if (e is a close event) then
8     for (every X ∈ nodesList in forward topological order) do
9       endEval(X, x)
```

Let Q be the input query to be evaluated on a stream of events for an XML tree T. We assume that a topological order (i.e., a linear order of the query nodes which respects the partial order induced by the structural relationships of the query) for the nodes of Q is fixed with the root node R of Q being the first node. Our algorithm is called **P**artial TPQ **S**treaming evaluation on **X**ML (PSX) and is shown in Listing 9. Algorithm *PSX* is event-driven: as events arrive, event handlers (which are the procedures *startEval* or *endEval*), are called on a sequence of query nodes that match the current node.

More specifically, when the algorithm receives an open event for a tree node x, it calls procedure *startEval* on all the query nodes in Q that match x. For each such node X, *startEval* examines whether x can be pushed on stack S_X and whether the current node sustains the partial paths that annotate X. In order to prevent x from 'seeing' a copy of itself on parent stacks of X, the query nodes that match x should be considered in their reverse topological order. When the algorithm receives the close event of x, it calls procedure *endEval* on the same query nodes but now it considers them in their forward topological order. For instance, consider evaluating the query $//A//*$ on the XML path

$/a_1/a_2/a_3$. When $\langle a_2 \rangle$ comes, node $*$ is considered before A. When $\langle /a_2 \rangle$ comes, node A is considered before $*$. For each query node X in the list, *endEval* pops the entry of x from S_X and checks if x is a candidate match of X. If this is the case and X is a backbone node, each candidate output stored in the entry for x is propagated to an ancestor of x in a stack, or is returned to the user if X is the root of Q.

Algorithm *PSX* has three main features: (1) it retains in memory only elements that are relevant for query evaluation, (2) it avoids processing redundant matches, and (3) it keeps only one copy of each candidate output in the stacks during execution. Another important feature which is especially useful in streaming environments is that the solutions are incrementally generated rather than being accumulated and delivered after the entire stream has been processed. We elaborate these features below.

6.3.2 Open Event Handler

Procedure $startEval$, shown in Listing 10, is invoked every time an open event for a tree node x arrives. At this time, all the ancestor nodes of x have arrived and x is the current node.

Filtering irrelevant data. Let X be a query node that matches x. Procedure *startEval* checks if x qualifies for being pushed on stack S_X (lines 1-3). Node x can be pushed on S_X only if x is an ancestor match of X. This check would require examining whether every ancestor of X in Q has an ancestor match on the path from the root of T to x. Fortunately, the stack-based organization allows this checking to be done efficiently, with its cost being bounded by the in-degree of the query dag Q. The reason is that only the descendant

or child relationships between x and the top entries of the parent stacks of X need to be checked.

Avoiding redundant matches. Because the answer of a query comprises only the embeddings of the output node of the query, we might not need to identify all the matches of the query pattern when computing the answer of the query. In this sense, we take advantage of the existential semantics of the query during evaluation: whenever a matching of a predicate node in the query is found, other matches of the same node that do not contribute to a possible new matching for the output node can be ignored. For instance, consider evaluating the query Q_2 of Figure 6.5(b) on the XML tree of Figure 6.5(a). The nodes a_1, b_1, e_1 and f_1 which are matches for the predicate nodes A, B, E and F, respectively, contribute to the match d_1 of the output node D. The nodes $a_2, \ldots, a_n$, $b_2, \ldots, b_n$, $e_2, \ldots, e_n$, $f_2, \ldots, f_n$ which are also matches of the predicate nodes can be ignored, since they all contribute to the same match d_1 of the output node. Note that these nodes correspond to $O(n^4)$ embeddings of the query with the same match for the output node. Avoiding these redundant matches of the predicate nodes saves substantial time and space.

Our algorithm exploits this observation using the concept of *redundant* match of a predicate node. Let X be a predicate node in Q. An ancestor match x of X is *redundant* for the evaluation if a node x' that precedes x in T is a candidate match of X and all the ancestor matches of X's parents in Q that are ancestors of x are also ancestors of x'. During the evaluation of the algorithm, an ancestor match x of a predicate node X is identified as redundant if the boolean field $PCFlags[X]$ associated with the top entry of each parent stack of X has been set to $true$. In the previous example, e_2 is a redundant match of node

Listing 10 Procedure startEval(X, x)

```
1  if (X ≠ R ∧ ∃P ∈ parents(X): empty(S_P)) then
2      return
3  if ((X = R) ∨ (∀P ∈ parents(X): top(S_P).XMLNode and x satisfy the structural relationship between P and X in Q)) then
4      if (isMatchRedundant(X) = true) then
5          return
6      push(S_X, newStackEntry(X, x))
7      for (every p_i ∈ PPsSink(X)) do
8          if (for every sink node Y of p_i: ¬empty(S_Y)) then
9              for (every sink node Y of p_i) do
10                 top(S_Y).SPFlags[p_i] ← true

Function isMatchRedundant(X)
1  if ((X is a predicate node ) ∧ (∀P ∈ parents(X): top(S_P).PCFlags[X] = true)) then
2      return true
```

E, since it is an ancestor match of E and when $\langle e_2 \rangle$ is read, $c_1.PCFlags[E]$ has been set to $true$ by e_1. Redundant matches (which can be nested at arbitrary levels) are not stored and processed by our algorithm (line 4 in *startEval*). Note that previous streaming algorithms in [32, 35, 37, 29] do not take advantage of this observation and process all the nodes in the XML tree regardless of their redundancy. For instance, in evaluating the sub-query $R//C[E//F]//D$ of Q_2 over the XML tree of Figure 6.20(a) using Algorithm X_{aos} [32] (X_{aos} cannot support the PTPQ Q_2), a number of $O(n^2)$ matches of the pattern $E//F$ will be unnecessarily accumulated in memory. The streaming algorithms in [30] take advantage of the existential semantics of the query during evaluation, but they are restricted to TPQs.

Once *startEval* determines that x is not a redundant match, it creates a new stack entry for x and pushes it on S_X (line 6). At this time, the stacks contain the ancestor matches of query nodes that lie on the current path. If x is a candidate output, the matches

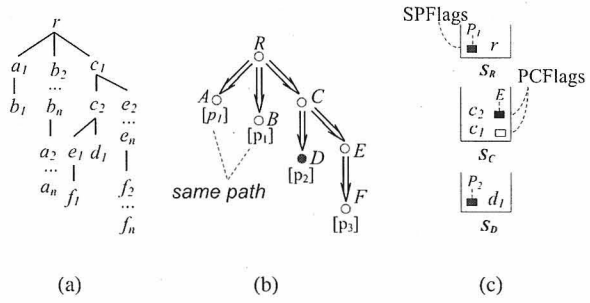


Figure 6.5 (a) XML Tree, (b) Query Q_2, (c) Snapshots of stacks

of the backbone nodes encode the set of output paths for x in T that the algorithm could follow upwards to determine whether x is a solution (see Proposition 6.2.1). For instance, consider again evaluating the query Q_2 of Figure 6.5(b) over the XML tree of Figure 6.5(a). Figure 6.5(c) shows the snapshots of stacks after $\langle d_1 \rangle$ is read and an entry for d_1 is pushed on stack S_D. Black boxes in the boolean arrays $PCFlags$ and $SPFlags$ associated with stack entries denote fields which are true. Node d_1 is a candidate output. The algorithm could follow upwards the path $/r/c_2/d_1$ or the path $/r/c_1/d_1$ to determine whether d_1 is a solution.

Checking the same path constraint. Procedure *startEval* proceeds to check whether the current node x sustains the partial paths annotating X in Q (lines 7-8) and updates the boolean array $SPFlags$ accordingly (lines 9-10). In order to do this, it suffices to check if the stacks contain an entry for every *sink* node of these partial paths. For instance, consider again the example shown in Figure 6.5. After the entry for b_1 is pushed on stack S_B, the stacks S_A and S_B respectively contain an ancestor match of the sink query nodes A and B of the partial path p_1. Therefore, the current node b_1 sustains p_1 (see Section **??**). As a result, $r.SPFlags[p_1]$ is set to $true$. Note that the checking time is bounded by the maximum number of sink nodes in a partial path of Q. It is of course constant when Q is a TPQ.

6.3.3 Close Event Handler

Procedure *endEval*, shown in Listing 11, is invoked every time a close event for a tree node x comes. At that time all the descendant nodes of x in T have arrived and x is the current node. Let X be a query node that matches x and s be the top entry of stack S_X. If the node $s.XMLNode$ is the same as x (line 4), entry s is popped out from S_X (line 5), and procedure *mergeFlags* is called to copy the truth values of the boolean arrays $s.SPFlags$ and $s.PCFlags$ to the new top entry of S_X (lines 6-7). For instance, consider the example shown in Figure 6.5. After entry c_2 is popped out from stack S_C, $PCFlags[E]$ is set to $true$ for the new top entry c_1. Function *isCandMatch* is then invoked to determine if s is a candidate match of X (line 8). In order to do so, function *isCandMatch* essentially applies Definition 6.2.2.

If X is a backbone node, s possibly stores a list of candidate outputs that are descendants of x. Recall that the backbone stacks encode all the output paths which the algorithm could use to determine if candidate outputs are solutions. In each of these paths, the nodes which are ancestor matches of the backbone nodes might become candidate matches. Whether the candidate outputs will eventually become solutions depends on whether an output path can be found consisting of nodes that are candidate matches of the backbone nodes. If no such path exists, the candidate outputs will be discarded. In any case, the matching information about x and the candidate outputs stored in s is propagated up along an output path encoded in the stacks. We detail this process below.

Handling a candidate match. If x is a candidate match of the query node X, procedure *endEval* considers four cases depending on the type of X. The last three cases are handled through a call to procedure *upwardPropagate*.

Listing 11 Procedure endEval(X, x)

```
1  if (empty(S_X)) then
2      return
3  s ← top(S_X)
4  if (s.XMLNode = x) then
5      pop(S_X)
6      if (¬empty(S_X)) then
7          mergeFlags(X, s)
8      if (isCandMatch(X, s) then
9          if (X = R) then
10             output(s.candList)
11         else
12             upwardPropagate(X, s)
13     else
14         downwardPropagate(X, s)
```

Procedure mergeFlags(X, *popped*)

```
1  for (every p_i ∈ PPsSink(X)) do
2      top(S_X).SPFlags[p_i] ← true, if popped.SPFlags[p_i] =
       true
3  for (every Y ∈ PChildren(X)) do
4      if (X//Y ∈ Q) then
5          top(S_X).PCFlags[Y] ← true, if popped.PCFlags[Y]
           = true
```

Function isCandMatch(X, s)

```
1  if (∀ p_i ∈ PPsSink(X): s.SPFlags[p_i] = true) ∧ (∀ Y ∈
   PChildren(X): s.PCFlags[Y] = true) then
2      return true
```

Procedure upwardPropagate(X, s)

```
1  if (X is a predicate) then
2      for (every P ∈ parents(X)) do
3          top(S_P).PCFlags[X] ← true
4  else {/*X is the output node or a backbone node */}
5      lowAncPar ← getLowAncEntry(parents(X))
6      if (X is the output node) then
7          append(lowAncPar.candList, newCandidate(x))
8      else if (X is a backbone node) then
9          if (MaxBChild = 1) then
10             append(lowAncPar.candList, s.candList)
11         else
12             for (every c ∈ s.candList) do
13                 c.flags[X] ← true
14                 nodes ← {Y|Y ∈ BSiblings(X) ∧
                   c.BFlags[Y] ≠ true}
15                 if (nodes = ∅) then
16                     append(lowAncPar.candList, c)
17                 else
18                     lowAncSib ← getLowAncEntry(nodes)
19                     if (lowAncSib ≠ null) then
20                         append(lowAncSib.candList, c)
```

Procedure downwardPropagate(X, s)

```
1  Y ← host(X)
2  lowAnc ← getLowAncEntry(BSiblings(Y) ∪ {Y})
3  if (lowAnc ≠ null) then
4      append(lowAnc.candList, s.candList)
```

Function getLowAncEntry(*nodes*)

```
1  Y_max ← maxarg_Y {top(S_Y).XMLNode.level}, ∀Y ∈
   nodes ∧ ¬empty(S_Y)
2  return top(S_Y_max)
```

(1) if X is the root node R of Q, the candidate outputs stored in entry s are simply returned to the user (line 10 of procedure *endEval*).

(2) if X is a predicate node of Q, for each parent P of X, $top(S_P).PCFlag[X]$ is set to $true$. This indicates that a candidate match of X has been found (lines 1-3 of procedure *upwardPropagate*).

(3) if X is the output node of Q, by definition, x is a candidate output. At this time, it can not be determined, based on the part of T seen so far, whether x qualifies to be a solution. Before such a decision can be made, x must be stored. The entries in the parent stack(s) of X can be used to store x. Note that it is possible that X has more than one parent node in Q. The stack for each of the parent nodes contains entries that are ancestors of x in T. Each of those entries lies on an output path for x. Clearly, attaching a copy of x even only to the top entry of each parent stack of X would lead to duplicate outputs, when there are multiple output paths for x consisting of backbone node candidate matches. To avoid duplicate outputs, procedure *upwardPropagate* propagates x only to the top stack entry which is the lowest ancestor of x among the top entries of the parent stacks of X (lines 5-7).

(4) if X is a backbone node of Q, as in case (3), the problem we face is where to propagate the candidate outputs in $s.candList$ after entry s is popped out from its stack. Recall that $MaxBChild$ is the maximum number of backbone child nodes of any node in Q. If $MaxBChild = 1$, the list of candidate outputs is propagated to the top entry of the parent stack of X (lines 9-10). This cannot be done when $MaxBChild > 1$, since it could lead to false outputs. The reason is that X may have a sibling backbone node Y for which the existence of a candidate match that is an ancestor of the candidate outputs in $s.candList$ may have not yet been determined. For instance, consider evaluating the

query Q_3 of Figure 6.6(b) (borrowed from [32]) on the XML tree of Figure 6.6(a). After entry z_4 is popped out from stack S_Z, and is identified as a candidate match of Z, $z_4.candList$ $(= \{w_6\})$ should not be propagated to the top entry r of the parent stack S_R, since it is not known at this time if w_6 has an ancestor which is a candidate match for Y.

We provide a solution to the problem by exploiting the data structure designed for candidate outputs. Recall that each candidate output c in $s.candList$ is a 2-tuple *(XMLNode, BFlags)* and that for each backbone node $Y \in Q$, $c.BFlags[Y]$ is used to indicate whether a candidate match of X that is an ancestor of c in T has been found (see Section 6.2). Using this data structure, the propagation of a candidate output c proceeds in two steps. In the first step, $c.BFlag[X]$ is set to $true$ (line 10). In the second step, c is propagated to the lowest ancestor among the top entries of the parent stacks of X or among the top entries of selected sibling stacks of X (lines 11-27). If none of these choices is applicable, c is discarded. Notice that an iteration over each candidate output in $s.candList$ is needed here (line 9), since the values of $BFlags$ can be different for each candidate output. For instance, consider again the example of Figure 6.6. Figure 6.6(c) shows a snapshot of stack S_Y during execution, where each of the candidate outputs in $y_1.candList$ has a different $BFlags$ value. For simplicity, for each candidate output, only the fields of Y and Z of its $BFlags$ are shown in the figure.

Note that an invariant of the upward propagation is that whenever a candidate output c is propagated to a stack of a backbone node X, for any backbone node Y which is a descendant of X in Q, $c.BFlags[Y]$ has been set to $true$.

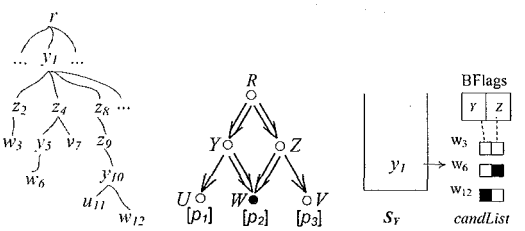


Figure 6.6 (a) XML Tree, (b) Query Q_3, (c) Candidate outputs in $y_1.candList$ with different $BFlags$ values

Handling a non-candidate match. If x is not a candidate match of X, the candidate outputs in $s.candList$ should be propagated along an output path that does not comprise x. Those candidate outputs could be propagated to an ancestor node of x either in the same stack S_X, or in the stack of a sibling backbone node of X, or in the stack of a descendant backbone node of X. This operation is handled by calling procedure *downwardPropagate* (line 14 in procedure *endEval*).

Let Y be the node $host(\mathrm{X})$ (line 1 in procedure *downwardPropagate*). By definition, node Y is the closest descendant-or-self backbone node of X such that $\forall Z \in$ BChildren(Y), $Y//Z \in Q$. The candidate outputs in $s.candList$ are propagated to the entry, among the top stack entries of S_Y and the stacks of the backbone siblings of Y, which is lowest ancestor of x (lines 2-4). If no such entry exists, the candidate outputs are discarded. For instance, consider again the example of Figure 6.6. When $\langle /y_5 \rangle$ is read, y_5 is identified as a non-candidate match of Y. Therefore, candidate output w_6 is downward propagated to z_4 in stack S_Z since $host(Y) = Y$ and Z is the sibling backbone of Y. Note that if instead of propagating w_6 to the top stack entry (z_4) of the sibling backbone node Z of Y, we propagate it to the new top stack entry y_1 of S_Y, we will lose the information that w_6

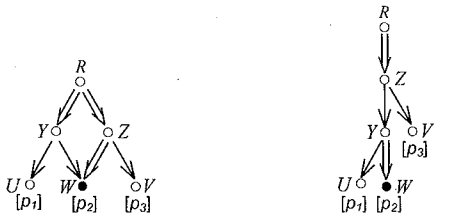


Figure 6.7 (a) Query Q_3', (b) Query Q_3''

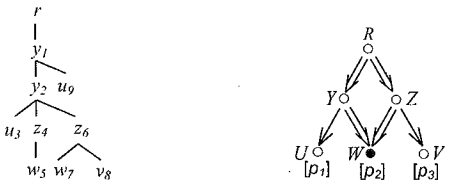
has an ancestor z_4 which is a candidate match of Z. As another example, suppose that we replace the edge '//' between Y and W in Q_3 by '/' to form the query Q_3' of Figure 6.7(a). We evaluate Q_3' on the XML tree of Figure 6.6(a). When $\langle /y_5 \rangle$ is read, w_6 is discarded since function $host$ returns $null$ on Y. As a third example, consider evaluating the query Q_3'' of Figure 6.7(b) on the XML tree of Figure 6.6(a). When $\langle /y_5 \rangle$ is read, w_6 is downward propagated to y_1 in stack S_Y since $host(Y) = Y$, and Y has no sibling backbone nodes.

The downward propagation does not update the array $BFlags$ of the candidate outputs in $s.candList$. It handles the candidate outputs in batch rather than individually, no matter whether $MaxBChild > 1$ or not.

Note that in both the upward and downward propagations, the candidate outputs stored in the popped entry of the current node are propagated to at most one stack entry. This way, for each candidate output, there is only one copy stored in the stacks during execution. This technique eliminates the need to explicitly perform duplicate solution removal which is required in the streaming algorithms for TPQs presented in [34, 32, 29].

6.3.4 An Example and Comparison with Previous Approaches

As an example, we evaluate query Q_3 of Figure 6.8(b) on the XML tree of Figure 6.8(a) using Algorithm PSX. The answer returned is $\{w_7\}$. In Figure 6.9, we show different



(a) (b)

Figure 6.8 (a) XML tree, (b) Query Q_3

snapshots of the query stacks during the execution of the algorithm. For simplicity, for each candidate output, only the fields for Y and Z in $BFlags$ are shown in the figure. Black boxes in the boolean arrays $PCFlags$ and $SPFlags$ (abbreviated as PC and SP in the figures) associated with stack entries denote fields which are true. Similarly, for the boolean array $BFlags$ associated with candidate outputs.

When $\langle u_3 \rangle$ is read, since u_3 is not a redundant match of node U, a new entry for u_3 is created and pushed on stack S_U (lines 4 and 6 of $startEval$). As U is the only sink node in partial path p_1, node u_3 sustains p_1. Therefore, $u_3.SPFlags[p_1]$ is set to $true$ (line 10 of $startEval$). When $\langle /u_3 \rangle$ is read, u_3 is popped out from S_U (line 5 of $endEval$). Since u_3 is a candidate match of node U, and U is a predicate child of Y, $y_2.PCFlags[U]$ is set to $true$ (lines 1-3 of procedure $upwardPropagate$ in $endEval$).

When $\langle w_5 \rangle$ is read, it can be determined that w_5 sustains partial path p_2. Thus $w_5.SPFlags[p_2]$ is set to $true$. When $\langle /w_5 \rangle$ is read, since w_5 is a candidate match of the output node W, a new candidate output for w_5 is created. It is appended to z_4.candList since z_4 is the lowest ancestor of w_5 (lines 6-7 of $upwardPropagate$).

When $\langle /z_4 \rangle$ is read, since z_4 is not a candidate match of Z (it has no children matching V) and there are no entries below z_4 in stack S_Z, z_4 along with its candList (= $\{w_5\}$) is discarded (line 14 of $endEval$).

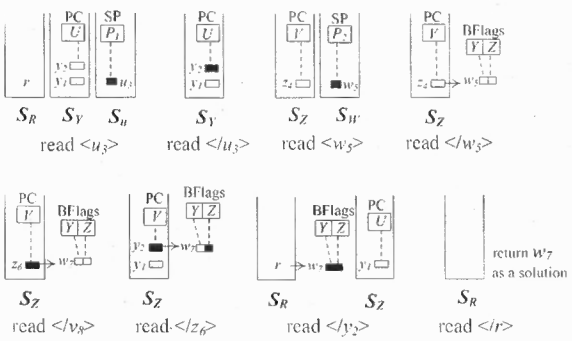


Figure 6.9 Snapshots of stacks during the evaluation of PSX on Q_3 and the XML tree of Figure 6.8

When $\langle /v_8 \rangle$ is read, since v_8 is a candidate match of node V, and V is a predicate child of Z, $z_6.PCFlags[V]$ is set to $true$. When $\langle /z_6 \rangle$ is read, since z_6 is a candidate match of Z, for each candidate output in z_6.candList (= $\{w_7\}$), its field $BFlags[Z]$ is set to $true$ (line 13 of $upwardPropagate$). Then, since its field $BFlags[Y]$ is $false$, the candidate output w_7 is appended to the candlist of its lowest ancestor y_1 (lines 18-20).

When $\langle /y_2 \rangle$ is read, since y_2 is a candidate match of Y, the field $BFlags[Y]$ of the candidate output w_7 in y_2.candList is set to $true$. As a result, fields $BFlags[Y]$ and $BFlags[Z]$ associated with the candidate output w_7 are set to $true$, and w_7 is appended to r.candList (line 16). Subsequently, since r is a candidate match of R, r.candList (= $\{w_7\}$) is returned to the user when $\langle /r \rangle$ is read(lines 9-10 of $endEval$).

As a comparison, we consider evaluating the same query Q_3 and the XML tree of Figure 6.8 using Algorithm X_{aos} [32]. Note that even though X_{aos} can evaluate this query, it cannot support queries as general as PTPQs. The XPath expression for query Q_3 is $//Y[U]//W[\backslash\backslash Z/V]$. Algorithm X_{aos} builds a parse tree for this XPath expression. Also, it constructs a dag for the XPath query in which all reverse axes are converted into their symmetrical forward axes. The dag is used to determine whether an XML document node

is an ancestor match of a dag node but pattern matches are constructed based on the parse tree. X_{aos} uses propagation techniques to construct pattern matches during evaluation. Specifically, after $\langle /w_5 \rangle$ is read, it optimistically propagates node z_4 to w_5 by assuming z_4 is a real match of Z (i.e., it matches the sub-query rooted at Z in the parse tree). Subsequently, node w_5 is propagated to nodes y_1 and y_2. At this time, X_{aos} determines that y_2 is a real match of Y and propagates it to node r. After $\langle /z_4 \rangle$ is read, it can be determined that z_4 is not a real match of Z since it has no child match for V. As a result, node z_4 is removed from w_5. The undo propagation is then recursively applied to entries w_1, y_1, y_2, and r. Clearly, such undo operation affects negatively the time and memory space performance of X_{aos}. X_{aos} produces the query solutions after all the XML document nodes have been scanned. The query solutions are produced by traversing the matches of the query and by projecting them on the query output node. However, X_{aos} may redundantly store multiple copies of the same output in different matches of the query. For this reason, an additional effort is needed to eliminate duplicate solutions at the final stage. In our example, two matches of the query are constructed: $[R : r[Y : y_2[U : u_3, W : w_7[Z : z_6[V : v_8]]]]]$ and $[R : r[Y : y_1[U : u_9, W : w_7[Z : z_6[V : v_8]]]]]$. Projecting each of them on the output node W returns the solution w_7 twice.

Notice that to evaluate query Q_3, $SPEX$ [69] has to first decompose it into three subqueries: two path queries Y/U and Z/V and one single-join dag $\{Y, Z\}//W$. Each subquery is evaluated separately and the solutions are composed. In a more recent version of $SPEX$ [11], Q_3 has to be transformed to two TPQs similar to those shown in Figure 6.2, which are again evaluated separately.

6.3.5 Analysis

Correctness. The following proposition characterizes the population of stacks during execution.

Proposition 6.3.1 *Let x be the current node on an open event of an XML tree T. Procedure $startEval$ correctly pushes x onto the stacks of the query nodes that match x while avoiding redundant matches.*

Proof. Let X be a query node in Q that matches x. Procedure $startEval$ first determines if x is an ancestor match of X using Definition 6.2.1. Node x is not pushed on stack S_X if it is not an ancestor match of X. If X is not a predicate node, x cannot have redundant matches and is pushed directly on stack S_X. Otherwise, procedure $startEval$ proceeds to check if the ancestor match x is a redundant match of X. Subsequently, x is pushed on S_X only if it is not redundant. □

The next proposition characterizes the transformation of candidate outputs into solutions during execution.

Proposition 6.3.2 *Let R be the root of Q, X be the output node of Q, and x be an entry in stack S_X. Procedure $endEval$ returns x to the user as a solution only if x is eventually propagated to an entry in S_R which is a candidate match of R.*

Proof. The claim trivially holds if X equals R. Note that procedure $endEval$ determines if a stream node in T is a candidate match of its corresponding node in Q only when the end event of that stream node is read (line 8). If x is not a candidate match of X, x is discarded by $endEval$. Let x be a candidate match of X. Procedure $upwardPropagate$ is invoked to appropriately propagate x to the top stack entry y of a parent Y of X in Q (line 5).

