View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ScholarBank@NUS

Optimizing XPath Queries Using Composite Axes

Sun Chong

NATIONAL UNIVERSITY OF SINGAPORE

2006

Optimizing XPath Queries Using Composite Axes

Sun Chong

(B. Eng. Tianjin University, P. R. China)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2006

Acknowledgement

I would like to express my gratitude to all those who gave me the possibility to conduct this piece of research and complete this thesis. I want to thank the Department of Computer Science (CS) of the National University of Singapore (NUS) for the strong support for my research work.

I am deeply indebted to my supervisor Dr. Chan Chee Yong, whose guidance, stimulating suggestions, and encouragement helped me in all the time of my research for and writing of this thesis.

Lastly, I would like to thank my family and all the friends in Singapore and China, for their understanding and support for my research work.

Contents

Summary v

List of figures vii

List of tables ix

1 Introduction 1

1.1 Contributions 4

1.2 Organization 4

2 Related Work 6

2.1 Structure Join Order Selection 6

2.2 Query Minimization 7

2.3 Optimization Based on Rewriting Techniques 10

2.3.1 Eliminating Wildcard Steps 10

2.3.2 Eliminating Reverse Axes 11

2.3.3 Removing Duplication and Pipelining 12

3 Preliminaries 14

4 Specialized Navigational Axes 18

4.1 Rewriting with SNAs 19

5 Region Axis 22

5.1 Data Model and Labeling 23

5.1.1 Fence Definition 24

5.2 Characterizations of Node Regions 25

5.3 Basic Form 28

5.4 Generalized Form 29

5.4.1 Rewriting Rules 30

5.5 Generalized Form with Constraints 31

5.5.1 Height Constraint 33

5.5.2 Horizontal Constraint 35

5.5.3 Update of Constraints 36

5.6 Rewriting Wildcard Queries 38

5.6.1 Eliminating NB*-Steps 39

5.6.2 Minimizing B*-Steps 40

5.6.3 Eliminating B*-Steps 42

6 Rewriting with Composite Axes **47**

6.1 Rewriting Algorithm 47

6.2 Implementation Issues 49

6.2.1 Checking of Inclusion Constraints 50

6.2.2 Evaluating Region Axis 51

6.2.3 Optimized Partial Data Loading 52

6.2.4 Implementing the SNA 53

7 Performance Study **55**

7.1 Experimental Setup 55

7.2 Experimental Results 58

8 Conclusions **63**

Summary

This thesis examines the XPath query evaluation and optimization in XML databases. The evaluation cost of an XPath query is a function of both the query size (in terms of the number of axis steps) as well as the complexity of axis step evaluation. Existing approaches to optimize XPath query evaluation have focused on query minimization techniques to reduce the number of query steps, access methods and processing algorithms to reduce the evaluation cost of axis steps. This thesis presents a novel approach to optimize XPath query evaluation by rewriting an input query using a set of composite axes.

In this thesis, we have designed the *specialized navigational axis*(SNA), which can be used to rewrite an input query to access much fewer elements to compute the evaluation results. At the same time, we have designed the novel composite axis *Region Axis*(RA), which is mainly used for rewriting the wildcard steps in XPath queries, whose evaluation is generally expensive. After rewriting the query with RA, we can generally skip the wildcard steps in the query and greatly improve the evaluation performance. We also provide a set of rewriting rules for the RA as well as the constraints to make the rewriting keep the equivalence. Note that we could combine both the SNA and RA into the query rewriting.

By rewriting with both SNA and RA, an optimized query not only has fewer steps, but the composite-axis steps can also be more efficiently evaluated than the replaced steps. We have conducted comprehensive experiments and the results demonstrate significant performance improvement using our proposed optimization and evaluation techniques.

List of Figures

3.1 XPath axes 15

4.1 Relationship among SNAs of α axis 19

5.1 Data model for region axis 23

5.2 Examples of node region types 26

5.3 Example of query rewriting with composite axes 42

5.4 Example of query rewriting with composite axes, where $R(...) = R(id_{min}(i,l),$ $id_{max}(i,l) - 1, l + 1, n)$ 43

5.5 Eliminating B*-steps with inclusion constraints 43

6.1 Rewriting with composite axes 48

6.2 Data structure for region axis 49

7.1 Varying n_{wc} 58

7.2 Varying n_{wc} 59

7.3 Varying n_b 60

7.4 Varying n_b 61

7.5 Varying number of branching wildcard-steps 61

7.6 Varying input data size 62

List of Tables

5.1 Rewriting rules for region axis $\mathcal{M}(\pi, \alpha)$ 30

5.2 Basic updating Rules 39

5.3 Rewriting Rules to eliminate branching wildcard steps 46

Chapter 1

Introduction

XML is the *de facto* standard language for data representation and exchange, thus lots of data have been stored in the XML format. XPath[26], as the core mechanism of several popular languages for processing XML data, such as XSLT[8] XQuery [5], attracts lots of research interest for its processing and optimization. Efficient XPath evaluation would greatly benefit other applications based on the XPath. Much research effort has been addressed on the efficient XPath evaluation[31, 1, 6, 20] and optimization. The optimization mainly includes two aspects, either optimizing the order of evaluation[29, 21] or transforming the XPath into equivalent form suitable for processing. For transformation, considering the importance of the query size to the evaluation cost, shown to be exponential relationship in [11], minimization of XPath has been one of the research focus [22, 10, 24, 12, 2]. Eliminating the reverse axes to make the XPath looking forward suitable for streaming processing has been conducted in[23] with two sets of rewriting rules. In[16], Helmer et al rewrite the XPath into algebra expression and then pipeline the evaluation.

To more efficiently evaluate the XPath queries, we have introduced the composite

axis, *specialized navigational axes* (or SNAs for short). SNA is essentially a composition of a navigational axis with a pruning optimization to produce a specialized axis that can be more efficiently evaluated. The pruning optimization is derived from the semantics of the other navigational axes that are structurally related to the axis being optimized in the query. As SNAs can be evaluated more efficiently, a query can be optimized by replacing each navigational axis step in the query with an appropriate SNA step. For example in the evaluation of the XPath query "/Desc::a/Foll::b", we just need to fetch the leftmost node "a" for the evaluation of the second query step "/Foll::b". Therefore, the query could be rewritten with our SNA as"$Desc_{lb}$::a/Foll::b". Clearly, with the help of SNA , much fewer elements in the data would be accessed.

Most recent work regarding XPath optimization as in [7] minimizes the wildcard steps in the XPath. A *wildcard step* is an XPath step with the wildcard node tag, such as "Parent::*" and "Preceding::*". For convenience or necessity, wildcard has been wildly used in the XPath, such as in[23], which is used to remove the reverse axes and it is used to represent some hidden elements in [9].For example, when querying against the secured XML views in the forms of DTDs, some element labels would be intentionally substituted by the wildcard in case of leaking any inform that the client issuing the query has no access permission. In more general cases, wildcards are used to represented a set of elements that we don't care about. For example, we may want to query the works of an actor regardless whether it is film or TV paly as in the XPath queries "child::works/child::film/child::name" and "child::works/child::TV/child::name". Actually, the queries could be expressed with the single query with the wildcard step as "child::works/child::*/child::name" to get access all the works of the actor. Wildcard step is generally very expensive for the evaluation, as it brings ambiguity for the XPath

evaluation, resulting in the necessity to pre-load all the XML data into the main memory for evaluation, thus damaging the scalability. Even worse, large amount of intermediate results would greatly deteriorate the evaluation efficiency. Take the simple XPath query "/Dec::*/Dec::a" for example, if the context node is the root, then all the nodes in the XML tree need to be accessed and for each node, its descendant nodes need to be checked whether the tag name is "a". The research work conducted in [7] is not complete and general enough, as it could only partially remove wildcards in the XPath with the help of the layer axis in the vertical axis steps and it has no way to process the more general XPath composed of both vertical and horizontal axes. Even for the vertical wildcards steps, still some common cases could not be processed by the layer axis, not to say efficiently. Therefore though the layer axis is the first effort touching the wildcard issue, it is far from efficiently eliminating all the wildcards.

In this thesis, we introduce another composite axis called $RegionAxis$ to remove the wildcards in the XPath according to the rewriting techniques. The evaluation based on the rewritten XPath would be more efficient and scalable, for the size of the rewritten XPath would be minimized as well as the ambiguity produced by the wildcard would be removed, making selectively loading possible for processing data of huge size that generally could not be hold by the main memory. To clearly present the idea of wildcards removing approach based on rewriting techniques, we would give the following example for illustration. In the query with XPath "/presibling::*/presibling::a", it would be rewritten into the following form as : $/R\{i,j,h,l\}$::a, in which $R\{i,j,h,l\}$ is the basic form of $RegionAxis$. With the new form, one step evaluation would be enough instead of two consecutive steps. More importantly, there is no need to store the intermediate results, usually huge, in the consecutive evaluation for the wildcard steps.

1.1 Contributions

In this thesis, we present a novel approach to optimize XPath query evaluation by rewriting an input query with a set of composite axes. Our contributions are summarized as follows:

- We design a novel composite axis, named *specialized navigational axis* (or SNA for short), to rewrite XPath queries for efficient evaluation. SNAs fully exploit the properties of the XPath axes and combine pruning techniques with the axes to enable the evaluation access much fewer elements.
- We design another novel composite axis, named *region axis* (or RA for short), to rewrite the wildcard steps in XPath query to optimize the query evaluation with a set of rewriting rules. To maintain the equivalence of the query rewriting, constraints and their updating rules are also provided.
- We present a systematic approach to rewrite XPath queries with both the SNAs and RAs, thus making the queries efficiently evaluated. We have done comprehensive experiments with our rewriting algorithms based on both the SNAs and RAs, to examine the efficiency improvement for query evaluation.

1.2 Organization

This thesis is organized as follows. Initially, we introduce the related work in Chapter 2. Then Chapter 3 presents some basic definitions and notations to facilitate the following presentation. Our new composite axes, namely, specialized navigational axes and region axes, are presented, respectively, in Chapters 4 and 5. Chapter 6 presents rewriting

algorithms that combine both types of new axes and describes implementation issues. Chapter 7 presents our experimental evaluation of the proposed optimizations. Finally, we present our conclusions in Chapter 8.

Chapter 2

Related Work

Lots of research effort has been conducted in the XPath optimization, while most is related to minimizing the XPath [22, 10, 24, 12, 2, 4], optimizing the order of evaluation [29, 21] and efficient evaluation strategies[6, 27, 25, 20] or index structures [19] . Not much work is related to the rewriting techniques for the XPath except [23, 16]. [23] presents the two separate complete sets of rewriting rules, based on which the XPath with reverse axes could be rewritten into equivalent reverse axis free forms. The rewritten XPath is more suitable for processing streaming XML data. The research effort in [16] rewrites the XPath into equivalent algebraic expression which could avoid duplication in the processing to pipeline the XPath evaluation.

2.1 Structure Join Order Selection

As value-based join order selection is at the heart of the relational query evaluation, structure join order selection [21, 29]is central to the query optimization of XML database. No matter what kinds of join algorithms [31, 20] have been adopted, join order selection is

always one core step for the query evaluation. Wu et al [29] introduce five different kinds join order selection algorithms for XML query optimization. The intuitive algorithm used for join order selection as in relation database is based on Dynamic Programming(DP), which grantees the best solution but not that efficient. Improvement for DP is the Dynamic Programming with Pruning(DPP)states that DPP is guaranteed to be suboptimal or would reach the deadend. Even though with the pruning, the optimization might still be expensive. Then some heuristics are used for the DPP to make aggressive pruning such as limiting the number of intermediate results considered(DPAP-EB) and only considering left-deep plans(DPAP-LD). Fully-pipelined(FP) algorithm only considers the non-blocking query plans and is guaranteed to select the cheapest non-blocking query plan which is fast enough.

2.2 Query Minimization

Till now, the major research work of XPath optimization has been focused on the minimization[22, 10, 24, 12, 2, 4]. In the XPath evaluation, the query size is considered as the main determinant of the efficiency as proved by three experiments with different XPath processors: XALAN, XT and IE6 [11]. The results show that the time cost of the query evaluation is exponential to the size of queries. Therefore it is important to minimize the XPath. The basic idea of XPath minimization is to identify and eliminate the redundant steps in the paths. There are mainly two kinds of redundancy [3]:

- Constraint independent. In a general XPath, some components may be inherently redundant such as that being subsumed by some other components. For example, “ find some classroom with 40 students and with more than one student”. We can

easily see that the condition "with more than one student" is redundant.

- Constraint dependent. In XML document, the DTD always builds some inherent constraints for the data structure such as required child, required descendant and co-occurrence, an irredundant query may become redundant with the presence of the integrity constraint.

The first research effort on minimizing the tree pattern query is conducted by Wood in [28], considering just a fragment of the XPath, which does not include descendent relationship. This kind of XPath is called simple XPath expression. The complexity of minimizing query using simple XPath expression has been shown to be polynomial to the size of the query.

Amer-Yahia et al [2] prove that for a tree query Q, there is always a unique minimal query based on the isomorphism, no matter whether there is integrity constraint or not. Theoretically, whatever way we use to remove the redundant nodes, finally we would get the isomorphous minimal query. The algorithms provided in [2] are built on the basis of endomorphisms from which we can get the following property: A node v of a query Q is redundant *iff* there is an endomorphism on Q that is not identity on v. Tree pattern query in XML document is closely related to the conjunctive query in classical relational database, where the minimization has been generally accepted as a NP-complete problem. While in the XML document, there are some limitations for the tree pattern query and integrity constraints reducing the complexity of the algorithm to polynomial time [2]. Prakash Ramanan et al [24]provide even better polynomial time bounded algorithm for minimizing query with or without integrity constraints. The basic idea for the algorithms is based on the simulation and the algorithm shows that the time complexity is O(N^4) with the presence of the three integrity constraints and even better O(N^2) without the

co-occurrence constraint.

