Taming XPath Queries by Minimizing Wildcard Steps

Chee-Yong Chan[*]
National University of Singapore
chancy@comp.nus.edu.sg

Wenfei Fan
University of Edinburgh & Bell Laboratories
wenfei@inf.ed.ac.uk

Yiming Zeng
National University of Singapore
zengyimi@comp.nus.edu.sg

Abstract

This paper presents a novel and complementary technique to optimize an XPath query by minimizing its wildcard steps. Our approach is based on using a general composite axis called the *layer axis*, to rewrite a sequence of XPath steps (all of which are wildcard steps except for possibly the last) into a single layer-axis step. We describe an efficient implementation of the layer axis and present a novel and efficient rewriting algorithm to minimize both non-branching as well as branching wildcard steps in XPath queries. We also demonstrate the usefulness of wildcard-step elimination by proposing an optimized evaluation strategy for wildcard-free XPath queries that enables selective loading of only the relevant input XML data for query evaluation. Our experimental results not only validate the scalability and efficiency of our optimized evaluation strategy, but also demonstrate the effectiveness of our rewriting algorithm for minimizing wildcard steps in XPath queries. To the best of our knowledge, this is the first effort that addresses this new optimization problem.

1 Introduction

XPath [15] is a widely-used language for XML data, and it is a core component of several important XML languages including XSLT [6] and XQuery [4]. While there has been a host of work on the efficient evaluation of XPath queries (e.g., structural join algorithms [1, 5, 11], specialized indexing techniques [16, 14]), research on the optimization of XPath queries itself has only begun to attract more attention. Since the size of an XPath query (in terms of the number of steps) is a key determinant of its evaluation complexity (e.g., [8]), an obvious optimization that has been explored is to minimize the size of a XPath query by eliminating redundant steps [17, 2, 13]. More recent work has shifted to understanding the properties of XPath expressions to identify useful rewriting rules [3], eliminating reverse axes in queries to facilitate their evaluation on streaming data [12], and to transforming queries to algebraic form for efficient evaluation [10].

In this paper, we present a novel and complementary approach to optimizing XPath queries by minimizing (non-redundant) wildcard steps. A *wildcard step* refers to an XPath step with the wildcard nodetest; examples include *child::** and *ancestor::**. Wildcard steps are commonly used when the element names are unknown or do not matter. For example, wildcard steps are common when querying against some secured XML views in the form of DTDs where some element labels have been intentionally replaced with wildcard-equivalent labels to hide their original labels [7]. Wildcard steps are also useful as shorthand notation to represent a set of element names. For example, if a publication element has either a journal or conference subelement, then the query */child::publication/child::journal/child::title union /child::publication/child::conference/ child::title* can be expressed more succinctly using the wildcard-based path expression */child::publication/child::*/child::title*. Furthermore, queries generated from certain optimization techniques (e.g., rewriting techniques to eliminate reverse axes [12]) may also contain wildcard steps. Thus, wildcard steps are very convenient and useful in XPath queries.

However, wildcard steps can be rather expensive to evaluate; for example, evaluating `desc::*` would require accessing all the descendant nodes of a context node. Thus removing/reducing wildcard steps in a query q is an important optimization issue. The basic idea of our approach is to rewrite a sequence of two steps s_1 and s_2, where s_1

[*]Supported in part by NUS grants R-252-000-164-112/101.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

is a wildcard step and either s_2 is the next step following s_1 in q (i.e., $s_1[\ldots]\ldots[\ldots]/s_2$) or s_2 is the first step in some qualifier expression of s_1 (i.e., $s_1[\ldots]\ldots[s_2\ldots]$), into an equivalent *single, composite step*. Removing wildcard steps leads to performance improvement on query-evaluation not only by reducing query size, but also by allowing selective loading of only the relevant input XML data. To enable this query rewriting approach, we introduce a new axis, called the *layer axis*, which is a natural generalization of XPath's "vertical" navigation axes (i.e., self, child, descendant, parent, and ancestor).

To give an idea of how the wildcard steps in a query can be eliminated via rewriting with the layer axis, consider the following XPath query $q =$ *desc::a/desc::*[par::b/child::c][anc::d/child::e][child::f]/desc::g*, which is depicted as a rooted tree in Figure 4(a). Observe that there is a single wildcard step *desc::** in q which happens to be also a "branching" step in the sense that it has more than one child steps in Figure 4(a) comprising of the next step after itself and the first step of each of its qualifier expressions. The wildcard step *desc::** in q can be eliminated by transforming q into an equivalent query q''' (shown in Figure 4(d)) which contains three instances of the layer axis represented by L^X. Note that our rewriting approach does not require knowledge of the data schema, and the size of the rewritten query (in terms of the number of steps) is no more than that of the input query.

In this paper, we make the following contributions:

- We introduce a novel and complementary approach to optimizing an XPath query by minimizing its wildcard steps. Our approach is based on using a new composite axis, called the layer axis, to facilitate efficient query rewriting to eliminate wildcard steps.

- We develop a novel and efficient query rewriting algorithm to minimize wildcard steps based on the layer axis.

- We propose an efficient and scalable evaluation algorithm for wildcard-free XPath queries, capitalizing on a selective loading strategy.

- We also experimentally demonstrate the benefits of our rewriting and evaluation optimizations for processing XPath queries.

Organization. The rest of this paper is organized as follows. Section 2 presents some definitions and notations. In Section 3, we define a new navigation axis called the layer axis. Section 4 presents rewriting techniques using the layer axis to eliminate wildcard steps in XPath expressions. In Section 5, we present an efficient and scalable approach to evaluate wildcard-free XPath queries. We review related work in Section 6, and present our experimental performance results in Section 7. Finally, we conclude with some future research directions in Section 8.

2 Preliminaries

In this paper, we consider the class of XPath queries that are formed using only the following axes: self, child, descendant, parent, and ancestor (which are abbreviated to *self*, *child*, *desc*, *par*, and *anc*, respectively). We refer to these five axes as *vertical axes*[1], and refer to a step with axis χ as a χ*-axis step*. This fragment of XPath is syntactically defined as follows:

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

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 $q[q_1][q_2]$.

We consider the two common semantics of XPath query evaluation that are used in practice: the first returns only the selected nodes without their subtrees, while the second returns both the selected nodes and subtrees. Based on the type of query evaluation being considered, we shall refer to an XPath query as a *node-selecting* XPath query if it is the first case; and as a *standard* query, otherwise.

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)[2] if v_j is a child node of v_i in $Tree(q)$. 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.

Example 2.1 Figure 1 shows the tree representation of the XPath query */desc::a[child::*[child::b][desc::c]/anc::d/desc::*/child::e*, which has two wildcard steps: a B*-step *child::** and a NB*-step *desc::**. □

Consider a data node v in an XML data tree. We define the *level of* v, denoted by $level(v)$, as follows: $level(v) = 0$ if v is the root node; otherwise, $level(v) = level(v') + 1$, where v' is the parent node of v. 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

[1]For simplicity and without loss of generality, we omit the descendant-or-self and ancestor-or-self axes.

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

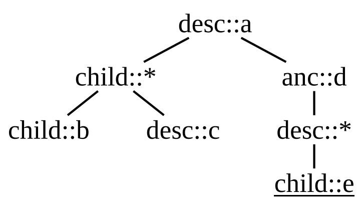


Figure 1: Tree representation $Tree(q)$ of query q

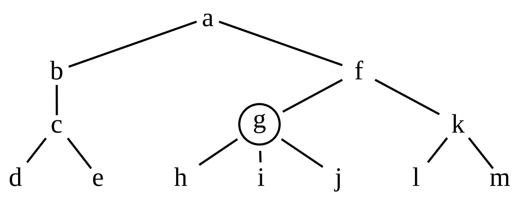


Figure 2: Example XML Data Tree

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 $ht(V) = \max_{v \in V}\{ht(v)\}$.