When the end event of y is read and y is found to be a candidate match of Y, two cases are considered: (1) Y is the root R. In this case, procedure $endEval$ returns x along with the other candidate outputs in $y.candList$ to the user (line 10). (2) Y is a non-root backbone node of Q. In this case, procedure $upwardPropagate$ is invoked to propagate x to its lowest ancestor which is the top stack entry of a sibling backbone node of Y or a parent of Y, depending on whether ancestors of x have been found to be candidate matches for the sibling backbone nodes of Y (lines 9-20).

If y is not a candidate match of Y, procedure $downwardPropagate$ is invoked (line 14). Let Z be the query node $host(Y)$. Node x along with other candidate outputs in $y.candList$ is downward propagated to the top stack entry of Z or of a sibling backbone node of Z. If Z is $null$ or the stack is empty, all the candidate outputs in $y.candList$ including x are discarded.

The above propagations of x continue until either x is discarded or is returned to the user. □

The correctness of Algorithm PSX follows from the previous two propositions.

Space and time complexity. Given a query Q and an XML tree T, Figure 6.10 shows the list of parameters used for the complexity analysis. Among them, the recursion depth is defined as follows. The *recursion depth of a query node X in T* is defined in [36] as the maximum number of nodes in a path of T that are ancestor matches of X. We define the *recursion depth D* of Q in T as the maximum recursion depth of the query nodes of Q in T.

The space complexity of Algorithm *PSX* is composed of two parts. The first part is the space consumed by the stacks. Since the number of entries in each stack at any given

Parameter	Description
$\|Q\|$	Number of nodes and edges in Q
N	Number of nodes in Q
P	Max. no. of PPs in Q which share a query node as a sink node
M	Maximum number of sink nodes in a partial path of Q
B	Maximum number of backbone children of any node in Q
S	Maximum number of backbone siblings of any node in Q
H	Height of T
$\|T\|$	Number of nodes in T
D	Recursion depth of Q on T

Figure 6.10 Complexity parameters

time is bounded by D, and the size of each stack entry is bounded by the out-degree of the corresponding query node, the space used by stacks is $O(D \times |Q|)$. The second part is used for storing candidate outputs whose number is bounded by $|T|$. When $B > 1$, each candidate output is associated with a boolean array $BFlags$ of size $O(N)$. Therefore, the total space needed for the candidate outputs when $B > 1$ is $O(|T| \times N)$. When $B = 1$, the total space needed for the candidate outputs is $O(|T|)$.

The time complexity of Algorithm *PSX* is determined by the time for accessing stack entries, and the time for processing candidate outputs.

For a current node x, let X be a query node that matches x. Procedure *startEval* and *endEval* spend respectively $O(fanin(X) + fanout(X) + M \times P)$ and $O(fanin(X) + fanout(X) + P)$ on accessing stack entries, where $fanin(X)$ and $fanout(X)$ denote respectively the in-degree and out-degree of X in Q. Since the number of query nodes that match the current node is $O(N)$, the total time spent on accessing stack entries for each node in T is $O(|Q| + N \times M \times P)$, which is dominated by $O(|Q| \times M \times P)$.

Candidate outputs are processed by procedure *endEval*. When $B = 1$, each candidate output is visited exactly once regardless of whether it is returned to the user or discarded. Thus, the total time spent on candidate outputs is $O(|T|)$. When $B > 1$, the total time spent on candidate outputs for the upward propagation is $O(|T| \times S \times H)$, since each candidate output can be propagated H times and each propagation takes $O(S)$ on finding the target stack entry.

Theorem 6.3.1 *Algorithm* PSX *correctly evaluates a query Q on a streaming XML document T. When $B = 1$, Algorithm* PSX *uses $O(|T| + D \times |Q|)$ space and $O(|T| \times |Q| \times M \times P)$ time. When $B > 1$, it uses $O(|T| \times N + D \times |Q|)$ space and $O(|T| \times (|Q| \times M \times P + S \times H))$ time.*

If Q is a tree-pattern query (TPQ), the values of parameters P, M, B, and S are 1, and $O(|Q|)$ equals N. In this case, the time and space complexity of Algorithm *PSX* are $O(|T| \times |Q|)$ and $O(|T| + D \times |Q|)$, respectively. Therefore, they are equal to the time and space complexity of the best known streaming algorithms [30] (the space used in [30] consists of caching space which is $O(D \times |Q|)$ and buffering space which is $O(|T|)$). Note however that the streaming algorithms in [30] support only TPQs while *PSX* supports a broad fragment of XPath that strictly contains TPQs.

6.4 Experimental Evaluation

We have implemented Algorithm PSX in order to experimentally study its execution time, memory usage, and scalability. Since there are no other streaming algorithms that support such a broad fragment of XPath, we compare PSX with Algorithm X_{aos} [32]. Even though X_{aos} cannot support PTPQs, it supports a subclass of XPath broader than TPQs, since it

can evaluate TPQs extended with reverse axes. As X_{aos} is not publicly available at this time, we implemented it based on the algorithm described in [32].

6.4.1 Experimental Setup

We implemented both algorithms (PSX and X_{aos}) in Java. We used the SAX XML parser [70], a event-based parser that scans XML document trees and produces a stream of events. All the experiments reported here were performed on an Intel Core 2 CPU 2.13 GHz processor with 2GB memory running JVM 1.6.0 in Windows XP Professional 2002. Each experiment was run $five$ times and each value displayed in the plots is averaged over these $five$ measurements.

We evaluated the performance of the algorithms on three datasets whose statistics are shown in Figure 6.11. The first one is a benchmark dataset using $XMark$[1] with $factor$ = 1. This dataset does not include recursive elements. The second one is a real data from the $Treebank$ project[2]. This dataset includes multiple recursive elements. The third one is a synthetic dataset generated by IBM's XML Generator[3] with $NumberLevels$ = 8 and $MaxRepeats$ = 4, based on the DTD shown in Fig. 6.12. By construction, this dataset includes highly recursive structures.

On each one of the three datasets, we tested 5 PTPQs. The queries on the synthetic dataset are shown in Figure 6.13. The queries on the other two datasets are analogous in structure to those for the synthetic dataset and are adapted for their respective dataset. We use the following naming convention for those queries: the queries are named NQ_i, $i = 1, \ldots, 5$, where N='X' denotes the XMark dataset, N='T' denotes the treebank

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

[2] http://www.cis.upenn.edu/~treebank

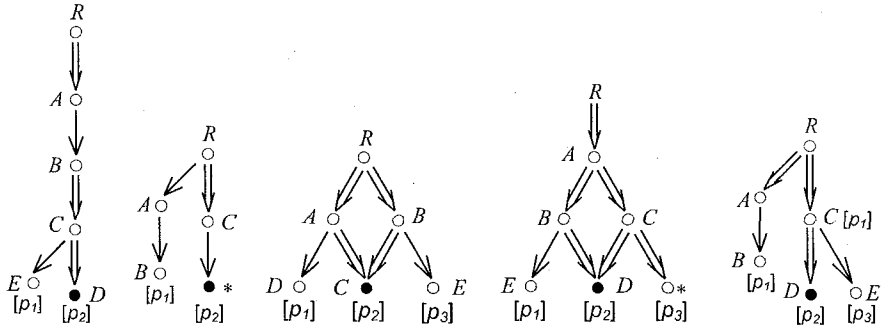
[3] http://www.alphaworks.ibm.com/tech/xmlgenerator

	XMark	*Treebank*	*Synthetic*
Size	113MB	82MB	20.3MB
#nodes	1627K	2380K	580K
#labels	74	250	6
Max/Avg depth	12/5.6	36/8.4	9/8.8

Figure 6.11 Dataset statistics

```
<!ELEMENT R ((A, B, C)+, D*, E*)>      <!ELEMENT C (A*, B*, D*, E*)>
<!ELEMENT A (B*, C*, D*, E*)>          <!ELEMENT D (#PCDATA)>
<!ELEMENT B (A*, C*, D*, E*)>          <!ELEMENT E (#PCDATA)>
```

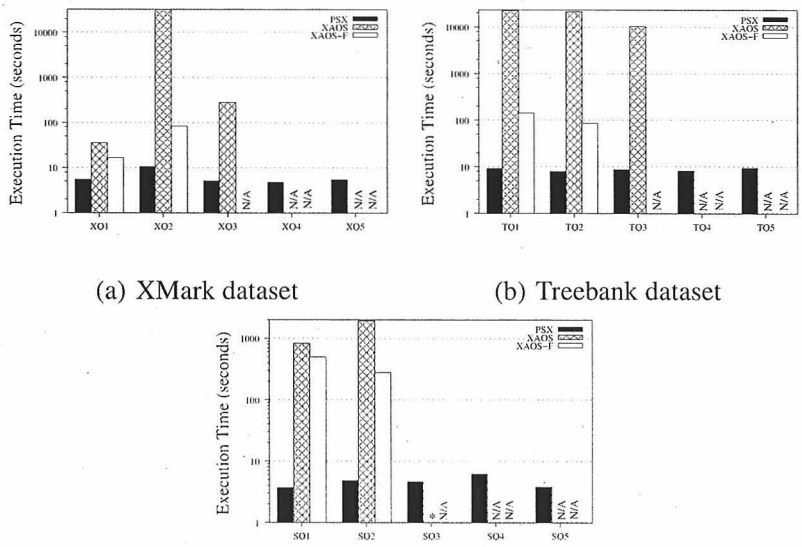
Figure 6.12 DTD for synthetic dataset



(a) SQ_1 (b) SQ_2 (c) SQ_3 (d) SQ_4 (e) SQ_5

Figure 6.13 Queries for Synthetic Dataset

dataset, and N='S' denotes the synthetic dataset. NQ_1 and NQ_2 are TPQs, while NQ_3 to NQ_5 are 'pure' PTPQs, i.e., they cannot be expressed by a single TPQ. Notice that even though NQ_5, for instance SQ_5, is syntactically similar to a TPQ, it is in fact a pure PTPQ because both nodes B and C are annotated by the same partial path p_1. This implies that these two nodes and their ancestor nodes lie on the same path. PSX supports all five queries of each dataset but X_{aos} only supports the first three. X_{aos} takes as input an

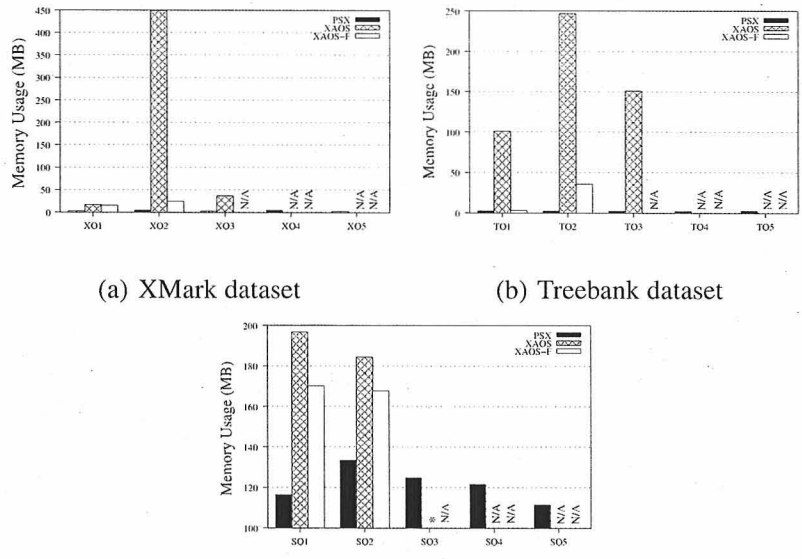


(a) XMark dataset (b) Treebank dataset

(c) Synthetic dataset

'*' denotes an execution that didn't finish within 7 hours; 'N/A' denotes incapacity of the algorithm to support the query

Figure 6.14 Query execution time



(a) XMark dataset (b) Treebank dataset

(c) Synthetic dataset

Figure 6.15 Maximum memory usage

XPath expression. In general, a given query can be equivalently represented by more than one XPath expression. For instance, $//A/B//C[/E]//D$ (which involves only forward

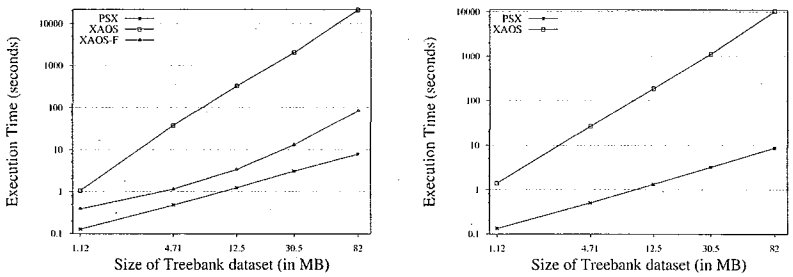
axes) and $//B[\backslash A]//C[/E]//D$ (which involves also reverse axes) are equivalent XPath expressions for SQ_1. In the experiments, for each of the first two queries (TPQs), we used two XPath expressions: one with only forward axes and one with both forward and reverse axes. We tested X_{aos} on both types of XPath expressions for the same query in order to examine the behavior of X_{aos} in the presence and absence of reverse axes. Note that the behavior of PSX is not affected by the syntax of the XPath expressions, since the input of the algorithm is a dag.

6.4.2 Query Execution Time

We compare the execution time of PSX, X_{aos}, and X_{aos}-F (X_{aos} on XPath expressions with only forward axes). Figure 6.23 shows the results of the three datasets (notice the logarithmic scale used for the Y-axis). As we can see, PSX has the best time performance, and in most cases it outperforms X_{aos} by at least one order of magnitude. The performance of PSX is stable, and does not degrade on more complex queries and on data with highly recursive structures.

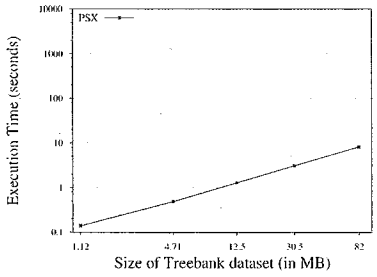
X_{aos} is more expensive than both PSX and X_{aos}-F in all the cases it applies. Its performance degrades significantly on recursive data and complex queries. For instance, when evaluating SQ_3 on the synthetic dataset (Figure 6.23(a)), X_{aos} was not able to finish within 7 hours. This can be explained as follows. First, X_{aos} exhaustively enumerates matches of a query pattern which can be exponential in the size of the query. Second, X_{aos} does not consider the existential semantics of the query during evaluation.

Although X_{aos}-F suffers from the same two drawbacks of X_{aos}, it performs better than X_{aos} in the cases it applies. The reason is that in the presence of reverse axes in the input XPath expression, X_{aos} may accumulate false pattern matches which have to be



(a) TQ_2

(b) TQ_3



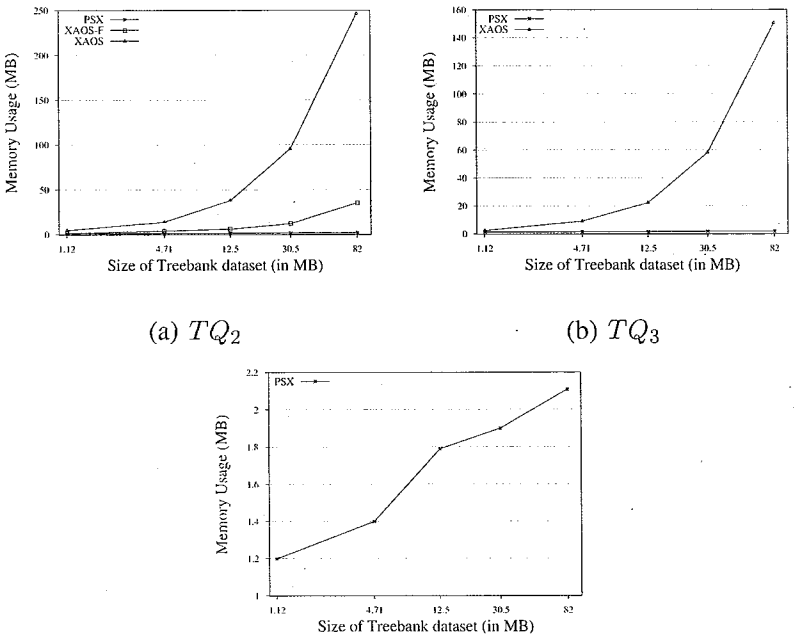
(c) TQ_4

Figure 6.16 Query execution time on Treebank data with increasing size

cleaned through 'backtracking'. This additional computation penalizes its performance. For example, when evaluating XQ_2 on the $XMark$ dataset with the XPath expression $//quantity[\backslash\backslash item[//mail]\backslash *]$, X_{aos} accumulates also all the false matches to *quantity*. This results in poor performance compared to X_{aos}-F (Figure 6.14(a)).

6.4.3 Memory Usage

We compare the maximum memory usage of PSX, X_{aos}, and X_{aos}-F. Figure 6.24 shows the results on the three datasets. The following observations can be made. First, PSX uses substantially less memory than X_{aos} and X_{aos}-F in all the cases (recall that X_{aos} can support only the first three, and X_{aos}-F only the first two queries). The memory usage of PSX on both $XMark$ and $Treebank$ datasets is stable, ranging from $1MB$ to $4MB$ (Figures



(a) TQ_2

(b) TQ_3

(c) TQ_4

Figure 6.17 Maximum memory usage on Treebank data with increasing size

6.15(a) and 6.15(b)). It increases up to $133MB$ for query SQ_2 on the systhetic dataset (Figure 6.24(a)). This can be explained by the following: (1) Since the $XMark$ dataset has no recursive structures, the recursion depth of all the queries on this dataset is 1. Thus, at any point of time, there is at most one entry stored in each query stack. (2) Although the $Treebank$ dataset has deep recursive structures, the number of solutions returned by the queries is small (up to 265 for TQ_5). (3) The synthetic dataset has highly recursive structures. Further, almost all the nodes in this dataset are relevant to the queries. Thus, the number of pattern matches that could potentially contribute to query answers is expected to be large (it can be exponential in the size of data and queries). Since any streaming algorithm has to store those potentially useful matches, the memory usage is expected to be high. This expectation is confirmed by the large number

of solutions returned by the queries on this dataset which ranges from $7K$ (for query SQ_5) to $200K$ (for query SQ_2). Note that in SQ_2, the output node is labeled by $*$ (wildcard), which explains the large number of solutions. The difference on the memory usage for PSX on SQ_1 to SQ_5 is due to the different structure of the queries. These results are in line with the space complexity of PSX stated in Theorem 6.3.1. Both X_{aos} and X_{aos}-F use less memory for SQ_2 than for SQ_1, while PSX uses more memory for SQ_2 than for SQ_1. The reason is that the memory usage of X_{aos} and X_{aos}-F depends on the number of pattern matches of the query (all stored in memory by these algorithms) which are more for SQ_1 than for SQ_2. In contrast, PSX avoids storing redundant query matches. It stores mainly query matches that contribute to a solution and these increase from $63K$ for SQ_1 to $200k$ for SQ_2.

X_{aos} consumes more memory space than X_{aos}-F in all the cases they apply. In particular, when evaluating TQ_1 on the $Treebank$ dataset, X_{aos} consumes about 40 times more space than X_{aos}-F (Figure 6.15(b)). The reason is that, as mentioned earlier, the presence of reverse axes in an XPath expression may lead to the generation of false pattern matches, and this increases the memory consumption of X_{aos}.

6.4.4 Scalability

We also measured the scalability of PSX, X_{aos}, and X_{aos}-F as the size of the input datasets increases. Figure 6.25 reports on the execution time of the algorithms increasing the size of Treebank XML data for three different queries TQ_2, TQ_3, and TQ_4. The scale of both X-axis and Y-axis is logarithmic. The performance of the queries on the other datasets is similar and is omitted here. The results show that as the input data size increases,

the execution time of PSX increases very slowly for both simple and complex queries, whereas the performance of X_{aos} degrades sharply in all the cases it applies.

Figure 6.26 shows the maximum memory usage increasing the size of Treebank XML data for the previous three queries. When the data size increases from $1MB$ to $82MB$, the memory usage of PSX is relatively constant. X_{aos}-F uses slightly more memory than PSX for TQ_2 (Figure 6.17(a)). In contrast, the memory consumption of X_{aos} increases much faster than the data size.

In summary, the experimental results show that Algorithm PSX is practically efficient with guaranteed polynomial time and space complexity in the size of the data and query. It is capable of evaluating a broader structural fragment of XPath than any existing streaming algorithm. Compared to the only known streaming algorithm that supports TPQs extended with reverse axes, PSX performs better by wide margin and shows much better scalability for processing both simple and complex queries on XML data with deep recursive structures.

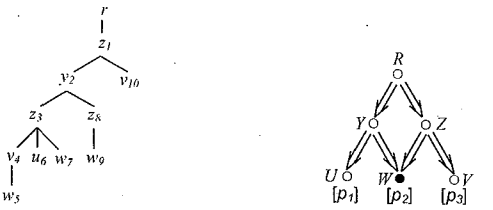
6.5 The Eager Evaluation Algorithm

Algorithm PSX evaluates query predicates and returns solutions to the user only when close events are encountered. This evaluation strategy is called *lazy* in [29, 30]. The lazy strategy makes the evaluation process natural. The query response time and memory space usage of Algorithm PSX can be improved at a small expense of the execution time. This can be achieved by an evaluation strategy which eagerly determines (that is, before the corresponding close events of query predicates are encountered) whether node matches should be returned as solutions to the user. It also proactively detects redundant matches. We call this strategy 'eager', and we present below an algorithm, called $EagerPSX$, that implements it.

In fact, stringent requirements on query response time and memory usage are important or even necessary for a number of streaming applications, including transaction monitoring systems [71] and sensor network systems [72]. These applications typically deliver data in streams that are produced continuously and represent real-world events, like financial tickers and traffic accidents, which need to be responded to.

We present below a motivating example which is used later to illustrate how differently $EagerPSX$ identifies redundant matches and returns solutions.

Example 6.5.1 *Consider evaluating query Q_4 of Figure 6.18(b) on the XML tree of Figure 6.18(a) using Algorithm PSX. Figure 6.19 shows different snapshots of the query stacks during the execution of the algorithm. Algorithm PSX returns $\{w_5, w_7, w_9\}$ as answer when $\langle /r \rangle$ is read. Candidate outputs w_5 and w_7 are propagated along the path $z_3y_2z_1r$ while candidate output w_9 is propagated along the path $z_8y_2z_1r$ in order to become solutions. However, when $\langle u_6 \rangle$ is read, we have enough information to determine that node w_5 is a solution of Q_4 and could be returned to the user immediately. Similarly, nodes w_7 and w_9 can be returned as solutions as soon as they arrive. Therefore, we only need to store one candidate output w_5 in memory instead of three as Algorithm PSX does. Also, we can determine that node z_3 is not useful for computing query solutions once $\langle v_4 \rangle$ is read. The reason is that at this point, we can determine that both nodes z_1 and z_3 are candidate matches of Z. Any candidate output, such as w_5, that can become a solution following the output path ry_2z_3 can instead follow the output path rz_1y_2. This way, w_5 can be returned to the user earlier. Further, even though z_3 is a match of the backbone node Z, it can be identified as redundant and discarded. This allows us to save not only computations*



(a) (b)

Figure 6.18 (a) XML tree, (b) Query Q_4

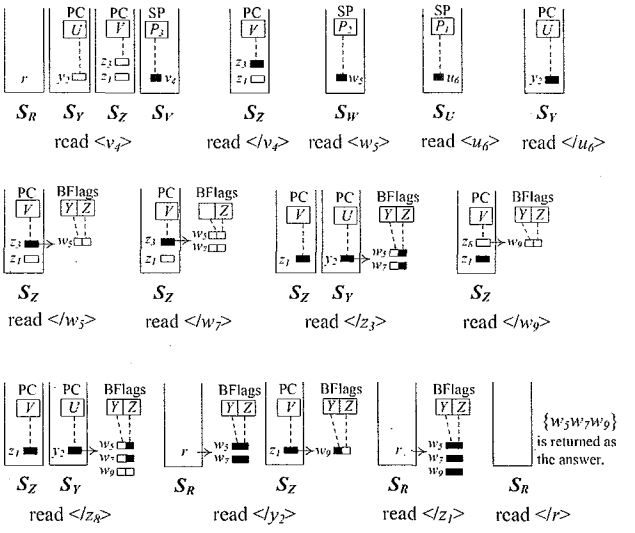
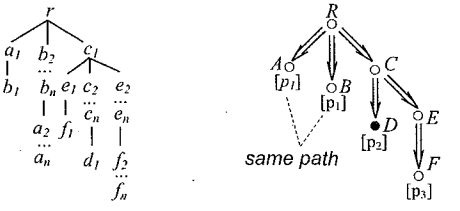


Figure 6.19 Snapshots of stacks during the evaluation of PSX on Q_4 and the XML tree of Figure 6.18



(a) (b)

Figure 6.20 (a) XML Tree, (b) Query Q_2

but also memory space. This type of query redundancy is related to redundant matches of backbone nodes.

To further show the importance of identifying redundant backbone node matches, let's consider evaluting query Q_2 of Figure 6.20(b) on the XML tree of Figure 6.20(a). Recall that a query match is redundant if it does not contribute to a possible new matching for the output node. From the discussion of Section 6.3.2, we know that nodes a_1, b_1, e_1 and f_1 which are matches for the predicate nodes A, B, E and F, respectively, contribute to the match d_1 of the output node D, whereas nodes $a_2, \ldots, a_n$, $b_2, \ldots, b_n$, $e_2, \ldots, e_n$, $f_2, \ldots, f_n$ are redundant predicate node matches since they all contribute to the same match d_1 of the output node D. These redundant matches occur because of the existential semantics of query predicates. As we can see from the figure, the backbone node C has n matches $\{c_1, \ldots, c_n\}$. These matches participate in n output paths ($rc_1d_1, \ldots, rc_nd_1$) for node d_1. Note that before nodes $c_2, \ldots, c_n$ are read, both r and c_1 have already satisfied their predicates. Therefore, any match of the output node D that is a descendant of c_1 (e.g., d_1) can be identified as a solution and thus should be returned to the user right away. The nodes $\{c_2, \ldots, c_n\}$ need not be stored in the stack of C. Keeping these nodes in memory unnecessarily delays the output of query solutions, and wastes computation time and memory space. It is important to note that redundant backbone node matches contribute to a number of pattern matches which in the worst case can be exponential on the size of the query.