It is exciting that polynomial time algorithms [28, 2, 24] have been designed for tree pattern query in XML document within some fragments of XPath. However a general class of queries may contain branchings, label wildcards and have descendent relationship. [28] processes simple XPath expression without including descendent relationship forming the fragment $XP^{\{/,[],*\}}$, while [2, 24] provide efficient algorithms for XP $XP^{\{/,[],//\}}$ in polynomial time. Lots of research work has shown that the minimization of tree pattern query within fragments combined with any two of the three features has efficient algorithms. However, for the fragment containing all the three features, the minimization is NP-complete which has been proved in [22]. Since it is difficult to design efficient algorithms for minimizing query within the fragment $XP^{\{/,[],*,//\}}$, two kinds of problems appear:find an efficient containment algorithm that is always sound, but not necessarily complete and find a sound and complete algorithm that is EXPTIME in general, but is provably efficient in special cases. [22] builds an algorithm based on the automata and mainly implements the transformation from tree pattern to regular tree language according to the ranking tree. Miklau et[22] also identifies two parameterized classes of queries for which containment is achieved efficiently, and shows that even with the bounded parameters, containment is NP-complete. The research work done in [10] is also on the fragment of $XP^{\{/,[],*,//\}}$. Flesca et al [10] provides the proof for that: the decision problem whether the minimal size of a query Q is greater than K is NP-complete. And again they make some more improvement for the two problems mentioned in [10]: they give an complete and sound algorithm works in exponential to the size of the pattern to be minimized based on the idea of subpattern and adopts the top-down strategy. Also in [10], they give an special case within the fragment $XP^{\{/,[],*,//\}}$ that could achieve the

minimization in polynomial time.

2.3 Optimization Based on Rewriting Techniques

As mentioned before, another important research direction for the optimization of XPath is based on the rewriting techniques. What it needs to do is not just for minimization while keeping the equivalence, but to make the transformed XPath have some specified property that would facilitate the evaluation.

2.3.1 Eliminating Wildcard Steps

Most related work is in [9], which is the first effort addressing the wildcard problem and try to minimize the wildcard steps in the XPath. Chan et al propose a complementary approach of rewriting the XPath based on the layer axis to minimize the wildcard steps in the XPath composed of pure vertical axes. Layer axis is new composite axis designed in [9] with the basic form of $L^{[i,j]}$, which represents the descendant nodes that are “i” to “j” levels lower than the context node or nodes. Layer axis is mainly for handling the wildcard steps combined with the vertical axis. Take the XPath “/chi::*/chi::b” for example, the rewritten XPath has the following form as “$L^{[2,2]}$::b”. Clearly, with the Layer axis, the XPath queries are possible to be rewritten into the equivalent wildcard free queries, saving the expensive wildcard step evaluation. However, the rewriting algorithm for the Xpath queries based on layer axis is not general and complete enough, as it could only handle the XPath composed of purely vertical axes; even for this kind of XPath, the algorithm could not eliminate all the wildcards. Extended Layer axis has been presented to handle complex XPath queries with parent and child axes, while it still can not handle

all the cases with vertical axes in the XPath, at the same time, it is expensive to evaluate the Extended Layer axis.

2.3.2 Eliminating Reverse Axes

[23] presents the algorithms based on two separately complete sets of rewriting rules for transforming the location paths with reverse axes into reverse-axis-free forms. In[18], efficient processing algorithms have been provide for each kind of axis in the XPath. Though efficient, many of the algorithms are built on knowledge of the whole data graph, especially for reverse axes.This strict constraint for XML data greatly limits the application of the algorithms and efficiency of the XPath processing especially for data-centric applications to be handled in the main memory. As reverse axes have blocked the stream-based processing, the straightforward solution is to eliminate the reverse axis to make the XPath looking forward. Rewriting the XPath to be reverse axis free shows better performance in terms of both the time and memory cost than other techniques such as store more information and so on.

The basic concept for rewriting axis is to transform the XPath with reverse axes into another equivalent form constructed with just forward axes.Equivalence for two paths is defined as follwoing: *Take any two paths P_1 and P_2, R_1 and R_2 are the nodes set located by the two paths separately, if $R_1 \equiv R_2$, P_1 is equivalent with P_2.*

According to the fundamental concept of equivalence, [23] builds up two separate sets of rewriting rules for all the reverse axes: General equivalence set and Specific equivalence set. Based on the two sets of rewriting rules, Olteanu et al [23] gives an intuitive algorithm for implementing the transformation. The basic idea is simple: use a stack to record all the location steps and start from the end of the path, whenever

meeting a reverse axis, it would rewrite the path with the corresponding equivalence rules until all the reverse axes have been removed. Each set of the rewriting rules based on the algorithm could eliminate all the reverse axes with different steps. For the general equivalence rule set, the rewriting results would include lots of expensive joins which are not easy for later processing, while the specific equivalence rule set would usually result in a larger size for the output as well as requiring worst exponential time complexity. [23] makes a general comparison and show that which rule set combined with algorithm is better may depend on the practical paths.

2.3.3 Removing Duplication and Pipelining

Similarly as removing reverse axes, Helmer et al [17] optimize the XPath by rewriting it into algebraic expressions suitable for generating code directly. The basic idea for this paper is to rewrite each individual local step into its equivalent form which could be processed by pipelining. One of the core considerations in the rewriting is to handle the duplications. For some axes, even the input is duplicate-free, the output would have some duplications which hesitate the pipelining. So Helmer et al [17] build several equivalence rules based on the idea of step function, which transforms the axes into some algebraic forms which would avoid the duplications. The motivation of their approach to enable a pipelined evaluation of the XPath query shares similarity with our SNAs. However, we differ significantly in our approach of rewriting queries with these axes/operators. In particular, our approach of query rewriting with SNAs simply substitutes a normal axis step with an appropriate SNA, which does not change the number of query steps. In constrast, their approach of rewriting to enable a pipeline evaluation can introduce additional query steps (as part of new predicates) which could transform an input linear

path query to a more complex tree-pattern query. Furthermore, new wilcard steps can be introduced into the rewritten query in their rewriting.

The idea of removing redundant context nodes was also discussed in [15, 13, 14]. However, the pruning approach presented there involves a two-step procedure: an axis step is first evaluated to generate a set of output nodes, which are then pruned before evaluating the next axis step. In contrast, our approach of pruning using composite SNAs actually enables the evaluation of the initial step to be combined with the subsequnt pruning in a single step, which can be more efficiently implemented. In addition, our approach is also more general, as optimization with SNAs can be applied to a branching step that takes into account of the semantics of all the child-steps of the branching step.

Chapter 3

Preliminaries

XPath is generally composed of several axis steps, either of vertical axes or horizontal axes, according to its relative locating area in the XML tree to the context node. Vertical axes mainly include: child, parent, ancestor and descendant. Semantically, vertical axes refer to the elements vertical to the context node in the XML tree. Take the context node "3" in Figure 3.1 for example, its child node set is {8, 9, 10, 11, 12} and its descendant includes the child node set as well as the subset {20, 21, 22, 23}. The parent of the context node is {1} and the ancestor set is the same as the parent of the context in this scenario. Horizontal axes include preceding sibling, following sibling, preceding and following axes. As shown in Figure 3.1, still take node "3" as the context , the pre-sibling node set is {2} and the following-sibling node set is {4,5}. Preceding and following axes usually cover lots of elements, and the node sets are {4, 5, 6, 7, 17, 18, 19} and {2, 13, 14, 15, 16, 24, 25, 26, 27, 28, 29 } respectively. Note that, the preceding contains the pre-sibling and the following contains the following-sibling. In this thesis, we consider the class of XPath queries that are formed using only the following axes: self, child, descendant, parent, ancestor, preceding, following, descendant-or-self and ancestor-or-self (which are

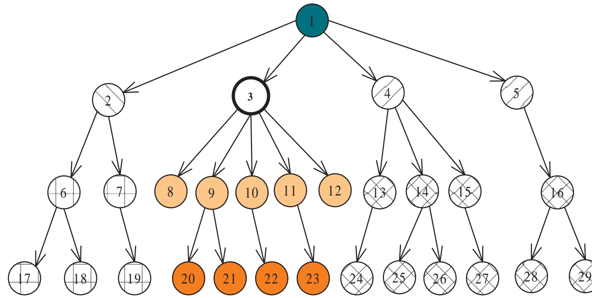


Figure 3.1: XPath axes

abbreviated to *self*, *child*, *desc*, *par*, *anc*, *prec*, *foll*, *ancos*, and *descos*, respectively). Note that descos, and ancos, which combine the vertical axes with the self-explained *self* axis are still denoted as vertical axes. The axes child, desc, foll and descos are called *forward axes*, while the axes par, anc, prec, and ancos are called *reverse axes*.

At the same time, we would like to call child and desc axes as "Down" axis and par and anc axes as "Up" axes. Some "Down" axes followed by some "Up" axes would form a "Up-Down" pattern in the XPath. We refer to a step with axis χ as a χ-*axis step*. This fragment of XPath is syntactically defined as follows:

$$q \quad ::= \quad \chi :: l \ \mid \ \chi :: * \ \mid \ q/q \ \mid \ q[q],$$

where l is an XML tag, $*$ is the wildcard, and '/' and '[.]' denote concatenation and qualifier, respectively. This fragment does not contain the union, negation, and the logical *or* operator. Observe that logical *and* is implicitly supported: $q[q_1 \ and \ q_2]$ is equivalent to $q[q_1][q_2]$.

Given an XPath query q, one can represent q by an unordered rooted tree, denoted by $Tree(q)$, where each step s_i in q is represented by a node v_i in $Tree(q)$ such that there is an edge (v_i, v_j) in $Tree(q)$ if steps s_i and s_j are "consecutive" steps in q of the form s_i/s_j or $s_i[s_j]$. Observe that there could be zero or more qualifier expressions between s_i

and s_j (or $[s_j]$) in q. Given two steps s_i and s_j in q, we say that s_j is a *child step* of s_i (or equivalently, s_i is a *parent step* of s_j)[1] if v_j is a child node of v_i in $Tree(q)$. Given a query node v_i, we use $\chi(v_i)$ to denote the axis associated with q_i.

A step s_i in q is said to be a *branching step* if its corresponding node v_i in $Tree(q)$ has out-degree of at least 2. Furthermore, if the branching step is also a wildcard step (i.e., its nodetest is *), then we refer to it as a *branching wildcard step*, abbreviated as B*-step. A wildcard step that is not a B*-step is abbreviated as NB*-step (for non-branching wildcard step). In the tree representation of the XPath query q, nodes that are underlined indicate the selected nodes to be returned as the query result.

Given an axis-step $\alpha::\tau$, we use $\alpha^k::\tau$ to denote the sequence of steps with $\alpha::\tau$ preceded by $(k-1)$ steps of $\alpha::*$; i.e.,

$$\alpha^k::\tau \;\equiv\; \underbrace{\alpha::* \;/\; \cdots \;/\; \alpha::*}_{k-1}/\alpha::\tau$$

Given a node v in a data tree T, we use $level(v)$ to denote the level of v in T, which is defined to be 1 if v is the root node; otherwise, its level is one more than its parent's level. We use $\delta(x, y)$ to denote the difference in levels between nodes x and y; i.e., $\delta(x, y) = level(x) - level(y)$. We define the *height of* v, denoted by $ht(v)$, as $ht(v) = \max_{v' \in V}\{level(v')\} - level(v)\}$, where V is the set of descendant leaf nodes of v. Thus, $level(v)$ and $ht(v)$ represent the maximum vertical distances between v and respectively, the top-most and bottom-most nodes reachable from v. More generally, for a given set of nodes V, we define the *height of* V, denoted by $ht(V)$, as $ht(V) = \max_{v \in V}\{ht(v)\}$.

In the XPath, an axis step could be the consecutive step of its preceding step of form s_1/s_2 or as a prediction step as of form $s_1[s_2]$. An axis step is called linear step if it

[1] A child (parent) step is not to be confused with a child-axis (parent-axis) step!

just has one consecutive step or one predicate step, otherwise, it is named as branching step, denoted as B-step. Linear step is represented as NB-step. Wildcards could appear in any kind of axes steps as well as in either B-step or NB-step. If we refer a step with axis χ as χ-axis step, then one wildcard step could be expressed as χ::*. If the wildcard is in the linear step, it is called linear wildcard step asχ::*/χ_1::a orχ::*[χ_1::a], otherwise asχ::*[[χ_1::a][χ_1::b]], it is called branching wildcard step. For example, the wildcard step in /Chi::a /Dec::* /Pre-sib::b is linear step in, while the wildcard step is branching step in XPath child::*[Pre-sib::a][Chi::b][Dec::c].

For any given XPath, the evaluation starts from the original context nodes (usually the root of the XML element tree), then for each axis step, we make the evaluation according to the axis step against the context nodes and the result returns as a node set. The returned node set would be the context for the consecutive axis step. It is clear that the evaluation of each axis step would reach a certain area in the XML tree according to the definition of the axes, and the evaluation of the XPath is actually the navigation in the XML tree. Finally the navigation would reach the area that contains the final results.

Chapter 4

Specialized Navigational Axes

To reduce the evaluation cost of axis-steps, we propose a new set of specialized navigational axes (SNAs), each of which is a composition of an axis step with a pruning optimization.