We use v_c to denote the context node, and η (or η_i for some value i) to denote a nodetest that is either an element label or the wildcard $*$.

3 Layer Axis

In this section, we introduce a new navigation axis, called the *layer axis*, which is a natural generalization of the basic vertical axes (i.e., self, child, descendant, parent, and ancestor). We show that the layer axis is capable of expressing all the vertical axes, and thus XPath queries in our fragment can be rewritten into an intermediate form in terms of the layer axis. Furthermore, the layer axis can be implemented very efficiently. In the next section we shall present an algorithm for minimizing wildcard steps by capitalizing on the layer axis, followed by an efficient selective loading strategy for evaluating wildcard-free XPath queries in Section 5.

3.1 Basic Layer Axis

The basic form of the layer axis, denoted by L^i, selects the "layer" of nodes that are exactly i levels away from the context node v_c either below the context node if i is positive; or above the context node if i is negative. More formally, L^i (where i is an integer) can be defined inductively in terms of the self, child, and parent axes as follows:

- L^0 is defined to be *self*;
- $L^{i+1} = L^i.child$ if $i > 0$ (downward);
- $L^{i-1} = L^i.par$ otherwise (upward).

For example, when the root is v_c, the sequence of three steps */child::*/child::*/child::*η is equivalent to the single step /L^3::η.

More generally, it is often convenient to refer to consecutive layers of data nodes that are located at a certain number of levels away from the context node. For example, the sequence of steps *child::*/child::*/child::*/anc::*η will select a "subtree" of nodes (with nodetest η) starting from the root layer to two layers below the layer containing the context node. The layer axis L^i can be easily extended to support such navigations in terms of $L^{\geq i}$ and $L^{\leq i}$ as follows:

$$L^{\geq i} = \begin{cases} \emptyset & \text{if } i > ht(v_c) \\ L^i \cup L^{\geq i+1} & \text{otherwise} \end{cases}$$

$$L^{\leq i} = \begin{cases} \emptyset & \text{if } i < -level(v_c) \\ L^i \cup L^{\leq i-1} & \text{otherwise} \end{cases}$$

Clearly, both the descendant and ancestor axes are special cases of $L^{\geq i}$ and $L^{\leq -i}$, respectively, with $i = 1$. For example, L^{-3}*::*/anc::** can be expressed as $L^{\leq -4}$::η.

For notational convenience, we define $L^{[i,j]}$, with $i \leq j$, as follows:

$$L^{[i,j]} = L^{\geq i} \cap L^{\leq j}$$

Observe that $L^{\geq i}$ and $L^{\leq i}$ can be expressed as $L^{[i,ht(v_c)]}$ and $L^{[-level(v_c),i]}$, respectively. As a special case, $L^i = L^{[i,i]}$. Thus in the sequel we shall focus on the general form $L^{[i,j]}$ of the layer axis.

We sometimes also use the layer axis with respect to an explicit context node v, which is expressed as $L^{[i,j]}(v)$. Thus, $L^{[i,j]}$ refers to $L^{[i,j]}(v_c)$.

Example 3.1 Consider the XML data tree in Figure 2. If g is the context node, then L^1::* $=$ {h,i,j}, L^{-2}::* $=$ {a}, $L^{\leq 1}$::* $=$ {a,f,g,h,i,j}, and $L^{\geq -1}$::* $=$ {f,g,h,i,j}. $\square$

The layer axis provides a concise specification for a sequence of XPath steps $s_1/s_2/\cdots/s_n$, where all the steps, except for possibly the last step s_n, are wildcard steps. For example, the query q_1 = $\underbrace{\textit{child::*}/\cdots/\textit{child::*}}_{i-1}$*/child::*$\eta_1$, is equivalent to L^i::η_1.

3.2 Height Constraints

In many cases, it is necessary to augment the layer-axis specification with additional constraints on the height of the selected nodes to preserve equivalence. For example, the query $q_2 = L^i$*::*/par::*η_2, where $i \geq 1$, is equivalent to

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

Here, the additional constraint on the height of the selected nodes indicates that the selected nodes must not be leaf nodes. Clearly, $q_2 \not\equiv L^{i-1}::\eta_2$.

More generally, there is a need to be able to specify constraints on the height of both the selected nodes as well as their ancestors. For example, the query $q_3 = L^i$*::*/par::*/par::*/child::*η_3, where $i \geq 2$, is equivalent to

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

Here, each selected node needs to satisfy the constraint that its parent node can reach some descendant node that is at least 2 levels below the parent node. Observe that the constraint on $L^{-1}(v)$ is not equivalent to the following height constraint on v: $ht(v) \geq 1$. Thus, supporting height constraints on ancestors of selected nodes is necessary. However, note that height constraints on descendants of selected nodes can be rewritten to equivalent constraints on the selected nodes themselves. For example, L^i::*/*par*::*/*par*::* $\equiv \{v \in L^{i-2}::* \mid ht(L^1(v)) \geq 1\}$, and the constraint on $L^1(v)$ can be rewritten as a constraint on v: $ht(v) \geq 2$.

As a final example, let us consider the query $q_4 = \{v \in L^{[i,j]}::* \mid ht(v) \geq h\}$ / desc::η_4. This query is equivalent to

$$\{v \in L^{\geq i+1}::\eta_4 \mid \exists\ r \in [\delta(v, v_c) - j, \delta(v, v_c) - i], \quad ht(L^{-r}(v)) \geq h\}$$

Here, we have a height constraint that specifies that each selected node v must have some ancestor node w that is r levels above v (i.e., w is in $L^{[i,j]}$) such that the height of w is at least h.

More formally, height constraints can be defined as follows. Let $cexp$ denote an integer expression defined in terms of integer constants as well as '$+$' and '$-$'. Let exp_1 and exp_2 denote integer expressions defined in terms of $level(v)$, $level(v_c)$, integer constants, 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) $ht(L^{cexp}(v)) \geq exp_1$; or

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

Note that height constraints of the form $ht(v) \geq exp_1$ are allowed in (F1) since $ht(v) = ht(L^0(v))$.

For example, $\exists\ i \in I\ (ht(L^i(v)) \geq h)$ states that v must have some ancestor/descendant node w that is i levels above/below v, where i is some value from a set of consecutive integers I, such that the height of w is at least h.

Putting these together, the layer axis is typically associated with a set of height constraints, and is thus denoted as $L^X(S)::\eta$, where X is a set of consecutive integers (of the form i, $\leq i$, $\geq i$, or $[i, j]$), and S is a (possibly empty) set of height constraints as defined above; it extracts nodes that are reachable via the layer axis and satisfies every height constraint in S.

Example 3.2 Using the new notation $L^X(S)::\eta$, the example queries q_3 and q_4 given above can be specified as $L^{i-1}(\{ht(L^{-1}(v)) \geq 2\})::\eta_3$ and $L^{\geq i+1}(\{\exists\ r \in [\delta(v, v_c) - j, \delta(v, v_c) - i](ht(L^{-r}(v)) \geq h)\})::\eta_4$, respectively. □

As a final remark, the vertical axes in XPath can be specified in terms of the layer axis as follows: *self* $\equiv L^0(\emptyset)$, *child* $\equiv L^1(\emptyset)$, *desc* $\equiv L^{\geq 1}(\emptyset)$, *par* $\equiv L^{-1}(\emptyset)$, and *anc* $\equiv L^{\leq -1}(\emptyset)$.