6.5.1 Algorithm $EagerPSX$

Algorithm $EagerPSX$ has the same body as Algorithm PSX shown in Listing 9. The only difference is that the open and close event handlers $startEval$ and $endEval$ are replaced by $startEvalEager$ and $endEvalEager$ which are shown in Listings 12 and 13, respectively.

Listing 12 Procedure startEvalEager(X, x)

```
1  if (X ≠ R ∧ ∃P ∈ parents(X): empty(S_P)) then
2      return
3  if ((X = R) ∨ (∀P ∈ parents(X): top(S_P) and x satisfy the
   structural relationship between P and X in Q)) then
4      if (isMatchRedundant(X)) then
5          return
6      e ← newStackEntry(S_X, x)
7      push(S_X, e)
8      updateStackEntry(X, e)
9      if (X is the output node ∧ e.down ∧ e.up) then
10         output(x)
11     for (every p_i ∈ PPsSink(X)) do
12         if (∀ Y ∈ sinkNodes(p_i): ¬empty(S_Y)) then
13             for (every Y ∈ sinkNodes(p_i)) do
14                 top(S_Y).SPFlags[p_i] ← true
15                 if (isCandMatch(Y, top(S_Y)) = true) then
16                     found ← true
17             if (found) then
18                 dagTraversal(p_i)
```

Function isMatchRedundant(X)

```
1  if (X is a predicate node) then
2      return (∀P ∈ parents(X): top(S_P).PCFlags[X] = true)
3  else if (X is a backbone node) then
4      Y ← host(X)
5      return (¬ empty(S_Y) ∧ top(S_Y).down ∧ top(S_Y).up)
```

Procedure updateStackEntry(X, e)

```
1  for (every P ∈ parents(X)) do
2      e.parPtrs[P] ← top(S_P)
3      e' ← top(S_P)
4      if (X is the output node or a backbone node then
5          if (e'.childPtrs[X] = null) then
6              e'.childPtrs[X] ← e
7          if (e'.up ∧ e'.down = true) then
8              e.parFlags[P] ← true
9  if (X is the output node or a backbone node) then
10     if (∀P ∈ parents(X): top(S_P).down ∧ top(S_P).up)
       then
11         e.up ← true
12     if (X is an internal node ∧ PChildren(X)= ∅) then
13         e.down ← true
```

Procedure dagTraversal(p_i)

```
1  let G_{p_i} be the query dag whose nodes are annotated by p_i.
   Dag G_{p_i} is rooted at a node X which is the lowest backbone
   ancestor of nodes in sinkNodes(p_i).
2  construct a topological order of the nodes in G_{p_i} with the
   property that for any two nodes A and B in G_{p_i}, if A/B ∈ Q,
   then B is the next node of A in the topological order. Let Z
   be the node with the largest topological order
3  let mEntry be an array indexed by the nodes of G_{p_i}
4  bottomUpTraversal(Z, top(S_Z), p_i)
5  if (mEntry[X].up is true) then
6      let G_B be the query dag rooted at Z and whose nodes
       consist of backbone nodes of Q.
7      topDownTraversal(X, mEntry[X], G_B)
```

The Structure of Stack Entries For the purpose of the eager evaluation, we extend the structure of the stack entries introduced in Section 6.2. Every extended stack entry stores sufficient information for efficiently checking redundant (backbone and predicate) matches and determining candidate matches as well. More specifically, let Q be a query, X be a node in Q, and S_X be the stack for X. Each entry e in stack S_X is a 8-tuple *(XMLNode, SPFlags, PCFlags, CandList, down, up, parPtrs, childPtrs)*. The first four fields are described in Section 6.2. We describe below the last four fields.

$e.down$ is a boolean variable which is $true$ iff $e.XMLNode$ (i.e., x) has been found to be a candidate match of X.

$e.up$ is a boolean variable which is $true$ iff every $P \in parents(X)$ has a candidate match on the path from the root of T to x. To facilitate the computation of its value, we associate with e an auxiliary field *parFlags* (not listed among the eight fields above). Field *e.parFlags* is a boolean array indexed by the nodes in *parents*(X). Given $P \in parents(X)$, $e.parFlags[P]$ indicates whether P has a candidate match on the path from the root of T to x.

$e.parPtrs$ is an array of pointers indexed by the parent nodes of X in Q. Given $P \in parents(X)$, $e.parPtrs[P]$ points to the highest among the entries in stack S_P that correspond to ancestors of e in T.

$e.childPtrs$ is an array of pointers indexed by the backbone child nodes of X in Q. Given $Y \in BChildren(X)$, Pointer $e.childPtrs[Y]$ points to the highest entry e' among the entries in stack S_Y that correspond to descendants of e in T such that $e'.parPtrs[X]$=e. It is *null* if no such e' exist.

We illustrate in Figure 6.21 these stack entry fields when $\langle z_3 \rangle$ is read during the eager evaluation of query Q_4 of Figure 6.18(b) on the XML tree of Figure 6.18(a). For ease of

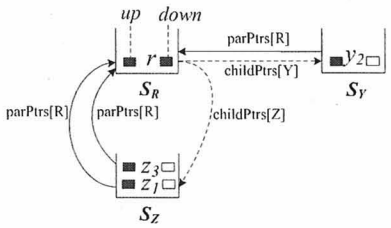


Figure 6.21 An illustration of stack entries during the evaluation of Q_4 on the XML tree of Figure 6.18 using $EagerPSX$

illustration, fields $SPFlags$, $PCFlags$, and $CandList$ are omitted in the figure. For every stack entry, we show the up and the $down$ fields as boxes which are black if the field is $true$ and white otherwise. Field $down$ keeps track of whether the corresponding XML node has been found to be a candidate match of a query node. For example, consider entries r and z_1 in stacks S_R and S_Z, respectively. $r.down$ = $true$ because node R has no predicate children and thus r trivally satisfies the candidate match requirements for R, while $z_1.down$ = $false$ because at this point of computation, it is not possible to determine whether z_1 has a descendant node that matches node V (node v_4 has not been read yet). Field up keeps track of whether the corresponding XML node has ancestor nodes which have been found to be candidate matches of all the parent nodes of the query node in consideration. For instance, $r.up$ = $true$ because r trivally satisfies the above requirement while $z_1.up$ = $true$ because r is a candidate match of R. A parent pointer in $parPtrs$ array of a stack entry of a query node is shown in the figure by a solid arrow pointing to the top entry of the stack of the parent node. For example, z_1 has a parent pointer in $parPtrs[R]$ pointing to r. A child pointer in $childPtrs$ array of a stack entry of a query node is shown in the figure by a dashed arrow pointing to the closest descendant of the entry in T that matches the corresponding child query node. For example, r has a child pointer $childPtrs[Z]$ pointing to z_1.

Open Event Handler The open event handler Procedure *startEvalEager* is shown in Listing 12. As the case of PSX, it starts by checking whether the current node x is an ancestor match of query node X (lines 1-3).

Checking redundant matches. Procedure *startEvalEager* proceeds to check whether x is a redundant match of X via a call to Function *isMatchRedundant*. The main difference of this function compared to the one in Procedure *startEval* (Listing 10) is that it also detects redundant matches for backbone nodes (lines 3-5). For instance, in the example of Figure 6.20. When $\langle c_2 \rangle$ is read, r and c_1 (the top entry in stack S_C) satisfy their predicates. Then, c_2 is identified as redundant for C and is not pushed onto S_C.

Setting up a new stack entry. Once *startEvalEager* determines that x is not a redundant match, it creates a new stack entry for x and pushes it on S_X (lines 6-7). Subsequently, procedure $updateStackEntry$ is invoked to set up the fields for e (line 8). Procedure $updateStackEntry$ first updates the pointers $parPtrs$ and $childPtrs$ for e (lines 1-6). Then, it updates fields $parFlags$ and up (lines 7-11). Finally, if X is an internal node (i.e., it is not a sink node of any partial path of Q) and X has no predicate children, then x is a candidate match of X and therefore $e.down$ is set to $true$ (lines 12-13).

If X is the output node of Q and $e.down$ and $e.up$ are set to $true$ by procedure $updateStackEntry$, then x can be identified as a solution and is returned to the user (lines 9-10 in $startEvalEager$).

Figure 6.21 described earlier shows the structures of stack entries after $\langle z_3 \rangle$ is read. This snapshot is constructed as follows: First, $\langle r \rangle$ is read and since r is an ancestor match of the query root R, a new stack entry for r is created and is pushed onto S_R. Then, as a result of a call to $updateStackEntry$, $e.up$ and $e.down$ are set to $true$. When $\langle y_2 \rangle$ is read,

two pointers are constructed: one is $parPtrs[R]$ pointing from y_2 to r in stack S_R, and the other is $childPtrs[Y]$ pointing from r to y_2. Also, $y_2.up$ is set to $true$. Finally, when $\langle z_3 \rangle$ is read, no child pointer $childPtrs[Z]$ from r to z_3 is created since there is such a pointer from r to z_1.

Traversing the query dags. Besides determining whether the current node x sustains the partial paths annotating X in Q (lines 11-14), procedure *startEvalEager* also checks whether the top entry of any sink node of a partial path p_i is a candidate match (line 15). If it is the case, procedure *dagTraversal* is invoked to traverse two dags G_{p_i} and G_B in that sequence (line 18). The purpose of the dag traversal is to examine whether there are matches that are solutions. Dags G_{p_i} and G_B are sub-dags of Q rooted at an ancestor node of X and consist of predicate nodes and backbone nodes, respectively. Procedure *dagTraversal* first calls procedure $bottomUpTraversal$ to traverse dag G_{p_i} recursively in a bottom-up way. During the traversal, $bottomUpTraversal$ evaluates the predicates of the matches encoded in the query stacks and updates stack entries and ancestor stack entries by following the parent pointers. Then, depending on the results returned from $bottomUpTraversal$, procedure *dagTraversal* possibly calls procedure $topDownTraversal$ to traverse the nodes of dag G_B recursively in a top-down manner. For each node under consideration, $topDownTraversal$ examines its stack entries to determine whether there are candidate outputs (associated with the stack entries) that can be returned as solutions to the user, and updates these stack entries and their descendant stack entries by following child pointers. During each traversal, redundant matches are detected and pruned. A traversal is terminated when either there are no more matches to examine, or a

match that has already been examined is encountered. Because of lack of space, we omit here more details on the two procedures [4].

Example 6.5.2 *Consider evaluating query Q_4 of Figure 6.18(b) on the XML tree of Figure 6.18(a). Figure 6.22 shows the snapshot of the query stacks. When $\langle v_4 \rangle$ is read, v_4 is identified as a candidate match of V, and therefore procedure $bottomUpTraversal$ is invoked to traverse the sub-dag (path in this case) $Z//V$ starting with V. After setting $v_4.down$ to $true$, $bottomUpTraversal$ goes up to Z. Then, it evaluates the predicates for entries z_3 and z_1 in stack S_Z in that order. Since z_3 and z_1 are both candidate matches of Z, it sets $z_3.down$ and $z_1.down$ to $true$. Procedure $bottomUpTraversal$ ends its traversal at z_1 and returns z_1 to the calling procedure* dagTraversal. *Given that $z_1.down$ and $z_1.up$ are $true$, procedure $topDownTraversal$ is invoked to traverse the sub-dag (path in this case) $Z//W$ starting with Z. Since $z_1.candList$ is empty, no solutions are returned at this time. Procedure $topDownTraversal$ proceeds to remove the entries above z_1 in stack S_Z (only z_3 in this case) since they constitute redundant matches. It terminates its traversal on W since stack S_W is empty.*

When $\langle w_5 \rangle$ is read, a new stack entry for w_5 is constructed and pushed onto stack S_W. Also, parent pointers and child pointers to and from stack entry z_1 and y_2 are constructed for the new entry w_5. When $\langle u_6 \rangle$ is read, procedures $bottomUpTraversal$ and $topDownTraversal$ are invoked to traverse the sub-dags $Y//U$ and $Y//W$, respectively. As a result, both $y_2.down$ and $w_5.up$ are set to $true$. At this time, node w_5 can be identified as a solution and is returned to the user. When $\langle w_7 \rangle$ and $\langle w_9 \rangle$ are read, both w_7 and w_9 are returned as solutions right away before other nodes are read. Notice that when $\langle z_8 \rangle$ is read, it is identified as a redundant match and thus it is ignored.

[4] The full version of the algorithm and its description can be found in http://web.njit.edu/~xw43/paper/eagerAlgo.pdf

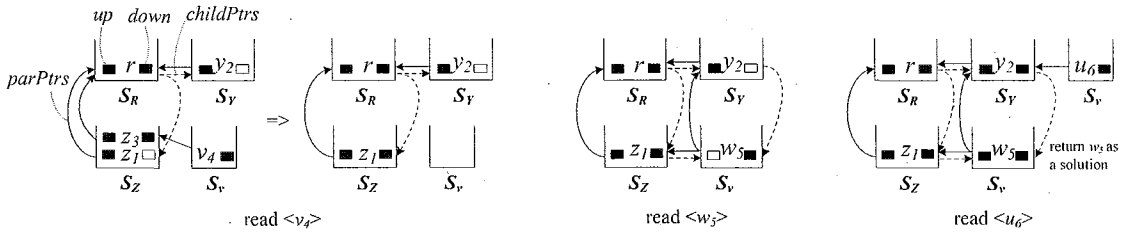


Figure 6.22 Snapshots of stacks during the evaluation of EagerPSX on Q_4 and the XML tree of Figure 6.18

Close Event Handler Procedure *endEvalEager* is shown in Listing 13. It differs from Procedure *endEval* in Listing 11 in that: (1) since child pointers are now used, they have to be updated whenever the entries they point to are popped out from their stacks (lines 6-10), and (2) the work performed before by procedures $mergeFlags$ and $isCandMatch$ is now performed by procedure $bottomUpTraversal$ called by *startEvalEager*.

6.5.2 Analysis

Let Q be a query and T be an XML tree. For the complexity analysis of *EagerPSX*, we refer to the parameters listed in Figure 6.10.

As with Algorithm *PSX*, the space complexity of *EagerPSX* is composed of two parts. One part of the space is consumed by the stacks. Since the number of entries in each stack at any given time is bounded by D, and the size of each stack entry is bounded by the out-degree and the in-degree of the corresponding query node, the space used by the stacks is $O(D \times |Q|)$. The other part is used for storing candidate outputs whose number is bounded by $|T|$. When $B > 1$, each candidate output is associated with a boolean array $BFlags$ of size $O(N)$. Therefore, the total space needed for the candidate outputs when $B > 1$ is $O(|T| \times N)$. When $B = 1$, the total space needed for the candidate outputs is $O(|T|)$.

Listing 13 Procedure endEvalEager(X, x)

```
1  if empty(S_X) then
2      return
3  s ← top(S_X)
4  if (s.XMLNode = x) then
5      pop(S_X)
6      if (X is the output or a backbone node) then
7          for (every P ∈ parents(X)) do
8              e ← s.parPtrs[P]
9              if (e.childPtrs[X] = s) then
10                 e.childPtrs[X] ← null
11         if (s.down = true) then
12             if (X = R) then
13                 output(s.candList)
14             else
15                 upwardPropagate(X, s)
16         else
17             downwardPropagate(X, s)
```

As we can see, *EagerPSX* has the same worst case space complexity as *PSX*. However, *EagerPSX* achieves better space performance because it applies evaluation strategies to eagerly determine whether node matches should be returned as solutions to the user and to proactively detect and prune redundant matches.

The time complexity of *EagerPSX* is determined by the time for accessing stack entries, and the time for processing candidate outputs. For a current node x, let X be a query node matching x. Procedure *endEval* spends $O(fanin(X))$ on accessing stack entries. Leaving apart the calls to Procedure *dagTraversal*, Procedure *startEval* spends $O(fanin(X) + fanout(X) + M \times P)$ on accessing stack entries.

Let e denote a stack entry for a query node Y. During its lifetime, the total time spent on e by procedures $bottomUpTraversal$ and $topDownTraversal$ is $O(fanin(Y)+fanout(Y)+P)$ and $O(fanin(Y)+fanout(Y))$, respectively. Therefore, Procedure *dagTraversal* spends $O(fanin(Y)+fanout(Y)+P)$ time on each stack entry. In summary, for each node in T, *EagerPSX* spends $O(|Q| \times M \times P)$ on accessing stack entries.

The time on processing candidate outputs is dominated by procedure *endEval*, and is the same as that of *PSX* in the worst case.

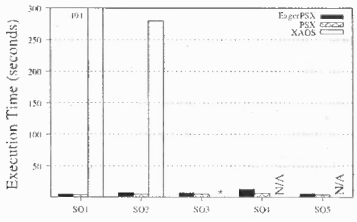
Theorem 6.5.1 *Algorithm* EagerPSX *correctly evaluates a query Q on a streaming XML document T. When $B = 1$, Algorithm* EagerPSX *uses $O(|T|+D\times|Q|)$ space and $O(|T|\times|Q|\times M\times P)$ time. When $B > 1$, it uses $O(|T|\times N+D\times|Q|)$ space and $O(|T|\times(|Q|\times M\times P+S\times H))$ time.*

Eager streaming algorithms have been presented in [34, 30] but they are restricted to TPQs. Besides supporting a class of queries that are more expressive than TPQs, our eager algorithm $EagerPSX$ is, to the best of our knowledge, the first one that detects and avoids processing different types of redundant query matches during streaming evaluation.

6.6 Experimental Evaluation

We have implemented Algorithm $EagerPSX$ in order to study its execution time, memory usage, and scalability. In this section, we experimentally compare $EagerPSX$ with Algorithms PSX and X_{aos} [32]. As mentioned before, X_{aos} is chosen for comparison because even though it does not support a broad fragment of XPath as PSX and $EagerPSX$ do, it nevertheless supports a restricted type of dag queries.

All the experiments were carried out on the same machine used for the experiments described in Section 6.4 with the same experimental settings, datasets and queries. Each



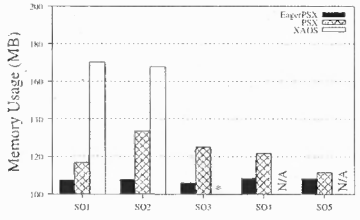
(a) Execution time

Query	$EagerPSX$	PSX	$XAOS$
SQ_1	0.02	3.83	493.8
SQ_2	0.01	4.6	279.5
SQ_3	0.012	4.64	*
SQ_4	0.09	6.43	N/A
SQ_5	0.015	4.19	N/A

(b) Time to 1st solution (seconds)

'*' denotes an execution that didn't finish within 7 hours; 'N/A' denotes incapacity of the algorithm to support the query

Figure 6.23 Query execution time and response time on synthetic dataset



(a) Max. runtime memory usage

Query	$EagerPSX$	PSX	$XAOS$
SQ_1	592	63864	94696
SQ_2	0	198458	198458
SQ_3	2050	90068	*
SQ_4	4783	113696	N/A
SQ_5	519	7271	N/A

(b) Max. stored candidate outputs

Figure 6.24 Memory usage on synthetic dataset

experiment was run $five$ times and each value displayed in the plots is averaged over these five measurements. In the interest of space, in the following we only report the results on the synthetic dataset. We obtain similar results on the other two datasets.

6.6.1 Query Execution Time

We compare the execution time of $EagerPSX$, PSX and X_{aos}. The execution time consists of data and query parsing time and query evaluation time. Figure 6.23(a) shows

the results on the synthetic dataset. As we can see, PSX has the best time performance, and in most cases it outperforms X_{aos} by at least one order of magnitude. $EagerPSX$ uses slightly more time than PSX, due to the overhead incurred by the traversals of the query dags for finding solutions. The performance of both $EagerPSX$ and PSX is stable, and does not degrade on more complex queries and on data with highly recursive structures.

X_{aos} is more expensive than both $EagerPSX$ and PSX in all the cases it applies. Its performance degrades significantly on recursive data and complex queries. For instance, when evaluating SQ_3 on the synthetic dataset (Figure 6.23(a)), X_{aos} was not able to finish within 7 hours.

6.6.2 Query Response Time

We compare the query response time of $EagerPSX$, PSX and X_{aos}. The query response time represents the time elapsed from the moment the query is issued to the moment the first solution is received. Figure 6.23(b) shows the query response time results on the synthetic dataset. As we can see, $EagerPSX$ gives the best query response time for both simple and complex queries. Compared to PSX and X_{aos}, $EagerPSX$ reduces the response time by orders of magnitude. $EagerPSX$ starts delivering query solutions almost immediately after a query is posed.

PSX returns solutions to the user when the end event of a node matching the root of a given query arrives. The worst case occurs when the only node matching the query root is the document root. For this reason, even though PSX performs best in terms of execution time, its query response time cannot compete with that of $EagerPSX$.

X_{aos} delivers query solutions only after the entire XML document is processed. Given that it also has the longest execution time, its query response time is the worst among the three.

6.6.3 Memory Usage

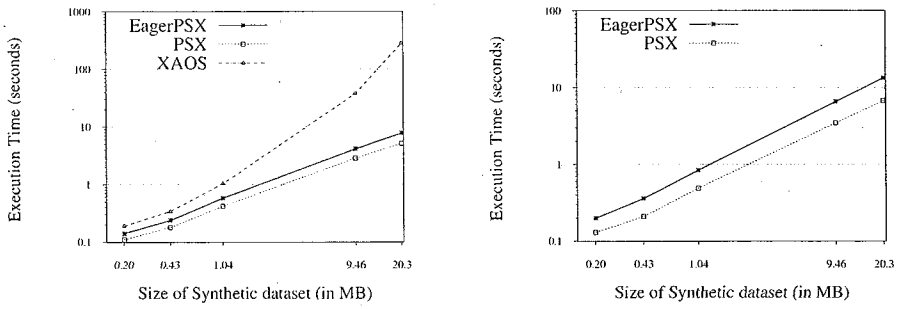
We compare the memory usage of $EagerPSX$, PSX and X_{aos}. We measure the memory usage in terms of maximal runtime memory consumption. We also measure the memory usage in terms of maximal number of candidate outputs stored at any point of time during execution.

Runtime memory consumption. Figure 6.24 shows the maximal memory consumption of the three algorithms on the synthetic dataset. As we can see, $EagerPSX$ uses substantially less memory than X_{aos} in all the cases (recall that X_{aos} can support only the first three queries). The memory usage of $EagerPSX$ is stable for both simple and complex queries.

PSX consumes more run time memory than $EagerPSX$ in all the test cases (up to 1.3 times on query SQ_2). The reason is three fold: (1) PSX cannot avoid storing redundant matches of backbone nodes during execution, (2) PSX has to store solutions in memory until the end event of the query root matches arrives, and (3) as we show below, PSX stores in memory more candidate outputs than $EagerPSX$.

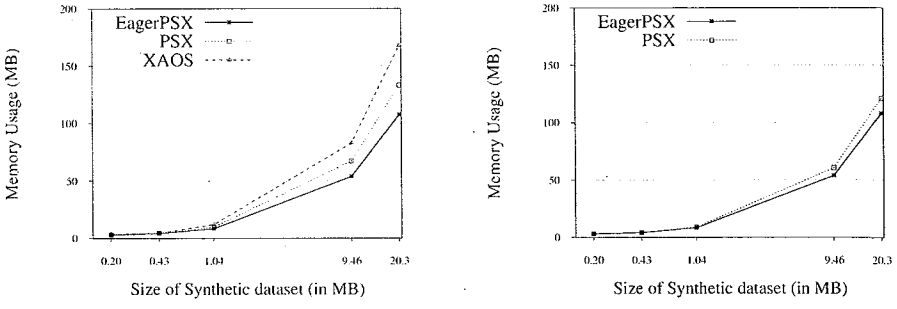
Number of stored candidate outputs. Figure 6.24(b) shows the maximal number of stored candidate outputs for the three algorithms on the synthetic dataset. Among the three algorithms, $EagerPSX$ stores the lowest number of candidate outputs. This is expected, since $EagerPSX$ employs the eager evaluation strategy which allows it to identify whether a candidate output is a solution as early as possible.

Compared to PSX, X_{aos} enumerates and stores all the matches of the query. Therefore, X_{aos} has to store multiple copies of the same candidate output. This is the case with query SQ_1 in Figure 6.24(b). PSX and X_{aos} identify the same 63864 candidate outputs.



(a) SQ_2 (b) SQ_4

Figure 6.25 Query execution time on synthetic data with increasing size



(a) SQ_2 (b) SQ_4

Figure 6.26 Memory consumption on synthetic data with increasing size

However, PSX stores each candidate output only once, while X_{aos} ends up storing 94696 copies of candidate outputs.

Note that in the best case, $EagerPSX$ avoids storing any candidate outputs and returns as solution to the user every candidate ouput as soon as it is identified as such. This is the case with SQ_2 in Figure 6.24(b), where $EagerPSX$ does not need to store any candidate output at all, while PSX and X_{aos} have to store approximately 200K nodes. Note also that in all the cases of Figure 6.24(b), PSX stores more candidate outputs than $EagerPSX$, and this is in accordance with the higher runtime memory consumption of PSX compared to $EagerPSX$ displayed in Figure 6.24.

6.6.4 Scalability

We also measured the scalability of $EagerPSX$, PSX and X_{aos} as the size of the input datasets increases. Figure 6.25 reports on the execution time of the algorithms increasing the size of synthetic XML data for two different queries: SQ_2 (a TPQ) and SQ_4 (a dag query). The scale of both X and Y axes is logarithmic. For the case of SQ_4, only results for $EagerPSX$ and PSX are reported, since X_{aos} cannot support this query. The results show that PSX always has the best time performance and $EagerPSX$ closely follows PSX. As the input data size increases, the execution time of $EagerPSX$ and PSX increases very slowly for both queries, whereas the execution time of X_{aos} increases sharply in the case it applies.

Figure 6.26 shows the runtime memory consumption increasing the size of synthetic XML data for the previous two queries. The memory consumption of X_{aos} increases faster than that of both $EagerPSX$ and PSX. PSX uses slightly more memory than $EagerPSX$.

In summary, the experimental results show that Algorithm $EagerPSX$ is efficient on a broad fragment of XPath. Compared to X_{aos}, the only known streaming algorithm that supports TPQs extended with reverse axes, $EagerPSX$ performs better by a wide margin in terms of time and space performance and scalability. It is runtime competitive with our lazy algorithm PSX for PTPQs, while achieving better space performance and greatly reducing the query response time for both simple and complex queries on XML data with deep recursive structures. Therefore, $EagerPSX$ can be very useful for current streaming applications that have stringent requirements on query response time and memory consumption.