We define four new variants of the *self* axis, denoted by $self_t$ $self_b$, $self_{lb}$, and $self_{rb}$, which for a given set of context nodes S, selects, respectively, the top-most, bottom-most, left bottom-most, and right bottom-most nodes defined as follows:

$$
\begin{aligned}
self_t &= \{v \in S \mid \ \nexists\, v' \in S,\ v' \text{ is an ancestor of v}\} \\
self_b &= \{v \in S \mid \ \nexists\, v' \in S,\ v' \text{ is a descendant of v}\} \\
self_{lb} &= \{v \in self_b \mid \ \nexists\, v' \in self_b,\ v' \text{ precedes v}\} \\
self_{rb} &= \{v \in self_b \mid \ \nexists\, v' \in self_b,\ v' \text{ succeeds v}\}
\end{aligned}
$$

For each variant of the *self* axis $self_x$, where $x \in \{t, b, lb, rb\}$, and an axis α, where $\alpha \in \{desc, foll, prec, anc\}$, we can define a new composite axis, denoted by α_x, as follows:

$$\alpha_x{::}\tau = \alpha{::}\tau \ / \ self_x{::}*$$

We refer to these new axis variants as *specialized navigational axes* (or SNAs). Note that

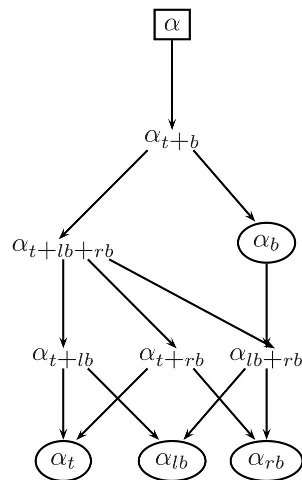


Figure 4.1: Relationship among SNAs of α axis

anc_b, anc_{lb}, and anc_{rb} are all trivially equivalent to anc.

For each axis α, where $\alpha \in \{desc, foll, prec, anc\}$, we can combine its SNAs using the union operator to provide five additional specialized axes denoted by α_{lb+rb}, α_{t+b}, α_{t+lb}, α_{t+rb}, and $\alpha_{t+lb+rb}$. Here, α_{x+y+z} means $\alpha_x \cup \ \alpha_y \cup \ \alpha_z$. The relationship among the SNAs can be captured by a partial order as shown in Fig. 4.1, where there is a directed path from one axis X to another axis X' iff the set of nodes selected by X is a superset of those selected by X'.

4.1 Rewriting with SNAs

Our algorithm for rewriting a query into an equivalent query using SNAs is shown in Algorithm 1. It uses a function $M(\alpha, \alpha')$ that takes a pair of axes (α, α'), where $\alpha = \chi(v)$, $\alpha' = \chi(v')$, and v' is a child node of some query node v, and computes the SNA for v:

Algorithm 1 `Rewrite-SNA` (q)

Input: query q

1: **for each** query node v in a bottom-up traversal of q **do**

2: **if** (v is a non-output step) **then**

3: $S = \{M(\chi(v), \chi(v')) \mid v' \text{ is a child node of } v\}$

4: $\chi(v) = \texttt{LUB}(S)$

5: **return** q

$$M(\alpha, \alpha') = \begin{cases} \alpha_b & \text{if } \alpha' = anc, \\ \alpha_t & \text{if } \alpha' = desc, \\ \alpha_{rb} & \text{if } \alpha' = prec, \\ \alpha_{lb} & \text{if } \alpha' = foll, \\ \alpha & \text{otherwise.} \end{cases} \tag{4.1}$$

More generally, if v has multiple child nodes, then $\chi(v) = \texttt{LUB}(S)$, where $S = \{M(\chi(v), \chi(v')) \mid v' \text{ is a child node of } v\}$ is the set of all SNAs of v's child nodes. Here, `LUB` is a function that computes the *lowest upper bound* of S w.r.t. the partial order Fig. 4.1[1].

Example 1.1 Consider the XPath query $q = prec{::}a[anc{::}b]/desc{::}c$. Since $M(prec, anc) = prec_b$ and $M(prec, desc) = prec_t$, we have $\chi(prec :: e) = \texttt{LUB}(\{prec_b, prec_t\}) = prec_{t+b}$. Therefore, the step *prec::a* in q can be optimized to $prec_{t+b}{::}a$, which can be evaluated more efficiently than *prec::a*. □

Based on Eq. (4.1), the axis of a query node v cannot be specialized if one of its child node (say v') has an axis type child or par. Nevertheless, it might still be possible

[1] For example, $\texttt{LUB}(\{\alpha_t, \alpha_b\}) = \alpha_{t+b}$.

to specialize v in some cases if v' has exactly one child node (say v'') such that the path v'/v'' can be transformed into an equivalent step (for the purpose of specialization) using the following rules:

T1. $\text{chi}^k\text{::*/desc::}\tau \equiv \text{desc}^{k+1}\text{::}\tau$

T2. $\text{chi::*/anc::}\tau \equiv \text{ancos::}\tau$

T3. $\text{par}^k\text{::*/anc::}\tau \equiv \text{anc}^{k+1}\text{::}\tau$

T4. $\text{chi::*/par::*/}\beta\text{::}\tau \equiv \beta\text{::}\tau$

Example 1.2 Consider the XPath expression q = foll::a/chi::*/par::*/chi::*/desc::*. By Eq. (4.1), the step foll::a cannot be specialized. However, by applying the rule T4 to q, q becomes transformed to q' = foll::a/chi::*/desc::*; and applying rule $T1$ to q', we obtained q'' = foll::a/desc^2::*. Now, Eq. (4.1) is able to specialize the $foll$ axis in q'' to follow$_t$. Therefore, q can be optimized to follow$_t$::a/chi::*/par::*/chi::*/desc::*. □

The overall approach to rewrite a query with using SNAs proceed by traversing the query nodes in q in a bottom-up manner. For each query node v visited, if v is of axis type child or par, the rules T1-T4 are applied if possible to transform v to a form so that it can facilitate subsequent specializations. Furthermore, if v is not an output node, then Eq. (4.1) is applied to v as described above.

Chapter 5

Region Axis

In this chapter, we present another composite axis called the *region axis* to rewrite a *wildcard path query* into a single axis step. A *wildcard path query* (or WP-query) refers to a sequence of axis steps where all the steps (except possibly for the last step) are wildcard steps. The ability of the region axis to concisely express a WP-query as a single step can be used as to optimize more complex queries with wildcard steps. More generally, queries involving wildcard steps can be broadly classified into two types: *branching wildcard queries* (B^*-query) and *non-branching wildcard queries* (NB^*-query). A wildcard query is a B^*-query if it has a branching step that is also a wildcard step; otherwise, it is a NB^*-query. Thus, WP-queries are a special case of NB^*-queries.

We start by examining properties of WP-queries. Conceptually, a WP-query selects a region of nodes S (w.r.t. a context node c) such that each node in S is reachable from c via a sequence of "typeless" navigational axis steps. We first give an introduction of the basic data model for the region axis in Section 5.1 , then we characterize the different types of node regions in Section 5.2; and then present the region axis starting from the simplest form in Section 5.3, followed by a more general form in Section 5.4. Finally, we

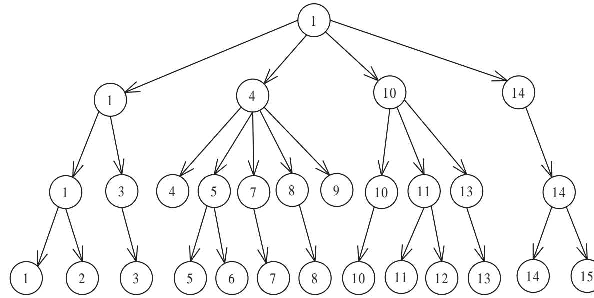


Figure 5.1: Data model for region axis

present the most general form that supports constraints in Section 5.5.

5.1 Data Model and Labeling

XML data of tree structure is generally organized into layers. Layer axis[7], which is based on the level information could naturally represent the vertical axes steps, such as child, parent axes. However, Layer axis could not be easily used to represent the horizontal axes. To efficiently evaluate the horizontal axes, more information such as labeling is needed. Range based labeling scheme mainly focuses on the vertical relationships, such as descendant-ancestor and parent-child. Here we introduce a new labeling scheme, *fence labeling*, as in Figure 5.1. The basic idea for *fence Labeling* scheme includes the following two principles:

1. *All leaf nodes are labeled sequentially from left to right.*
2. *Label of each node is the smallest of its descendants.*

With this fence label, each node could be represented by the set (*label,level*). For example in Figure 5.1, (8,2) represents the node with label 8 in level 2 (root is in the level 0). As no same labels would appear in the same level, each set represents one unique element

in the XML tree. The process for the fence labeling, as that for range based labeling scheme, could be achieved in a single parse of the XML data.

The fence label is much more concise with one label for each node and the size of the fence label is bounded by the leaf nodes number. More importantly, *fence label* could horizontally locate nodes in the XML tree bounded by O(h)(h is the height of the XML document), and node locating is essential for navigating in the XML tree. For example, the searching of node(5,2) starts from the leaf node level and finding the leaf node with label 5 as (5,3) would cost constant time O(1) according to the implementation in the following section. Then along the linked path from leaf to root, node(5,2) could be found with the cost O(h). While locating a node with range based label would always resort to binary search with the cost O(lgn)(n is the number of nodes in level l).

5.1.1 Fence Definition

Horizontally, a XML tree could be considered to be composed of several paths from the leaf node to the root. We would like to call each path with the *fence label* as a *fence*. According to the *fence labeling* scheme, each fence could be decided by the leaf node label of its path, since each node label of one path is the largest among the labels of the same level less or equal to the leaf node label of that path. Thus the leaf node label is called the *ID* of the fence. The labels of the elements in that fence are called *fvalue*, denoted as *fvalue(fence ID, level)*,which could be determined by the fence and the level as follows:

$$fvalue(i,l) = \begin{cases} 0 & i{=}1 \\ max(label(n)), & \\ where, \ \mathit{label}(n) < i \& \mathit{level}(n) = l & i > 1 \end{cases}$$

fvalue(i,l*)*returns the largest one among the labels that are less than or equal to i in level l. So, in all, fence ID determines one fence in the XML tree and the fence plus the level would decide the *fvalue*.

For example, the fence from leaf node(6,3) to the root is determined by the leaf node label 6(fence ID),named as fence(6), and fence(6) is composed of the following labels:"6,5,4,1"(bottom-up). Obviously, all the labels of the elements in fence(6) are less than '6'. These labels are the *fvalue* of the fence in each level, for example, '5' is the *fvalue* of fence(6) in level 2 and is the largest one of the labels that are less than or equal to the ID '6' in level 2. Take fence(12) for example, it is made up of "12,11,10,1" and fvalue(12,1) is '10'. Note that, fence label is for the element node and *fvalue* is for the fence.

5.2 Characterizations of Node Regions

To appreciate and understand the design the region axis, we first analyze the various types of node regions that can result from WP-queries. Our analysis reveals that the node region selected by a WP-query can be classified into four main types. A node region is defined to be **regular region** if it satisfies two properties:

(P1) the nodes selected at each level are contiguous in the sense that there can not be any non-selected node (i.e., "hole") that lies between two selected nodes at some level; and

(P2) the region has a "smooth contour" in the sense that for every pair of leftmost (or rightmost) selected nodes v and w in the region, where v is one level above w, v is either a leaf node or a parent of w.

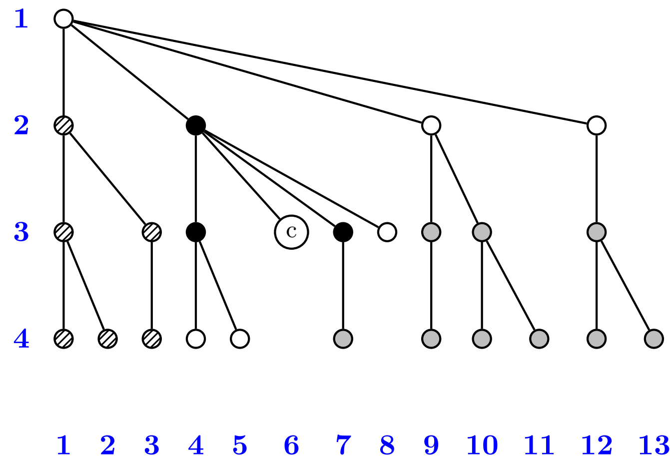


Figure 5.2: Examples of node region types

A node region that does not satisfy (P1) is said to be a **holey region**; and a node region that does not satisfy (P2) is said to be an **irregular region**. A node region that satisfies neither of the properties is said to be a **holey irregular region**.

Example 2.1 Fig. 5.2 illustrates the three main types of node regions with WP-queries on a data tree w.r.t. a context node *c*: (a) (par::* / prec::*) the hatched region is regular; (b)(foll::* / child::*) the gray region is irregular because its leftmost boundary node at level 3 is not a child of its leftmost boundary node at level 2; and (c)(par::* / desc::* / par::*) the black region is holey because there is a non-selected node (i.e., the context node *c*) between two selected nodes at level 3. If the step *par::** is evaluated w.r.t. the gray region, the outcome is a holey irregular region that consists of the last 2 nodes at level 2, and the last five nodes at level 3 (excluding the last fourth node). □

The type of node region selected by a WP-query can be characterized by the following result.

Theorem 5.2.1 *Let R_q denote the node region selected by a WP-query q. Then*

1. *R_q is regular only if q has only forward-vertical-axis steps or has only reverse-*

vertical-axis steps.

2. *R_q is holey only if q has at least one reverse-vertical-axis step.*

Proof 5.2.1 *we have explicitly and separately explained the the properties about region axes in the above theorem.*

1. Evaluate CR(context region) with forward-vertical-axis to get the new region R, $\forall$ node $n \in R$ except the nodes in the top level of R, $\exists$ node $\alpha \in CR$, n is the descendant or child of α. Therefore we can say that R is a regular region. Similarly when we apply the revers-vertical-axis on the CR, we would also get the regular region. However, when we apply following axes on CR, it would be easy to find that $\exists$ α and β on the left edge of R and α is not descendant or ancestor of β. Similar case would happen to the preceding axis.

2. Evaluate CR(context region) with the following axis to get a new region R. $\exists$ node $\alpha \in CR$, $\forall$ node n in the XML tree, if label(n)>label(α), then $n \in R$. Clearly, R is a region with no holes. Similar case would happen to the preceding axis. When we evaluate CR with forward vertical axis, as $\forall$ node $m \in CR$ has a parent or an ancestor node except for the nodes in the top level of CR. Therefore, all nodes in R is part of the final result. However, some nodes in CR may not have child nodes. When we evaluate CR with reverse vertical axis to get a new region R, there may exist some node α that α should not belong to the final result, which may result in holy for the region. In all, if q has no reverse-vertical-axis step, the node region reached by q would never be holey.