3.3 Implementation of Layer Axis

In order for a rewriting-based approach using the layer axis to be effective, it is critical that the layer axis be implemented efficiently. In this section, we describe how the layer axis can be efficiently supported by precomputing certain additional information as the input XML document is parsed and loaded into main memory for query evaluation.

There are essentially two key operations that need to be efficiently supported in a layer-axis step evaluation. Specifically, to determine if a data node v is selected by a layer axis $L^{[i,j]}(S)$, we need to (1) check if v is in $L^{[i,j]}(v_c)$; and (2) check if v satisfies each height constraint in S. These two checkings can be efficiently supported by precomputing $level(v)$ and $ht(v)$ for each data v. With these precomputed values, v is in $L^{[i,j]}(v_c)$ if $\delta(v, v_c) = level(v) - level(v_c)$ is in $[i, j]$; and v satisfies the height constraint "$ht(L^k(v)) \geq h$" if $ht(v) \geq h + k$ (for non-negative k values), and if $ht(u) \geq h$ (for negative k values), where u is the ancestor node of v that is k levels above v.

The $level(.)$ and $ht(.)$ information can be easily precomputed by a single parse of the input XML data file as it is loaded into main memory, by using a stack of size H (where H is the height of the input XML data tree T) to store information about the current path of data nodes being parsed. In particular, $ht(v)$ is computed by using the property that $ht(v) = \max_w\{ht(w)+1\}$, where w is a child node of v in T. Note that the height H of an XML data tree T is usually a small value (independent of the size of T).

Thus, an efficient implementation of the layer axis can be easily supported by precomputing some information during the parsing of the input XML data file.

3.4 Extended Layer Axis

Recall that the motivation for the layer axis is to have a general composite axis (that can be efficiently implemented) to be used for eliminating wildcard steps. Specifically, the goal is to be able to rewrite any sequence of XPath steps that consists of all wildcard steps (except for possibly the last step) into a single layer-axis step. However, even with the most general form of the layer axis presented in Section 3.2, there are certain sequences of steps that can not be expressed using a single layer-axis step. For example, consider the query $q = L^{-i}(\emptyset)$::*/*child*::η, where $i > 0$. Note that $q \not\equiv L^{-i+1}(\emptyset)::\eta$. This is because $L^{-i+1}(\emptyset)$::* will select at most one single node, which is the ancestor node of the context node v_c that is $i - 1$ levels above v_c, instead of a set of child nodes as intended. Thus, q can not be rewritten using a single layer-axis step.

This limitation arises whenever a sequence of steps first navigates upwards to some ancestor node of the current context node v_c, and is then followed by a downward navigation. In this case, some of the nodes selected by the downward navigation are neither ancestors nor descendants of v_c, which means that they can not be be captured using a single layer-axis step (which is defined with respect to v_c).

To overcome this restriction, we propose a simple extension of the layer axis of the form $L^{X/Y}(S)$, where for some non-negative integer i, X (which is of the form $-i$ or $\leq -i$) specifies an upward navigation; Y (which is of the form i or $\geq i$) specifies a downward navigation; and S is a set of height constraints. More formally, the extended variant of the layer axis $L^{X/Y}(S)$ is defined as follows:

$$L^{X/Y}(S)::\eta \;\equiv\; L^{X}(\emptyset)::* \;/\; L^{Y}(S)::\eta \qquad (1)$$

We refer to the two variants of the layer axis as *basic layer axis* and *extended layer axis*. Thus, using the extended layer axis, q can now be rewritten as $L^{-i/1}(\emptyset)::\eta$.

Supporting the extended variant is, however, more involved. Checking whether or not two data nodes v_1 and v_2 are related by the axis $L^{-i/j}$, for example, is equivalent to checking if v_1 and v_2 have a common ancestor node that is i and j levels above them, respectively. Clearly, there is a need to balance the tradeoff between the generality of the layer axis and the efficiency of its implementation. Intuitively, a more general layer specification should be able to eliminate wildcard steps for a larger class of queries, but its implementation cost is likely to be higher. In this paper, for practical reasons, we will focus on the basic layer axis together with a special case of the extended variant of the form $L^{-1/Y}(S)$, which can be efficiently implemented by additionally storing a pointer to the parent node for each data node. We intend to explore the tradeoffs of more general variants as part of future work.

4 Minimizing Wildcard Steps

The basic idea of our rewriting algorithm is to iteratively eliminate one wildcard step at a time until either all the wildcard steps have been eliminated or none of the remaining wildcard steps can be eliminated. Each iteration therefore merges a sequence of two steps s_1 and s_2, either of the form s_1/s_2 or of the form $s_1[s_2]$ (at least one of which is a wildcard step) into a single layer-axis step.

We first discuss the simpler case of minimizing NB*-steps in Section 4.1 and then extend our techniques to handle B*-steps in Section 4.2. Finally, we combine these techniques to present an algorithm to minimize wildcard steps in Section 4.3.

4.1 Non-branching Wildcard Steps

In this section, we consider the elimination of wildcard steps that appear in “linear” path expressions (without qualifier expressions) of the form s_1/s_2, where s_1 is a wildcard step; i.e., s_1 is of the form $L^{[i,j]}(S)$*::** with $i \leq j$. Step s_2 is of the form $\chi::\eta$, where χ is any vertical XPath axis and η is either an element name or a wildcard.

Our goal is to rewrite a two-step path expression of the form

$$p \;=\; L^{[i,j]}(S)::* \,/\, \chi::\eta$$

into an equivalent layer-axis step:

```
Algorithm Rewrite-NB*-step ($s_1, s_2$)
Input:  A NB*-step $s_1 = L^{[i,j]}(S)::*$
        A step $s_2 = \chi::\eta$
Output: A step $s' = L^{[i',j']}(S')::\eta$ (equivalent to $s_1/s_2$)
        if $s_1$ can be removed; or $s_1/s_2$ otherwise
1)  if ($i < 0$) and ($\chi \in \{child, desc\}$) then
2)      return $s_1/s_2$;
3)  Apply appropriate rule from (R1) to (R4) to
    rewrite $L^{[i,j]}(\emptyset)::*/\chi::\eta$ to $L^{[i',j']}(S_{new})::\eta$;
4)  $S_{update} = \emptyset$;
5)  for each $\phi \in S$ do
6)      rewrite $\phi$ to $\phi'$ as described in Section 4.1.1;
7)      $S_{update} = S_{update} \cup \{\phi'\}$;
8)  return $L^{[i',j']}(S_{new} \cup S_{update})::\eta$;
```

Figure 3: Algorithm to remove NB*-steps

$$p' \;=\; L^{[i',j']}(S')::\eta$$

to eliminate the wildcard step $L^{[i,j]}(S)::*$.

Note that if step s_1 was of the form $L^{-1/[i,j]}(S)$, then p' is simply replaced with $L^{-1/[i',j']}(S')::\eta$. Thus, the rewritings involving the basic layer axis can be easily extended over to the extended variant; therefore, for simplicity and with loss of generality, we will focus our discussion on the basic layer axis.

Our rewriting algorithm to eliminate a NB*-step is shown in Figure 3. We explain the rewriting rules in terms of two cases, depending on whether or not $i \geq 0$.

4.1.1 Case 1: $i \geq 0$

We first consider the case where the set of constraints S in p is empty and then extend the results to the general case where S could be non-empty.