CHAPTER 7

ASSIGNING SEMANTICS TO PARTIAL TREE-PATTERN QUERIES

In this chapter, we define our novel semantics for the PTPQ language. The chapter is organized as follows. Section 7.1 presents the data model and the query language. Index graphs and complete TPQs for a PTPQ are introduced in Section 7.2. In Section 7.3, we present our novel semantics for the PTPQ language. Our approach is compared with previous one in Section 7.4 and experimentally evaluated in Section 7.5.

7.1 Data Model and Query Language

For the purpose of defining semantics to partial tree-pattern queries, we make some modifications to the data model and the PTPQ language presented in Chapter 3.

Data Model. Let $\mathcal{E}$ be an infinite set of *elements* that includes a distinguished element r, $\mathcal{X}$ be an infinit set of *variables*, and $\mathcal{V}$ be an infinit set of *values*. Variables range over elements, and play the role of wildcards in tree-pattern queries. Here, we use variables to distinguish between different wildcard nodes. Symbols e, x, and v (possibly with indices) refer systematically to an element, a variable, and a value respectively. The term *construct* (denoted c) refers either to an element or a variable.

As is usual, we model XML documents as trees. Nodes in an XML tree are labeled by elements or values. In particular, the root node of an XML tree is the only node labeled by element r. Values can label only leaf nodes. Attributes of elements in an XML document are modeled as (sub)elements. For simplicity we assume that the same element does not

label two nodes on the same path (that is, the XML trees are not recursive). We discuss in the next section how this restriction can be relaxed.

Figure 7.1 shows three XML trees T_1, T_2, and T_3 from different data sources that record bibliographic information in different formats (a slight extension of an example introduced in [5, 6]). T_1 and T_3 categorize the data based on the publication year, while T_2 categorizes the data based on the type of publication (article or book). Still, in T_1 the year of the publications is specified as a child element of a "bib" node, while in T_3 there is no "bib" node, and the "book" and "article" nodes are children of a "year" node that indicates their year of publication. We are interested in retrieving information by issuing the same

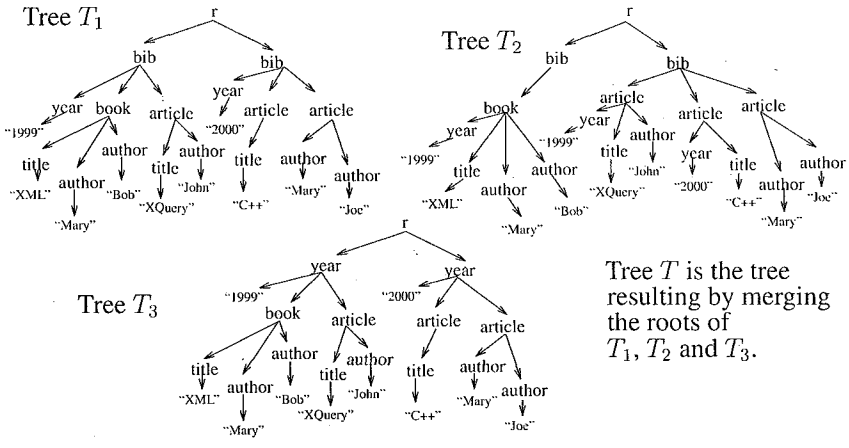


Figure 7.1 An XML Tree T

query against all these data sources, even though information is structured differently in each one of them. Therefore, we view all these XML trees as one tree T rooted at r.

Query Language. We make following extensions to the PTPQ defined in Definition 6.1.1: (1) each node is possibly annotated with a set of value predicates; and (2) at least one partial path is defined to the output path. Below is the full definition of the query language:

Definition 7.1.1 (PTPQ) *A* Partial Tree-Pattern Query *(PTPQ) is a triple* $Q = (\mathcal{P}, S, O)$, *where:*

(a) $\mathcal{P}$ *is a nonempty set of triples* $(p, \mathcal{A}, \mathcal{R})$ *called* Partial Paths *(PPs).*

p *is the name of the PP. The names of the PPs in* Q *are distinct. Therefore, we identify PPs in* Q *with their names.*

$\mathcal{A}$ *is a set of predicates of the form* $c = V$, *where* V, *the* annotation *of* c, *is a set of values* $\{v_1, \ldots, v_k\}$, $k \geq 1$. *The meaning of predicate* $c = V$ *is that* $c = v_1$ *or* $\ldots$ *or* $c = v_k$.

$\mathcal{R}$ *is a set of expressions of the form* $c_i \rightarrow c_j$ (child precedence relationship), $c_i \Rightarrow c_j$ (descendant precedence relationship), *and* $c_i \Longrightarrow c_j$ (descendant-or-self precedence relationship), *where constructs* c_i *and* c_j *are distinct. In particular,* $\mathcal{R}$ *comprises a descendant-or-self precedence relationship* $r \Longrightarrow c$, *for every predicate* $c = V$ *in* $\mathcal{A}$.

The expression $c[p]$ *denotes the construct* c *in PP* p.

(b) S *is a set of expressions of the form* $c_i[p_i] \equiv c_j[p_j]$, *where* p_i *and* p_j *are PPs in* $\mathcal{P}$. *These expressions are called* node sharing expressions. *Roughly speaking, they state that the node labeled by construct* c_i *in PP* p_i *and the node labeled by construct* c_j *in PP* p_j *coincide (the two PPs* share *this node). Set* S *can be empty.*

(c) O *is a set of PPs in* $\mathcal{P}$. *These PPs are called* output *PPs of* Q. □

We graphically represent PTPQs using graph notation. Each PP of a PTPQ Q is represented as a (not necessarily connected) graph of nodes identified with, and labeled by, the constructs of the PP. If a node n in the graph has an annotation V in Q, it is labeled by the predicate $c = V$ instead of the construct c. The name of each PP is shown by the corresponding PP graph. The names of the output PPs of Q are shown boxed. Child,

descendant and descendant-or-self precedence relationships in a PP are depicted using the arrows $\rightarrow$, $\Rightarrow$, and $==\Rightarrow$, respectively, between the corresponding nodes in the PP graph. In particular, descendant precedence relationships of the form $r \Rightarrow c$ and $r ==\Rightarrow c$ in a PP are shown only with the presence of node c in the PP graph. Variable names are prepended by a * sign, while values are shown between quotes. A node sharing expression $c_i[p_i] \equiv c_j[p_j]$ is represented by an edge between node c_i of the PP graph p_i and node c_j of the PP graph p_j labeled by the $\equiv$ symbol.

Suppose that we want to find the title and year of publications authored by "Mary" [5, 6]. We are not interested to restrict the type of publication we are looking for, and actually we do not know what type of publications are recorded in the XML data. Further, assume that we know that title and author are not categorization features in our XML document(s), and therefore they should appear below any categorization element. We formulate this PTPQ as shown in Figure 7.2. Symbols x and y denote variables, while p_1 and p_2 are the output PPs of the PTPQ. As another example, consider the query that finds additional authors of publications of which "Mary" is an author and also the title and year of those publications. In this case, assume that we expect author "Mary" and the other author to be sibling nodes and descendants of a publication node which has a descendant node "title". We do not have any idea about the placement of node "year" besides the fact that it should relate in some way to the publication. This PTPQ is shown in Figure 7.3.

The answer of a PTPQ is based on the concept of PTPQ embedding.

Definition 7.1.2 *An* embedding *of a PTPQ Q to an XML tree T is a mapping M of the constructs of the PPs of Q to nodes in T such that:*

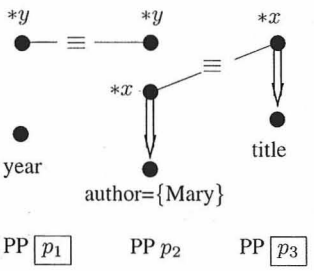


Figure 7.2 PTPQ Q_1

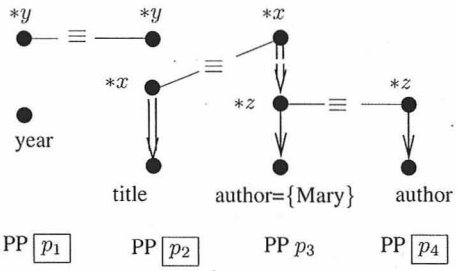


Figure 7.3 PTPQ Q_2

(a) An element e of Q is mapped by M to a node in T labeled by e; a variable v of Q is mapped by M to a node in T labeled by any element.

(b) The constructs of a PP in Q are mapped by M to nodes in T that are on the same path.

(c) If a construct c has an annotation V in a PP p (that is, a predicate $c = V$ is specified in p), then the image of $c[p]$ under M has a child node labeled by a value in V.

(d) $\forall\ c_i[p] \rightarrow c_j[p]$ in Q, $M(c_j[p])$ is a child of $M(c_i[p])$ in T; $\forall\ c_i[p] \Rightarrow c_j[p]$, in Q, $M(c_j[p])$ is a descendant of $M(c_i[p])$ in T; and $\forall\ c_i[p] \Longrightarrow c_j[p]$ in Q, $M(c_j[p])$ is a descendant of $M(c_i[p])$, or $M(c_i[p])$ and $M(c_j[p])$ coincide in T.

(e) $\forall\ l[p_i] \equiv l[p_j]$ in Q, $M(l[p_i])$ and $M(l[p_j])$ coincide. □

We call *image* of a PP p in Q under M, denoted $M(p)$, the path from the root of T that comprises all the images of the constructs of p under M and ends in one of them. Notice that more than one PP of Q may have their image on the same root-to-leaf path of T (M does not have to be a bijection). The concept of image of a PP is extended to apply to PTPQs in a straightforward way.

We initially define the answers of a PTPQ on an XML tree as follows.

Definition 7.1.3 *The* answer set *A of a PTPQ Q on an XML tree T is the set of subtrees of T formed by the images of the output PPs of Q under all possible embeddings of Q to*

T. *The subtrees comprise also the child value nodes of the elements. The subtrees in A are called* answers *of* Q *on* T.

Figure 7.4 shows the images of PTPQ Q_1 of Figure 7.2 under three of the possible embedding of Q_1 to the XML tree T of Figure 7.1. The values of the elements are additionally included in the images for clarity. The images of the output PPs of Q in the figures are shown with thicker arrow edges. More specifically, Figures 7.4(a), (b) and (c) correspond to embeddings of Q_1 to the XML trees T_1, T_2 and T_3 respectively that constitute tree T of Figure 7.1.

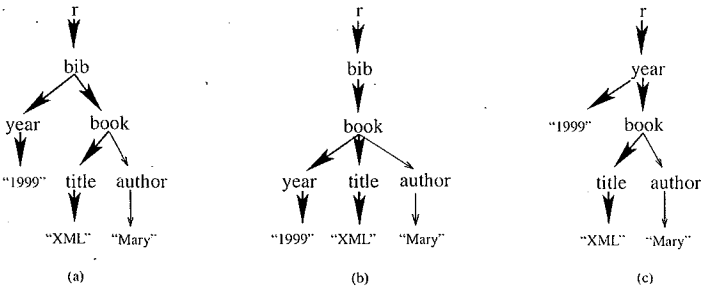


Figure 7.4 The images of Q_1 under three of the embeddings of Q_1 on T. The answer of Q_1 in every image is shown with thicker arrows.

Observe that the language is able to retrieve with one query the title and year of the publications of Mary from different parts of the XML tree, even though these parts structure the data in different ways.

The previous definition of the answer set of a PTPQ accepts any possible embedding of Q to T. This generality allows embeddings that do not relate elements and values in the way the user was expecting when formulating the query. We call the answers corresponding to these embeddings *meaningless* answers. For instance, each of the images of Q_1 shown in Figure 7.4 correctly corresponds to a publication (a book in this case) authored by "Mary". However, this is not the case with the images of Q_1 in Figure 7.5 under three other

embeddings of Q_1 into T. In each one of them, year and/or title values do not correspond

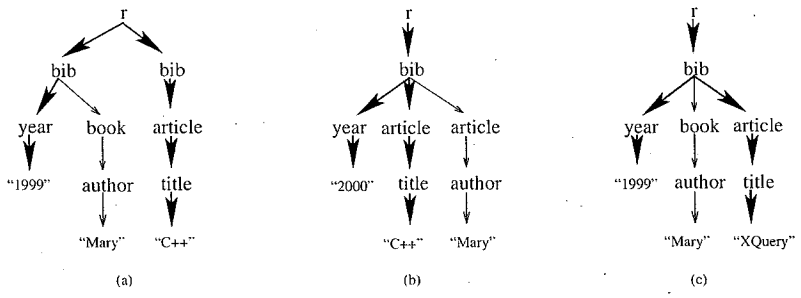


Figure 7.5 The images of Q_1 under three of the embeddings of Q_1 on T. The answer of Q_1 in every image is shown with thicker arrows.

to a publication authored by "Mary" even though these values appear in an answer with "Mary". In Section 7.3, we will present a technique that excludes these subtrees and returns answers to the user that are meaningful.

7.2 Evaluating PTPQs Using Complete TPQs

We show now how PTPQs can be evaluated using TPQs. We first discuss index graphs for XML trees. Then, we use index graphs to construct a set of complete TPQs whose answers, taken together, form the answer of a given PTPQ. Besides allowing us to evaluate PTPQs, the complete TPQs of a PTPQ provide the basis for defining meaningful semantics for PTPQs in the next section.

7.2.1 Index Graphs

Given a partitioning of the nodes of an XML tree T, an index graph for T is a graph G such that: (a) every node in G is associated with a distinct equivalence class of element nodes in T, and (b) there is an edge in G from the node associated with the equivalence class a to the node associated with the equivalence class b, iff there is an edge in T from a node in

a to a node in b. Index graphs have been referred to with different names in the literature including "path summaries", "path indexes" and "structural summaries". They differ in the equivalence relations they employ to partition the nodes of the XML tree which includes simulation and bismulation [63, 64] or even semantic equivalence relations [7]. Index graphs have been extensively studied in recent years in both the "exact" [65, 63, 73] and the "approximate" flavor [74, 64]. A common characteristic of those approaches is that the index graph is used as a back end for evaluating a class of path expressions without accessing the XML tree. To this end, the equivalence classes of the XML tree nodes are attached to the corresponding index graph nodes.

For the needs of PTPQs we define index graphs where the equivalence classes are formed by all the nodes labeled by the same element in the XML tree. Figure 7.6 shows the index graph G of the XML tree T of Figure 7.1.

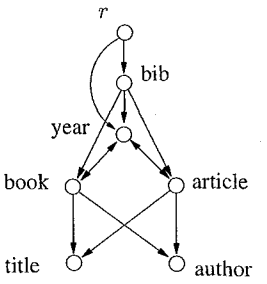


Figure 7.6 Index graph G

In contrast to other approaches, the equivalence classes of the XML tree nodes are not kept with the index graph. Therefore, PTPQs are ultimately evaluated on the XML tree. Even though the index graph for an XML tree is not a schema in the form of a DTD or an XML Schema, we take advantage of it in the same way schema information is exploited in relational databases. We use index graphs to support the evaluation of a PTPQ through the generation of a set of complete TPQs.

7.2.2 Complete TPQs for a PTPQ

If G is the index graph of an XML tree T, we say that T *underlies* G. Given a PTPQ Q and an index graph G, Q can be evaluated by computing a set of complete TPQs whose answers, taken together, are equal to the answer of Q on any XML tree underlying G. By *complete* TPQ we mean a TPQ that involves only child relationships and no variables (and therefore, completely specifies a tree pattern). Intuitively, a complete TPQ satisfies both: the structural and value constraints of the PTPQ, and the structural constrainst of the index graph.

Definition 7.2.1 *Let Q be a PTPQ and G be an index graph. A* complete *TPQ (CTPQ)* for *Q on G is a TPQ U without variables (wildcards) and descendant precedence relationships which is rooted at a node labeled by r and satisfies the following conditions:*

(a) There is a mapping M from the nodes of Q to the nodes of U that respects paths, labeling elements, precedence relationships, and node sharing expressions. If $V_1, \ldots, V_k$ are the annotations of all the nodes in Q that are mapped to the same node n in U, n is annotated by $V_1 \cap \ldots \cap V_k$. Two nodes in a path in U are not labeled by the same element, and every leaf node of U is the image of a node of Q under M. The output nodes of U are the images under M of the nodes of the output PPs of Q. Notice that it is possible that all the nodes of two distinct PPs of Q are mapped by M to nodes on the same path in U.

(b) There is a mapping M' from the nodes of U to the nodes of G that respects labeling elements and child precedence relationships. □

Figure 7.7 shows two of the CTPQs of the PTPQ Q_1 of Figure 7.2 on the index graph G of Figure 7.6. The output nodes have their labels boxed. For simplicity of presentation, the paths are not named.

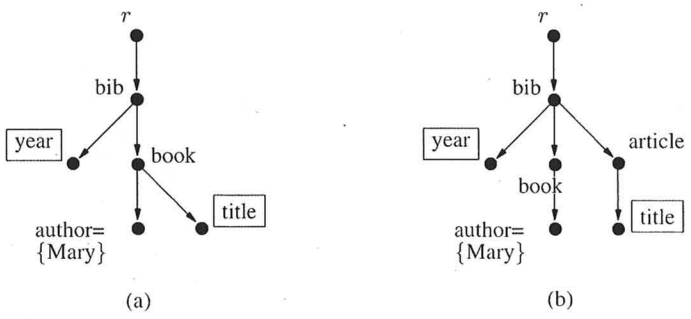


Figure 7.7 Two CTPQs for Q_1 on G: (a) U_1, and (b) U_3

Clearly, a CTPQ can be seen as a PTPQ (without variables and descendant or descendant-or-self precedence relationships) where the node sharing expressions are defined by the common nodes of different root-to-leaf paths. The output PPs of the corresponding PTPQ are defined by the paths of the CTPQ that comprise output nodes. Then, we can define the *answer* of a CTPQ to be to the answer of the corresponding PTPQ. We can now show the following proposition.

Proposition 7.2.1 *Let Q be a PTPQ, G be an index graph, and U_1, ..., U_k, $k \geq 1$, be all the CTPQs of Q on G. Let also A, A_1, ..., A_k be the answer sets of Q, U_1, ..., U_k, respectively, on an XML tree underlying G. Then $A = \cup_{i \in [1,k]} A_i$.* □

Therefore, the answers of a PTPQ Q on an XML tree T can be computed by determining the set $\mathcal{U}$ of all the CTPQs of Q on the index graph that underlies T and by computing the answers of each CTPQ in $\mathcal{U}$ on T.

Consider the the XML tree T (Figure 7.1) and its index graph G (Figure 7.6). Consider also the PTPQ Q_1 (Figure 4), and its CTPQs, U_1 and U_3, on G (Figure 7.7). One can see that the answer of Q_1 on T shown in Figure 7.4(a) is also an answer of CTPQ U_1. Similarly, the answer of Q_1 on T shown in Figure 7.5(c) is also an answer of CTPQ U_3.

Note that the approach presented in this paper can be easily extended to handle recursive XML trees. In this case, CTPQs for a PTPQ are generated using an index tree instead of an index graph. An index tree is a tree structure similar to an *1-index*[1] [63] with the exception that no pointers to the data are stored in the index. The absence of cycles in the index allows one to deal with the presence of multiple nodes labeled by the same element in the same PSP of a PTPQ.

7.3 Using Complete TPQs to Exclude Meaningless Answers

In this section, we assign semantics to our PTPQ language that returns meaningful answers. In contrast to previous approaches which exclude embeddings of the query to the data tree [12, 13, 5, 6], our approach excludes CTPQs of a PTPQ. In this sense, our approach relies both on data and on structural patterns of data, instead of relying exclusively on data.

Based on the results of the previous section, we consider that, given an XML tree T (and its index graph G), the answer of a PTPQ is the union of the answers of its CTPQs on G. However, some of these CTPQs may return meaningless answers. Consider, for instance, again, the PTPQ Q_1 (Figure 7.2) and the XML tree of Figure 7.1 along with its index graph G in Figure 7.6. The CTPQ U_3 (Figure 7.7(b)) of Q_1 on G returns (among others) the answer of Figure 7.5(c) which is meaningless. Therefore, this CTPQ of Q_1 should not be used for computing the answers of Q_1. Analogously to query answers, we

[1] 1-indexes coincide with *strong DataGuides* when the data is a tree.

characterize a CTPQ of PTPQ Q on G as *meaningful* with respect to T if it returns a *meaningful* answer on T. Otherwise, it is characterized as meaningless with respect to T. In order to formally define meaningful CTPQs we need to introduce a transformation for CTPQs.

7.3.1 A Transformation for Complete TPQs

Let Q be a PTPQ, T be an XML tree and G be its index graph. Figure 7.8 shows two CTPQs, U and U', of a PTPQ Q on an index graph G. CTPQ U comprises three subtrees

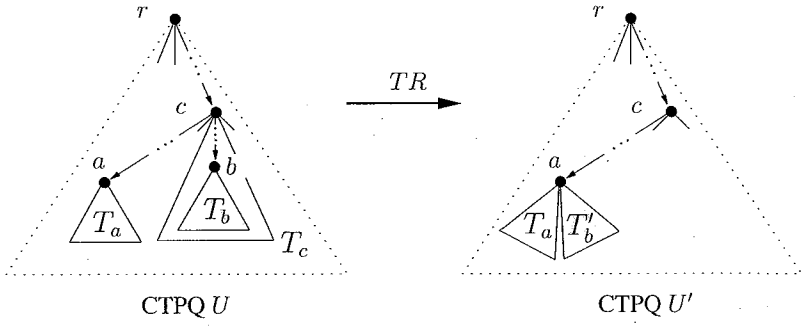


Figure 7.8 Transformation TR transforms the CTPQ U to the CTPQ U'

T_a, T_b and T_c. T_a is the subtree of U rooted at the node labeled by a, T_c is a subtree of U rooted at the node labeled by c, and T_b is the subtree of T_c rooted at a node labeled by b. Subtrees T_a and T_b can be empty (that is, they can trivially contain only their root node a and b respectively). The node labeled by c can coincide with the root of U. However, the node labeled by a cannot coincide with the node labeled by c, and the node labeled by b cannot coincide with the node labeled by c (that is, the node labeled by c is an ancestor of the nodes labeled by a and b). Labels a and b can be equal. Subtree T'_b in U' is a tree identical to T_b except that its root is labeled by a instead of b. CTPQ U' can be obtained from U by removing the subtree T_c below the node labeled by c, and by making T'_b a subtree

of the node labeled by a. We define the *transformation* TR on CTPQs as a transformation that transforms a CTPQ of the form of U into a CTPQ of the form of U'. Notice that CTPQ U' has at least one node less than CTPQ U.

We formally define meaningful CTPQs in the next subsection but we provide some intuition now on the transformation TR. Consider a CTPQ U' resulting by applying TR to a CTPQ U. Our intention is to characterize U as meaningless with respect to T, and to exclude it from consideration in computing the answers of Q, if U' returns an answer on T. To understand this idea, observe that there is a 1-1 mapping f from the nodes of U' to the nodes of U that respects node labels and child precedence relationships (with the exception of the child precedence relationships from the node labeled by a in T'_b). Then, the following proposition holds:

Proposition 7.3.1 *Assume that CTPQ U' results by applying transformation TR to a CTPQ U. If n' is the lowest common ancestor (LCA) of the nodes $n'_1, \ldots n'_k$ in U', and n is the LCA of the nodes $f(n'_1), \ldots, f(n'_k)$ in U then n is not a descendant of $f(n')$ in U.* □

Since, there is an image of Q under an emdedding to T (and therefore an answer of Q on T) that closely relates the nodes as determined by U', any image of Q under an emdedding to T (and the corresponding answer) that relates the nodes in the looser way determined by U is not meaningful, and should be excluded from generating an answer.

To clarify the use of transformation TR, we show next some applications of it on the CTPQs of our running example. We consider PTPQ Q_1 (Figure 7.2) on index graph G (Figure 7.6) that underlies the XML tree T (Figure 7.1). Figure 7.9 shows three CTPQs U_1, U_2 and U_3 of Q_1 on G. Dotted lines denote the subtrees T_a, T_b, and T_c of transformation

TR as they are graphically shown in Figure 7.8. The CTPQ U_2 will be excluded from consideration in the evaluation of Q_1 on T because TPQ U_1 returns an answer on T.

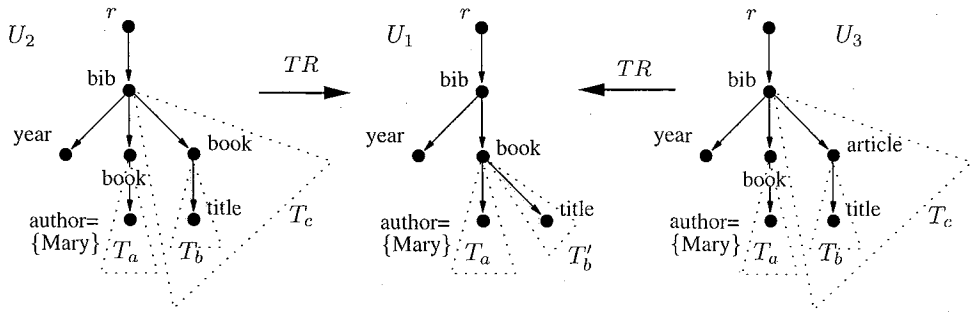


Figure 7.9 CTPQs for Q_1: U_2 and U_3 are meaningless

Similarly to U_2, CTPQ U_3 will be excluded from consideration. Notice that in the case of CTPQ U_2, the roots of T_a and T_b are labeled by the same element "book", while in the case of CTPQ U_3 they are labeled by different elements "book" and "article".

Figure 7.10 show applications of transformation TR in sequence. The CTPQ U_6 is excluded from consideration because of the CTPQ U_5. Then, the CTPQ U_5 is also excluded because of the CTPQ U_4.

Finally, Figure 7.11 shows some other applications of transformation TR in sequence. Notice that T_b in Figure 7.11(c) (and consequently T'_b in Figure 7.11(d)) are empty. Observe that the CTPQ U_8 has an extra branch from the root with respect to CTPQ U_7.