Based on the above analysis, we now present three forms of the region axis (from the simplest to the most general): the basic form can specify regular regions, the generalized form can additionally specify irregular regions, and augmenting constraints to the

generalized form can further specify holey regions.

5.3 Basic Form

An XML document tree could be split horizontally with fence and vertically with the level, thus forming a grid. Our region axis is based on using a $m \times n$ grid to identify the nodes in in a document tree T with n levels and m fences as illustrated in Fig. 5.2. The fences are labeled consecutively from 1 to m starting from the leftmost leaf node based on the fence labeling schema. We use $id(v)$ to refer to the fence label of a leaf node v.

Each node v in T can be referred by a pair (i, ℓ) as discussed before, where $i \in [1, m]$ and $\ell \in [1, n]$; and it means v is the ancestor of the leaf node w with $id(w) = i$, and $level(v) = \ell$. Note that the above definition is well defined only when $level(w) \geq \ell$. For notational convenience, we assume that if $level(w) < \ell$, then (i, ℓ) refers to a "virtual" descendant node of w at level ℓ. Clearly, a node v in T could be referred in k different ways if v is the ancestor of k distinct leaf nodes in T.

Given a node v identified by (i, ℓ), we use $id_{min}(i, \ell)$ or $id_{min}(v)$ to denote the fence label of the leftmost descendant leaf node of v, which is also the fence value of v; i.e., $id_{min}(v) = id_{min}(i, \ell) = \min\{id(v') \mid v' \text{ is a descendant leaf node of } v\}$. Similarly, we use $id_{max}(i, \ell)$ or $id_{max}(v)$ to denote the fence label of the rightmost descendant leaf node of v. We use $id_{int}(v)$ to denote the interval $[id_{min}(v), id_{max}(v)]$.

Example 3.1 Consider again the data tree in Fig. 5.2, where for each node v, $level(v) \in [1, 4]$ and $id(v) \in [1, 13]$. For the node marked c, we have $id_{min}(c) = 5$, $id_{max}(c) = 6$, and $id_{int}(c) = [5, 6]$. Thus, c can be referenced by $(5, 3)$ or $(6, 3)$. □ The

region axis is defined w.r.t. to some context node, say $c = (i, \ell)$. The simplest form of the region axis is given by $R(i_1, i_2, \ell_1, \ell_2)$, where $1 \leq i_1 \leq i_2 \leq m$ and $1 \leq \ell_1 \leq \ell_2 \leq n$. This form enables regular regions to be concisely specified and is defined as follows:

$$R(i_1, i_2, \ell_1, \ell_2) = \{v \in T \mid id(v) \in [i_1, i_2],\ level(v) \in [\ell_1, \ell_2]\} \tag{5.1}$$

Referring again to the example in Fig. 5.2, the hatched regular region can be specified as $R(1, 3, 2, 4)$. However, the basic form of the region axis is inadequate to specify irregular regions as illustrated below.

Example 3.2 Consider the gray irregular region in Fig. 5.2. Observe that it is incorrect to specify it with $R(7, 13, 3, 4)$ because this specification includes two additional nodes, $(7, 3)$ and $(8, 3)$, which are not in the gray region. □

5.4 Generalized Form

To increase the expressiveness of the region axis to handle irregular regions, instead of just being integer values, both i_1 and i_2 in $R(i_1, i_2, \ell_1, \ell_2)$ need to be generalized to be integer expressions involving constants, variables, and functions. Specifically, w.r.t. a context node $c = (i, \ell)$, each of i_1 and i_2 in $R(i_1, i_2, \ell_1, \ell_2)$ must be of one of the following forms: (1) an integer constant value, (2) the value i, (3) the value ℓ, (4) a *level variable*, denoted by $\$L$, that ranges over $[\ell_1, \ell_2]$, or (5) an integer expression involving any combination of integer constants, i, ℓ, and $\$L$, with possibly addition and subtraction arithmetic operators as well as the functions $id_{min}(\cdot, \cdot)$ and $id_{max}(\cdot, \cdot)$. For clarity, we use $f_1(\$L)$ (resp., $f_2(\$L)$) in place of i_1 (resp., i_2) when the first (resp., second) parameter in $R(\cdot)$ is an integer expression that involves the variable $\$L$. Therefore, $R(f_1(\$L), f_2(\$L), \ell_1, \ell_2)$ is defined as follows:

$$R(f_1(\$L), f_2(\$L), \ell_1, \ell_2) = \bigcup_{\$L=\ell_1}^{\ell_2} \{(i', \$L) \mid i' \in [f_1(\$L), f_2(\$L)]\} \tag{5.2}$$

$\mathcal{M}(\pi,\alpha)::\tau \;\equiv\; \pi::* \;/\; \alpha::\tau$		
Axis	**Region axis** π	
α	$R(i_1, i_2, \ell_1, \ell_2)$	$R(f_1(\$L), f_2(\$L), \ell_1, \ell_2)$
$prec$	$R(1, id_{min}(i_2, \$L) - 1, 1, n)$	$R(1, id_{min}(i', \$L) - 1, 1, n)$, $i' = \max_{\$L\in[\ell_1,\ell_2]} \{id_{min}(f_2(\$L), \$L)\}$
$foll$	$R(id_{max}(i', \$L) + 1, m, 1, n)$, $i' = id_{max}(i_1, \ell_2)$	$R(id_{max}(i', \$L) + 1, m, 1, n)$, $i' = \min_{\$L\in[\ell_1,\ell_2]} \{id_{max}(f_2(\$L), \$L)\} - 1$
par	$R(i_1, i_2, \ell_1 - 1, \ell_2 - 1)$	$R(f_1(\$L + 1), f_2(\$L + 1), \ell_1 - 1, \ell_2 - 1)$
anc	$R(i_1, i_2, 1, n - 1)$	$R(f_1(\$L + 1), f_2(\$L + 1), 1, n - 1)$
$child$	$R(id_{min}(i_1, \$L), id_{max}(i_2, \$L - 1), \ell_1 + 1, \ell_2 + 1)$	$R(f_1(\$L - 1), id_{max}(f_2(\$L - 1), \$L - 1), \ell_1 + 1, \ell_2 + 1)$
$desc$	$R(id_{min}(i_1, \ell_2), id_{max}(i_2, \ell_1) - 1, \ell_1 + 1, n)$	$R(f_1(\$L - 1), {f'}_2(\$L), \ell_1 + 1, n)$, ${f'}_2(\$L) = \max_{\ell\in[\ell_1+1,\$L]} \{f_2(\ell + 1)\}$

Table 5.1: Rewriting rules for region axis $\mathcal{M}(\pi, \alpha)$

5.4.1 Rewriting Rules

Given a region-axis step $s_1 = \pi::*$ and a navigational-axis step $s_2 = \alpha::\tau$, we use $\mathcal{M}(\pi, \alpha)$ to denote a function to rewrite s_1/s_2 into a single region-axis step; i.e.,

$$\mathcal{M}(\pi,\alpha)::\tau \;\equiv\; \pi::* \;/\; \alpha::\tau \tag{5.3}$$

The rewriting rules of the function $\mathcal{M}(\pi, \alpha)$(π is a region axis, $\alpha \in \{prec, foll, par, anc, child, desc\}$) are defined Table 5.1. For clarity, Table 5.1 shows the rewritings for both the basic as well as generalized form of the region axis π in the left and right columns, respectively. We illustrate the application of the rewriting rules with the following example.

Example 4.1 Consider the WP-query *foll::* / child::** w.r.t. the context node c in Fig. 5.2 which selects the gray irregular region. We explain how this WP-query can be rewritten into a single region-axis step using Table 5.1. Observe that the step *foll::** w.r.t. $c = (6, 3)$ is equivalent to $R(6, 6, 3, 3)$*::* / foll::**. This sequence is first rewritten into $R(id_{max}(6, \$L) + 1, m, 1, n)$*::**. Finally, $R(id_{max}(6, \$L) + 1, m, 1, n)$*::* / child::** is rewritten into $R(id_{max}(6, \$L - 1) + 1, id_{max}(13, \$L - 1), 2, 4)$. □

However, note that the generalized form still can not capture holey regions as illustrated by the following example.

Example 4.2 Consider the black holey region in Fig. 5.2. Applying the rewriting rules in Table 5.1 to rewrite *par::* / desc::* / par::** w.r.t. context node c, we have *par::** w.r.t. c $= R(6, 6, 2, 2)$; $R(6, 6, 2, 2)$ / desc::* $= R(4, 7, 3, 4)$; and finally, $R(4, 7, 3, 4)$ / par::* $= R(4, 7, 2, 3)$. However, the specification of $R(4, 7, 2, 3)$ for the black region is incorrect as it includes the extra node $(6, 3)$ corresponding to the "hole" between two selected nodes at level 3. □

5.5 Generalized Form with Constraints

To further enhance the expressiveness of the region axis to specify holey regions, we extend the region axis with additional *constraints* to enable specification of non-selected nodes that fall between selected nodes at the same level.

Recall from Example 4.2 the generalized region axis is unable to exclude the "hole" in the holey region in Fig. 5.2. One simple way to address this limitation is to augment the specification with additional constraints; in particular, Example 4.2 can be fixed by

adding a "height" constraint as follows:

$$\{v \in R(4, 7, 2, 3) \mid ht(v) \geq 1\}$$

which states each selected node in the region $R(4, 7, 2, 3)$ must have a height of at least one.

More generally, we extend the region axis specification to $R_C(i_1, i_2, \ell_1, \ell_2)$, where C denote a set of constraints. In other words,

$$R_C(i_1, i_2, \ell_1, \ell_2) = \{v \in R(i_1, i_2, \ell_1, \ell_2) \mid v \text{ satisfies } C\} \tag{5.4}$$

For notational simplicity, we sometimes omit the four parameters in the region axis specification when they do not matter and simply refer to the region axis as R_C. Furthermore, any reference to v in C refers to a selected node in R.

Consider a WP-query $q = s_1/\cdots/s_k$. Note that C is initialized to empty at the start of the rewriting of q; i.e., $q \equiv R_\emptyset(i, i, l, l)\ /s_1/\ \cdots/s_k$, where (i, l) represents a context node. As the rewriting progresses, the first height constraint is added to C when the first reverse-vertical-axis step is rewritten in q. More generally, consider the rewriting of $R_\emptyset$::* / α^i::τ, where $\alpha \in \{par, anc\}$ and $i \geq 0$. We have

$$R_\emptyset{::}{*}\ /\ \alpha^i{::}\tau \ \equiv\ R'_C{::}\tau \tag{5.5}$$

where $R' = \mathcal{M}(R, \alpha^i)$ and $C = \{ht(v) \geq i\}$. Here, the constraint in C states that a selected node in R' must have a height of at least i.

Conceptually, the region axis R_C now consists of two parts: the first part R defines the region of the selected nodes, and the second part C specifies the additional constraints that the selected nodes must satisfy. Therefore, we now also need to update C when merging a region-axis step π_C::* with another axis step α::τ. Thus, the rewriting of

π_C::* / α::τ into the single region-axis step $\pi'_{C'}$::τ consists of two modifications: (1) the updating of the region axis from π to π' which is handled by the function $\mathcal{M}(\pi, \alpha)$ defined in Table 5.1; and (2) the updating of the set of constraints from C to C' which is handled by a new function $\mathcal{U}(\pi_C, \alpha)$. In other words, we have

$$\pi_C{::}{*}\ /\ \alpha{::}\tau \ \equiv\ \pi'_{C'} \text{ where } \pi' = \mathcal{M}(\pi_C, \alpha) \text{ and } C' = \mathcal{U}(\pi_C, \alpha) \tag{5.6}$$

5.5.1 Height Constraint

As shown in the Example 4.2, height constraint is necessary to keep the region axis expression correct. To facilitate the expression, we have introduced the $R^k(v)$ for the region axis. Assume v represents the node (i, l), $R(v)$=$R(i, i, l, l)$ and $R^k(v)$=$R^k(i, i, l+k, l+k)$. With this notation,the query $q_2 = R(v(i, l))$ :: $*/par$ ::η_2 , where $l \geq 1$, is equivalent to

$$\{v^{'} \in R^{-1}(v){::}\eta_2 \mid ht(v^{'}) \geq 1\}$$

according to the semantic of the height constraint.

The height constraint is meant to make sure that each node selected must have one descendant node. Except this simple case for height constraint, we need to specify some constraints on the height of both the selected nodes as well as their ancestors.
Example 5.1 The query $q_3 = R(i, i, l, l)$ :: $*/par$ :: $*$ $/par$ :: $*$ $/chi$ ::η_3,where $l \geq 2$, is equivalent to

$$\{v^{'} \in R^{-1}(v){::}\eta_3 \mid ht(R^{-1}(v^{'})) \geq 2\}$$

□

Here, the constraint is specified on the parents of the selected nodes to make sure

that each selected node must have a descendant node that is two level below. Note that the constraint on $R^{-1}(v)$ is not equivalent to the following height constraint on v: $ht(v) > 1$. Therefore, it is necessary to support the height constraints on the ancestors of the selected nodes.

For a query with descendant axis, the height constraint becomes more complex and general.

Example 5.2 The query $q_4 = \{v \in R(i,j,l,h)::* \mid ht(v) \geq h\}$ / desc::η_4 is equivalent to

$$\{v^{'} \in R(i,j,l+1,maxl)::\eta_4 \mid \exists\, r \in [l - ht(v^{'}), h - ht(v^{'})], ht(R^{-r}(v^{'})) \geq h\} \tag{5.7}$$

□

Note that there is a little difference between the height constraints for layer axis and region axis. As an layer axis L^l represents some nodes that are l levels below the context node. However, it shows that the current nodes are in l level of the document tree for $R(i,j,l,l)$. Both height constraints specify that each selected node v must have some ancestor node u that is r level above v such that the height of w is at least h.