When $i \geq 0$ and $S = \emptyset$, p can be transformed into p' using the following set of four rewriting rules[3]:

(R1) $L^{[i,j]}(\emptyset)::* \,/\, \text{child}::\eta \;\equiv\; L^{[i+1,j+1]}(\emptyset)::\eta$

(R2) $L^{[i,j]}(\emptyset)::* \,/\, \text{desc}::\eta \;\equiv\; L^{\geq i+1}(\emptyset)::\eta$

(R3) $L^{[i,j]}(\emptyset)::* \,/\, \text{par}::\eta \;\equiv\; L^{[i-1,j-1]}(S')::\eta$ where $S' = \{ht(v) \geq 1\}$

(R4) $L^{[i,j]}(\emptyset)::* \,/\, \text{anc}::\eta \;\equiv\; L^{\leq j-1}(S')::\eta$ where $S' = \{ht(v) \geq i - \delta(v, v_c),\; ht(v) \geq 1\}$.

The rewriting from $L^{[i,j]}$ to $L^{[i',j']}$ in each rule is self-explanatory: in (R1), for example, the transformation essentially adjusts the relative location of the selected nodes (w.r.t. v_c) down by one level from $[i, j]$ to $[i + 1, j + 1]$ due to the second child-axis step. Note that only rules (R3) and (R4), which have a reverse-axis for their second steps, require a height constraint to be specified to preserve equivalence. For (R3), the height constraint is necessary to select only non-leaf nodes that are in $L^{[i-1,j-1]}(S')::\eta$. For

[3]Note that the expression $x + 1$ is actually $\min\{x + 1, ht(v_c)\}$, and the expression $x - 1$ is actually $\max\{x - 1, -level(v_c)\}$.

(R4), a selected node v in $L^{\leq j-1}(S')$::η must have a descendant node w in $L^{[i,j]}$. Clearly, v is a non-leaf node, and so its height must be at least one. Furthermore, if v is in $L^{\leq i-1}(S')$::η, then v must be able to reach some descendant node w in L^i, and so the height of v must be at least $i-\delta(v, v_c)$. Combining these two constraints, we have $ht(v) \geq \max\{i - \delta(v, v_c), 1\}$, which is shorthand for the set of two constraints given above for (R4).

We now explain how the above results can be extended to the scenario where S is non-empty. The rewriting rules (R1) to (R4) are still applicable except that the set of existing constraints in S need to be updated. Thus the set of height constraints S' can be represented as $S' = S_{update} \cup S_{new}$; where S_{update} denotes the set of updated constraints in S, and S_{new} is the set of new constraints as defined by the above rules. Note that the updating of S to S_{update} is independent of the generation of any new height constraint in S_{new}. Moreover, each constraint in S is updated independently of the other updates.

To simplify the discussion, let us consider $p = L^{[i,j]}(S)$::* / χ::η where $S = \{ht(L^{-k}(v)) \geq h\}$, k and h are non-negative integers[4]. The constraint S is updated by incorporating the distance between the old selected nodes and the new selected nodes as follows.

1 If χ = *child*, then $S_{update} = \{ht(L^{-(k+1)}(v)) \geq h\}$.

2 If χ = *par*, then $S_{update} = \{ht(L^{-(k-1)}(v)) \geq h\}$.

3 If χ = *desc*, then

$$S_{update} = \{\exists\, r \in [\delta(v, v_c) - j, \delta(v, v_c) - i] \\ (ht(L^{-(r+k)}(v)) \geq h)\}.$$

That is, v has an ancestor w in $L^{[i,j]}$ such that $ht(L^{-k}(w)) \geq h$.

4 If χ = *anc*, then

$$S_{update} = \{\exists\, r \in [i - \delta(v, v_c), j - \delta(v, v_c)] \\ (ht(L^{r-k}(v)) \geq h)\}.$$

That is, v has a descendant w in $L^{[i,j]}$ such that $ht(L^{-k}(w)) \geq h$.

In general, height constraints may involve $level(v)$, which can occur on the left hand side of "$\geq$" (in $[exp_1, exp_2]$ of the form (F2)) or on the right hand side of "$\geq$" (in exp_1 of the form (F1)).

As an example of the first case, consider $L^{[i,j]}(S)$::* / $desc$::η, where $S = \{\exists\, r \in [\delta(v, v_c) - a, \delta(v, v_c) - b](ht(L^{r-k}(v)) \geq h)\}$ for some integer constant expressions a, b, and k. The updated constraint S_{update} is to assert that a new selected node v has an ancestor w in $L^{[i,j]}$ such that there exists $r \in [\delta(w, v_c) - a, \delta(w, v_c) - b]$ with $ht(L^{r-k}(w)) \geq h$. This requires adjustment to the range of r by incorporating $\delta(v, w)$. In a nutshell, this is done by capitalizing on the following relations:

$$\begin{aligned} \delta(v, w) + \delta(w, v_c) &= \delta(v, v_c) \\ \delta(w, v_c) &\leq j \\ \delta(w, v_c) &\geq i \end{aligned}$$

It is easy to verify that the height constraint S is updated to $S_{update} = \{\exists\, r \in [\delta(v, v_c) - j - a, \delta(v, v_c) - i - b](ht(L^{r-k}(v)) \geq h)\}$.

As an example of the second case, consider $L^{[i,j]}(S)$::* / $desc$::η, where $S = \{ht(L^k(v)) \geq \delta(v, v_c) + h\}$ for some integer constant expressions k and h. By similar reasoning as in the example for the first case, it can be verified that the height constraint S is updated to $S_{update} = \{\exists\, r \in [\delta(v, v_c) - j, \delta(v, v_c) - i](ht(L^{k-r}(v)) \geq \delta(v, v_c) + h - r)\}$.

Due to the lack of space we omit updating of height constraints of other forms, which can be derived by a straightforward structural induction. We should remark that the new and updated constraints can be checked efficiently with the implementation strategy described in Section 3.3.

4.1.2 Case 2: $i < 0$

When $i < 0$, the rules (R3) and (R4) remain intact and are applicable. However, rules (R1) and (R2) no longer hold when $i < 0$ because the sequence s_1/s_2 now corresponds to an upward navigation followed by a downward navigation, which can not be handled by the basic layer axis as explained in Section 3.4. Thus, for such cases, the wildcard step s_2 can not be eliminated. However, if the upward navigation is restricted only to a single level, then the extended layer axis variant $L^{-1/[i,j]}(S)$ can be applied.

Similarly, when it comes to updating existing height constraints, the rules for *par* and *anc* given above remain unchanged when $i < 0$, whereas those for *child* and *desc* do not apply here.

4.2 Branching Wildcard Steps

In this section, we extend the techniques for eliminating NB*-steps to eliminate B*-steps. Our algorithm (shown in Figure 5) to eliminate a B*-step s in an XPath query consists of two rewriting steps:

PullUpChildStep The first rewriting step tries to transform s into a NB*-step by "pulling up" some of the child steps of s to become above s. The goal is try to reduce the problem of eliminating a B*-step to the simpler case of eliminating a NB*-step.

MergeChildStep If s still remains a B*-step after the *PullUpChildStep*, the second rewriting step then tries to merge s with one of its child steps to eliminate s.

Example 4.1 Consider the XPath query q, where $Tree(q)$ is shown in Figure 4(a). To eliminate the B*-step *desc::**, `PullUpChildStep` first transforms $Tree(q)$ into $Tree(q')$ (shown in Figure 4(b)) by pulling up the

[4]Recall that if $k < 0$, then the constraint $ht(L^{-k}(v)) \geq h$ can be rewritten as $ht(v) \geq h - k$.