7.3.2 Determining the Meaningful Complete TPQs

Next, we formally define the concept of meaningful CTPQ of a PTPQ on an index graph. Consider a PTPQ Q, an XML tree T, and its index graph G. Let $\mathcal{U}$ be the set of CTPQs of Q on G. We define a binary relation $\prec$ on $\mathcal{U}$ as follows: for every $U, U' \in \mathcal{U}$, $U' \prec U$ if and

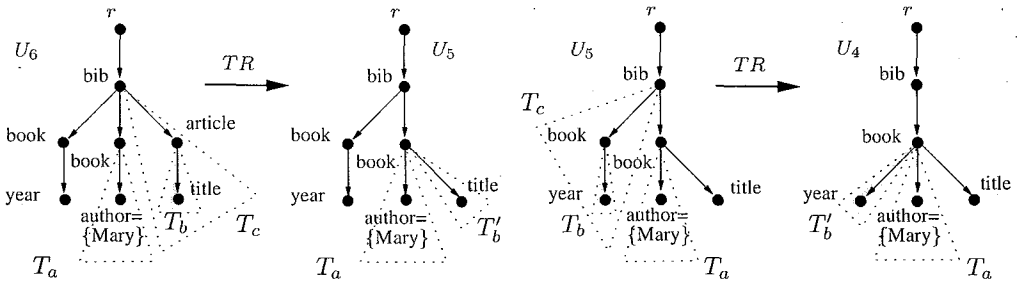


Figure 7.10 CTPQs for Q_1: U_6 and U_5 are meaningless

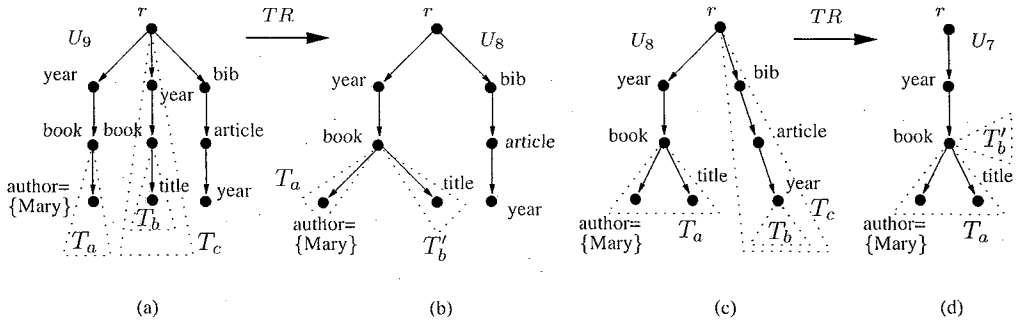


Figure 7.11 CTPQs for Q_1: U_9 and U_8 are meaningless

only if U' can be obtained by applying a sequence of transformations TR to U. Clearly, $\prec$ is a strict partial order.

Definition 7.3.1 *A CTPQ $U \in \mathcal{U}$ is called* meaningless *with respect to T if there is another CTPQ $U' \in \mathcal{U}$ such that (a) $U' \prec U$, and (b) U' has an answer on T. Otherwise, it is called* meaningful *with respect to T.* □

We can now update the definition of the answer set of a PTPQ given in Section 7.1. We provide a new definition for the answer set of a PTPQ so that it comprises only answers of meaningful CTPQs. The new definition is based on Proposition 7.2.1 and Definition 7.3.1.

Definition 7.3.2 *Let Q be a PTPQ, T be an XML tree and G be an index graph. Let also $U_1, \ldots, U_k$, $k \geq 1$, be the* meaningful *CTPQs of Q on G with respect to T. If*

A, A_1, ..., A_k are the answer sets of Q, U_1, ..., U_k, respectively, on T, then $A = \cup_{i\in[1,k]}A_i$. □

Consider the CTPQ U_3 shown in Figure 7.7(b). As mentioned in Section 7.3.1, U_3, evaluated on the XML tree T of Figure 7.1, returns the meaningless answer of Figure 7.5(c). CTPQ U_3 is also shown in Figure 7.9 and it is characterized by Definition 7.3.1 as meaningless. Therefore, it will not be used to generate answers for the PTPQ Q_1 (Figure 7.2) on T. In contrast, CTPQ U_1 of Figure 7.7(a) returns only the meaninful answer of Figure 7.4(a). CTPQ U_1, is also shown in Figure 7.9. One can se that Transformation TR cannot be applied to U_1. Therefore, it is correctly characterized by Definition 7.3.1 as meaninful, and will be used to generate answers for the PTPQ Q_1 on T. One can check that when it comes to evaluate Q_1 on T, Transformation TR excludes all CTPQs for Q_1 on G (Figure 7.6) except the CTPQs U_1, U_4 and U_7 of Figures 7.9, 7.10, 7.11, respectively, and the variations of those CTPQs where label "book" is replaced by "article".

Since the meaningful CTPQs are TPQs, their evaluation can be implemented on top of an XQuery engine and benefit from the extensive optimization techniques that have been developed up to now for XQuery [47, 19, 20].

7.4 Comparison with Previous Approaches

In this section, we compare the semantics of our query language with the semantics of three other well known query languages for XML that aim at excluding meaningless answers [12, 13, 5, 6]. In most practical cases, the information in the XML tree is incomplete (e.g. optional elements/values in the schema of the document are missing), or irregular (e.g.

different structural patterns coexist in the same document). Therefore, we also take this parameter into account in our comparison.

Schmidt et al. [12] suggest the *meet* operator to let the users query an XML document without knowledge of the elements and the structure. Queries are sets of keywords to be matched against the values of the XML document. This approach exploits the structure of the XML tree and is based on merely computing the Lowest Common Ancestor (LCA) of the nodes in the XML tree that match the keywords. The computation of the LCAs is done bottom up. When the LCA of a set of nodes that match the keywords is computed, these nodes are excluded from further consideration. The meet operator might fail to return a meaningful answer when a node is a descendant of another node of similar type (logical hierarchy) and the information in the XML tree is not complete.

Consider, for instance, the XML tree of Figure 7.12 and a keyword query consisting of the keywords "Mary", "title" and "year". This approach considers only keywords that are values but we allow also keywords that are elements in this example as this does not affect the computation of LCAs. The *meet* operator will fail to return the subtree rooted at the node labeled by *bib* which is the meaningful answer. The reason is that another LCA node is identified first (the node labeled by "book") and the subtree rooted at this node is excluded from further consideration. The meet operation will also fail to exclude meaningless answers in case of incomplete information even for a flat XML tree (that is, a tree that does not contain logical hierarchies). Consider, for instance, the same keyword query and the XML tree T_1 of Figure 7.1. The meet operator will return the meaningless answer shown in Figure 7.5(b).

XSEarch [13] is a semantic search engine for XML. It uses a simple query language that allows keyword specifications (values and/or elements) and a primitive structural

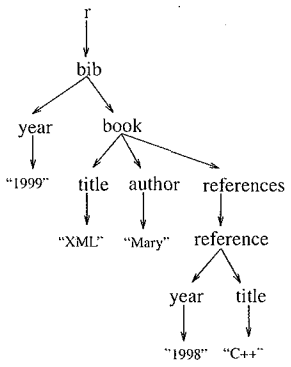


Figure 7.12 XML Tree

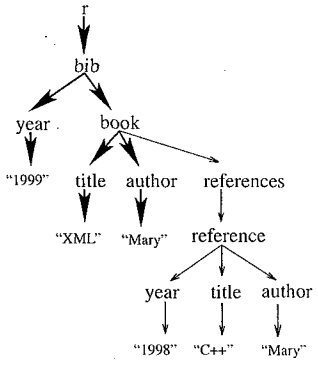


Figure 7.13 XML Tree

restriction (a node labeled by an element keyword has a descendant node labeled by a value keyword). The answers are subtrees that contain the keyword labeled nodes. XSEarch uses the concept of *Interconnection Relationship* to capture the meaningful XML subtree for a set of nodes that match the keywords. Two nodes n_1 and n_2 are interconnected if the subtree rooted at their LCA does not contain two nodes labeled by the same element. Nodes n_1 and n_2 can have the same label though. The interconnection relationship is extended to multiple nodes through an *all-pair* or a *star* n-ary relationship. XSEarch allows only query answers where the nodes matched by the keywords are all-pair or star related. Intuitively, nodes in the XML tree represent entities and element labels represent their type. Nodes with the same label represent entities of the same type. Descendant nodes of a node n are assumed to belong to the entity n represents. Two nodes that are meaningfully related should not belong to different entities of the same type. XSEarch difficultly fails to return a meaningful answer.

However, XSEarch usually fails to exclude meaningless answers even if the XML tree is flat, and does not have incomplete information. Consider the keyword query Q_1 specifying the elements "title" and "year" and the value "Mary" on the XML tree of Figure

7.1. XSEarch fails to exclude the meaningless answer of Figure 7.5(c). The reason is that any two of the nodes that match the keywords are interconnected (in contrast, XSEarch succeeds in excluding the meaningless answer of 7.5(a) because this one contains two nodes which are both elabeled by "bib"). Similarly, query Q_1 issued against the XML tree of Figure 7.12 which has a logical hierarchy fails to exclude the several meaningless subtrees-answers rooted at the node labeled by the "book".

Li et al. [5, 6] extend XQuery to enable users to query XML documents without full knowledge of the structure. This work is closer to ours compared to the previous two because it allows the user to specify extensive structural restrictions in a query besides keywords. To compute a query, this approach finds the LCA node of the set of nodes that match the keywords, and treats the subtree rooted at this node as the context for query evaluation. It employs a particular version of LCA, called Meaningful Lowest Common Ancestor Structure (MLCAS). The MLCAS of two nodes n_1 and n_2 (and therefore that of any superset of those two nodes) does not exist if two other nodes of the same type (that is, nodes labeled by the same element) have an LCA which is a descendant of the LCA of n_1 and n_2. The MLCAS approach fails to return meaningful answers when the XML data contains logical hierarchies even if there is no incomplete data in it. Consider, for instance, query Q_1 specifying the elements "title" and "year" and the value "Mary" on the XML tree of Figure 7.13. Under the MLCAS semantics, the answer set of the query does not contain the subtree-answer rooted at the node labeled by element "bib" (shown with bold arrows in Figure 7.13), which, intuitively, is the answer the "most related" to query Q_1. The answer set will contain only the subtree-answer rooted at the node labeled by "reference". When the XML data is incomplete, the MLCAS approach fails to exclude meaningless answers.

For instance, the keyword query Q_1 specifying the elements "title" and "year" and the value "Mary" on the XML tree of Figure 7.1 will return the meaningless answer of Figure 7.5(b) because there are articles in the XML tree T that have only a title (and no author) and articles that have an author (and no title). Another drawback of this approach is that the semantics for the keyword queries (MLCAS) is different than the semantics for the structural queries (XQuery). Therefore, the structural restrictions cannot be taken into account in determining the meaningful answers of a query in the first place. If a meaningful answer of a query is not contained in the subtrees returned by the keyword search part of the query, it cannot by recovered by further evaluating the structural part of the query. Notice that, in contrast, in our approach both the structural restrictions and the keywords in a query determine the meaningful TPQs that, in turn, compute the answers of a query.

Our approach successfully returns all the meaningful and eliminates all the meaningless answers of the examples discussed in this section. Its success is due to the original way it uses to evaluate the meaningful answers of a query. Previous approaches identify meaningful answers by operating *locally* on the *data* by computing LCAs of nodes in the XML tree. In contrast, our approach operates *globally* on *structural summaries* of data (index graphs) to compute meaningful TPQs. This overview of data gives an advantage to our approach compared to previous ones.

7.5 Experimental Evaluation

We implemented our approach (abbreviated as PTPQ), and we experimentally compared it to previous approaches on two aspects: the quality of the returned results, and the efficiency of their computation.

7.5.1 Quality

In order to asses the quality of our approach, we implemented the three approaches discussed in Section 7.4 (Meet [12], XSEarch [13], and MLCAS [5, 6]). We ran detailed experiments to compare their *Recall* (defined as the proportion of relevant materials retrieved) and *Precison* (defined as the proportion of retrieved materials that are relevant). These parameters have been used for years for measuring the quality of keyword search in information retrieval systems. As our language is a database language, we did not use ranking functions combined with threshold values to trade recall for precision (or vice versa).

Experimental Setting We used real-world DBLP data of the size of 324 MB collected from http://dblp.uni-trier.de/xml/ in May 2006. To simplify the document for the experiments, we retained only three publication types: "book", "article", and "inproceedings". For each publication type, we retained only the properties "title", "authors", and ''year". As the original DBLP data is flat, for evaluation purposes, we restructured it into three types of data sets that comply respectively with the schemas shown compactly as dags in Figure 7.14. All subelements of publications in the three schemas are optional as in the original

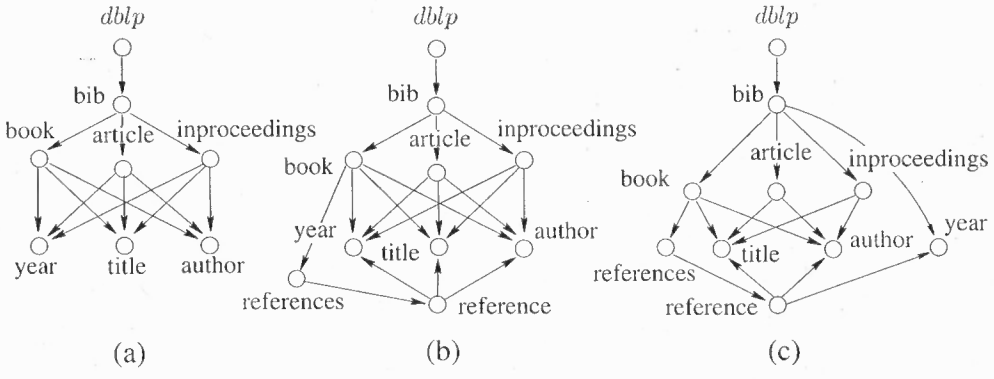


Figure 7.14 Three schemas for the DBLP data: (a) Type 1, (b) Type 2, (3) Type 3

DBLP schema. Publications of the schema Type 1 do not have references. Publications

of schemas Type 2 or Type 3 may have references. We consider also references to be publications. Therefore, schemas Type 2 and 3 contain logical hierarchies. One difference between schema Type 2 and Type 3 is that publications of schema Type 3 are categorized by year.

Besides the structure of the document, the "incompletness" of the data also affects the effectiveness of the keyword based searches. We define a publication in the data set as *complete* if it has all the subelements "title", "year", and "author", otherwise it is *incomplete*. For the experiments, we considered data sets that have different percentages of incomplete publications.

The data sets for the three schema types are generated as follows. A program loads a set of sampled "book", "article", and "inproceedings" elements each with three subelements "author", "title", and "year" from the original DBLP data. Another program randomly chooses a set of publications among them for removal of some of their subelements. One or at most two subelements can be removed from each publication. The percentage of incomplete publications for different publication types can be specified through input parameters. Finally, an XML creator reassembles the publications to an XML document. The structure of the generated XML file, determined also by an input parameter, can be any one of the three types shown in Figure 7.14.

As the distinction between keywords that are values and keywords that are elements is insignificant for the semantics of the queries in all approaches, in our experiments, we query only for elements. We used keyword queries that comprise at least two of the elemenst "title", "year", and "author". We also ran experiments on queries with more than three keywords and the results were similar. We also considered two of the previous keyword queries with structural restrictions. We used these queries to experimentally compare only

our approach (PTPQ) and the MLCAS approach since these are the only two that allow the specification of non-primitive structural restrictions.

For all three LCA-based approaches, we consider that the answer of a query is the subtree whose leaves are the nodes that match the keywords and whose root is their LCA (the way it is defined in each approach). Thus two distinct matchings of the keyords with the same LCA determine two different answers. For each query and each type of data set, we wrote a fully specified query in XQuery that expresses what the user is seeking. We used the answers of these queries as a reference for computing precision and recall.

For each query and each type of data set, we have run the four approaches on six XML documents with increasing percentage of incomplete publications in the range from 0 (all the publications are complete) to 50% (half the publications are incomplete).

We ran the experiments on a Pentium 2.40GHz computer with 512MB of RAM running Windows XP Professional. We implemented all keyword search techniques in Java and used the SAX API of the Xerces Java Parser for the parsing of XML files. Berkeley DB XML 2.2.13 was used to store XML files and run XQuery.

7.5.2 Experimental Results for Keyword queries with or without structural restrictions

Keyword queries without structural restrictions We first consider keyword queries without structural restrictions. Figure 7.15 shows precision and recall of the two keyword query {author, year} for the three types of documents varying the percentage of incomplete publications in the documents.

Both XSEarch and PTPQ have perfect recall on all types of documents both for complete and incomplete data. Meet has also perfect recall on Type 1 and 2 documents but performs slightly worse on Type 3 documents when the data is not complete. MLCAS

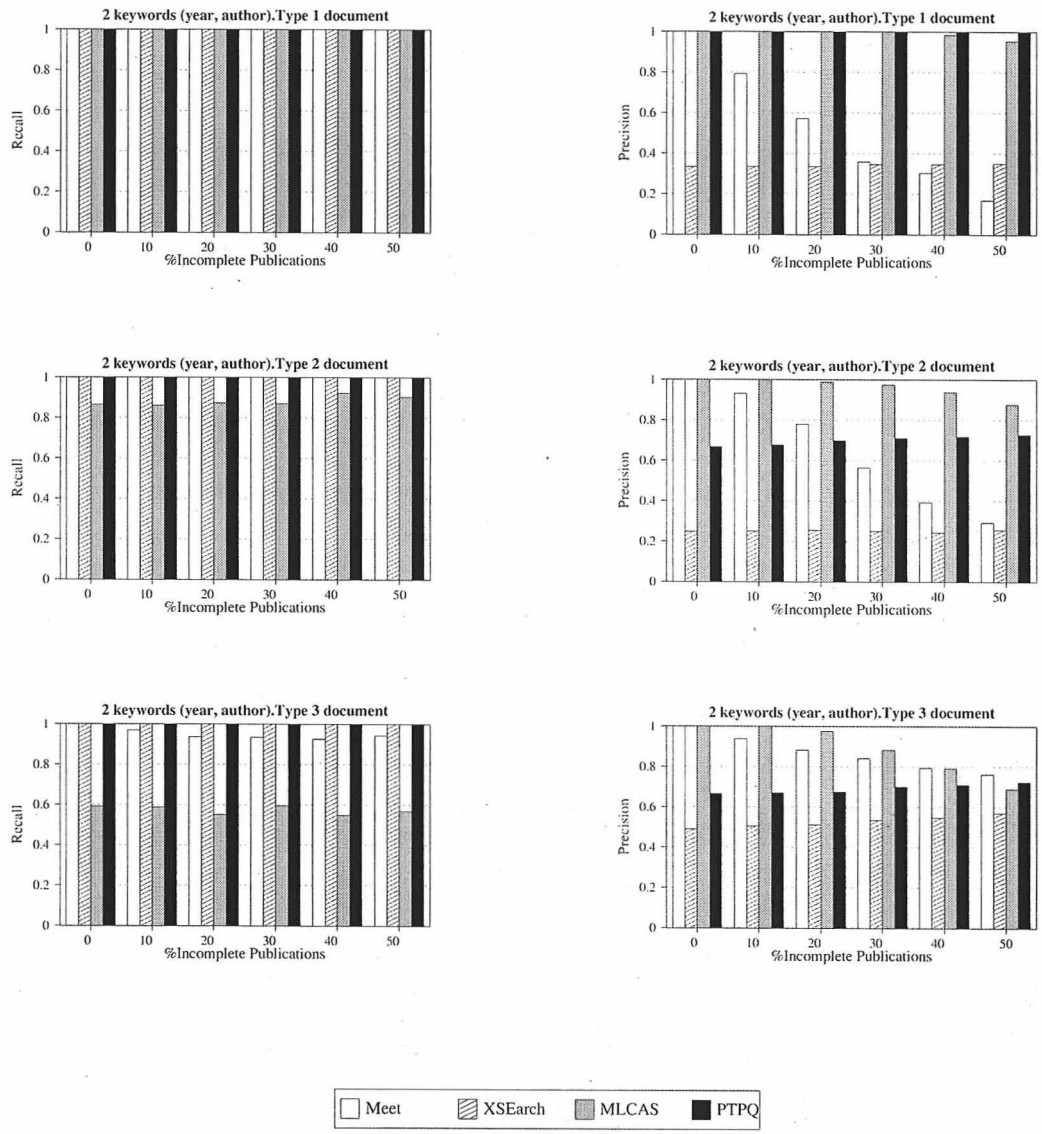


Figure 7.15 Recall and Precision for the two-keyword query {author, year}

has also perfect recall on Type 1 documents. In contrast, its recall is degraded on Type 2 documents and it drops below 60% on Type 3 documents both for complete and incomplete data. This is due to the fact MLCAS cannot handle the logical hierarchies appearing in Type 2 and 3 documents.

PTPQ shows perfect precision on Type 1 document (no logical hierarchies). Its precision starts above 60% for complete data and goes slightly up as the percentage of incomplete publications increases on Type 2 and 3 documents. The opposite trend is followed by Meet and MLCAS on all types of documents. They start at 100% with complete data and drop as the percentage of incomplete publications increases. The precision of XSEarch is, in general, low and is not affected significanyly significantly by the increase of the percentage of incomplete information.

Figure 7.16 shows the precision and recall of the two-keyword query {title, author} for the three types of documents varying the percentage of incomplete publications in the documents. We omit the plots of the query {title, year} as they are analogous to those of the query {title, author}.

All four approaches show in Figure 7.16 similar trends to those shown in Figure 7.15. Meet and XSErch show on the average even lower precision. Their recall is perfect for all types of documents both for complete and incomplete data. Interestingly, the recall of MLCAS improves when the percentage of incomplete publications increases, reaching 100% for a percentage of 50% of incomplete publications. The reason is that when the number of incomplete publications increases, a number of "book", "article", and "inproceedings" elements (which are missed anyway by the MLCAS approach) are not anymore correct answers.

Figure 7.17 shows the precision and recall of the three-keyword query {title, author, year} for the three types of documents varying the percentage of incomplete publications in the documents. The trends are similar to those of two-keyword queries with a slight degradation of the recall of Meet, and an average degradation of the precision of XSEarch and Meet.

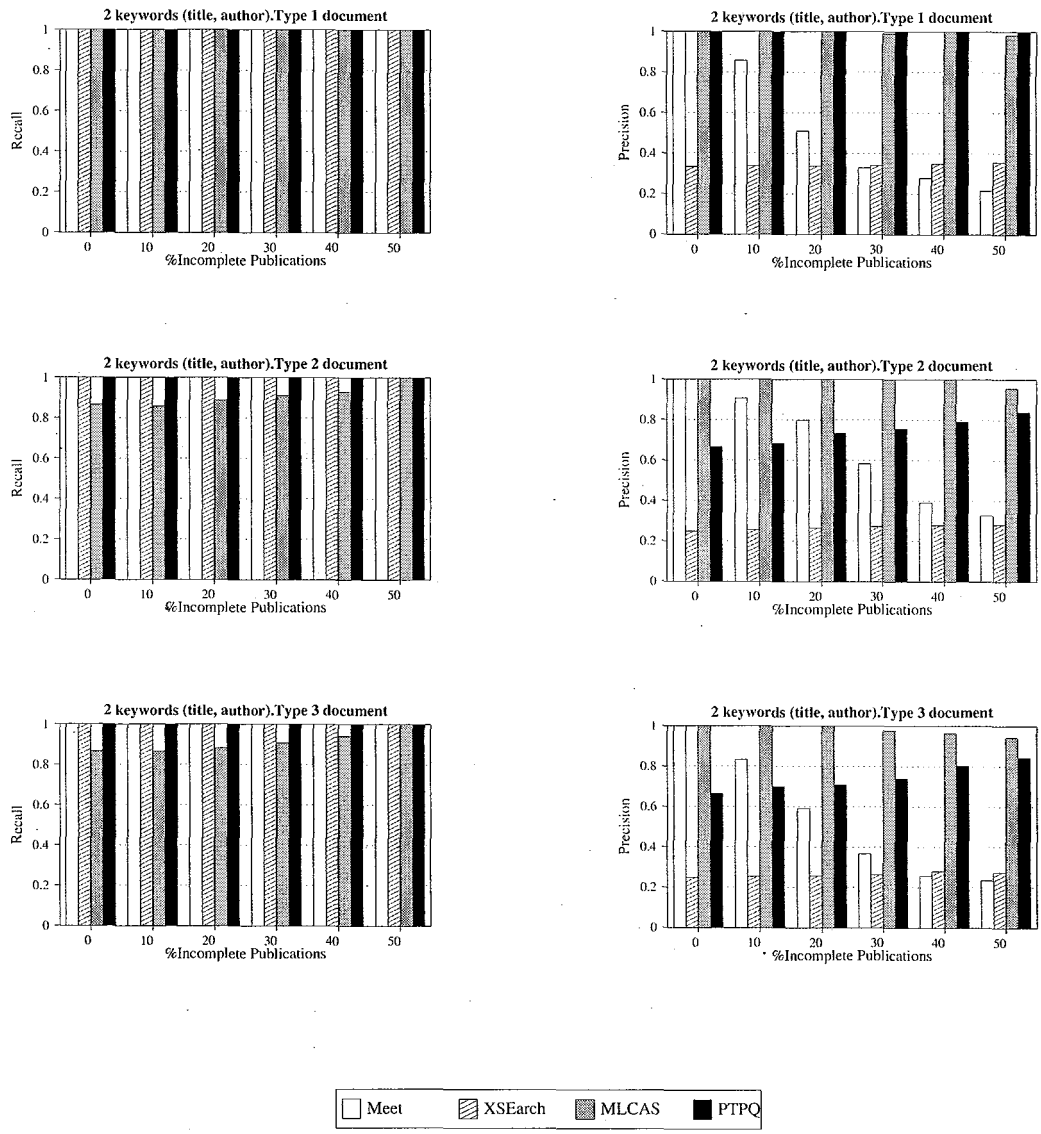


Figure 7.16 Recall and Precision for the two-keyword query {title, author}

Experimental Results for Keyword Queries with Structural Restrictions We now consider queries that involve also structural restrictions. We use two of the previous keyword queries where the "author" and "title" keyword elements are both child nodes of some (the same) element. This structural restriction can be formulated on the keyword queries {author, title}, and {title, author, year}. We call the first one Q_{s2} and the second one Q_{s3}.