To formally express the height constraints, we have introduced the *cexp* to denote the integer expression defined in terms of integer constraints as well as "+" and "-". We have used exp_1 and exp_2 as the integer expressions defined in terms of $level(v)$ as well as "+" and "-". Then a height constraint ϕ on a selected node v can be specified as one of the following two forms:

F1 $ht(R^{cexp}) \geq exp_1$;

F2 $\exists\, i \in I\ \phi$, where $I = [exp_1, exp_2]$ is an range of consecutive integers, and ϕ is a height constraint of the form (F1).

Note that the above form of height constraint is the same as that for the layer axis and the only difference is that of exp_1 *and* exp_2 as in the above example 5.2.

5.5.2 Horizontal Constraint

More generally, height constraints are not enough to keep the evaluation of region axes correct for queries with horizontal axes.

Example 5.3 Let's consider the simple query $q_5 = R(v(i,l)) :: */par :: */pre :: \eta_2$. Before the evaluation of the preceding axis step , we have already produced one height constraint C. Then in the final step evaluation, we would get the result as $R(1, id_{min}(i, \$L)-1, 1, n)$ according to the rewriting rules in Table 5.1. However, the rewriting rules have left constraints out, which is not correct and we need to add the constraint as:

$$\{v \in R \mid \exists v', v' satisfies\ C; minId(v) < minId(v')\}$$

R represents the set of nodes selected by the XPath or the result set. □

Here, the new constraint specifies that any node in R must have one node that satisfies the constraint C on its right side, which actually expresses the semantics of the preceding axis step. We would call this kind of constraint as horizontal constraint (HC for short).

More generally, the horizontal constraint has two kinds of forms which are separately produced by preceding axis and following axis:

F3 $\{v \in R \mid \exists v', v' satisfies\ C; minId(v) < minId(v')\}$

F4 $\{v \in R \mid \exists v', v' satisfies\ C; minId(v) > minId(v')\}$

R is the selected node set and C is the constraint before. Note that C could be any form of F3 and F4 or height constraint F1 and F2.

As our notion of height constraints are similar to those introduced in [7], we have introduced similar forms $R^k(v)$ as the layer axis to facilitate the height constraint. Actually we have R(v)=R(i,i,l,l)and $R^k(v) = R(i,i,l{+}k,l{+}k)$ for v is the node(i,l). Then the query $q_2 =$ R(v(i,l))::*/par::η_2 , where $l \geq 1$, is equivalent to

$$\{v \in R^{-1}(v)::\eta_2 \mid ht(v) \geq 1\}$$

according to the semantic of the height constraint. Similarly, the query $q_4 = \{v \in R(i,j,l,h)::* \mid ht(v) \geq h\}$ / desc::η_4. This query is equivalent to

$$\{v^{'} \in R(i,j,l+1,maxl)::\eta_4 \mid \exists\ r \in [\delta(v,v_c) - h, \delta(v,v_c) - l], ht(R^{-r}(v^{'})) \geq h\} \tag{5.8}$$

5.5.3 Update of Constraints

According to the update of the constraint C_{update}=$\mathcal{U}(\pi_C,\alpha)$, each constraint $C^{'}$ should be composed of two parts represented as $C^{'} = C_{update} \cup C_{new}$; where C_{update} denotes the set of updated constraints in $C^{'}$, and C_{new} is the set of new constraints as defined by the above rules. Note that the updating for region axes adopts the similar idea as that for the layer axes. The updating of C to C_{update} is independent of the generation of any new height constraint in C_{new}. Moreover, each constraint in C is updated independently of the other updates.

As we both have vertical axes and horizontal axes, there are four kinds of cases for updating the constraints. To simplify the discussion, let us consider $p = R_C$::* / χ::η

1. C is height constraint and χ is one of the vertical axes. This case is similar as that for the layer axes and would be discussed in the following.

2. C is height constraint and χ is one of the horizontal axes. The updating for this case would produce the *HorizontalConstraint* as shown in the example 5.3.

3. C is *HorizontalConstraint* and χ is one of the horizontal axes. This case is the similarly as case 2 and produces *HorizontalConstraint*.

4. C is *HorizontalConstraint* and χ is one of the vertical axes. This case has two different situations. If χ is child axis or descendant axis, then constraint S would not be changed. If χ is parent axis or ancestor axes and assume the original *Horizontal Constrains* C as $\{v \in R \mid \exists t, minId(v) > t\}$, the updated constraint would be as follows:

   $$\{v \in R \mid \exists t, maxId(v) < t\}$$

   While if the C is $\{v \in R \mid \exists t, minId(v) > t\}$, then even if χ is parent axis or ancestor axes, the updating would not have any effect on the constraint C .

For case 1, we could generally adopt the updating rules presented in Table 5.2. While there is a special kind of queries we need to pay special attention to. We find that if the consecutive wildcard steps in the XPath have two or more “Up-Down” patterns, the updating could becomes complex. If the “Up-Down” patterns are all formed of childand paraxes, the updating is simple and we could adopt the rules in Table 5.2 for updating. If the “Up-Down” patterns are composed only of ancanddescaxes, the updating is also easy as shown in the following example.

Example 5.4 Considering the XPath “/anc::* /desc::* /anc::* /desc::*” and the constraint for the context of this XPath is “*ht(v)>t*”, then the constraint after evaluation the whole XPath is simply updated as $\{ht(r) > (t + 1)$, r is the root$\}$. □

Note that the updated constraint is just specified for the root and there is no need for us to check the selected nodes.

However, when the "Up-Down" patterns are composed of all of the four vertical axes, the updating could be complex, especially in the case that there is a descendant axis in the "Down" part of the pattern and there is no ancestor axis in the "Up" part of the pattern.

Example 5.5 Considering updating the height constraint for the XPath "par^2::* /desc::* /par^3::* /desc::*" and assuming the constraint for the context is $ht(v) > t$, the height constraint would be updated as $\{\exists\ r \in [l - ht(v^{'}), h - ht(v^{'})], ht(R^{-r+3}(v^{'})) \geq t + 2.\}$ (l and h are the level range for nodes reached by the step par^2::*). After the processing of the last step "desc::*", the height constraint is updated as $\{\exists\ r^{'} \in [l^{'} - ht(v^{''}), h^{'} - ht(v^{''})], \exists\ r \in [l - ht(R^{-r^{'}}(v^{''})), h - ht(R^{-r^{'}}(v^{''}))], ht(R^{-r+3}(R^{-r^{'}}(v^{''}))) \geq t + 2.\}$. ($l^{'}$ and $h^{'}$ are the level range for nodes reached by the step par^2::* /desc::* /par^3::*). In the height constraint,$R^{-r^{'}}(v^{''})$ would get a node $v^{'}$ in the height constraint before the updating. □

Though the height constraint for the consecutive "Up-Down" patterns in the XPath is complex as shown in the example, this kind of scenario seldom appears in real XPath queries.

5.6 Rewriting Wildcard Queries

In this chapter, we present our approach to optimize wildcard queries by minimizing the number of wildcard steps via query rewriting with the region axis. We first consider rewriting to eliminate NB*-steps, followed by rewriting to minimize B*-steps.

Axis	**Updating Constraint** $R(i,j,l,h)(S)$::* / χ::η ={$v^{'} \in R^{'}$ | $v^{'}$ satisfies S_{new} &S_{update} }	
	S={$ht(R^{-k}v(i,l)) \geq$ t}	S={$\exists$ $v_x(i_x,l_x) \in$ R_{con} satisfies $i_x \leq$ d & $ht(R^{-k}(v_x)) \geq$ t; $minId$(v)$\leq i_x$ }
prec	$\exists\, v^{''}(i^{''},l^{''}) \in$ R(i,j,l,h), satisfies $i^{''} \leq i$ & $ht(R^{-k}v^{''}) \geq$ t; $minId(v^{'}) \leq i^{''}$;	$\exists$ $v_x(i_x,l_x) \in$ R_{con} satisfies $i_x \leq$ d & $ht(R^{-k}(v_x))$ $\geq$ t; $minId$(v)$\leq$ i_x-1;(R_{con} is the context area in S)
foll	$\exists$ $v^{''}(i^{''},l^{''})\in$R(i,j,l,h)($i' = id_{max}(i,l)$) satisfies $i^{''} \geq$ $i^{'}$ &$ht(R^{-k}(v^{''}))$ $\geq$ t; $minId(v^{'}) \geq i^{''}$;	$\exists\, v_x(i_x,l_x) \in R_{con}$ $v^{''}(i^{''},l^{''})\in$R(i,j,l,h),satisfies $i_x \leq$ d & $ht(R^{-k}(v_x))\geq$t; $minId(v^{''}) \leq$ i_x & $minId(v^{'})$>$minId(v^{''})$;
par	$ht(R^{-k}v^{'}(i^{'},l^{'})) \geq$ t+1;	$\exists$ $v_x(i_x,l_x) \in$ R_{con} satisfies $i_x \leq$ d & $ht(R^{-k}(v_x)) \geq$ t; $minId$(v)$\leq i_x$;
anc	$\exists\, r \in [ht(v^{'})-h, ht(v^{'})-l]$, $ht(R^{-k}(v^{'})) \geq t+r$;	$\exists$ $v_x(i_x,l_x) \in$ R_{con} satisfies $i_x \leq$ d & $ht(R^{-k}(v_x)) \geq$ t; $minId$(v)$\leq i_x$;
child	$ht(R^{-k-1}v^{'}(i^{'},l^{'})) \geq$ t;	$\exists$ $v_x(i_x,l_x) \in$ R_{con} satisfies $i_x \leq$ d & $ht(R^{-k}(v_x)) \geq$ t; $minId$(v)$\leq i_x$;
desc	$\exists\, r \in [l-ht(v^{'}), h-ht(v^{'})]$, $ht(R^{-k-r}(v^{'})) \geq t$;	$\exists$ $v_x(i_x,l_x) \in$ R_{con} satisfies $i_x \leq$ d & $ht(R^{-k}(v_x)) \geq$ t; $minId$(v)$\leq i_x$;

Table 5.2: Basic updating Rules

5.6.1 Eliminating NB*-Steps

The rewriting algorithm (shown in Algorithm 2) is rather straightforward: it traverses the query tree top-down and rewrites away any wildcard steps encountered by applying the rewriting functions $\mathcal{M}(.,.)$ and $\mathcal{U}(.,.)$.

Algorithm 2 `Eliminate-NB*-Steps` (q)

Input: query q, where each step $s_j = \alpha_j{::}\tau_j$

1: **for** (each step s_i visited by a top-down traversal of q) **do**

2:     **if** (the parent-step s_j of s_i is a NB*-step) **then**

3:         $R' = \mathcal{M}(\alpha_j, \alpha_i)$

4:         $C' = \mathcal{U}(\alpha_j, \alpha_i)$

5:         replace s_i/s_j in q with $R'_{C'}{::}\tau_i$

6: **return** q

Example 6.1 Consider the query q_4 in Fig. 5.3(d). The NB*-step *child::** in q_4 be rewritten using the region axis to the query q_5 in Fig. 5.4(e), where the context node for evaluating the NB*-step is assumed to be (i, l). □

5.6.2 Minimizing B*-Steps

In contrast to NB*-steps, the elimination of B*-steps is more complex and requires query rewriting with additional type of constraints (to be discussed in Section 5.6.3). In this section, we present a simpler and more efficient first approach to minimize B*-steps in a query. Although this approach might not completely eliminate all B*-steps, its advantage is that can be more efficiently implemented.

The rewriting algorithm (shown in Algorithm 3) tries to minimize the B*-steps in an input query by applying rewriting rules to convert B*-steps to NB*-steps.

We have identified seven rewriting rules ($R1$ to $R7$), each of which is represented by a column in Table 5.3, to rewrite an input query q (shown in the top row) to an equivalent transformed query q' (shown in the bottom row) such that the B*-step s in

Algorithm 3 `Minimize-B*-Steps` (q)
Input: query q

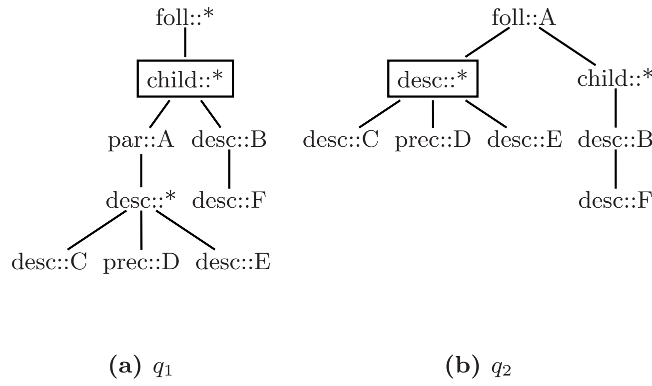
1: **while** (q can be transformed by some rule Ri from Table 5.3) **do**
2: apply rule Ri to rewrite q
3: **return** q

q (shown as a boxed node) is transformed into a step s' in q' (also shown as a boxed node), where s' has one fewer child-step than s. For simplicity, we use a triangle symbol to denote a subtree of zero or more steps in a query.

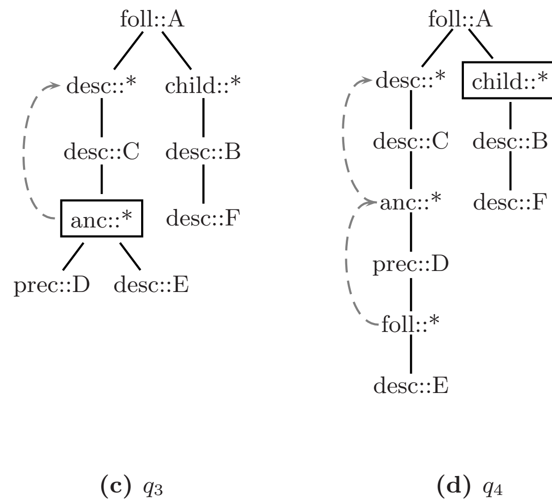
For example, rule $R1$ rewrites q into q' such that the B*-step s = desc::* in q is converted into the step s' = child::* in q'. If s had only one child-step (i.e., the white subtree is actually empty), then the rewriting has eliminated the B*-step in q; otherwise, q' (with a reduced fan-out for its B*-step) could be further transformed by applying other rules.