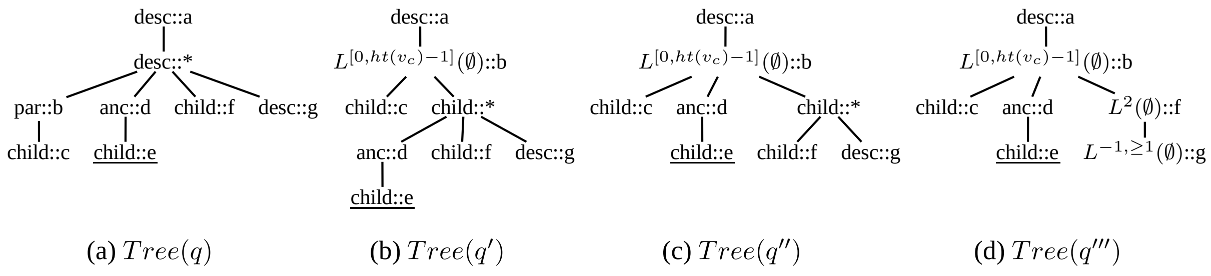


Figure 4: Example of branching wildcard step elimination

```
Algorithm Rewrite-B*-step $(q, s)$
Input:  An XPath query $q$ with a B*-step $s$.
Output: An equivalent query $q'$,
        where $s$ may be removed
1)  if ($s$ has a par-axis child step $s_{par}$) then
2)      Pull up step $s_{par}$ in $q$;
3)      for each anc-axis child step $s_{anc}$ of $s$ do
4)          Pull up step $s_{anc}$ in $q$;
5)  if ($s$ is now a NB*-step in $q$) then
6)      let $s_{child}$ be the child step of $s$ in $q$;
7)      $s' =$ Rewrite-NB*-step $(s, s_{child})$;
8)      Modify $q$ by replacing $s/s_{child}$ with $s'$;
9)  else
10)     if ($s$ has some child-axis child step $s_{child}$) then
11)         Merge $s$ with $s_{child}$ in $q$;
12) return $q$;
```

Figure 5: Algorithm to remove a B*-step

subtree rooted at *par::b* to become above *desc::**; the axes of these two steps are updated to $L^{[0,ht(v_c)-1]}(\emptyset)$*::b* and *child::**. Next, $Tree(q')$ is transformed into $Tree(q'')$ (shown in Figure 4(c)) by pulling up the subtree rooted at *anc::d* to become a child subtree of $L^{[0,ht(v_c)-1]}(\emptyset)$*::b*. Note that the transformed wildcard step *child::** in $Tree(q'')$ is still a B*-step with two child steps.

The second rewriting step `MergeChildStep` then transforms $Tree(q'')$ into $Tree(q''')$ (shown in Figure 4(d)) by the following two changes: (1) the step *child::** is merged with its child step *child::f* into $L^2(\emptyset)$*::f*; and (2) the axis of *desc::g* is changed to $L^{-1,\geq 1}(\emptyset)$. □

We now elaborate on the two rewriting steps in the following subsections.

4.2.1 Pulling Up Child Steps

To eliminate a B*-step s_1 in an XPath query q, our algorithm first tries to transform s_1 into a NB*-step (which can then be eliminated by algorithm `Rewrite-NB*-step` described in the preceding section) by "pulling up" all the child steps of s_1 that have reverse axes (i.e., *par* or *anc*) to become above step s_1 in $Tree(q)$.

This transformation is best explained with a diagram as shown in Figure 6 to eliminate the B*-step s_1 in query q. Here, the par-axis child step s_2 in Figure 6(a) is pulled up to become above step s_1 in Figure 6(b); and the axes for steps s_1 and s_2 are updated accordingly to obtain an equivalent query q'. Note that the set of height constraints S associated with the B*-step s_1 is not affected by the transformation of $Tree(q)$ to $Tree(q')$.

More formally, given an XPath query of the form $s_0/s_1[s_2\ T_2][s_3\ T_3]\cdots$, where $s_1 = L^{[x,y]}(S)$::* is a B*-step that has a *par*-axis child step $s_2 =$ *par*::η_2, the rewrite rule for pulling up s_2 above s_1 can be stated as follows:

$$s_0/s_1[s_2\ T_2][s_3\ T_3]\cdots \ \equiv\ s_0/s'_2[T_2]/s'_1[s_3\ T_3]\cdots$$

where s_2 is rewritten to $s'_2 = L^{[x-1,y-1]}(\emptyset)$::$\eta_2$, and s_1 is rewritten to $s'_1 = L^1(S)$::*.

Once a par-axis child step[5] has been pulled up above the B*-step s_1, each of the anc-axis child steps of s_1 (if any) can also be pulled up to transform q' to q'' as illustrated in Figure 6(c). Specifically, the subtree rooted at step s_3 in Figure 6(b) is pulled up to become a child subtree of step s'_2, and its axis is changed from *anc* to *anc-or-self* to preserve equivalence. Note that if both the nodetests η_2 and η_3 in Figure 6(c) have distinct element labels, then the axis of s'_3 can be simplified from *anc-or-self* to *anc*, as illustrated by the step *anc*::d in Figure 4(c). Here again, the transformation of $Tree(q')$ to $Tree(q'')$ does not affect the set of height constraints S associated with the B*-step s_1.

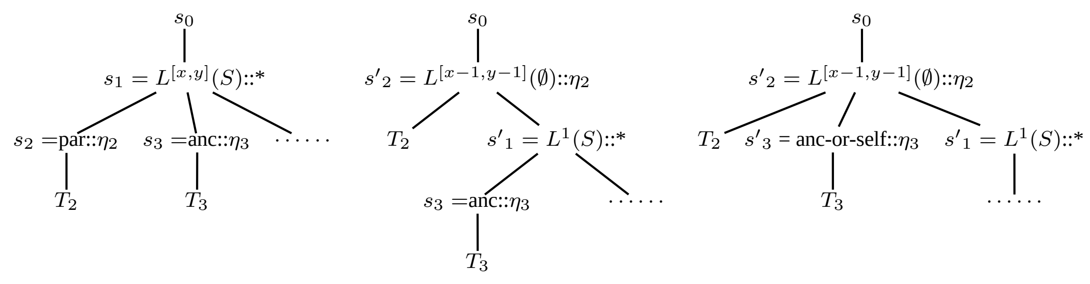
Note that if the B*-step $s_1 = L^{[x,y]}(S)$::* does not have any par-axis child steps, then it is only possible to pull up any of its anc-axis child steps provided if s_1 is a child-axis step (i.e., $x = 1$ and $y = 1$).

4.2.2 Merging With Child Step

The second *MergeChildStep* rewriting step is used to eliminate a B*-step s_1 that could not be transformed into a NB*-step by the *PullUpChildStep* rewriting. As its name suggests, the idea of this second rewriting is to try to merge s_1 with one of its child steps to transform s_1 into a non-wildcard step. Note that, after applying the *PullUpChild-Step* rewriting, s_1 does not have any *par*-axis child step (i.e., each child step of s_1 must be either a *child*-, *desc*-, or *anc*-axis step).

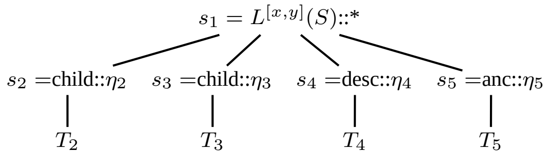
The simplest way to perform the merging is to merge the B*-step with a child-axis child step as illustrated in Figure 7, which shows the elimination of a B*-step $s_1 =$

[5]In the event that s_1 has multiple par-axis child steps, the collection of par-axis child steps can be first combined and optimized to remove redundancies (e.g., [13]). Such issues are, however, orthogonal to the techniques described here and are beyond the scope of this paper.

