

# Relational Index Support for XPath Axes\*

Leo Yuen and Chung Keung Poon

Department of Computer Science  
City University of Hong Kong  
{leo, ckpoon}@cs.cityu.edu.hk

**Abstract.** In this paper, we designed efficient indexing structure for XML documents so that each basic XPath axis step is supported. The indexing structure is built on top of the B<sup>+</sup>-tree which is available in practically all commercial relational database systems. For most of the basic axis steps, we are able to derive theoretical worst case execution time bounds. We also perform experimental evaluation to substantiate those bounds.

## 1 Introduction

The Extensible Markup Language (XML) [6] is becoming the *de facto* standard for information representation and exchange over the Internet. Owing to its hierarchical (recursive) and self-describing syntax, XML is flexible enough to express a large variety of information. XPath is a query language promoted by W3C for addressing parts of an XML document and is a core fragment of several other major XML query languages like XSLT, XPointer and XQuery. Thus, it is important to support XPath queries efficiently.

An XML document can be naturally modelled as a tree in which nodes represent XML elements while edges represent nesting between elements. A sample XML document is shown in Figure 1.

Starting from a given *context node*, an XPath expression specifies a set (or a sequence, in XPath 2.0 [2]) of nodes to be reported as follows. An XPath expression is comprised of a number of *location steps*, each consisting of an *axis*, a node test, and possibly a predicate as well. The axis specifies how the document tree is to be traversed from a context node. The node test then filters the set of reached nodes using a test on the node's tag name or type. The predicate, if present, specifies further conditions for filtering. The result set of a location step will then become the context nodes for the next location step. For example, the expression

`/descendant::book/child::author`

consists of two location steps. Starting from the root as the default initial context node, the first location step `“/descendant::book”` specifies all nodes in the tree with `book` as their tag names. Then, the second location step specifies all their children with `author` as their tag names.

---

\* This research was fully supported by a grant from the Research Grants Council of the Hong Kong SAR, China [Project No. 9040906 (RGC Ref. No. CityU 1164/04E)].

```

<?xml version="1.0"?>
<a>
  <b>
    <d><g/><h/></d>
  </b>
  <c>
    <e/>
    <f><i/><j/></f>
  </c>
</a>

```

**Fig. 1.** An example XML document

Handling large XML documents in secondary storage turns out to be a big challenge. Earlier approaches [7, 9, 15, 16, 21] construct structural summaries of the XML documents to improve the query efficiency. They often require large index storage unless query performance is sacrificed.

Newer approaches apply various labelling schemes on the document tree so that structural information about the nodes (such as ancestor-descendant or parent-child) can be checked by just examining their labels. Then an XPath expression can be evaluated by preparing the lists of nodes satisfying the node tests for each location step in the expression, and pairwise joining the lists according to the structural requirements on the nodes. Because of the labelling, this type of joins (called *containment* or *structural* joins) does not require traversing the actual document tree, thereby saving a lot of disk I/Os. A substantial body of research [1, 5, 14, 20, 27, 29] has been on the efficient implementation of such joins. All of them, except for [29], are for native XML databases. Also, most of these works mainly emphasized the “vertical” axes, i.e., the ancestor, descendant, parent and child axes. More recent work focus on evaluating certain patterns of XPath expressions including the holistic twig joins [3, 14, 15, 22, 26], sequences of consecutive child axis steps [4] and the next-of-kin pattern [30].

There are also investigations on evaluating the whole XPath expressions. On the main memory model, Gottlob et al. studied the query and data complexities of XPath expression evaluation. They designed generic algorithms [10, 12] and determined the complexity for evaluating the whole path expression [11]. For example, for a fragment of XPath which they called Core XPath, they designed an algorithm for its evaluation using  $O(n \cdot |Q|)$  time where  $n$  and  $|Q|$  represent the size of the XML document and query respectively. In the relational domain, Schmidt et al. [23] and Yoshikawa et al. [28] proposed path-based approaches which, again, favours the vertical axes. DeHaan et al. [8] proposed generic translations of an XPath expression into a single SQL statement. In practice, their method is unlikely to be efficient without modifying the relational engine.