Note that in rule $R2$, the function $f(\tau, \tau')$ returns a wildcard if both τ and τ' are wildcards; otherwise, if only τ is a wildcard, then τ' is returned (and vice-versa). If both τ and τ' are distinct element names, then the query's result is empty. Due to space constraint, we omit the correctness proofs for the rewriting rules.

Example 6.2 Consider the query q_1 in Fig. 5.3(a). The B*-step *child::** can be rewritten away by applying rule $R2$ to obtain the equivalent query q_2 in Fig. 5.3(b) which has only NB*-steps. □



(a) q_1 **(b)** q_2



(c) q_3 **(d)** q_4

Figure 5.3: Example of query rewriting with composite axes

5.6.3 Eliminating B*-Steps

In this section, we present an approach to completely eliminate B*-steps by rewriting using a new type of constraints called *inclusion constraints*. In contrast to Algorithm `Minimize-B*-Steps` discussed in the previous section, which only minimizes the number of B*-steps, the new approach can completely reduce all B*-steps to NB*-steps; however, the tradeoff of this aggressive approach is that it incurs run-time overhead to enforce the inclusion constraints. We first illustrate the key idea of this method with an example.

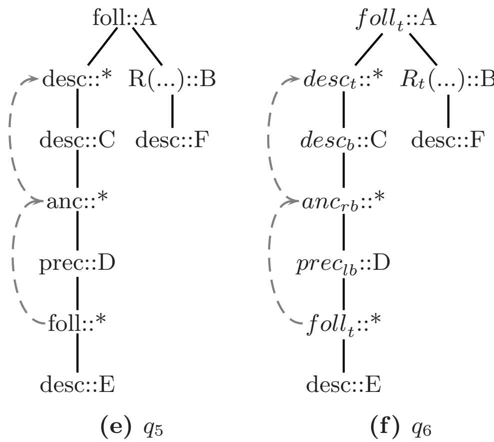


Figure 5.4: Example of query rewriting with composite axes, where $R(...) = R(id_{min}(i, l),$ $id_{max}(i, l) - 1, l + 1, n)$

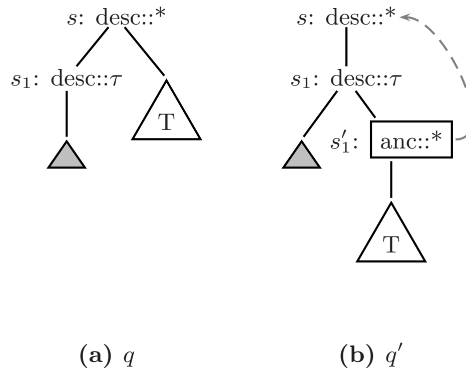


Figure 5.5: Eliminating B*-steps with inclusion constraints

Example 6.3 Consider the query q in Fig. 5.5(a) with a B*-step labeled as s and one of its child-step labeled as s_1. Observe that the B*-step s can not be removed by the rewriting rules in Table 5.3. To convert step s into a NB*-step, we can push the subtree rooted at one of its child-steps, say T (as shown in Fig. 5.5(b)), to become part of the subtree of another of its child-step (s_1 in this example) through a new step $s_1' = anc::*$ (shown as a boxed node), where the axis of s_1' is the reverse of that of s_1. This rewritten query q' is clearly a more general query than q. To preserve the equivalence of q' and q, we need to specify an *inclusion constraint*, denoted by $s_1' \sqsubseteq s$ (indicated in Fig. 5.5(b) by a dashed arrow from s_1' to s), to ensure that each data node selected by s_1' is included

in those selected by s. □

As the above example suggests, a B*-step s with k child-steps $s_1, \cdots, s_k$ can be "linearized" into a NB*-step by sequentially pushing each child-step s_i into the subtree of child-step s_{i-1}, $i \in [2, k]$. For each step s_i being pushed, a new widlcard-step s'_{i-1} is added so that s_{i-1} is the parent-step of s'_{i-1}, and s'_{i-1} is in turn the parent-step of s_i. Corresponding to each newly introduced wildcard-step s'_{i-1}, a new inclusion constraint given by $s'_{i-1} \sqsubseteq s'_{i-2}$ is also added[1].

Thus, in general, each B*-step s (with k child-steps) can be converted into a NB*-step by adding $k - 1$ new NB*-steps together with their corresponding inclusion constraints. The newly added NB*-steps can subsequently be rewritten away using Algorithm `Eliminate-NB*-Steps`; details of how this can be performed efficiently together with inclusion constraint checking is discussed in Section 6. The overall algorithm is given in Algorithm 4.

Example 6.4 Consider the query q_2 in Fig. 5.3(b), where the B*-step *desc::** can not be reduced to a NB*-step using the previous approach. One approach to eliminate this B*-step is shown in Fig. 5.3(c) and Fig. 5.4(d). First, as shown in Fig. 5.3(c), a new NB*-step *anc::** is added as child-step of *desc::C* (with an inclusion constraint indicated by the dashed arrow from *anc::** to *desc::**); the other two child-steps of *desc::** are relocated to become child-steps of the new NB*-step. The transformation from q_2 to q_3 reduces the branching factor of the wildcard-step. Finally, as shown in Fig. 5.4(d), a similar procedure is applied to rewrite q_3 to q_4 generating yet another NB*-step and inclusion constraint. Now q_4 has only NB*-steps. □

[1]Note that s'_0 refers to the initial B*-step s itself.

Algorithm 4 `Eliminate-B*-Steps` (q)

Input: query q

1: initialize the set of inclusion constraints I to be empty
2: **for** (each step s_i visited by a top-down traversal of q) **do**
3:     **if** (s_i is a B*-step) **then**
4:         let $L = \{s_{i,1}, \cdots s_{i,k}\}$ be the child-steps of s_i
5:         choose a step $s_{i,j} \in L$
6:         let $s'_{i,j}$ be a new wildcard-step with axis equal to the reverse of the axis of $s_{i,j}$
7:         insert $s'_{i,j}$ to become a child-step of $s_{i,j}$
8:         **for** (each $s_{i,k} \in L, s_{i,k} \neq s_{i,j}$) **do**
9:             relocate $s_{i,k}$ to become a child-step of $s'_{i,j}$
10:             add a new constraint $s'_{i,j} \sqsubseteq s_i$ into I
11: **return** (q, I)

R1	**R2**	**R3**	**R4**
α::τ desc::* par::τ'	α::τ child::* par::τ'	α::τ par::* par::τ'	α::τ child::* anc::τ'
α::τ descos::τ' child::*	α::$f(\tau, \tau')$ child::*	α::τ par^2::τ' par::*	α::τ ancos::τ' child::*
R5	**R6**	**R7**	
α::τ par::* anc::τ'	α::τ anc::* par::τ' anc::τ''	α::* child::τ	
α::τ anc^2::τ' par::*	α::τ anc::* par::τ' ancos::τ''	α::* child::τ par::*	

Table 5.3: Rewriting Rules to eliminate branching wildcard steps

Chapter 6

Rewriting with Composite Axes

In this chapter, we present the rewriting algorithm based on the composite axes: SNAs and RAs. To efficiently implement the rewriting algorithm, we would introduce some interesting implementation issues in the later section.

6.1 Rewriting Algorithm

Putting together all the four rewriting algorithms that we have introduced, Fig. 6.1 shows the sequence of rewriting an input query q into q'.

The first step tries to convert as many of the B*-steps in q as possible into NB*-steps by applying the simple rewriting rules in Table 5.3. If this step fails to completely eliminate all the B*-steps, the second step applies a more aggressive and complex rewriting method using inclusion constraints. After all the B*-steps have been reduced to NB*-steps, we use the third step to optimize the evaluation of the remaining steps by replacing them with more efficient, specialized axis steps when applicable. Finally the

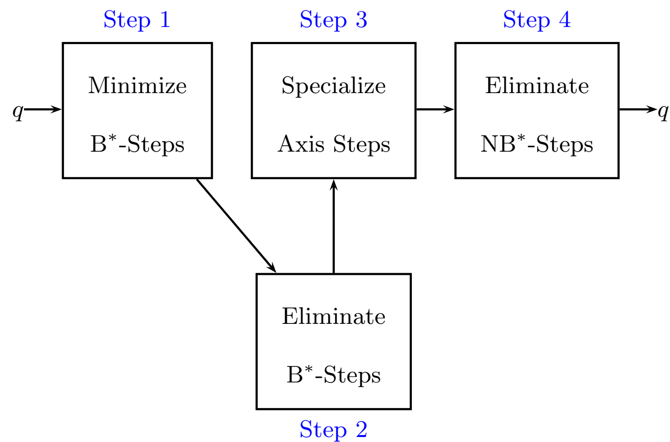


Figure 6.1: Rewriting with composite axes

fourth step eliminates the NB*-steps by merging with the non-wildcard steps in q. We could apply the partial loading strategy when wildcard steps are removed.

Note that,B*-steps are the most complex part of the query for the evaluation, therefore, our algorithm transfers B*-steps into NB*-steps to prepare for further optimization. The transferring algorithm generally introduces lots of the inclusion constraints for checking in the evaluation period, so `Minimize-B*-Steps` is applied first to minimize the inclusion constraints checking resulting from the following B*-steps eliminating algorithms by minimizing the B*-steps first. After the first two steps, only NB*-steps are left. Our algorithm only specializes the non-wildcard axis steps, as all the wildcard steps would be rewritten by the our final step. Note that we could better specialize axes steps if we apply the axis specializing algorithm before wildcard steps are rewritten, as we could generally exploit more usefully information to further optimize our specialized axes. However, we could not make any specialization based on the region axis resulting from rewriting the wildcard steps. As shown in the Example 1.2, we could specialize the step "foll::a" as $foll_t :: a$ according to the specializing algorithm. However, if we rewrite the wildcard steps first, we would get the "foll::a/R(...)", where we can not apply the

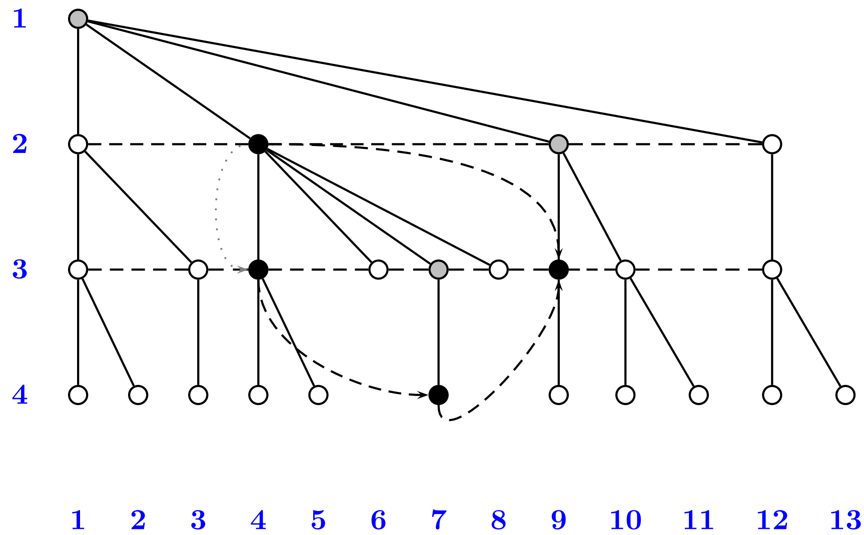


Figure 6.2: Data structure for region axis

SNA for "foll::a". Note that the specialized axis step would not affect the wildcard step rewriting, as the specialized axis is for locating the nodes that we need to fetch for a step in a certain region. This region is not affected by any specialized axis step.

The running example that we have been referring to in Figs. 5.3 and 5.4 show how a query q_1 can be transformed into q_6. Clearly, each of these NB*-steps can be eliminated by composing with its child-step. However, note that the inclusion constraints associated with these NB*-steps still need to be checked at evaluation time.

6.2 Implementation Issues

In this section, we describe some of the interesting implementation details related to query evaluation using the proposed composite axes.

6.2.1 Checking of Inclusion Constraints

The main challenge in the evaluation of region axis is the run-time checking of inclusion constraints (if they are present). This, however, can be efficiently handled by simply performing "intersection" operations on the region specification expressions corresponding to the wildcard steps without actually having to physically evaluate the wildcard steps to identify the matching data nodes. For example, consider query q_3 in Fig, 5.3(c), where there is an inclusion constraint stating that *anc:** $\sqsubseteq$ *desc::**. Assume that there is a set of context nodes S for evaluating *desc::**. The checking of the inclusion constraint proceeds as follows. Since there is an inclusion constraint involving step *desc::**, we first "evaluate" (at the rewriting level) step *desc::** w.r.t. each context node $(i, l) \in S$ by simply deriving a region expression that corresponds to $R(i, i, l, l)$ / *desc::** thereby obtaining a set E of region expressions, where $|E| = |S|$. Next, we evaluate the step *desc::C* w.r.t. S by physically evaluating the region-axis expression corresponding to *desc::** / *desc::C* to obtain a set of output nodes S', which will serve as context nodes for step *anc::**. At this point, since we need to check the inclusion constraint (which also involves step *anc::**), we again perform a rewriting-level evaluation, this time for *anc::** w.r.t. each of the context nodes in $(i', l') \in S'$ to obtain a region-axis expression that corresponds to $R(i', i', l', l')$ / *anc::** thereby obtaining another set E' of region expressions, where $|E'| = |S'|$. With the two collections of region expressions E and E', we can check the containment constraint by taking each $e' \in E'$ and checking if there exists some $e \in E$, where the intersection of the two region expressions e and e' are non-emtpy. If it is indeed non-empty, the constraint is satisfied, and the qualifying region expression $e \cap e'$ is used later as a context node to evaluate the two child-steps *prec::D* and *desc::E*.

To summarize, the above inclusion constraint checking idea is efficient as it does

not require physical evaluation of any wildcard steps; instead, wildcard steps are logically evaluated by deriving a region expression (w.r.t. each context node) which can be efficiently achieved using the region-axis rewriting function.