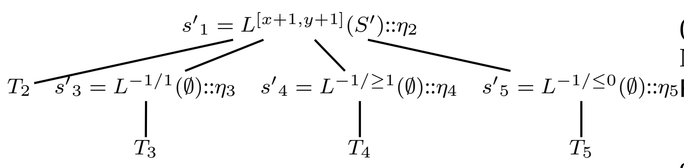


(a) $Tree(q)$ (b) $Tree(q')$ (c) $Tree(q'')$

Figure 6: Pulling up *par*-axis and *anc*-axis steps



(a) $Tree(q)$


$s'_1 = L^{[x+1,y+1]}(S')$::η_2
T_2
$s'_3 = L^{-1/1}(\emptyset)$::$\eta_3$
$s'_4 = L^{-1/\geq 1}(\emptyset)$::$\eta_4$
$s'_5 = L^{-1/\leq 0}(\emptyset)$::$\eta_5$
T_3
T_4
T_5

(b) $Tree(q')$

Figure 7: Merging a B*-step with a *child*-axis child step

$L^{[x,y]}(S)$::* in a query $q = s_1[s_2\ T_2][s_3\ T_3]\cdots$. Here, $Tree(q)$ in Figure 7(a) is transformed into $Tree(q')$ in Figure 7(b) by merging s_1 with its child step $s_2 =$ *child*::η_2, which can be viewed as eliminating the NB*-step s_1 in a "linear" expression s_1/s_2 Thus, by rule (R1) in Section 4.1, s_2 becomes rewritten into $s'_2 = L^{[x+1,y+1]}(S')$::η_2; note that S is updated to S' using the updating rules as discussed in Section 4.1. Other child steps s_j (or $[s_j]$) of s_1 are updated accordingly to cope with the merging, as follows:

- *child*::η is changed to $L^{-1/1}(\emptyset)$::η, namely, moving upward one level and then moving one level down;
- *desc*::η now becomes $L^{-1/\geq 1}(\emptyset)$::η; and
- *anc*::η is changed to $L^{-1/\leq 0}(\emptyset)$::η.

For a concrete illustration of the *MergeChildStep* rewriting, the reader can refer to Figure 4(c) again, where the B*-step *child::** is eliminated by merging with its child step *child::f* to transform q'' to q''' as shown Figure 4(d).

Observe that the transformed query produced by the *MergeChildStep* rewriting step is not unique as it depends on the choice of the child step selected for merging with the B*-step. Referring again to the example in Figure 7, note that if the child step s_3 had been selected (instead of s_2) for merging with the B*-step s_1, the transformed query tree in Figure 7(b) would have been different with T_2 and η_2 being swapped with T_3 and η_3, respectively.

4.3 Rewriting Algorithm

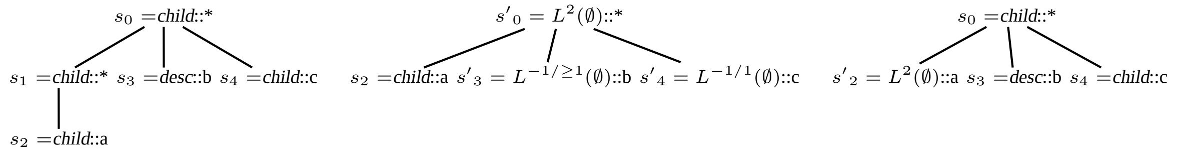
In this section, we combine the techniques developed in the preceding sections to present a rewriting-based approach (shown in Figure 9) to minimize wildcard steps in an input XPath query.

The algorithm consists of two stages: the first stage (lines 1 to 4) calls `Rewrite-NB*-step` to remove NB*-steps, and the second stage (lines 5 to 8) calls `Rewrite-B*-step` to remove B*-steps.

In both stages, the algorithm traverses the tree in a top-down manner to search for wildcard steps; this is important to guarantee that a wildcard step s_1 is rewritten before its descendant wildcard step s_2 of the same type (i.e., both B*-steps or NB*-steps). The purpose of this requirement is to maximize the number of wildcard steps that can be eliminated. For NB*-steps, recall that the rewriting rules for them are of the form $L^{[i,j]}(S)$::* / χ::$\eta \;\equiv\; L^{[i',j']}(S')$::$\eta$, with the wildcard-based layer axis "on top" and a basic vertical axis "below". Thus, rewriting a "lower" NB*-step first before its "upper" NB*-step could deprive the latter from being eliminated. A similar reasoning also applies to B*-steps, which is why a top-down traversal is preferred over an arbitrary traversal for both stages.

Furthermore, the reason for eliminating NB*-steps first before B*-steps is also for the purpose of maximizing the number of wildcard step eliminations. For example, consider the query q in Figure 8(a) containing two wildcard steps s_0 and s_1. If the B*-step s_0 is first rewritten, then q is transformed to q' in Figure 8(b), and the B*-step s'_0 in q' can not be eliminated by Algorithm `Rewrite-B*-step`. On the other hand, if the NB*-step s_1 is first rewritten, then q is transformed to q'' in Figure 8(c), and the remaining B*-step can still be eliminated using Algorithm `Rewrite-NB*-step`. Thus, to maximize the number of wildcard step eliminations, NB*-steps are eliminated first before B*-steps.

The algorithm takes at most quadratic time in the size of the input XPath query q. The overhead is negligible since q is typically small in practice. Note that since the output of the *MergeChildStep* rewriting is not unique, it follows that the output of the rewrite algorithm is also not unique.



(a) $Tree(q)$ (b) $Tree(q')$ (c) $Tree(q'')$

Figure 8: Example showing the advantage of eliminating NB*-steps before B*-steps

```
Algorithm Rewrite($q$)
Input:  An XPath query $q$.
Output: An equivalent query to $q$ that has the
        minimal number of wildcard steps.
1)  Traverse $Tree(q)$ top-down:
2)      for each step $s$ visited do
3)          if ($s$ is a NB*-step in $s/s'$) then
4)              Rewrite-NB*-step ($s, s'$);
5)  Traverse $Tree(q)$ top-down:
6)      for each step $s$ visited do
7)          if ($s$ is a B*-step) then
8)              Rewrite-B*-step ($q, s$);
9)  return $q$;
```

Figure 9: Rewrite Algorithm

5 Optimized Evaluation of Wildcard-free Queries

In this section, we demonstrate the usefulness of wildcard step elimination by presenting a simple and yet effective optimized evaluation strategy for XPath queries that are free of wildcard steps.

One limitation of XPath query evaluators that rely on a main-memory representation of the input XML data (e.g., DOM-based implementations) is that they can not scale to process large input XML data due to their large space requirement. An obvious idea to alleviate this problem is to selectively load only the necessary portion of the input data into main memory to evaluate the input query. However, determining the necessary portion of the data to load seems to be a difficult problem itself when the input XPath query contains wildcard steps since a *desc*-axis wildcard step in a query can potentially refer to the entire data. However, if the input query is wildcard-free or if its wildcard steps can be completely eliminated, then it becomes possible to employ a selective data-loading strategy to improve both the scalability as well as the efficiency of query evaluation.

In this section, we present an optimized evaluation strategy for wildcard-free XPath queries that is based on the simple idea of selectively loading only a portion of the data nodes into main memory based on the set of element labels that appear in the input query. We first explain the idea for evaluating node-selecting XPath queries and then discuss how this strategy can also be applied (to some extent) to standard XPath queries.