In this paper, we focus on indexing the XML document on a relational database to support each basic axis step efficiently. This is important despite the many research on matching patterns of path expressions or even the whole path expressions holistically. First, by concentrating on a basic step, we often obtain more efficient indexing method. For example, Gottlob et al.’s algorithm

[10] requires  $O(n)$  time when the XPath expression contains only one axis step, i.e.,  $|Q| = 1$ . This is slow compared to our index structure. Second, our techniques may well be applicable to these other methods. Relatively few works have been done on this direction.

Kha et al. [17] suggested a recursive version of the UID [19] to support all XPath axes. It requires adding dummy nodes to the original tree to make it a complete  $b$ -ary tree before labelling in a breadth-first manner, where  $b$  is the maximum fanout of the original tree. Thus, the label size can be very large ( $O(n \log n)$  bits per node in the worst case compared with  $O(\log n)$  bits in common interval or prefix labelling).

In contrast, Grust et al. [13]'s XPath Accelerator maps each node in an XML document to a point on the so-called pre/post plane; and translates the four global axes, i.e., descendant, ancestor, preceding and following, into 2-d range queries over this 2-d plane. To support the 2-d range queries, they have implemented their index structure on top of an R-tree (a spatial index structure) as well as a  $B^+$ -tree which does not naturally suit for 2-d range queries. Thus, no worst case performance bounds are given.

Here, we refined their work by mapping the nodes to 1-dimensional intervals instead of points on a 2-d plane. Then the four global axes are translated to either 1-dimensional range queries or interval queries (also called 1-d *point enclosure* in some literature). Consequently, we reduce the number of dimensions in the index structure by one. Note that Grust et al. already observed that the descendant axis can actually be computed as 1-d range queries. However, they did not go further to consider the other three axes in the way we do.

One-dimensional range queries are well-supported by the ubiquitous  $B^+$ -tree index in relational database with good worst case performance bound. For the interval queries, we employ the RI-tree of Kriegel et al. [18]. Thus, instead of indexing the pre/post-plane by an RI-tree as suggested by Grust et al., we directly map the nodes to intervals which is naturally supported by RI-tree. Interestingly, Jiang et al. [14] also made use of certain variants of interval tree index structure to support interval queries. However, they require modifying the database kernel to incorporate their XR-tree. In contrast, the RI-tree is much simpler and directly implementable on top of a  $B^+$ -tree.

We observe that the set of intervals derived from the document tree possesses many nice properties. First, the interval boundaries are confined to a limited range, thereby allowing us to derive good theoretical bounds for many of the axes. Second, the set of intervals are nested, i.e., for any two intervals in the set, either they have no intersection or one is completely contained in another. This allows us to use RI-tree, as an alternative to  $B^+$  tree, to support range queries, which may be of independent theoretical interest. This in turn permits us to support both XPath axes with and without name tests.

The rest of this paper is organized as follow. In the next section, we explain some basic concept of XPath axes and the main idea of Grust et al. Then we gradually build up to our final index structure by describing the design for handling the descending, preceding and following axes in Section 3, the ancestor

axis in Section 4, the local axes in Section 5 and the name test in Section 6. We present our experimental results in Section 7. The paper is then concluded in Section 8.

## 2 Preliminaries

### 2.1 The XPath Axes

There are altogether 13 axes in XPath. Besides the *attribute* and *namespace* axes, the other 11 axes deal with traversals of the document tree. In the same spirit as Zhang et al. [30], we divide them into two types. The *local axes* include the *self*, *parent*, *child*, *preceding-sibling* and *following-sibling* axes. The *global axes* include the *descendant*, *descendant-or-self*, *ancestor*, *ancestor-or-self*, *preceding* and *following* axes.

### 2.2 Grust et al.’s Document Region

Given an XML document tree, Grust et al. label every node  $u$  by the pair  $(pre(u), post(u))$  where  $pre(u)$  and  $post(u)$  represent the pre-order and post-order labelling of  $u$  in a depth-first traversal of the tree. For example, in Figure 2, each node is labelled with a pair of numbers, the left is its pre-order number and the right is its post-order number. As such, each node is naturally mapped to a point on the 2-dimensional pre/post-plane. Then the descendants of  $u$  are precisely those nodes  $v$  such that  $pre(u) < pre(v)$  and  $post(v) < post(u)$ . In other words,  $v$  lies on the lower-right quadrant of  $u$ . Similarly, the ancestors of  $u$  are those nodes lying on the top-left quadrant of  $u$  while the preceding and following nodes of  $u$  are on the lower-left and upper-right quadrants of  $u$  respectively. See Figure 2.