6.2.2 Evaluating Region Axis

After eliminating the wildcard steps, the XPath would be rewritten with the form of region axis. Therefore, it is critical to efficiently evaluate the region axis. As the XPath query evaluation is based on main-memory representation of the XML document tree, we have formed a new crossed link structure as well as the tree structure in the one SAX based pass of the document tree as shown in Figure 6.2. For each node, it has the link to its parent and child vertically, if present, as shown with solid lines in Figure 6.2. At the same time, each node links to the first node on its left and right, if present, within the same level, as shown with horizontal dashed line. Thus the whole tree becomes a cross linked structure. To facilitate the evaluation, all the leaf nodes are indexed with a simple array. Take the evaluation of the basic region axis form $R(4,7,3,4)$ in Fig 6.2 for example, we would use the array index to locate the start position at the node $n(4,4)$ and end position at the node $n(7,4)$ with $O(1)$ complexity. Then, we could go up from the $n(4,4)$ and $n(7,4)$ separately along the vertical links to reach levels range [3,4] in the region with $O(lg(h))$(h is the height of the tree). In this example, we have already started from the level 4. Therefore, the nodes in level 4 between the node $n(4,4)$ and the node $n(7,4)$ would be returned according to the horizontal links. Similarly, the region part(the nodes) in level 3 could also be returned. Even for the general form of region axis, the complexity for locating the region is $O(h \times lg(h))$ complexity. Clearly, it is known that the height of XML document tree is generally low.

6.2.3 Optimized Partial Data Loading

One of the advantages of eliminating wildcard steps in a query is that it enables an efficient query evaluation strategy that can selectively load only a portion of the data nodes into main memory based on the set of element labels that actually appear in the input query. This can alleviate the scalability problem faced by XPath query evaluators that uses a main-memory representation of the input XML data (e.g., DOM-based implementations). We refer to the conventional approach of loading the entire document into main-memory for evaluation as the *complete loading approach* and refer to the alternative, more efficient approach (that is feasible for wildcard-free queries) as the *partial loading approach*. To evaluate our region axis, we can not just load the nodes that have the labels in the query, which could result in some ambiguous problems.

Example 2.1 Assume the black nodes in the Figure 6.2 have the tag a appearing in the query q, and assume one context for the "$desc :: a$" in the q is "$R(4, 9, 2, 4) :: *$", when apply the evaluation of the step "$desc :: a$" with the context according to rules in Table5.1, we need the information of the node $n(9, 2)$ to determine the new region for search nodes with tag a. Clearly, we have no information of $n(9, 2)$ without loading it into the memory. □

Actually, what we need to load into memory includes two parts: all the nodes with the tags appearing in the query and all the nodes that are ancestors of some nodes loaded into the memory. As shown in the Figure 6.2, all the black nodes are necessary to load and we also need to load the grey nodes into the memory. It is obvious that still lots of nodes would not be loaded into the memory compared with the completely loading strategy.

However, in the following case, we can not apply our partial loading strategy: *when there is height constraint C in the XPath followed by some horizontal axes.* This is because that we need some node n that satisfying the C to produce the new horizontal constraint, however, it is rather possible n does not have the tag appearing in the query, in which case, the query evaluation would not be exact correct without loading all the nodes into the memory.

Example 2.2 In the evaluation of the XPath "desc::a /par::* /anc::* /chi::* /foll::*/prec::b", we would need a height constraint "C" before the "foll::*" step. However, according to the height constraint updating rules for the horizontal axes, we need to get some node "n" that satisfies "C", which node may be not "a" or "b". In this case, the partial loading strategy would result in the problem of finding "n". □

Therefore, in the implementation, we need to make a simple check whether the height constraint (if any) would appear before the horizontal constraint, in which case we would not apply the partial loading strategy. Note that, this check is conducted in the procedure for rewriting the XPath query. In the query rewriting, whenever we find horizontal axes appearing in the query, we would check if there is some height constraint C produced before this horizontal axis.

6.2.4 Implementing the SNA

To process the SNA, we have built an additional structure that links the nodes with the same tag according to the document order. That link contains two part: each node links to its first descendant node with the same tag ($descendant - link$) as the gray dotted curves shown in figure 6.2 and each node links to its first following node with the same tag ($following - link$), as the dark dashed curves shown in Figure 6.2. Note that, all the

dark nodes in Figure 6.2 have the same tag. For example, each node with the tag name "a" are linked with its descendent that has the tag name "a" and also has link to its first following node with tag "a". Note that this structure could be efficiently implemented in the one pass of the document.

With the linked structure, we could efficiently evaluate the specialized axes steps. To evaluate the α_{rb}, we need to search along the "$following - link$" for the first node with tag "α" that is covered by the desired region axis. Then we need to check whether the current node has some descendant node with the same tag in this region along the "$descendant - link$". This step would cost $lg(n)$ assuming the link for "α" contains n nodes. Similarly we could evaluate the α_{lb}. It is a little more complex to evaluate the α_t and α_b, however, the basic idea is the similar, search for the nodes in the region covered by the axis along the $following - link$ and then check its descendant along the $descendant - link$ and fetch all the nodes that satisfy the requirement of the specialized axis step.

Example 2.3 In Figure 6.2, assume the region is $R(4, 9, 2, 4)$ and the black nodes have the desired tag name α. Then the α_{rb} returns $n(4, 3)$; α_{lb} returns $n(9, 3)$; α_b returns $\{n(7, 3)$, $n(4, 3)$, $n(9, 3)\}$ and α_t returns $\{n(7, 3)$, $n(4, 2)$, $n(9, 3)\}$ according to their definitions. $\square$

Chapter 7

Performance Study

To verify the effectiveness of our proposed rewriting optimizations, we conducted an experimental performance study using the XMark benchmark data [30]. Our results indicate that our proposed optimizations achieve a significant performance improvement over traditional evaluation methods for XPath queries.

7.1 Experimental Setup

Data Sets: We used the XMark benchmark data [30] for our experiments and generated four data files of size 70MB, 110MB, 165MB, 240MB, and 300MB. The number of element nodes contained in these files are, respectively, about 1.1 million, 1.7 million, 2.4 million, 3.6 million and 4.8 million.

Queries: We generated XPath queries using the XMark benchmark schema by varying the following parameters: the number of linear wildcard steps, the number of branching wildcard steps and non-wildcard branching steps.

Experiment 1. To investigate the effect of the number of consecutive NB*-steps (denoted by n_{wc}) in the linear XPath query, we have used the XPath query Q1: "*desc::site /desc::*(1) /prec::*(2) foll::*(3)/desc::personref*"; for $n_{wc} = k$, $k \in [0, 3]$, and we added the NB*-steps gradually according to the order shown in the bracket. For example, the query for $n_{wc} = 1$ is "*desc::site/desc::** /desc::personref". Note that there would be no height constraint in the evaluation of Q1.

Experiment 2. To examine the effect of the height constraint for the query evaluation with the region axis, we have used another query Q2: "*desc::site/desc::* /foll::* /anc::*/desc::personref*". Q2 is similar as Q1 except using different axes steps, which introduce the height constraint in rewriting XPath queries with region axes.

Experiment 3. In this experiment, we have examined the effect of the number of the branching non-wildcard steps (denoted by n_b). We gradually produce the experiment queries from Q3: "*desc::personref/prec::* /chi::* /foll::person [pre::item/desc::mail] /anc::site*" by varying n_b; for $n_b = 0$, the query is formed from $Q3$ by eliminating all its predicate steps, and for $n_b > 0$, n_b copies of $Q3$ are concatenated to form the query.

Experiment 4. As we know that RA only works for optimizing the wildcard steps in the XPath, while SNA could be applied to the general axes steps. To clearly show the effect of the SNA for the evaluation of XPath queries especially for the non-wildcard steps, we have examined the evaluation performance of the query Q4, which is a modification of Q3 to have more non-wildcard axes steps and complex non-wildcard branching steps rather than wildcard steps: "*desc::personref/prec::* /pre::item/prec::time /foll::person [desc::email] [foll::closed _auction /desc::price] /anc::site*" by varying the n_b. With the first query resulting from Q4 as "*desc::personref/prec::* /foll::person*" having $n_b = 0$, for $n_b > 0$, n_b copies of $Q4$ are concatenated to form the query.

Experiment 5. We examine the effect of the number of B*-steps (denoted by n_{bwc}) in experiment 5 with the query Q5: "*desc::personref /foll::* /prec::people /chi::* [chi::gender] [foll::city][prec::item] /anc::site*". Q5 is similar to Q3 except that the wild-card steps in Q5 are B*-step and there is no B*-step in Q3. We generate other experiment queries with Q5 according to the following approach: for $n_{bwc} = 0$, the query is formed from $Q5$ by eliminating all its predicate steps, and for $n_{bwc} > 0$, k copies of $Q5$ are concatenated to form the query.

Experiment 6. To examine the scalability of our approaches, we have evaluated a simple query Q6 "desc::site /desc::* /desc::keyword" with the data of varied size from 70MB to 300MB. The results are shown in experiment 6.

Algorithms. We compared the various proposed methods L^{α}_{β}, where $L \in \{P, F\}$ indicates whether partial loading (P) or full loading (F) is being used; α indicates whether SNAs are being used ($\alpha = sna$ if SNAs are used; otherwise, α is empty); and β indicates whether RAs and rewriting optimized are being used ($\beta = ra$ if only RAs are used, $\beta = rw$ if only rewriting is used, $\beta = ra + rw$ if both rewriting and RAs are used, or β is empty, otherwise).Note that if $L = P$, then β must contain ra. The conventional evaluation, which is denoted by F. is implemented based on MinContext in [12].

The performance metric used is the response time which comprises of two components: the document parsing time as well as the evaluation time. The *parsing time* includes the time to parse and load the data into main memory (either partially or fully). The *evaluation time* refers to the actual time required to evaluate the input query using the loaded data.

Our experiments were conducted on a 2.6 GHz Intel Pentium IV machine with 1

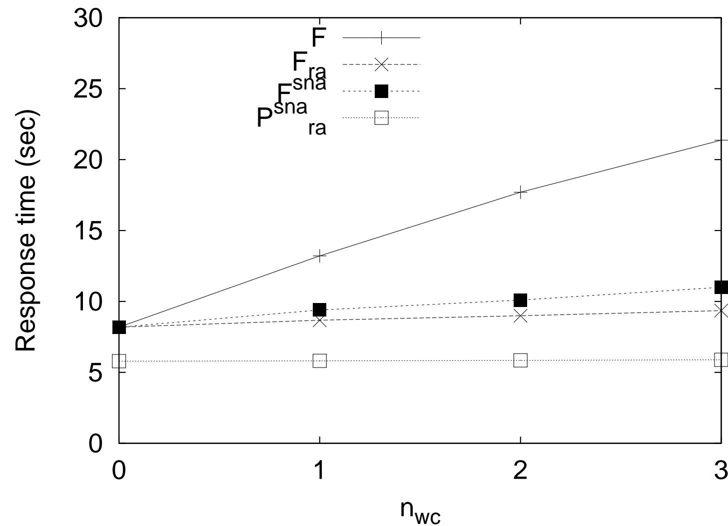


Figure 7.1: Varying n_{wc}

GB of main memory running Windows XP; and all algorithms were implemented using Java.

7.2 Experimental Results

Experiment 1. Fig. 7.1 compares the performance as n_{wc} is varied. For the two methods that used RAs to eliminate wildcard steps (i.e., P_{ra}^{sna} and F_{ra}), their performance is independent of n_{wc} demonstrating the effectiveness of rewriting away wildcard steps, with P_{ra}^{sna} giving the best performance. Comparing P_{ra}^{sna} and F_{ra}, F_{ra} improves over F (the conventional approach) by a factor of up to 1.9, while P_{ra}^{sna} improves over F even more by a factor of up to 3.0. The main reason for the improvement of P_{ra}^{sna} is due to partial data loading. The parsing time turns out to be the dominant component of the total evaluation cost for both P_{ra}^{sna} and F_{ra}, the partition in the total response time keeps nearly constraint with increasing the consecutive wildcard steps. This is due to the effect of the our approach in rewriting the XPath. While for F, the parsing time is about 90%

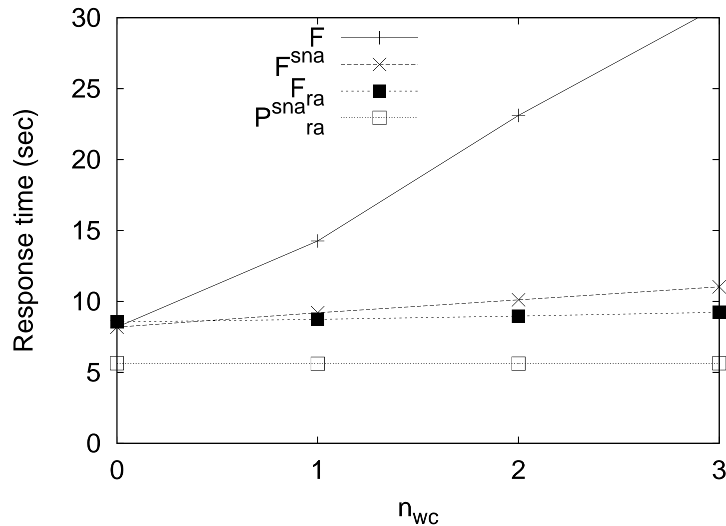


Figure 7.2: Varying n_{wc}

of the total cost when $N_{nc} = 0$ and it reduces to about 40% for experiment when $N_{nc} = 3$ due to the higher querying cost for queries with wildcard steps. The use of SNAs for this simple query turns out to be not too significant; in fact, we observed similar performance for both P^{sna}_{ra} and P_{ra} (not shown on the graph). Comparing the performance of F_{ra} and F^{sna}, the results indicate using RAs is more effective than SNAs since the savings from eliminating wildcard steps are relatively more significant. Note that the B*-step minimization step S1 was not applied here since query $Q1$ does not have any branching wildcard steps.