5.1 Node-selecting XPath Queries

To illustrate the basic idea, let us consider a simple node-selecting XPath query q =/*child*::a/*desc*::*/*child*::b. Instead of loading the entire input data into main memory to evaluate q, a more efficient approach is to first rewrite q into an equivalent wildcard-free query q' =/*child*::a/$L^{\geq 2}(\emptyset)$::b. Then, by exploiting the absence of wildcard steps in q', it is now possible to selectively load only the data nodes whose labels are explicitly referenced in the query (i.e., data nodes that are labeled a or b).

More specifically, as the input data file is parsed, the necessary node information is precomputed as described in Section 3.3. If an input data node's label is referenced in the rewritten query q', this data node is cached in main memory (to be used for evaluating the query); otherwise, this node is "irrelevant" for the query evaluation and it is only cached temporarily in the stack (described in Section 3.3) for the purpose of precomputing the $ht(.)$ and $level(.)$ information. Once the input data has been completely parsed and selectively loaded, the query can now be processed using the loaded relevant data.

However, note that even though an irrelevant node v itself is not needed for query evaluation, its precomputed $ht(v)$ value might still be needed for checking the height constraints for some other relevant data node. This can be solved by storing the precomputed height information of all ancestors of each relevant data node v (including v itself) in an array $Ht_v[0, ..., H]$ (where H is the height of the data tree) such that $Ht_v[i] = ht(L^{i-level(v)}(v))$ for $i \leq level(v)$. Clearly, the storage of the collection of arrays Ht_v for the relevant data nodes can be optimized given that there are overlapping entries in them.

Since the set of element labels that are explicitly referenced in the input query (in the earlier example, only $\{a, b\}$) is generally a small subset of the set of element labels that appear in the input XML data, this selective-loading strategy can lead to significant reduction in the data that need to be loaded for query evaluation, thereby improving its scalability. Furthermore, since the data size (more accurately, the size of the loaded data) is a key determinant of the time-complexity of XPath query evaluation (e.g., [8]), the query processing can also be significantly improved using this optimized evaluation strategy.

As a final remark on the implementation of the optimized evaluation, we note that it is necessary to precompute the array $Ht_v[0, ..., H]$ only for data nodes v (with element label η) if there is some layer axis $L^X(S)$::η in the rewritten query with $S \neq \emptyset$. For the example query

q', since the only layer-axis there does not have any height constraints, the array $Ht_v[0, ..., H]$ need not be precomputed at all.

5.2 Standard XPath Queries

The above optimized evaluation strategy can also be applied to standard XPath queries, but the portion of input data that needs to be loaded is likely to be larger (compared to node-selecting queries). This is due to the semantics of standard XPath queries which require returning not only the selected target nodes but also the subtrees rooted under them. Consequently, data nodes that are descendants of potential target nodes must also be cached in main memory (in addition to the data nodes whose element labels are explicitly referenced in the query) for evaluation.

6 Related Work

To the best of our knowledge, this is the first paper that addresses the problem of reducing the size of XPath queries by minimizing wildcard steps. Our proposed rewriting optimizations are different from but complementary to existing XPath query rewriting techniques.

In contrast to the research on minimizing redundant XPath steps [17, 2, 13] which relies on integrity constraints of the data schema, our rewriting techniques are designed for minimizing non-redundant wildcard steps and do not require knowledge of the data schema. Another difference from these work is that our techniques apply to a larger fragment of XPath queries beyond the child and descendant axes considered for twig queries there.

Our work also differs from the recent research on eliminating reverse axes in XPath queries [12]; indeed, the rewriting techniques there can actually introduce additional wildcard steps into the transformed queries.

7 Performance Study

To verify the effectiveness of our rewriting algorithms and optimized evaluation strategy for XPath queries, we conducted a performance study using the XMark benchmark data [18]. Our results indicate that our proposed optimizations achieve a significant performance improvement over traditional evaluation methods for XPath queries, with our selective data loading evaluation strategy (based on wildcard step elimination) outperforming a conventional evaluation method by a factor ranging from 2 to 4.

7.1 Experimental Testbed and Methodology

Data Sets: We used the XMark benchmark data [18] for our experiments and generated four data files of size 70MB, 175MB, 260MB, and 340MB. The number of element nodes contained in these files are, respectively, about 1.1 million, 2.5 million, 3.8 million, and 5 million.

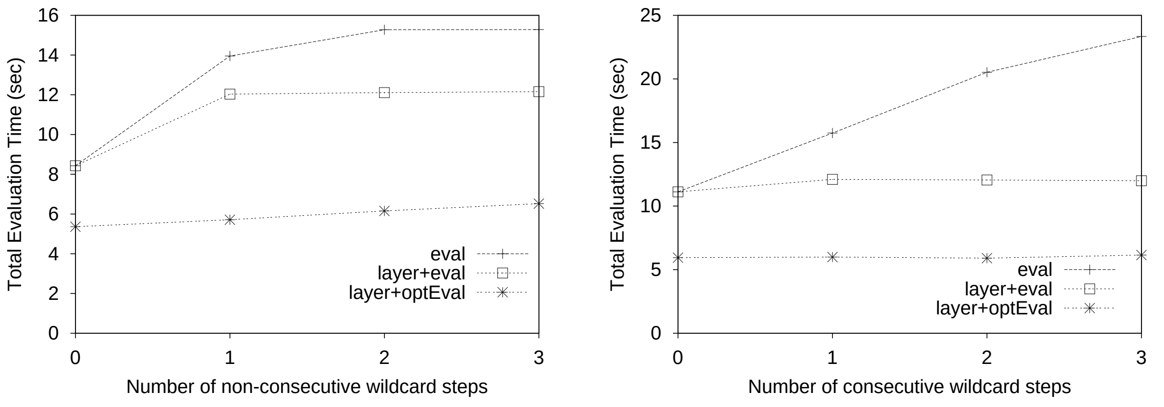
Queries: We generated node-selecting XPath queries using the XMark benchmark schema by varying three parameters: the number of non-consecutive NB*-steps (denoted by N_{nc}), the number of consecutive NB*-steps (denoted by N_c), and the number of B*-steps (denoted by N_b). N_{nc} is varied from 0 to 3, where the query with $N_{nc} = 3$ consists of a single step followed by three predicates (each of which contains a wildcard step) as follows: "/desc::mailbox [anc::*/anc::site] [desc::*/desc::from] [desc::*/desc::to]". Queries with $N_{nc} < 3$ are generated from this query by removing the appropriate number of predicates. N_c is varied from 0 to 3, where the query with $N_c = 3$ is of the form "/desc::site/desc::*/desc::*/desc::*/child::keyword", and queries with $N_c < 3$ are generated from this query by removing the appropriate number of wildcard steps. Finally, N_b is varied from 0 to 3 as follows. The query with no B*-steps is /desc::bidder [par::openauction] [anc::regions] /anc::site, and the query with a single B*-step is q_1 = /desc::personref /anc::* [par::openauction] [anc::regions] /anc::site. A query with n B*-steps, $n > 1$, is generated by concatenating n copies of q_1.

Algorithms: We compared the performance of three different methods for evaluating XPath queries. The first method ("`eval`") corresponds to an unoptimized approach where the wildcard steps in the input query are not eliminated, and the query is evaluated by first constructing a main-memory representation of the entire input XML data before query evaluation. We implemented this evaluation method based on [9]. The second method ("`layer+eval`") is an enhancement of the first method that optimizes the input query by eliminating its wildcard steps using the layer axis. Finally, the third method ("`layer+optEval`") is a further improvement of the second method which is based on our proposed optimized evaluation strategy using selective loading together with wildcard step elimination.