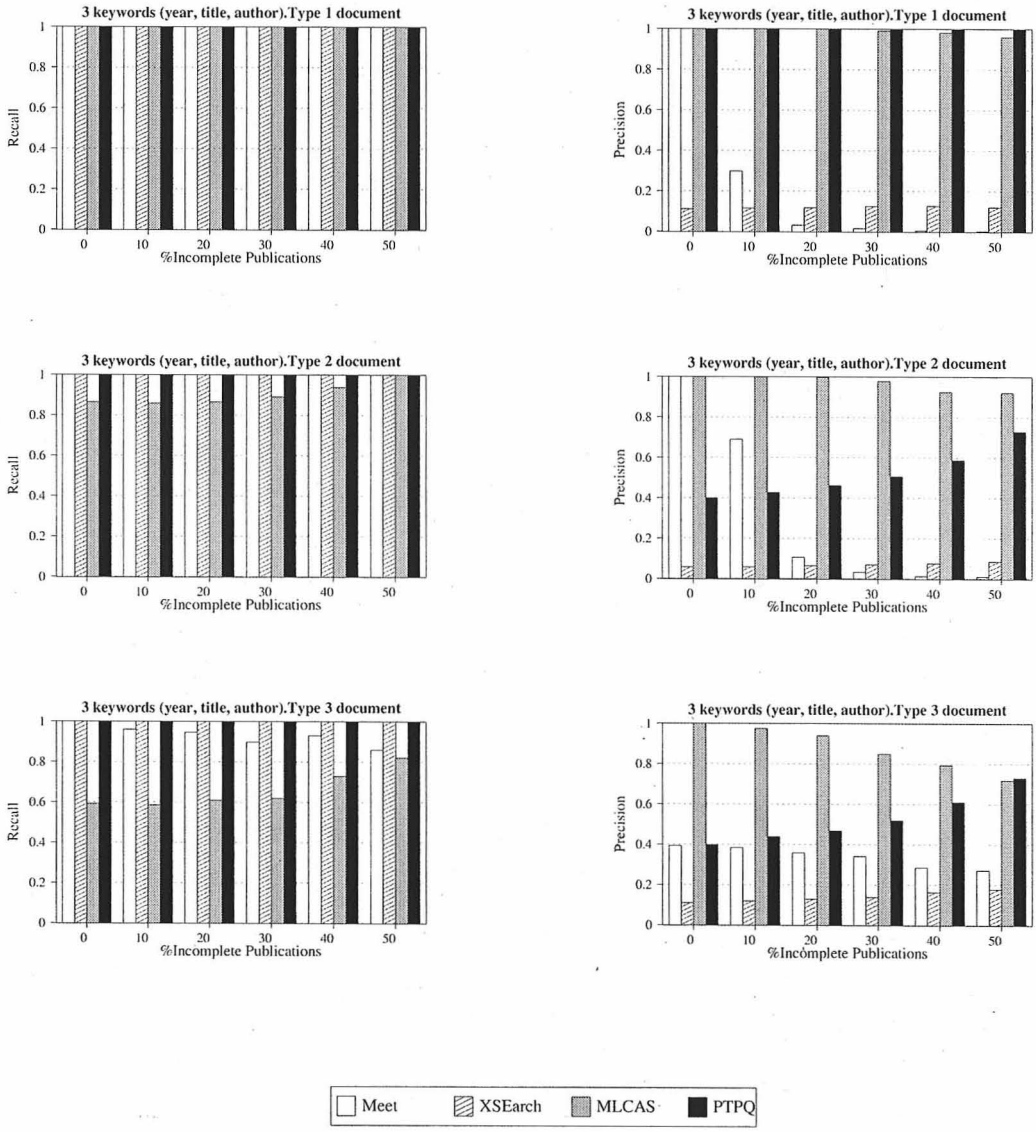


Figure 7.17 Recall and Precision for the three-keyword query {title, author, year}

Figure 7.18 shows the precision and recall of query Q_{s2} for Type 3 documents varying the percentage of incomplete publications in the document (for Type 1 document the precision and recall are perfect for both approaches). Both approaches show the same recall as for the corresponding query without structural restrictions, which for the PTPQ approach is 100%. Both approaches improve their precison achieving a perfect one.

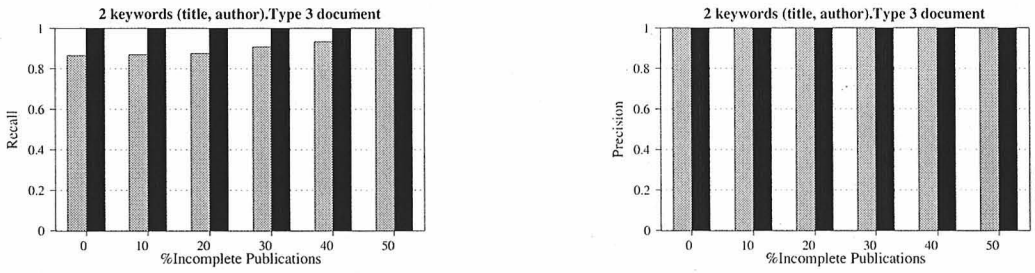


Figure 7.18 Recall and Precision for the query Q_{s2} (two keywords and structural restrictions)

Figure 7.19 shows the precision and recall of query Q_{s3} for Types 2 and 3 documents varying the percentage of incomplete publications in the document. The PTPQ approach has perfect recall. The MLCAS approach has the same recall as for the corresponding query without structural restrictions. Both approaches improve their precision but the improvement is more important for PTPQ.

In summary, MLCA shows good precision which can be improved with structural restrictions. However, its recall is low (it falls below 60% in some cases). Its recall cannot be improved when additional structural restrictrions are imposed since the semantics for the keyword part of the query is different than that of the structural part of the query. Therefore, answers missed in the evaluation of the keyword search part of the query cannot be recovered during the evaluation of the structural part. PTPQ does does not show this drawback. Its recall is perfect with and without structural restrictions, while additional structural restrictions improve its recall.

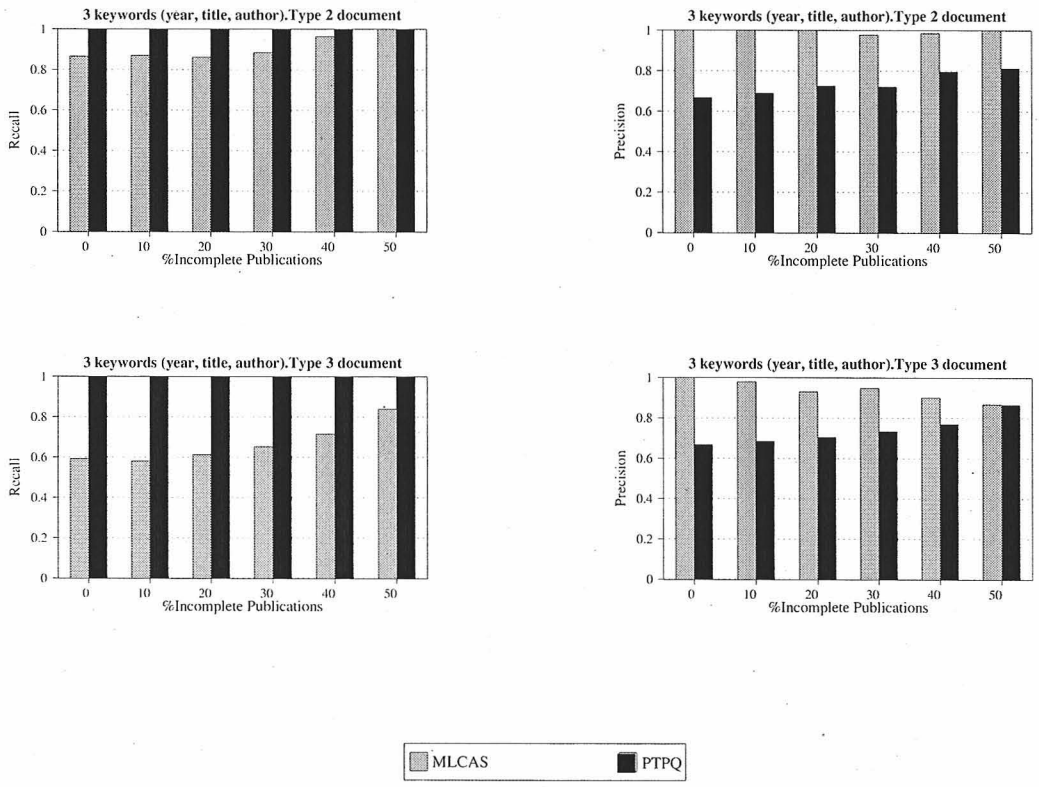


Figure 7.19 Recall and Precision for the query Q_{s3} (three keywords and structural restrictions)

7.5.3 Performance

In order to assess the performance of our approach, we compared it to the MLCAS approach which also allows the specification of structural constraints. In addition, the MLCAS approach is embedded into Timber [47] an XML database management system. In order to guarantee a unique experimental comparison environment, we used Timber also for the evaluation of the meaningful TPQs of our approach. In this section we present the experiments conducted to evalute the performances of MTPQ approach and MLCAS approach on generating meaningful answers and report results obtained.

Experimental Setting We compare the time cost of evaluating an MLCAS-embedded XQuery, with that of MTPQ approach which generates a set of meaningful TPQs and then evaluate their corresponding XQuery.

In the query quality experiments, we have chosen to generate synthetic DBLP datasets so that we can better control the relationship between the algorithms and the characteristics of the datasets.

We used original DBLP datasets. We retained only the properties of "year", "author", "title", "publisher", and "ISBN" for each publication as these are the ones that we used in queries, and removed other properties such as "volume" and "pages". In the experiments we used five different sizes of datasets: 95kb, 21mb, 29mb, 95mb and 148mb. We used the same computer as for the quality experiments.

We used following four queries in the experiments:

1. Query 1 (Q_1): Find titles of all the WWW publications (pure 2-keyword query: www and title).

2. Query 2 (Q_2): Find the title and year publications that have ISBN (2-keyword query with structural constraints: ISBN is the child of the publication; this constraint is specified outside the MLCA function, i.e., it is specified in the body of XQuery)

3. Query 3 (Q_3): Find titles of all the articles and their publication year (2-keyword query with structural constraints: title is under the article; this constraint is specified within the MLCA function)

4. Query 4 (Q_4): Find the publications of inproceedings (1-keyword query:inproceedings)

Due to some bugs in the current version of Timber (we contacted its author who confirmed this), we were unable to run MLCAS XQuery with more than two keywords for document size larger than 90kb. Therefore we didn't show the experimental results on three keywords queries and up.

For each approach, we ran the four queries on the five data sets. The running time of MTPQ consists of the time of generating meaningful TPQs and of evaluating TPQ(XQuery). The running time of an XQuery on Timber is measured in terms of its physical plan execution and does not include the time for query parsing and evaluation. Each query was run consecutively five times for each data set with hot caches. The average running time was used in the performance evaluation.

Fig. 7.20, Fig. 7.21, Fig. 7.22 and Fig. 7.23 respectively report the execution time of MLCAS approach and MTPQ approach for the queries on DBLP data. We can see that the time of generating meaningful TPQs is very small, only around 1% of the total evaluation time for the MTPQ approach; for very large data sets, such overhead can even be ignored. We can also see that the execution time of MLCAS approach is larger than that of MTPQ by orders of magnitude. For example, for Q1, it took MLCAS 431.743 seconds to generating 5729 results for the 148MB DBLP dataset; while MTPQ only used 3.23 seconds to generate the same results on the same dataset. Such a big difference is expected, as MLCAS works solely on data while MTPQ uses an index graph the size of which is much smaller than that of the underlying data. Moreover, we can see that the scalability of our approach is much better than that of MLCAS approach.

Our experiment results show that the MTPQ approach is superior to the MLCAS approach, both in query performance and search quality.

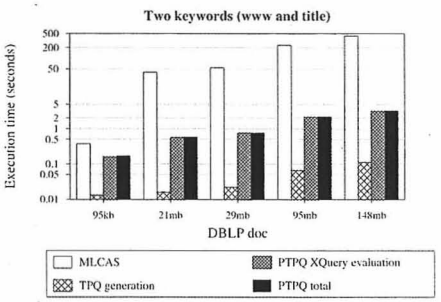


Figure 7.20 Performance Comparison for Q_1

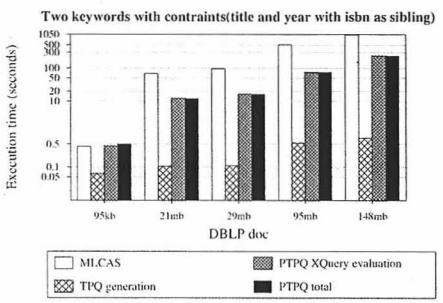


Figure 7.21 Performance Comparison for Q_2

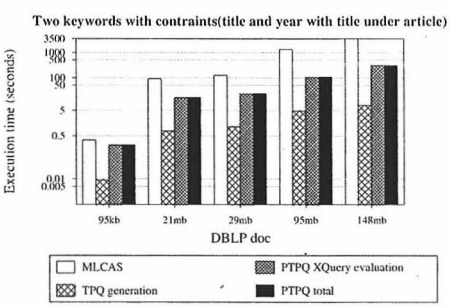


Figure 7.22 Performance Comparison for Q_3

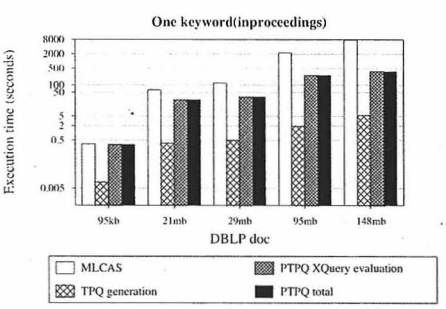


Figure 7.23 Performance Comparison for Q_4

CHAPTER 8

ANSWERING XML QUERIES USING MATERIALIZED VIEWS

In this chapter, the problem of answering XML queries using materialized views is addressed. The chapter is organized as follows. Section 8.1 presents the motivation for the studing problem. In Section 8.2, the data model, the class of queries and views considered, and the inverted lists evaluation model adopted are presented. The novel concept of view materialization is also introduced in this section. Necessary and sufficient conditions for tree-pattern query answerability are provided in Section 8.3. Section 8.4 presents a stack-based algorithm which compactly encodes in polynomial time and space all the homomorphisms from a view to a query. Experimental results are presented in Section 8.6.

8.1 Introduction

XML is by now the standard for exchanging, exporting and integrating data on the web. As increasing amounts of information are stored, exchanged, and exported using XML, it is becoming increasingly important to efficiently query XML data sources. Answering queries using views is a well-established technique in data integration, query caching and warehousing, where queries expressed over data sources are answered using materialized views defined over these data sources [75]. It is also (along with indexing) one of the best known techniques used for optimizing the evaluation of queries [76, 77]. The problem behind this technique can be formulated as follows: given a query and a set of materialized views along with their definitions, decide whether the query can be answered using the

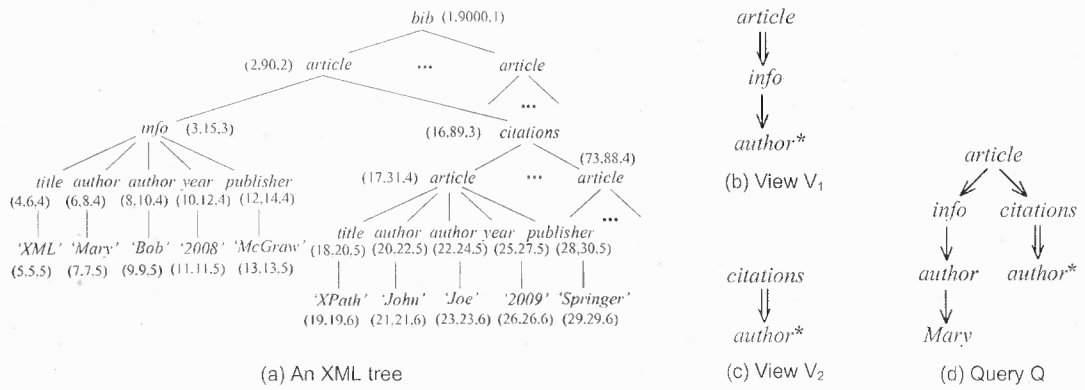


Figure 8.1 An XML tree with two views and one query on this tree

materialized views. If the answer is positive, usually there are alternative ways to compute the query from the materialized views inducing different evaluation costs. Consequently, another related problem consists in finding the best way to compute the query from the materialized views. These problems have been studied extensively in the realm of relational databases. However, there is a restricted number of contributions in that direction in the context of XML. The reason is the many limitations associated with the use of materialized views when a traditional way for evaluating queries on XML documents is adopted.

Limitations of Previous Approaches. The core of XPath consists of tree-pattern queries with one output node (TPQs). The answer of a TPQ is the set of subtrees rooted at the matches of the query output node against the XML document tree. The presence of an output node on queries and views, and the absence of complete structural information outside the subtrees in the view materializations, greatly reduces the chances of a query to have a hit of one or more views in the pool of materialized views that together can be used to answer the query. For this reason, some approaches suggest the materialization of additional information about the view answers, e.g. ancestor path information [49]. However, keeping this information only partially addesses the issue while increasing the size of data that needs to be stored. Storing, in addition, data values and references to

XML data [49] assumes a centralized enviroment and is not appropriate when the queries need to be answered using only the materialized views (that is, when the base XML data is not accessible). Further, the size of the answer subtrees can be very large. When multiple views are materialized (and inevitably ovelapping portions of the XML document are repeatedly and redundantly stored), view materialization becomes unfeasible due to space limitations. Even if space limitations are met, usually the view materializations are unindexed fragments of the XML document making the computation of a query more expensive compared to computing it against the original XML document. For this reason, in the performance studies of both [52] and [57] an upper bound has been set on the size of the XML fragment per view that can be materialized. This restriction limits both (a) the chances to answer the query using only the materialized views, and (b) the chances to find an efficient evaluation plan for the query using the materialized views. These obstacles defy the reason for materializing views in the first place.

Example 8.1.1 *Consider the XML tree of Figure 8.1(a) which records bibliographic information (ignore for the moment the triplets associated with the tree nodes). Let's assume that the view* $V_1 : //article//info/author$*, which retrieves article authors, and the view* $V_2 : //citations//author$*, which retrieves citing authors, are materialized in the client cache. Views* V_1 *and* V_2 *are shown as TPQs in Figures 8.1(b) and 8.1(c) respectively, where an asterisk denotes an output node. Suppose the user issues the query* $Q : //article[info/author = 'Mary']/citations//author$ *against the client cache. The query asks for the authors who cite articles authored by Mary and is shown as a TPQ in Figure 8.1(d). One can see that query* Q *cannot be answered using* V_1 *and/or* V_2*. The reason is that no structural information is available outside the view answer subtrees in the view materializations. Query* Q *cannot be answered using* V_1 *and/or* V_2 *even if ancestor path*

information is stored along with the subtrees in the view materializations because the absence of node identifiers does not allow a structural join on the materializations of the two views. Query Q can be answered using V_1 if node article is the output node of V_1. However, in this case, the materialization of V_1 is the whole base XML tree, and V_2 redundantly materializes part of it. Such a large materialization is likely prohibitive in the client cache, and if it is not, in the absence of an index on the materialization of V_1, it would probably be preferable to evaluate Q against the base XML data stored in the server instead of using the views materialized in the client cache.

The Inverted Lists Evaluation Model. A recent approach for evaluating queries on large persistent XML data assumes that the data is preprocessed and the position of every node in the XML tree is encoded [20, 21]. Further, the nodes are partitioned by node label, and an index of inverted lists is built on this partition. In order to evaluate a query, the nodes of the relevant inverted lists are read in the pre-order of their appearance in the XML tree. We refer to this evaluation model as *inverted lists* model. All the relevant query evaluation algorithms in this model are based on stacks that allow encoding an exponential number of pattern matches in a polynomial space. Comparison studies on XML query evaluation techniques [78, 79] show that holistic algorithms [20, 22, 21, 25, 26, 43] in the inverted lists model are superior to other algorithms and evaluation models (streaming/navigational approaches [34] or sequential/string matching approaches [80]). In this paper, we assume that the inverted lists model and holistic evaluation algorithms are adopted. Note that in the inverted lists model, the answer of a TPQ is not a subtree of the XML tree but a set of tuples. The fields of the tuples correspond to the query nodes. Each tuple contains the

(positional representation of) XML tree nodes that match the query nodes in an embedding of the query to the XML tree.

Problem Addressed. Driven by the prominence of the inverted lists evaluation model, we address the problem of answering TPQs using exclusively one or more materialized views in the context of this model. We also address the problem of the optimal evaluation of a TPQ using exclusively materialized views in the same context.

In this new context, query answerability by materialized views is not restricted by the presence of output nodes in queries and views since all query and view nodes can be seen as output nodes. As a consequence, queries have more chances to have a hit involving one or more materialized views in the view pool.

This new framework revises the "answering queries using materialized views" problem since previous conditions for query answerability are not valid anymore. Further, traditional approaches [49, 52, 53, 55, 57] evaluate queries by generating compensation TPQs over materialized views and look at the optimization of this evaluation as a problem of finding the lowest cost compensation TPQ. Unfortunately, these techniques are not applicable in the new context and novel stack-based techniques need to be devised for computing queries over view materializations.

Our Approach. We suggest a novel approach for materializing views where instead of materializing the view answer, we materialize sublists of the inverted lists for the labels of the view nodes. A query can be computed very efficiently using materialized views by running holistic stack-based algorithms over the inverted sublists of the view nodes.

Going back to Example 8.1.1, the triplets by the nodes of the XML tree of Figure 8.1(a) denote the positional representations of these nodes. As we show later, in the

context of our approach, not only the TPQ Q of Figure 8.1(d) can be answered using the materializations of views V_1 and V_2 of Figures 8.1(b) and 8.1(c), but also this computation can be performed very efficiently. Moreover, view materialization takes minimal space and any redundancy is avoided.

8.2 Data Model, Query Language, and Evaluation Model

In this section, we briefly present the data model, the class of queries and views we consider, and the inverted lists evaluation model we adopt. We also introduce our novel concept of view materialization.

Data Model. An XML database is commonly modeled by a tree structure. Tree nodes represent and are labeled by elements, attributes, or values. Tree edges represent element-subelement, element-attribute, and element-value relationships. For simplicity, we do not distinguish here between element, attribute, and value nodes, and we denote by $\mathcal{L}$ the set of node labels in the XML tree.

For XML trees, we adopt the region encoding widely used for XML query processing [20, 21]. This encoding associates every node with a triplet (*begin*, *end*, *level*). This triplet is called *positional representation* of the node. The *begin* and *end* values of a node are integers which can be determined through a depth-first traversal of the XML tree, by sequentially assigning numbers to the first and the last visit of the node. The *level* value represents the level of the node in the XML tree. The utility of the region encoding is that it allows efficiently checking structural relationships between two nodes in the XML tree.

For instance, given two nodes n_1 and n_2, n_1 is an ancestor of n_2 iff $n_1.begin < n_2.begin$, and $n_2.end < n_1.end$.

Query and View Language. For simplicity of presentation and in order to highlight the novel features of our approach, we consider that queries and views are tree-pattern queries (TPQs). We comment later on how our approach can be applied to broader classes of queries e.g. queries with reverse axes and wildcards.Contrary to all previous approaches on answering queries using views [49, 52, 55, 58], we do not impose any restriction on the output nodes. Queries and views can have any number of output nodes and this does not affect the usability of the views for the evaluation of the queries. For this reason, in our definition below we do not explicitly refer to output nodes, and all the nodes of queries and views are considered to be output nodes. Our approach applies without modification to the case where arbitrary sets of nodes in queries and views are considered to be output nodes.

A *tree-pattern query* (TPQ) specifies a pattern in the form of a tree. Every node in a TPQ Q has a label from $\mathcal{L}$. There are two types of edges in Q. A single (resp. double) edge between two nodes in Q denotes a child (resp. descendant) structural relationship between the two nodes.

The answer of a TPQ on an XML tree is a set of tuples. Each tuple consists of XML tree nodes that preserve the child and descendant relationships of the query.

More formally: an *embedding* of a TPQ Q into an XML tree T is a mapping M from the nodes of Q to nodes of T such that: (a) a node in Q labeled by a is mapped by M to a node of T labeled by a; (b) if there is a single (resp. double) edge between two nodes X and Y in Q, $M(Y)$ is a child (resp. descendant) of $M(X)$ in T.

We call *image* of Q under an embedding M a tuple that contains one field per node in Q, and the value of the field is the image of the node under M. Such a tuple is also called *solution* of Q on T. The *answer* of Q on T is the set of solutions of Q under all possible embeddings of Q to T.

A view is a named query. The class of views we consider is not restricted. Any kind of query can be a view.

Outline of the Inverted Lists Evaluation Model. In the inverted lists evaluation model, the data is preprocessed and the position of every node in the XML tree is encoded. For every label in the XML tree, an inverted list of the nodes with this label is produced. Given an XML tree T, we use L to denote its set of inverted lists and L_a to denote the inverted list in L for label a. List L_a contains the positional representation of the nodes labeled by a in T ordered by their *begin* field.

Let Q be a query. With every query node X in Q labeled by a, we associate the inverted list L_a in L. To access the nodes in L_a for X, we maintain a cursor C_X. Cursor C_X sequentially accesses the nodes in L_a starting with the first node.

With every query node X in Q, we also associate a stack S_X. At the beginning of the evaluation of a query, all stacks are empty. When the nodes in the inverted lists are accessed by the cursors, they are possibly stored in stacks. At any point in time, stack entries represent partial solutions of the query that can be extended to the solutions as the algorithm goes on.

In the following we ignore the XML tree T and we assume that the input for the evaluation of queries and views is the set of inverted lists L. When a query Q is evaluated

on L, if the cursor of a node X in Q iterates over the inverted list L_Y we say that node X is *computed on L using the list L_Y*.

View Materialization. We now define our novel concept of view materialization.

Definition 8.2.1 *Let V be a view, and L be a set of inverted lists. The* materialization $V(L)$ *of V on L is a set of sublists of the inverted lists in L–one for each view node in V. If X is a node in V labeled by a, L_X denotes its inverted list in $V(L)$ and it contains only those nodes of $L_a \in L$ that are images of X in a solution of V on L. Sublist L_X is called the* materialization of X in $V(L)$.