In experiment 1, we have checked the memory cost for the F and P. The F_{ra} in the evaluation of Q1 loads around 0.08M nodes into the memory and the total memory cost is about 52MB, while the P_{ra} loads about 1.1M nodes into the memory and it takes about 160MB of total memory size. Therefore, the partially loading strategy could dramatically reduce the number of nodes loaded into the memory, thus reducing the total memory cost for the query evaluation.

Experiment 2. The evaluation performance of Q2 is shown in Fig. 7.2. Note that, the evaluation time cost for Q2 has the similar trend as that for Q1, which tells that the height constraint in the evaluation of Q2 has not affected much of the total performance of the evaluation.

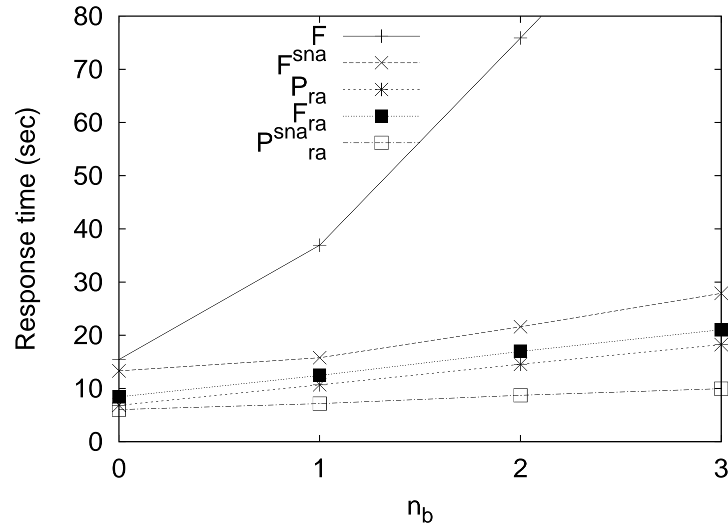


Figure 7.3: Varying n_b

Experiment 3. Fig. 7.3 compares the performance as n_b is varied. Similar to the results in Fig. 7.1, the methods that uses RAs performed better than those that do not, with P_{ra}^{sna} giving the best performance. Again here, the conventional approach F has the worst performance with a response time of about 120s when $n_b = 3$ (not shown on the graph).

Experiment 4. In this experiment, we emphasize on examining the efficiency of SNA for the non-wildcard steps in the XPath. Clearly, F^{sna} achieves much better performance than the F_{ra} and P_{ra} for the queries with fewer wildcard steps as for Q4 in Fig. 7.4. However, considering query evaluation performance for Q3 in Fig. 7.3 , which has many wildcard steps and fewer non-wildcard steps, F_{ra} and P_{ra} work better than the F^{sna} does. Note that in in Fig. 7.4, the query for step 0 has fewer non-wildcard

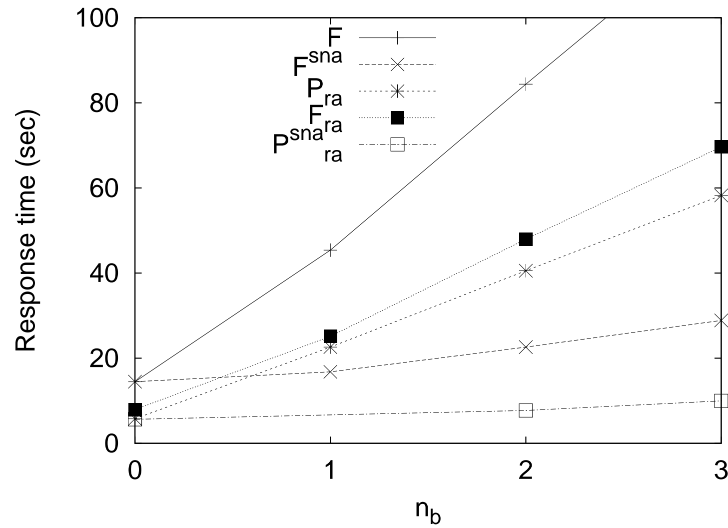


Figure 7.4: Varying n_b

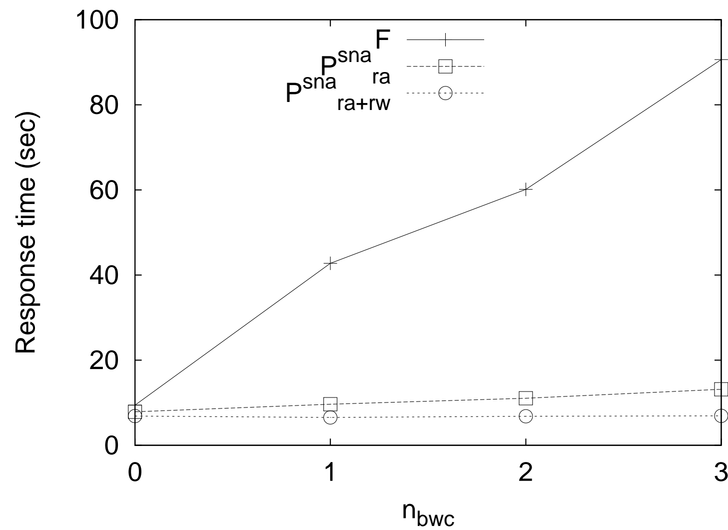


Figure 7.5: Varying number of branching wildcard-steps

step which enables F_{ra} and P_{ra} achieve better performance than F^{sna}. Therefore, we could conclude that using F_{ra} or P_{ra} for XPath queries with more wildcard steps achieves better performance and could beat F^{sna}, while on the other hand, F_{ra} or P_{ra} will not show good performance for XPath queries with fewer wildcard steps, for which cases F^{sna} achieves better performance. Since both the SNAs and RAs could benefit XPath query evaluations, we combine these two composite axes into the P^{sna}_{ra}, which has shown

the best performance in both Fig. 7.3 and Fig. 7.4.

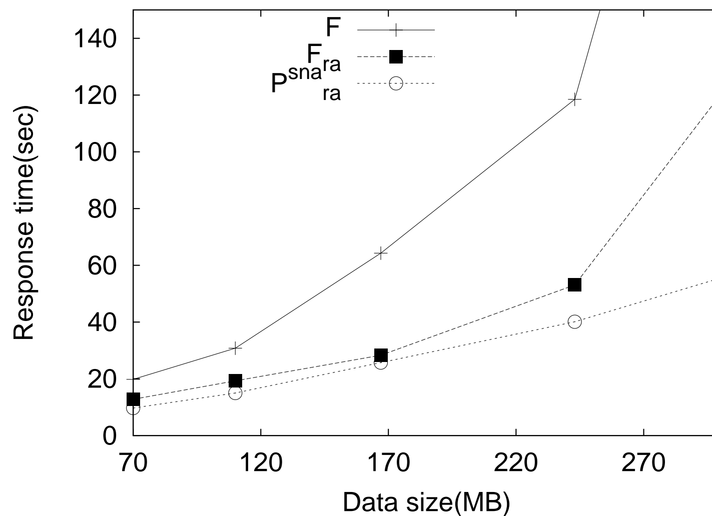


Figure 7.6: Varying input data size

Experiment 5. Fig. 7.5 compares the performance when n_{bwc} is varied. Again here, we see that P^{sna}_{ra} outperforms F significantly. Since $Q5$ has B*-steps, the rewriting optimization (step S1) to minimize B*-steps becomes applicable. Comparing P^{sna}_{ra} with P^{sna}_{ra+rw}, we observe that the additional use of step S1 (i.e., P^{sna}_{ra+rw}) improves P^{sna}_{ra} slightly. This is due to the fact that for P^{sna}_{ra}, the loading time is the dominant component of the response time.

Experiment 6. Finally, Fig. 7.6 compares the cost of evaluating the linear query $Q6$ (with a single NB*-step) as the data size varies. Observe that the performance gap between P^{sna}_{ra} and F widens with increasing data size. For the largest data file, the response time for F is actually 243s (not shown on the graph). The results also demonstrate that P^{sna}_{ra} is scalabile compared to F. Similar to the case for query $Q1$ (in Fig. 7.1), the performance of P^{sna}_{ra} is actually similar to P_{ra} (not shown on the graph) as the effectiveness of SNAs is limited for the simple query.

Chapter 8

Conclusions

In this thesis, we have presented a novel approach to optimize the evaluation of XPath queries by rewriting an XPath query using a set of composite axes: specialized navigational axis(SNA) and region axis(RA). Using the composite axes can not only reduce the number of query steps, but they are also more amenable to efficient implementations.

Each SNA is essentially a composition of a traditional navigational axis with a pruning operator into a single axis. This integrated axis can be evaluated much more efficiently than sequentially evaluating each of the composed steps. This optimization is particularly effective for "far-reaching" axis steps that evaluates to a large data area and/or involving wildcard nodetest. With the help of SNA, much fewer elements of the XML document would be accessed in the evaluation of the XPath queries. From another perspective, SNA applies the pruning optimizations ahead of the real evaluation of the axis step.

We have proposed the fence labeling scheme for tree structure XML document to be split horizontally. Based on the fence and level, each XML tree could form a grid.

Therefore, region axis is proposed on the base of grid to enable wildcard steps in a query to be eliminated, which results in very efficient query evaluation. For the region axis, we have presented the basic forms and general forms to express a region in the XML tree structure. We have also designed our algorithms to rewrite XPath queries using region axes. To keep the XPath rewriting maintain the equivalence, constraints are introduced to combine with the region axes and a set of constraint updating rules are also provided.

With the composite axes SNA and RA, XPath query rewriting approach is designed, in which both the SNA and RA are fully exploited according to the property of XPath queries. The effectiveness of all these optimizations and composite axes for the XPath rewriting are demonstrated by our experimental results.

Our current work in this thesis still can not handle all the axes in the XPath, such as sibling-related axes, which could generally break the region. For example, it is difficult to use one regular region to express the result area represented by the XPath "desc::*/foll-sibling::*". As part of our future work, we intend to further explore the query rewriting with composite axes for a larger fragment of XPath that includes sibling-related axes. We are expecting to combine more complex constraints with the region axis to eliminate the sibling-related axes using the rewriting techniques.

Bibliography

[1] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural joins: a primitive for efficient XML query pattern matching. In *ICDE*, pages 141–152, 2002.

[2] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of tree pattern queries. In *SIGMOD*, pages 497–508, March 2001.

[3] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Minimization of tree pattern queries. In *SIGMOD*, pages 497–508. ACM Press, 2001.

[4] Michael Benedikt, Wenfei Fan, and Gabriel M. Kuper. Structural properties of XPath fragments. In *ICDT*, pages 79–95, 2003.

[5] Scott Boag, D. Chamberlin, Mary Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon. *XQuery 1.0: An XML query language.* http://www.w3.org/TR/xquery, November 2003.

[6] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD*, pages 310–321, 2002.

[7] Chee-Yong Chan, Wenfei Fan, and Yiming Zeng. Taming XPath queries by minimizing wildcard steps. In *VLDB*, page 156, 2004.

[8] James Clark. *XSL Transformations (XSLT) 1.0*. http://www.w3.org/TR/xslt, November 1999.

[9] Wenfei Fan, Chee-Yong Chan, and Minos Garofalakis. Secure XML querying with security views. In *SIGMOD*, page 587, 2004.

[10] Sergio Flesca, Filippo Furfaro, and Elio Masciari. On the minimization of XPath queries. In *VLDB*, pages 153–164, 2003.

[11] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, pages 95–106, 2002.

[12] Georg Gottlob, Christoph Koch, and Reinhard Pichler. XPath query evaluation: improving time and space efficiency. In *ICDE*, pages 379–390, 2003.

[13] Torsten Grust. Accelerating XPath location steps. In *SIGMOD*, pages 109–120, 2002.

[14] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase join: teach a relational DBMS to watch its (axis) steps. In *VLDB*, pages 524–525, 2003.

[15] Torsten Grust, Maurice van Keulen, and Jens Teubner. Accelerating XPath evaluation in any RDBMS. *TODS*, 29(1), 2004.

[16] Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Optimized translation of XPath into algebraic expressions parameterized by programs containing navigational primitives. In *WISE*, pages 215–224, 2002.

[17] Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Optimized translation of xpath into algebraic expressions parameterized by programs containing navigational primitives. In *WISE*, pages 215–224, 2002.

[18] Jan Hidders and Philippe Michiels. Efficient xpath axis evaluation for dom data structures. In *Plan-X 2004,Informal processings*, pages 54–63, 2004.

[19] Haifeng Jiang, Hongjun Lu, Wei Wang, and Bengchin Ooi. XR-tree: Indexing XML data for efficient structural joins. In *ICDE*, 2003.

[20] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig joins on indexed XML documents. In *VLDB*, pages 273–284, 2003.

[21] Norman May, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Xquery processing in natix with an emphasis on join ordering. *XIME-P*, pages 49–54, 2004.

[22] Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of xpath. *J. ACM*, 51(1):2–45, 2004.

[23] Dan Olteanu, Holger Meuss, Tim Furche, and Frangois Bry. XPath: looking forward. In *Workshop on XML-based Data Management*, pages 109–127, March 2002.

[24] Prakash Ramanan. Efficient algorithms for minimizing tree pattern queries. In *SIGMOD*, pages 299–309, 2002.

[25] Praveen Rao and Bongki Moon. PRIX: indexing and querying XML using Prufer sequences. In *ICDE*, pages 288–300, 2004.

[26] W3C. *XML Path Language (XPath) 1.0*. http://www.w3.org/TR/xpath, 1999.

[27] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. ViST: a dynamic index method for querying XML data by tree structures. In *SIGMOD*, pages 110–121, 2003.

[28] Peter T. Wood. Minimising simple XPath expressions. In *WebDB*, pages 13–18, 2001.

[29] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Structural join order selection for xml query optimization. In *ICDE*, pages 443–454, 2003.

[30] XMark Project. *XMark–an XML benchmark project.* http://www.xml-benchmark.org, 2001.

[31] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, pages 425–436, 2001.