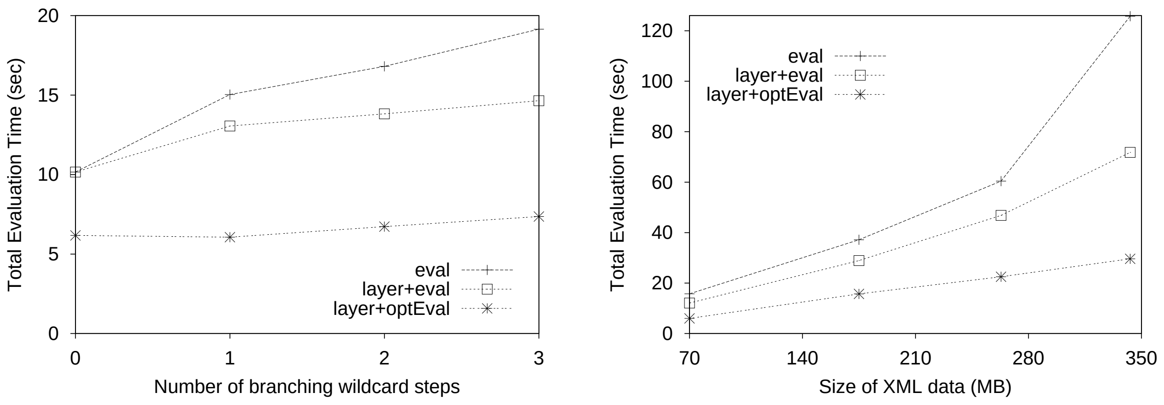
We compared the evaluation methods in terms of the total evaluation time that includes two components: the parsing time as well as the querying time. The *parsing time* includes the time to parse and load the data into main memory (either partially or entirely) as well as the time to perform any precomputations (e.g., both the `layer+eval` and `layer+optEval` methods might need to precompute $level(.)$ and $ht(.)$ values). The *querying time* refers to the actual time required to evaluate the input query using the loaded data and any precomputed information to compute the query's result. Our experiments were conducted on a 2.6 GHz Intel Pentium IV machine with 1 GB of main memory running Windows XP; and all algorithms were implemented using Java.

7.2 Experimental Results

Figures 10(a), (b), and (c), compare the performance results of the three evaluation algorithms on the 70MB data file by varying the parameters N_{nc}, N_c, and N_b, respectively. Figure 10(d) compares the performance of evaluating the single NB*-step query with $N_c = 1$ by varying the size of the XML data file. Our results demonstrate that both the wildcard-step elimination strategy (i.e., `layer+eval`) as well as the selective-loading evaluation strategy (i.e., `layer+optEval`) consistently outper-



(a) Varying N_{nc} (b) Varying N_c



(c) Varying N_b (d) Varying input data size

Figure 10: Evaluation of Node-selecting XPath Queries.

form the traditional evaluation method (i.e., `eval`), with `layer+optEval` giving the best performance. Specifically, `layer+eval` improves `eval` by a factor of up to 2.3, and `layer+optEval` improves `eval` by a factor of up to 4.2.

Figure 10(a) shows that as a query's complexity increases (with a larger number of non-consecutive wildcard steps), its total evaluation time also increases as expected; and generally, the performance gain of our proposed optimizations over `eval` also increases. Note that when the query has no wildcard steps (i.e., $N_{nc} = 0$), both `eval` and `layer+eval` are essentially the same and they have the same evaluation cost.

The parsing time turns out to be the dominant component of the total evaluation cost for all three methods. In particular, for both `layer+eval` and `layer+optEval`, the parsing time constitutes over 90% of the total evaluation cost. For `eval`, while the parsing time is about 90% of the total cost when $N_{nc} = 0$, this reduces to about 55% when $N_{nc} > 0$ due to the higher querying cost for queries with wildcard steps. Among the three methods, the parsing time for `layer+eval` is the highest, while `layer+optEval` incurs the lowest parsing cost. The reason for the latter is due to the effectiveness of selective data loading. `layer+eval` is more costly than `eval` in terms of parsing because `layer+eval` incurs the additional overhead of precomputation without the benefit of selective loading; however, the overall evaluation cost of `layer+eval` is still lower than that of `eval` due to the significant performance benefit with wildcard step elimination.

In terms of querying time, `layer+eval` is more efficient than `eval` because the wildcard steps in the queries are eliminated by `layer+eval` which results in faster query evaluations. The querying performance of `layer+eval` is further improved by `layer+optEval` which significantly reduces the data nodes loaded for evaluation; indeed, the proportion of data nodes being loaded by `layer+optEval` ranges from 1.3% to 3.8% as N_{nc} increases from 0 to 3.

Similar trends are also observed for the performance comparisons with varying number of consecutive non-branching wildcard steps N_c and varying number of branching wildcard steps N_b in Figures 10(b) and (c), respectively. In particular, the performance improvement of both `layer+eval` and `layer+optEval` over `eval` is greater as the query's complexity increases.

Finally, Figure 10(d) compares the performance of evaluating the single NB*-step query with $N_c = 1$ as a function of the size of the XML data file. The results indicate that the performance benefits of both `layer+eval` and `layer+optEval` over `eval` become more significant as the data size increases.

8 Conclusions

In this paper, we have proposed a new and complementary approach to optimize XPath queries by minimizing their wildcard steps. Our approach is based on using a general, composite axis called the layer axis, to rewrite a sequence of XPath steps into a single layer-axis step. We have described an efficient implementation of the layer axis and presented a novel and efficient rewriting algorithm to minimize both non-branching as well as branching wildcard steps in XPath queries. We have also demonstrated the usefulness of wildcard-step elimination by proposing an optimized evaluation strategy that capitalizes on the absence (or reduction) of wildcard steps in XPath queries. Our experimental results show that both the rewriting techniques and optimized evaluation strategy can result in significant performance improvement for XPath query evaluation. To the best of our knowledge, this is the first paper that addresses this new optimization problem.

As part of our future work, we intend to investigate the optimality and completeness of our rewriting algorithm, and also examine the possibility of extending the layer axis to handle the "horizontal" axes (i.e., preceding, following, and sibling-related axes) as well.

References

[1] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: a primitive for efficient XML query pattern matching. In *ICDE*, pages 141–152, 2002.

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

[3] M. Benedikt, W. Fan, and G. M. Kuper. Structural properties of XPath fragments. In *ICDT*, pages 79–95, 2003.

[4] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, and J. Simeon. *XQuery 1.0: An XML query language*. `"http://www.w3.org/TR/xquery"`, November 2003.

[5] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD*, pages 310–321, 2002.

[6] J. Clark. *XSL Transformations (XSLT) 1.0*. `"http://www.w3.org/TR/xslt"`, November 1999.

[7] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure XML querying with security views. In *SIGMOD*, 2004.

[8] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, pages 95–106, 2002.

[9] G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: improving time and space efficiency. In *ICDE*, pages 379–390, 2003.

[10] S. Helmer, C.-C. Kanne, and G. Moerkotte. Optimized translation of XPath into algebraic expressions parameterized by programs containing navigational primitives. In *WISE*, pages 215–224, 2002.

[11] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed XML documents. In *VLDB*, pages 273–284, 2003.

[12] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: looking forward. In *Workshop on XML-based Data Management*, pages 109–127, March 2002.

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

[14] P. Rao and B. Moon. PRIX: indexing and querying XML using Prufer sequences. In *ICDE*, pages 288–300, 2004.

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

[16] H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: a dynamic index method for querying XML data by tree structures. In *SIGMOD*, pages 110–121, 2003.

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

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