In this sense, the inverted lists in the materialization $V(L)$ contain only those nodes of the inverted lists in L that contribute to a solution of V on L.

Our approach for view materialization departs from all the previous approaches which consider materializing copies of XML tree fragments, typed values, ancestor paths, or references to the input XML tree [49, 52, 58, 55, 57]. Note that our approach is space efficient since the sublists can encode in linear space a number of solutions for the view which is exponential on the number of view nodes.

8.3 Answering Queries Using Views

Let Q be a query and X be a node in Q labeled by a. Recall that in order to evaluate Q on L, the cursor C_X of X iterates over the inverted list L_a in L. If there is a sublist, say L_X, of L_a such that Q can be computed on L by having C_X iterate over L_X instead of L_a, we say that node X *can be computed using L_X on L*. Let V be a view whose materialization on L is $V(L)$. The idea of our approach for answering Q using V on L is to identify nodes in Q that can be computed using the materializations of nodes in V for every L and use

their materializations in $V(L)$ for computing the answer of Q on L instead of using the corresponding inverted lists in L.

8.3.1 Answering a Query Using a Single View

We start by defining what answering a query using a view means in our context of view materialization.

Definition 8.3.1 *Let $V(L)$ be the materialization of a view V on a set of inverted lists L. A query Q* can be answered using *V if for a node X in Q there is a node Y in V with the same label as X, such that for every L, X can be computed using $L_Y \in V(L)$. In this case, we say that view node Y* covers *query node X, or that Y is a* covering node *of X.*

Let's assume that Q can be answered using V. If every node in Q is covered by a node in V, we say that Q can be answered completely *using V. Otherwise, we say that Q can be answered* partially *using V.*

When the answer of a query is computed using a view, a node of the query that is covered by a view node uses only the materialization of this view node. Since the materialization of the view node is a sublist of the inverted list for the node label, it is usually smaller than the inverted list. This reduces the cost for computing the answer of the query.

Deciding Whether a Query Can be Answered Using One View. In order to specify conditions for view usability, we need the concept of homomorphism between views and queries. A *homomorphism* from a view V to a query Q is a mapping that maps all the nodes of V to nodes with the same label in Q and preserves child and descendant relationships (preserving a descendant relationship means that it is mapped to a path of nodes).

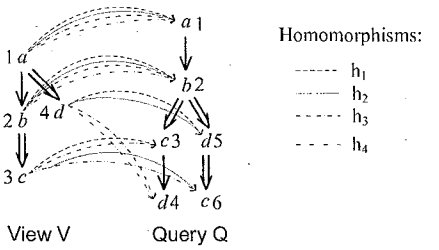


Figure 8.2 Four homomorphisms from view V to query Q

Figure 8.2 shows a query Q and a view V and four homomorphisms h_1, h_2, h_3 and h_4 from V to Q.

The following theorem relates node coverage to homomorphisms.

Theorem 8.3.1 *Let Q be a query and V be a view. A node X in Q is covered by a node Y in V iff there is a homomorphism from V to Q that maps Y to X.*

Necessary and sufficient conditions for view usability based on homomorphisms are provided by the next collorary of Theorem 8.3.1.

Corollary 8.3.1 *Let Q be a query and V be a view. Query Q can be answered using V iff there is a homomorphism from V to Q.*

For instance, in the example of Figure 8.2, query Q can be answered using view V since there is at least one homomorphism from V to Q. Both nodes labeled by d in Q are covered by node d in V.

Notice that our definition of homomorphism is less restrictive than previous ones, since we do not have to consider (and impose conditions on) output nodes [52, 53, 58]. This increases the chances for a homomorphism from a view to a query to exist. Based on Theoreom 8.3.1, it also increases the chances of the view to be useful in answering the query. This constitutes an important advantage of our approach compared to previous ones, since it allows the exploitation of views when other approaches fail.

In order to guarantee that a query can be answered completely using a view, we need to make sure that *every* node of the query has a covering node in the view. The next corollary of Theorem 8.3.1 expresses this requirement in terms of homomorphisms from the view to the query.

Corollary 8.3.2 *Let Q be a query and V be a view. Query Q can be answered completely using V iff there are homomorphisms from V to Q such that every node of Q is the image of a node in V under some homomorphism.*

Based on Corollary 8.3.2, one can easily see that in the example of Figure 8.2, query Q can be answered completely using view V.

Computing the Answer of a Query Using One View. In the traditional approach for answering a query using a view [49, 52, 53, 55, 57], the query is rewritten using the view. That is, in order to compute the answer of the query, a compensation query is determined which is applied to the materialized view and computes the answer of the query. This compensation query does so by navigating in the view materialization which is a set of subtrees of the original XML tree.

In contrast, in our approach, we use the view materialization and compute the query answer by running stack-based evaluation algorithms over the materializations of the covering view nodes.

Therefore, in order to perform the computation of the answer what is needed is an association of the query nodes with covering view nodes. The set of covering view nodes of a given query node is determined by the homomorphism of Theorem 8.3.1 as follows:

Let $h_1, \ldots, h_k$ be the homomorphisms from a view V to a query Q and $Y_i^1, \ldots, Y_i^{m_k}$ be the nodes in V whose image under h_i is X. Then, the set $m(X)$ of covering nodes for

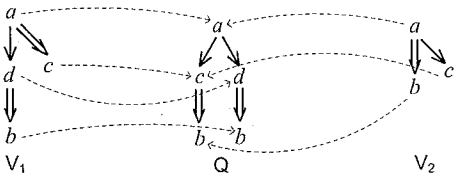


Figure 8.3 Query Q and views V_1 and V_2 and homomorphisms

X in V is

$$m(X) = \bigcup_{i \in [1,k],\, j \in [1,m_k]} \{Y_i^j\}$$

If $\exists X \in Q$, $m(X) \neq \emptyset$, Q can be answered using V. If $\forall X \in Q$, $m(X) \neq \emptyset$, Q can be answered completely using V. The materialization in $V(L)$ of any node in $m(X)$ can be used for computing X. However, we might also use the materializations of multiple (or all the) nodes in $m(X)$: let L_{X_1} and L_{X_2} be the materializations of two nodes X_1 and X_2 in $m(X)$. The *intersection* $L_{X_1} \cap L_{X_2}$ is the sublist of L_{X_1} and L_{X_2} which comprises the nodes that appear in both L_{X_1} and L_{X_2}. In order to compute the answer of Q using V any subset of $m(X)$ can be used: during the computation of the answer, X will be computed using the intersection of the materializations of the view nodes in this subset.

Note that a view V can have a number of homomorphisms to a query which is exponential in the number of view nodes. However, the number of covering nodes in $m(X)$ is bounded by the number of nodes in V.

8.3.2 Answering a Query Using Multiple Views

The presence of multiple views in the view pool increases the chances of a query to be answered using their materializations. We extend below our definition for answering a query using a view to multiple views. We first define the *union* of the materializations of two view nodes. Let X_1 and X_2 be two view nodes with the same label a, and L_{X_1} and L_{X_2}

be their materializations. The *union* $L_{X_1} \cup L_{X_2}$ of L_{X_1} and L_{X_2} is the sublist of L_a which comprises exactly the nodes of both L_{X_1} and L_{X_2}.

Definition 8.3.2 *Let $V_1(L), \ldots, V_n(L)$ be the materializations of views $V_1, \ldots, V_n$ on a set of inverted lists L. A query Q* can be answered using *$V_1, \ldots, V_n$ if for a node X in Q, there are nodes $Y_1, \ldots, Y_k$ in $V_1, \ldots, V_n$, such that, for every L, X can be computed using $L_{Y_1} \cup \ldots \cup L_{Y_k}$.*

Let's assume that Q can be answered using $V_1, \ldots, V_n$. If for every node X in Q, there are nodes $Y_1, \ldots, Y_k$ in $V_1, \ldots, V_n$, such that, for every L, X can be computed using $L_{Y_1} \cup \ldots \cup L_{Y_k}$ for every L, we say that Q can be answered completely *using $V_1, \ldots, V_n$. Otherwise, we say that Q can be answered* partially *using $V_1, \ldots, V_n$.*

Deciding Whether a Query Can be Answered Using Multiple Views. For the class of queries we consider here, checking whether a query can be answered using multiple views can be expressed in terms of checking whether a query can be answered using a single view.

Theorem 8.3.2 *Let Q be a query and $\{V_1, \ldots, V_n\}$ be a set of views. Query Q can be answered using $V_1, \ldots, V_n$ iff for some V_i, $i \in [1, n]$, Q can be answered using V_i.*

Figure 8.3 shows a query Q and two views V_1 and V_2. Each of these views has a homomorphism to Q which is also shown in the figure. Based on Corollary 8.3.1, Q can be answered using V_1 (or V_2). Therefore, based on Theorem 8.3.2, Q can be answered using V_1, V_2.

For the case of answering completely a query using views we can state the following theorem.

Theorem 8.3.3 *Let Q be a query and $\{V_1, \ldots, V_n\}$ be a set of views. Query Q can be answered completely using $V_1, \ldots, V_n$ iff it can be answered using $V_1, \ldots, V_n$ and for every node in Q, there is a covering node in some (not necessarily the same) V_i, $i \in [1, n]$.*

Based on Theorem 8.3.3, one can see that query Q of Figure 8.3 can be answered completely uisng the views V_1 and V_2 of the same figure.

Computing the Answer of a Query Using Multiple Views. In order to perform the computation of the answer of the query using a set of materialized views we associate query nodes with the set of corresponding covering nodes in the views. The set of covering nodes of a given query node in multiple views is defined in terms of the set of covering nodes of the query in a single view: let X be a node in query Q, and $m_1(X), \ldots, m_n(X)$ be the sets of covering nodes of X in $V_1, \ldots, V_n$, respectively. Then, the set $m(X)$ of covering nodes of X in $V_1, \ldots, V_n$ is

$$m(X) = \bigcup_{i \in [1,n]} m_i(X)$$

As with the case of a single view, if $\exists X \in Q$, $m(X) \neq \emptyset$, Q can be answered using $V_1, \ldots, V_n$. If $\forall X \in Q$, $m(X) \neq \emptyset$, Q can be completely answered using $V_1, \ldots, V_n$. The materialization of any node in $m(X)$ can be used for computing X. However, we might also use the materializations of some (or all the) nodes in $m(X)$: during the computation of the answer, X will be computed using the intersection of the materializations of these view nodes in $m(X)$.

In this paper, we focus on answering completely queries using views.

8.4 Computing Covering Nodes

As discussed in Section 8.3, given a query Q and a view V, the covering nodes for a node of Q in V are defined in terms of the homomorphisms from V to Q. However, the number of these homomorphisms can be exponential on the size of V. In this section, we present a stack-based algorithm which computes in polynomial time and space the covering nodes of the nodes in Q without explicitly enumerating all the homomorphisms from V to Q.

Match Sets. In the algorithm we use a data structure, called *match set*, which is similar to those employed in [81, 82, 32] for encoding query pattern matches.

Let q be a node in query Q and v be a node in view V. We say that v *matches* q if v has the same label as q. Let T_v and T_q denote the subtrees rooted at v and q, respectively. Let also v_i be a child node of v in V and q_j be a node in the subtree T_q. We say that the pair (v, q) is *consistent* with (v_i, q_j), if v and v_i match q and q_j, respectively, and if $v/v_i \in V$, then $q/q_j \in Q$.

The match set $MS(V, Q)$ is a directed acyclic graph (dag) that compactly stores the set of homomorphisms from V to Q. The nodes of this dag correspond to node pairs (v, q) such that v matches q. Each node (v, q) is associated with an array $ptrsArr$ indexed by the children of v in V. Given a child v_i of v in V, $ptrsArr[v_i]$ is a set of pointers. Each of the pointers points to a node (v_i, q_j), where q_j is a node in T_q and (v, q) is consistent with (v_i, q_j). There is an edge in the dag from node (v, q) to node (v_i, q_j) iff there is a pointer from $ptrsArr$ of (v, q) to (v_i, q_j). We call match set of a node (v, q), denoted $MS(v, q)$, the node (v, q) along with the array $ptrsArr$ of (v, q). Note that node (v_i, q_j) can be a child of multiple nodes $(v, q_1), \ldots, (v, q_n)$, where $q_1, \ldots, q_n$ are ancestor nodes of q_j in Q. Let r_V and r_Q denote virtual roots of V and Q, respectively. Then, the match set dag $MS(V, Q)$ is

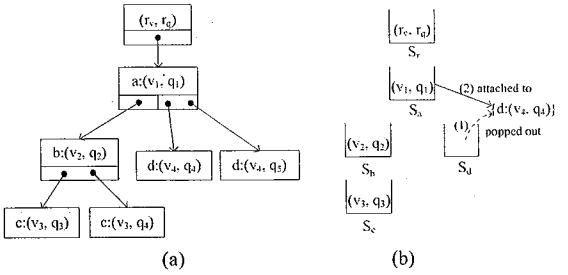


Figure 8.4 (a) The match set dag for the view and the query of Figure 8.2, (b) The snapshots of stacks after the query leaf node d has been visited during the execution of Algorithm *computeCovering*

rooted at the node (r_V, r_Q). As we show later, the size of the dag is polynomial in the size of V and Q.

Figure 8.4(a) shows the dag of the match set for the view V and query Q of Figure 8.2. In order to uniquely identify a node of the view or the query, every node of V and Q in Figure 8.2 is associated with a node id.

Given a match set dag $MS(V, Q)$, we can compute the set of homomorphisms from V to Q. Clearly, the time required for enumerating all the homomorphisms is exponential on the size of the view in the worst case. However, we do not need to enumerate all the homomorphisms in order to compute covering nodes of the query nodes. Instead, as we show below, we can compute covering nodes from the match set dag.

Computing Match Sets. The match set $MS(v, q)$ can be computed inductively by computing the match set of each child of v in V. If v is a leaf node of V, then $MS(v, q)$ consists of only node (v, q). Otherwise, suppose that we have computed all the match sets for each child v_i of v. Then, ptrsArr[v_i] of $MS(v, q)$ is populated by adding pointers to each child node (v_i, q_j) such that (v, q) is consistent with (v_i, q_j). If every ptrsArr[v_i] is non-empty after the population, we call the newly computed $MS(v, q)$ a *valid* match set.

Based on the above idea, we provide below an algorithm that efficiently computes match sets and covering nodes.

The Algorithm. Algorithm *computeCovering*, shown in Listing 14, takes a query Q and a view V as inputs and computes the covering nodes in V for each query node of Q. It is a stack-based algorithm which associates each view node of V with a stack. It proceeds in two steps. In the first step, it calls Procedure $constructMS$ (shown in Listing 15) to compute the match set dag $MS(V, Q)$ (line 2). In the second step, the dag is traversed top-down to determine the covering view nodes (lines 3-5).

Procedure $constructMS$ traverses the tree pattern Q in preorder, constructing the match sets as it visits nodes and traverses edges. When $constructMS$ visits a query node for the first time, it creates a match set for each matching view node. The created match set are pushed onto stacks. When $constructMS$ returns to a query node after traversing the entire subtree of this node, it determines whether the match sets created for the query node are valid and inserts into the arrays $ptrsArr$ of their parent nodes pointers that point to the corresponding nodes. When $constructMS$ finishes the traversal of Q, $MS(r_V, r_Q)$ encodes all the homomorphisms from V to Q. We describe the process below in more detail.

Initially, a match set $MS(r_V, r_Q)$ is pushed onto stack S_{r_V}, the stack of the virtual view root. For each query node q visited for the first time, $constructMS$ iterates in postorder over each view node v matching the query node (line 1). Let (u, p) be the node of the match set corresponding to the top entry of stack S_u. Procedure $constructMS$ checks whether (u, p) is consistent with (v, q). If this is the case, a match set $MS(v, q)$ is created and then pushed onto stack S_v (lines 2-7). Next, $constructMS$ recursively calls itself on

each child node of q (lines 8-9). After the traversal of the subtree of q, for each v matching q considered in preorder, it pops out the top entry $MS(v, q)$ from stack S_v (lines 10-11). If $MS(v, q)$ is valid, for each entry in stack S_u, where u is the parent of v, a pointer that points to (v, q) is created and added to the entry's ptrsArr[v_i] (lines 12-15).

Figure 8.4(b) shows a snapshot of the view stacks during the execution of Algorithm $computeCovering$. After the query leaf node d (node id 4) has been visited, the corresponding match set is popped out from the stack S_d of view node d. Since it is valid, it is attached to the only match set in stack S_a of view node a.

Complexity. Let v be a node in V. We define the *prefix* query of v, denoted $prefix(V, v)$, as the path from the root of V to v. Given a query Q, we define the *recursion depth of node v in Q* as the maximum number of nodes in a path of Q that are images of v under all the possible embeddings to $prefix(V, v)$ in that path of Q. We define the *recursion depth* D of V in Q as the maximum recursion depth of the view nodes of V in Q.

The number of query nodes matched by a view node is bounded by the number $|Q|$ of the nodes of Q. The total number of match sets constructed during execution is bounded by $|V| \times |Q|$. The number of incoming pointers to each constructed match set is bounded by D. Therefore, the space complexity of Algorithm $computeCovering$ is bounded by $O(|V| \times |Q| \times D)$.

The time complexity of Algorithm $computeCovering$ is determined by the time for processing stack entries (that is, match sets). The number of entries in each stack at any given time is bounded by D. Let v be a view node that matches a query node q under consideration. Procedure $constructMS$ spends $O(fanout(v) + D)$ on checking whether $MS(v, q)$ is valid and on visiting entries in the parent stack of v, where $fanout(X)$ denotes

Listing 14 Algorithm computeCovering

1 create a stack for each node of V and initialize the covering node set $m(q)$ to be empty for each node q of Q.

2 constructMS(root(Q))

3 let $visited$ be a boolean matrix where the rows are indexed by the nodes of V and the columns are indexed by the nodes of Q. Initialize each field of $visited$ to be $false$

4 **for** (every node $MS(v, q)$ encountered in the top down traversal of the match set dag of V and Q) **do**

5 if $visited[v, q]$ is $false$, then add v to $m(q)$, set $visited[v, q]$ to $true$, and continue the traversal on the children of $MS(v, q)$.

the out-degree of v in V. Since the number of view nodes that match node q is $O(V)$, the total time spent on processing stack entries for each node in Q is $O(|V| + |V| \times D)$, which is dominated by $O(|V| \times D)$. Therefore, the time complexity of Algorithm $computeCovering$ is bounded by $O(|V| \times |Q| \times D)$.

8.5 Optimization Issues

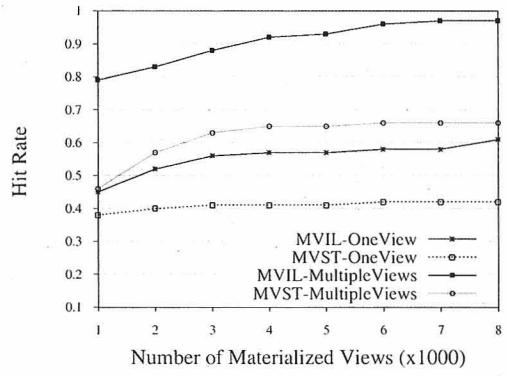
Computation Time Issues. As discussed in Section 8.3, if a query Q can be answered completely using some views, and $m(X)$ is the set of all the covering nodes of a node X in Q with respect to these views, then X can be computed using the intersection of the materializations of the nodes in $m(X)$. If additional views that have a homomorphism to Q are discovered in the view pool, the set $m(X)$ of covering nodes for X with respect to all the views will potentially get new view nodes and the intersection of their materializations will potentially decrease in size making, of course, the computation of X cheaper. However, there is a cost associated with discovering additional views that have a homomorphism to Q. Therefore, if a set of views that answers a query Q has been discovered in the view pool, a question that arises is whether it is worth spending additional time to find other views that have a homomorphism to Q in an effort to reduce the overall computation cost of Q using the view materializations. Our experimental results in Section 8.6, show that

Listing 15 Procedure constructMS(q)

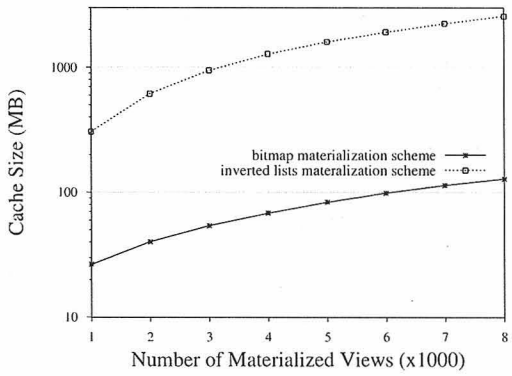
```
1  for (every v ∈ nodes(V) that matches q considered in post-order) do
2     let u be the parent of v in V
3     if (stack S_u is not empty) then
4        let (u,p) be in the top entry of S_u
5        if ((u, p) is consistent with (v, q)) then
6           create MS(v, q) and initialize ptrsArr[v_i] to be empty for every child v_i of v
7           push MS(v, q) to stack S_v
8  for (every child q' of q in Q) do
9     constructMS(q')
10 for (every v ∈ nodes(V) that matches q considered in pre-order) do
11    pop out the top entry e from stack S_v
12    if (e is a valid match set) then
13       let u be the parent of v in V
14       for (every stack entry e' ∈ S_u) do
15          add to ptrsArr[v] a pointer that points to the node of e
```

the answer to this question is positive: the implementation of our algorithm of Section 8.4 takes minimal time to compute all the covering nodes of a query even with a large view pool. This is largely compensated by the benefit in computation time we obtain by finding additional views with homomorphisms to Q.

Using Bitmaps. Consider two view nodes X_1 and X_2 both labeled by the same label a. The materializations L_{X_1} and L_{X_2} of X_1 and X_2 are sublists of the inverted list L_a. L_{X_1} and L_{X_2} might overlap. Instead of storing directly L_{X_1} and L_{X_2}, one can store the union $L_{X_1} \cup L_{X_2}$ of L_{X_1} and L_{X_2} along with two bitmaps B_{X_1} and B_{X_2} on $L_{X_1} \cup L_{X_2}$ for L_{X_1} and L_{X_2} respectively. Bitmap B_{X_i}, $i = 1, 2$, has a '1' bit at position x iff L_{X_1} comprises the XML tree node at position x of $L_{X_1} \cup L_{X_2}$. This idea can be applied to



(a) Hit rate



(b) Cache size

Figure 8.5 Hit rate and cache size with with increasing number of materialized views

multiple view node materializations resulting in important space savings. Note that because the view node materializations $L_{X_1}, \ldots, L_{X_k}$ of the view nodes $X_1, \ldots, X_k$ having the same label are sorted on the *begin* value of the positional representation of their XML tree nodes, the intersection $L_{X_1} \cap \ldots \cap L_{X_k}$ can be computed by merge-joining $L_{X_1}, \ldots, L_{X_k}$. Using bitmaps, the intersection of $L_{X_1}, \ldots, L_{X_k}$ can be computed by bitwise AND-ing $B_{X_1}, \ldots, B_{X_k}$ which produces a bitmap of the intersection $L_{X_1} \cap \ldots \cap L_{X_k}$ on $L_{X_1} \cup \ldots \cup L_{X_k}$. That is, the order is preserved. Besides the important space savings, the use of bitmaps also offers time saving for two reasons: (a) fetching into memory bitmaps of view nodes and the inverted list nodes corresponding to their bitwise AND has less I/O cost than fetching the materializations of these nodes, and (b) bitwise AND-ing bitmaps has less CPU cost than merge-joining the corresponding view node materializations.

8.6 Experimental Evaluation

We implemented our approach and ran experiments to study its time and space performance and scalability. We also ran experiments to compare our approach with traditional approaches.

As traditional approaches assume a different evaluation model and answer sets, this comparison makes sense when it concerns the view cache hit rate.

8.6.1 Experimental Setup

Our implementation was coded in Java. All the experiments reported here were performed on an Intel Core 2 CPU 2.13 GHz processor with 2GB memory running JVM 1.6.0 in Windows XP Professional. The Java virtual machine memory size was set to 512MB. Both XML inverted lists and TPQ view definitions as well as the view materializations were stored in a commercial DBMS. Each displayed time value in the plots is averaged over 5 runs with a cold DBMS buffer cache.

We ran experiments both on an XML benchmark data set generated using $XMark$ [83] and on a synthetic dataset using IBM's XML Generator [67]. We used a 56.2MB XML benchmark data set generated using $XMark$ [83]. This XML document does not include recursive elements. It contains 74 distinct element labels. The total number of parsed element nodes (excluding attributes and text values) is 832911 and the size of their positional representations (i.e., the inverted lists) is 15.1MB. We also ran experiments on a highly recursive synthetic dataset, whose results are similar to those reported here and are omitted in the interest of space.

We used the XPath generator $YFilter$ [84] to produce queries. $YFilter$ generates XPath queries according to specified parameters, such as the maximum query depth, the probability of descendant edges (//), and the probability of branches. In order to create more general workloads, we modified $YFilter$ in the following two ways: (a) we removed the limitation on supporting only one level of nesting of path expressions, so that it can

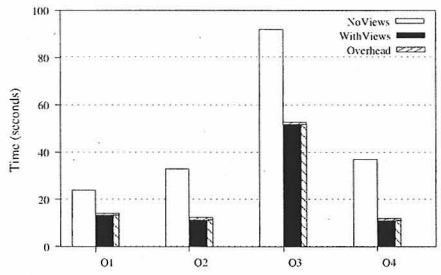
generate complex XPath queries with arbitrary nesting, and (b) we relaxed the restriction on the axis of a predicate path expression which allows only child axes (/).

8.6.2 Hit Rate

We first compare the view cache hit rate of our approach with that of previous approaches. The hit rate expresses the percentage of randomly generated queries that can be answered using one or multiple views materialized in the view cache. In order to compare with previous approaches where queries have output nodes we use the criterion for query answerability using a set of views of [57] which requires that: (a) the output node of a view in the view set is mapped to an ancestor-or-self node of the query output node through a homomorphism (in which case we say that this query node is covered by the view), and (b) each query node which is not covered by this view is covered by some other view in the set.

We generated a workload with 8000 views. We used the following setting for the workload: maximum view depth = 4, probability of descendant edges = 0.8, and probability of branches = 1. We also generated 100 random queries with the following setting: maximum query depth = 9, probability of descendant edges = 0.8, and probability of branches = 1. In the experiments, we scaled the number of views in the view pool from 1000 to 8000. To better illustrate the capacities of the different approaches under comparison, we also measured and compared the hit rate of these approaches when only one view can be used for answering the given query.

Figure 8.5(a) shows the hit rate of different approaches increasing the number of views in the view pool. We refer to our approach as MVIL (Materialized Views as Inverted Lists) and to the approach in [57] as MVST (Materialized Views as Subtrees). Our approach



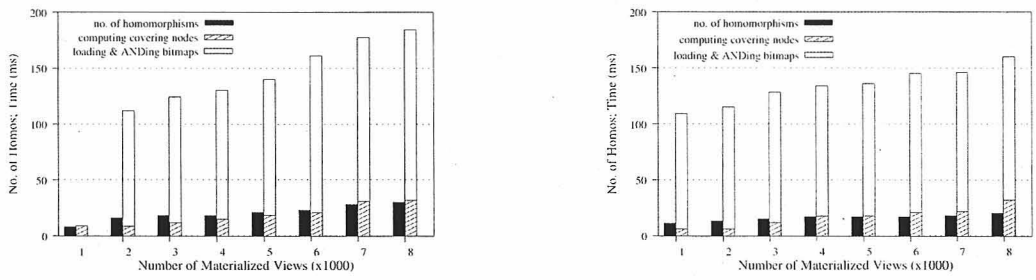
(a) Query processing time

	#INV	#SUB	#ANS
Q1	55854	29963	18977
Q2	77663	26393	4839
Q3	218024	120383	13230
Q4	79076	17084	3237505

#INV: number of nodes in the inverted lists used for evaluating the query
#SUB: number of nodes in the inverted sublists (materialized views) used for evaluating the query using the views
#ANS: number of tuples in the query answer

(b) Query evaluation statistics

Figure 8.6 Query processing time and evaluation statistics on the XMark dataset



(a) Q_2 (b) Q_4

Figure 8.7 Computation overhead with increasing number of materialized views

largely outperforms MVST both when one or multiple materialized views are used to answer the query. For the case of multiple views it outperforms MVST by at least 40% and achieves a hit rate of 97% for 7000 or more views in the view pool.

8.6.3 Space Performance

We also measured the space efficiency of our approach. We used the workload on the XMark dataset described above. Recall that the materialization of a view is stored as bitmaps, one per each view node. In addition, a set of inverted lists is stored, one inverted list per each distinct node label in the views of the view pool. Each such inverted list is the union of the materializations of all the view nodes with the same label in the view pool. We

refer to this materialization scheme as *bitmap materialization scheme*. As a comparison, we also stored directly the materializations of the nodes of all the views and measured the total space used. We refer to the later scheme as *inverted lists materialization scheme*.

Figure 8.5(b) reports on the view cache size under the two materialization schemes as the number of materialized views increases from 1000 to 8000. The scale of the Y-axis is logarithmic. The total size of the view cache under the bitmap scheme rises from 26.45MB to 128.3MB as the number of views in the view pool increases from 1000 to 8000. In comparison, the size of the cache under the inverted lists scheme increases faster than the bitmap scheme from 305.8MB to 2563.63MB. Further, the inverted lists scheme consumes much more space, up to 20 times more than the bitmap scheme for most of the test cases.

Notice that the size of the bitmap materializations can be further reduced using state of the art bitmap compression techniques [85] without compromising the efficiency of bitwise logical operations. Such an implementation is beyond the scope of the dissertation.

8.6.4 Query Processing Time

We next show the speedup obtained in query evaluation time with our approach. We assume that the views are materialized in the client side while the base XML data is stored remotely in the server side. Queries are evaluated at the server side without using materialized views, while they are evaluated at the client side using exclusively the view materializations. In both cases the inverted lists evaluation model is adopted and the state of the art holistic algorithm $TwigStack$ [20] is employed. The communications costs are ignored. If these costs are taken into account the savings achieved by our approach are even larger. For the comparison, we used the workload of the 8000 materialized views described above. Among the 8000 views, 6605 have non-empty answers. We also used four test queries on

the XMark dataset, which are shown in Figure 8.8. These queries are randomly generated and they can all be answered using exclusively the materialized views. Figure 8.6(a) reports on the query processing time per query for two different configurations: *NoViews* refers to evaluating queries on the server XML database without using materialized views. *WithViews* refers to answering queries using exclusively materialized views stored in the client view cache. *Overhead* denotes the computational overhead for using materialized views. It consists of the time needed for finding the covering view nodes of the query nodes and the time needed for loading in memory and bitwise ANDing the bitmaps of the node materializations.

Q1	//site/people/person[.//interest]/name
Q2	//site/regions/namerica/item[.//quantity] [.//mail//to]/name
Q3	//open_auction[.//description[text//keyword]] [initial][quantity]//bidder/date
Q4	/site[.//person[.//creditcard]/address[country] //zipcode]//africa//incategory

Figure 8.8 Queries on the XMark dataset

As we can see from Figure 8.6(a), *WithViews* achieves significant speedup compared to *NoViews*: from 77% for Q_3 up to a factor of 2.3 for Q_4 (our experiments on a highly recursive dataset show a speedup by a factor of 13 for some queries). For each query, the fraction of *Overhead* in the total processing time using *WithViews* is very small, ranging from 0.34% for Q_3 to 1.73% for Q_2.

Figure 8.6(b) shows the evaluation statistics of the four queries of Figure 8.8 over the XMark dataset. We observe that the query evaluation performance is largely determined by the number of inverted list nodes read from disk during execution, since each disk access triggers I/O whose cost dominates the computation costs of the query. As we can see in Figure 8.6(b), a query can be computed using substantially smaller inverted lists with our

approach (column #SUB) than with the *NoView* approach (column #INV). For instance, the number of nodes accessed using materialized views is reduced by 78% for query Q_4 of Figure 8.8. This reduction in size, reduces the I/O cost, but it also reduces the CPU cost resulting in a substantial speedup.

8.6.5 Scalability

Finally, we measured the scalability of our approach as the number of the materialized views in the view pool increases. The scalability is examined in terms of the computation overhead which, as explained in Section 8.6.4, consists of two parts: (a) the time spent on finding all the query covering nodes in the view pool–this operation is done by the algorithm described in Section 5.2, and (b) the time spent on loading selected bitmaps from disk to memory and on bitwise ANDing bitmaps.

Figure 8.7 reports on both components of the computation overhead, as well as the number of homomorphisms from the view to the query when the number of materialized views increases from 1000 to 8000 for two queries Q_2 and Q_4. Notice that the bitmap processing component is 0 for query Q_2 when the view pool contains 1000 views, since Q_2 has no hit on the view cache in this case. As expected, the number of homomorphisms for each query grows as the number of views increases. Both components of the overhead grow very smoothly. For instance, for query Q_4, the covering node computation component and the bitmap processing component for 1000 views are 6ms and 103ms, respectively. They grow to 32ms and 128ms for 8000 views (a ratio of 5.3 and 1.2 respectively). Note that using a bitmap compression technique [85] can further reduce the size of bitmaps and thereby the I/O cost for loading them in memory.

CHAPTER 9

CONCLUSION AND FUTURE WORK DIRECTIONS

In this chapter, the dissertation is concluded by summarizing the contributions and providing a discussion of the future work.

9.1 Summary of Contributions

Current applications export and exchange XML data on the web. In this context, a major challenge is the querying of the data when the structure is complex or is not fully known, and the integrated querying of multiple data sources that export data with structural differences and irregularities. The dissertation focuses on three aspects. One is the design of efficient non-main-memory evaluation methods for PTPQs. Another is the assignment of semantics to PTPQs so that they return meaningful answers. The third aspect is on answering PTPQs using materialized views.

A query language with wildcards that allows partial specification of a tree pattern has been introduced. PTPQs can express a broad fragment of XPath. Because of their expressive power and flexibility, they are useful for querying XML documents whose structure is complex or not fully known to the user, and for integrating XML data sources with different structures.

The problem of evaluating partial path queries with repeated labels under the indexed streaming model has been addressed. Partial path queries are not a subclass of tree-pattern queries but they form a subclass of PTPQs. Partial path queries are represented as dags. We have designed three algorithms for evaluating partial path queries on XML data. The first

algorithm *IndexPaths-R* exploits a structural summary of data to generate an equivalent set of path patterns of a partial path query and then uses a stack-based algorithm *PathStack-R* for evaluating path queries with repeated labels. The second algorithm *PartialMJ-R* extracts a spanning tree from the query dag and uses the *PathStack-R* to find the matches of the root-to-leaf paths in the tree. These matches are progressively merge-joined to compute the answer. Finally, the third algorithm *PartialPathStack-R* exploits multiple pointers of stack entries and a topological ordering of the nodes to apply a stack-based holistic technique. To the best of the author's knowledge, *PartialPathStack-R* is the first holistic algorithm that evaluates partial path queries with repeated labels. An analysis was provided to those three algorithms and extensive experimental evaluations were conducted to compare their performance. The results show that *PartialPathStack-R* has the best theorectic value and has considerable practical performance advantages over the other two algorithms.

Based on the work on the partial path queries, an efficient holistic algorithm, called $PartialTreeStack$, was designed for evaluating PTPQs in the indexed streaming model. Algorithm $PartialTreeStack$ takes into account the dag form of PTPQs and avoids redundant processing of subdags having multiple "parents." It avoids checking whether node matches satisfy the dag structural constraints when it can derive that they violate a same-path constraint. $PartialTreeStack$ finds solutions for the partial paths of the query and merge-joins them to produce the query answer. When no parent-child relationships are present in the query dag, it is guaranteed that every partial path solution produced will participate in the final answer. Therefore, $PartialTreeStack$ does not produce intermediate results. A theoretical analysis of $PartialTreeStack$ was provided to show its polynomial time and space complexity. It was further shown that under the reasonable assumption that

the size of queries is not significant compared to the size of data, $PartialTreeStack$ is asymptotically optimal for PTPQs without parent-child structural relationships.

In order to assess the performance of $PartialTreeStack$, two approaches were designed for comparison that exploit existing state-of-the-art techniques for more restricted classes of queries. The first one is algorithm $TPQGen$, which generates a set of TPQs equivalent to the given PTPQ, and computes the answer of the PTPQ by taking the union of their solutions. The second one is algorithm $PartialPathJoin$, which decomposes the PTPQ into partial-path queries and computes the answer of the PTPQ by merge-joining their solutions. All three algorithms were implemented and detailed experiments were conducted to compare their performance. The experimental results show that $PartialTreeStack$ outperforms the other two algorithms. To the best of the author's knowledge, $PartialTreeStack$ is the first algorithm in the indexed streaming model that supports such a broad fragment of XPath.

An efficient streaming algorithm called PSX was developed for evaluating PTPQs in the streaming model. To the best of the author's knowledge, no previous algorithms exist that can efficiently support the streaming evaluation of such a broad fragment of XPath. PSX exploits a dag representation of PTPQs enhanced with same-path constraints and wisely avoids processing redundant query matches. It has guaranteed polynomial time and space complexity in the size of the data and query and matches the complexity of the best known streaming algorithm on TPQs. The experimental results show that PSX can be used in practice on a wide range of queries and on large datasets with deep recursion. They also show that PSX largely outperforms, in terms of time and memory usage, the only known streaming algorithm that can support TPQs with reverse axes.

Another efficient streaming algorithm for PTPQs called $EagerPSX$ was also designed. $EagerPSX$ applies an eager evaluation strategy to quickly determine when node matches should be returned as solutions to the user. It proactively detects redundant query matches to save both computational time and memory space. It has guaranteed polynomial time and space complexity in the size of the data and query and is runtime competitive with the only known streaming algorithm for PTPQs which is a lazy algorithm. The experimental results show that $EagerPSX$ can be used in practice on a wide range of queries and on large datasets with deep recursion. They also show that, compared to the lazy algorithm PSX, $EagerPSX$ largely improves the query response time and has better space performance.

An original approach for assigning semantics to PTPQs has been suggested. In contrast to previous approaches that operate locally on data, the proposed approach operates globally on structural summaries of data to extract tree patterns. An experimental evaluation of the proposed approach and the previous approachs shows that the proposed approach has a perfect recall both for XML documents with complete and incomplete data. It also shows better precision compared to approaches with similar recall. The proposed approach can be directly implemented on top of an XQuery engine.

The problem of answering XML queries using exclusively materialized views have been addressed. Previous approaches to this problem are limited by the way query answers (and view materializations thereof) are defined. To overcome these limitations, the problem has been revised by placing it under the setting of the indexed streaming model which is currently the prominent model for evaluating queries on large persistent XML data. In this context, an original approach for materializing views has been suggested which stores the inverted lists of only those XML tree nodes that occur in the answer to the view. To the best of the author's knowledge this is the first time the problem is addressed in this context and

such a materialization scheme is adopted. Necessary and sufficient conditions have been provided for tree-pattern query answerability in terms of view to query homomorphisms. A time and space efficient algorithm was designed for deciding query answerability and it was shown how queries can be computed over view materializations using stack-based holistic algorithms. Optimization techniques were further developed which minimize the storage space and avoid redundancy by materializing views as bitmaps, and that optimize the evaluation of the queries over the views by applying bitwise operations on view materializations. The experimental results showed that the proposed approach has largely higher hit rates than previous approaches, significantly speeds up the evaluation of queries without using views, and scales very smoothly in terms of storage space and computational overhead.

9.2 Directions of Future Work

Future work includes exploiting materialized views for optimizing queries in centralized environments. In this setting, the focus is on answering possibly partially a query using views. An interesting problem is the devise of techniques for selecting views for materialization in order to satisfy a number of optimization goals. It would also be interesting to work on algorithms for the efficient updating of the view materializations when the XML data is modified.

It is also worth further investigating the efficient computation of meaningful answers of PTPQs. Based on the results of Chapter 7, the semantics of PTPQs is defined as a set of TPQs to be evaluated on an XML tree. One research direction involves further elaborating on methods for the efficient computation of these TPQs. Another research

direction involves ranking these TPQs based on the meaningfulness of their answers and designing techniques for the efficient computation of the k-most meaningful among them.

REFERENCES

[1] "World Wide Web Consortium site, W3C," http://www.w3.org/, September 2009.

[2] "XML Path Language (XPath). World Wide Web Consortium site, W3C," http://www.w3.org/TR/xpath20, September 2009.

[3] "XML Query Language (XQuery). World Wide Web Consortium site, W3C," http://www.w3.org/XML/Query, September 2009.

[4] C. Yu and H. V. Jagadish, "Querying complex structured databases," in *VLDB*, 2007, pp. 1010–1021.

[5] Y. Li, C. Yu, and H. V. Jagadish, "Schema-Free XQuery," in *VLDB*, 2004, pp. 72–83.

[6] ——, "Enabling schema-free XQuery with meaningful query focus," *The VLDB Journal*, vol. 17, no. 3, 2008.

[7] D. Theodoratos, T. Dalamagas, A. Koufopoulos, and N. Gehani, "Semantic querying of tree-structured data sources using partially specified tree patterns," in *CIKM*, 2005.

[8] D. Theodoratos and X. Wu, "Assigning semantics to partial tree-pattern queries," *Data Knowl. Eng.*, 2007.

[9] V. Hristidis, Y. Papakonstantinou, and A. Balmin, "Keyword proximity search on XML graphs," in *ICDE*, 2003, pp. 367–378.

[10] Y. Xu and Y. Papakonstantinou, "Efficient keyword search for smallest lcas in XML databases," in *SIGMOD*, pp. 527–538.

[11] D. Olteanu, "Forward node-selecting queries over trees," *ACM Trans. Database Syst.*, vol. 32, no. 1, p. 37, 2007.

[12] A. Schmidt, M. L. Kersten, and M. Windhouwer, "Querying XML documents made easy: Nearest concept queries," in *ICDE*, 2001.

[13] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv, "XSEarch: A semantic search engine for XML," in *VLDB*, 2003.

[14] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram, "Xrank: ranked keyword search over XML documents," in *SIGMOD*, 2003.

[15] D. Florescu, D. Kossmann, and I. Manolescu, "Integrating keyword search into XML query processing," *Computer Networks*, 2000.

[16] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram, "TeXQuery: a full-text search extension to XQuery," in *WWW*, 2004.

[17] M. P. Consens and T. Milo, "Algebras for querying text regions (extended abstract)," in *PODS*, 1995.

[18] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman, "On supporting containment queries in relational database management systems," in *SIGMOD*, 2001.

[19] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava, "Structural joins: A primitive for efficient XML query pattern matching." in *ICDE*, 2002.

[20] N. Bruno, N. Koudas, and D. Srivastava, "Holistic twig joins: optimal XML pattern matching," in *SIGMOD*, 2002.

[21] H. Jiang, W. Wang, H. Lu, and J. X. Yu, "Holistic twig joins on indexed XML documents." in *VLDB*, 2003.

[22] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo, "Efficient structural joins on indexed XML documents," in *VLDB*, 2002.

[23] H. Jiang, H. Lu, W. Wang, and B. C. Ooi, "XR-Tree: Indexing XML data for efficient structural joins." in *ICDE*, 2003.

[24] J. Lu, T. Chen, and T. W. Ling, "Efficient processing of XML twig patterns with parent child edges: a look-ahead approach," in *CIKM*, 2004.

[25] B. Yang, M. Fontoura, E. Shekita, S. Rajagopalan, and K. Beyer, "Virtual cursors for XML joins," in *CIKM*, 2004.

[26] L. Chen, A. Gupta, and M. E. Kurul, "Stack-based algorithms for pattern matching on DAGs," in *VLDB*, 2005.

[27] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan, "Twig2Stack: bottom-up processing of generalized-tree-pattern queries over XML documents," in *VLDB*, 2006.

[28] D. Olteanu, "Spex: Streamed and progressive evaluation of XPath," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 7, pp. 934–949, 2007.

[29] Y. Chen, S. B. Davidson, and Y. Zheng, "An efficient XPath query processor for XML streams," in *ICDE*, 2006, p. 79.

[30] G. Gou and R. Chirkova, "Efficient algorithms for evaluating XPath over streams," in *SIGMOD*, 2007, pp. 269–280.

[31] D. Olteanu, H. Meuss, T. Furche, and F. Bry, "XPath: Looking forward," in *EDBT*, 2002, pp. 109–127.

[32] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski, "Streaming XPath processing with forward and backward axes." in *ICDE*, 2003, pp. 455–466.

[33] S. Abiteboul, P. Buneman, and D. Suciu, *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, 1999.

[34] F. Peng and S. S. Chawathe, "XPath queries on streaming data," in *SIGMOD*, 2003, pp. 431–442.

[35] V. Josifovski, M. Fontoura, and A. Barta, "Querying XML streams," *VLDB Journal.*, vol. 14, no. 2, pp. 197–210, 2005.

[36] Z. Bar-Yossef, M. Fontoura, and V. Josifovski, "On the memory requirements of XPath evaluation over XML streams," in *PODS*, 2004, pp. 177–188.

[37] P. Ramanan, "Evaluating an XPath query on a streaming XML document," in *ICMD*, 2005.

[38] Y. Wu, J. M. Patel, and H. V. Jagadish, "Structural join order selection for XML query optimization," in *ICDE*, 2003.

[39] T. Chen, J. Lu, and T. W. Ling, "On boosting holism in XML twig pattern matching using structural indexing techniques," in *SIGMOD*, 2005.

[40] H. Jiang, H. Lu, and W. Wang, "Efficient processing of XML twig queries with or-predicates," in *SIGMOD*, 2004.

[41] D. Theodoratos, P. Placek, T. Dalamagas, S. Souldatos, and T. K. Sellis, "Containment of partially specified tree-pattern queries in the presence of dimension graphs," *VLDB J.*, vol. 18, no. 1, pp. 233–254, 2009.

[42] S. Souldatos, X. Wu, D. Theodoratos, T. Dalamagas, and T. K. Sellis, "Evaluation of partial path queries on xml data," in *CIKM*, 2007, pp. 21–30.

[43] X. Wu, S. Souldatos, D. Theodoratos, T. Dalamagas, and T. K. Sellis, "Efficient evaluation of generalized path pattern queries on xml data," in *WWW*, 2008, pp. 835–844.

[44] Y. Kanza and Y. Sagiv, "Flexible queries over semistructured data," in *PODS*, 2001.

[45] S. Amer-Yahia, S. Cho, and D. Srivastava, "Tree pattern relaxation," in *EDBT*, 2002.

[46] V. Hristidis and Y. Papakonstantinou, "Keyword proximity search in XML trees," *IEEE Trans. on Knowl. and Data Eng.*, vol. 18, no. 4, 2006, member-Nick Koudas and Member-Divesh Srivastava.

[47] S. Paparizos, S. Al-Khalifa, A. Chapman, H. V. Jagadish, L. V. S. Lakshmanan, A. Nierman, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu, "Timber: a native system for querying XML," in *SIGMOD*, 2003.

[48] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi, "Answering regular path queries using views," in *ICDE*, 2000, pp. 389–398.

[49] A. Balmin, F. Özcan, K. S. Beyer, R. J. Cochrane, and H. Pirahesh, "A framework for using materialized XPath views in XML query processing," in *VLDB*, 2004, pp. 60–71.

[50] C. Yu and L. Popa, "Constraint-based XML query rewriting for data integration," in *SIGMOD*, 2004, pp. 371–382.

[51] J. Tang and S. Zhou, "A theoretic framework for answering XPath queries using views," in *XSym*, 2005, pp. 18–33.

[52] B. Mandhani and D. Suciu, "Query caching and view selection for XML databases," in *VLDB*, 2005, pp. 469–480.

[53] W. Xu and Z. M. Özsoyoglu, "Rewriting XPath queries using materialized views," in *VLDB*, 2005, pp. 121–132.

[54] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola, "Rewriting nested XML queries using nested views," in *SIGMOD*, 2006, pp. 443–454.

[55] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou, "Structured materialized views for XML queries," in *VLDB*, 2007, pp. 87–98.

[56] B. Cautis, A. Deutsch, and N. Onose, "XPath rewriting using multiple views: Achieving completeness and efficiency," in *WebDB*, 2008.

[57] N. Tang, J. X. Yu, M. T. Özsu, B. Choi, and K.-F. Wong, "Multiple materialized view selection for XPath query rewriting," in *ICDE*, 2008, pp. 873–882.

[58] L. V. S. Lakshmanan, H. Wang, and Z. Zhao, "Answering tree pattern queries using views," in *VLDB*, 2006, pp. 571–582.

[59] J. Wang and J. X. Yu, "XPath rewriting using multiple views," in *DEXA*, 2008, pp. 493–507.

[60] J. Lu, T. W. Ling, C.-Y. Chan, and T. Chen, "From region encoding to extended dewey: on efficient processing of XML twig pattern matching," in *VLDB*, 2005.

[61] D. Phillips, N. Zhang, I. F. Ilyas, and M. T. Özsu, "Interjoin: Exploiting indexes and materialized views in XPath evaluation," in *SSDBM*, 2006, pp. 13–22.

[62] E. Lee and J. Geller, "Parallel operations on class hierarchies with double strand representation," Parallel Processing for Artificial Intelligence 3, J. Geller, H. Kitano and C. B. Suttner (editors), North-Holland Elsevier, 1997, 69-94.

[63] T. Milo and D. Suciu, "Index structures for path expressions," in *ICDT*. London, UK: Springer-Verlag, 1999, pp. 277–295.

[64] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth, "Covering indexes for branching path queries," in *SIGMOD*. ACM Press, 2002, pp. 133–144.

[65] R. Goldman and J. Widom, "DataGuides: Enabling query formulation and optimization in semistructured databases," in *VLDB*, 1997.

[66] "University of pennsylvania Treebank Project," http://www.cis.upenn.edu/ treebank, September 2009.

[67] A. L. Diaz and D. Lovell, "IBM's XML generator," http://www.alphaworks.ibm.com/tech/xmlgenerator, September 2009.

[68] D. Suciu, "XML data repository," http://www.cs.washington.edu/research/xmldatasets/, September 2009.

[69] D. Olteanu, T. Furche, and F. Bry, "Evaluating complex queries against XML streams with polynomial combined complexity," in *BNCOD*, 2004, pp. 31–44.

[70] D. Megginson and et al, "Simple API for XML," http://www.saxproject.org/, September 2009.

[71] C. Cortes, K. Fisher, D. Pregibon, and A. Rogers, "Hancock: a language for extracting signatures from data streams," in *KDD*, 2000, pp. 9–17.

[72] S. Madden and M. J. Franklin, "Fjording the stream: An architecture for queries over streaming sensor data," in *ICDE*, 2002, pp. 555–566.

[73] A. Barta, M. P. Consens, and A. O. Mendelzon, "Benefits of path summaries in an XML query optimizer supporting multiple access methods," in *VLDB*, 2005, pp. 133–144.

[74] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes, "Exploiting local similarity for indexing paths in graph-structured data," in *ICDE*. IEEE Computer Society, 2002, p. 129.

[75] A. Y. Halevy, "Answering queries using views: A survey," *VLDB J.*, vol. 10, no. 4, pp. 270–294, 2001.

[76] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, "Optimizing queries with materialized views," in *ICDE*, 1995, pp. 190–200.

[77] J. Goldstein and P.-Å. Larson, "Optimizing queries using materialized views: A practical, scalable solution," in *SIGMOD*, 2001, pp. 331–342.

[78] M. M. Moro, Z. Vagena, and V. J. Tsotras, "Tree-pattern queries on a lightweight XML processor," in *VLDB*, 2005, pp. 205–216.

[79] G. Gou and R. Chirkova, "Efficiently querying large XML data repositories: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 10, pp. 1381–1403, 2007.

[80] P. Rao and B. Moon, "Prix: Indexing and querying XML using prüfer sequences," in *ICDE*, 2004, pp. 288–300.

[81] C. M. Hoffmann and M. J. O'Donnell, "Pattern matching in trees," *J. ACM*, vol. 29, no. 1, pp. 68–95, 1982.

[82] G. Miklau and D. Suciu, "Containment and equivalence for an XPath fragment," in *PODS*, 2002, pp. 65–76.

[83] A. Schmidt and et al., "XMark: An XML benchmark project," http://monetdb.cwi.nl/xml/, September 2009.

[84] Y. Diao and et al., "YFilter," http://yfilter.cs.umass.edu/, September 2009.

[85] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 1–38, 2006.