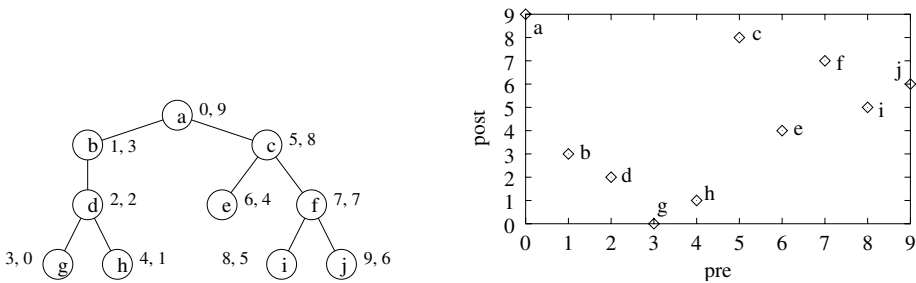


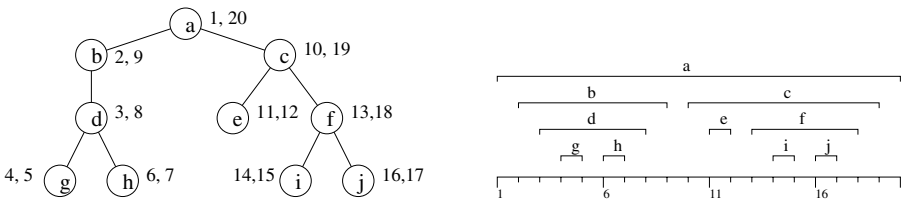
Fig. 2. Left: Pre/post-order of tree nodes. Right: The pre/post-plane

Thus, supporting these four XPath axes amounts to indexing the pre/post-plane to allow for 2-d range searching. To do this, Grust et al. have experimented with ordinary B<sup>+</sup>-tree index as well as R-tree (a spatial index structure).

### 3 The Descendant, Preceding and Following Axes

#### 3.1 Mapping Nodes to Intervals

We will label each node in the document tree  $T$  by an interval as follows. It is folklore that the structure of a tree can be represented by a set of nested parentheses. For convenience, we will call it a *bracket expression*. The bracket expression of  $T$  can be obtained by performing a depth-first traversal on  $T$ . When we start traversing the subtree rooted at  $u$ , we output an opening bracket for node  $u$ . When we have finished traversing the subtree of  $u$ , we output a closing bracket for  $u$ . Then the *first-order* number of a node  $u$ , denoted  $f(u)$ , is the position of its opening bracket in the bracket expression of  $T$ . Similarly, the *last-order* number of  $u$ , denoted  $l(u)$ , is the position of its closing bracket. See Figure 3 for an example.



**Fig. 3.** Left: Interval labelling of tree nodes. Right: The corresponding intervals

Note that for a tree with  $n$ -nodes, there are  $2n$  brackets. Thus, the range of the first- and last-order number is in  $[1..2n]$ . Moreover,  $f(u) < l(u)$  for every node  $u$ . We can view that each node  $u$  is labelled with the interval  $[f(u), l(u)]$ . The following properties are obvious.

**Proposition 1.** *For any tree labelled with the above interval labelling scheme, node  $u$  is an ancestor of  $v$  (or equivalently,  $v$  is a descendant of  $u$ ) iff the interval  $[f(v), l(v)]$  is contained in  $[f(u), l(u)]$ , i.e.,  $f(u) < f(v) < l(v) < l(u)$ .*

**Proposition 2.** *For any tree labelled with the above interval labelling scheme, the set of intervals is nested, i.e., any two intervals are either non-overlapping or one is completely contained in another.*

#### 3.2 Finding Descendants, Preceding and Following Nodes

Based on Proposition 1, the descendants  $v$  of a context node  $c$  can be characterised by

$$f(c) < f(v) < l(c) \tag{1}$$

which is a 1-d range searching. An alternative characterisation is:

$$f(c) < l(v) < l(c). \tag{2}$$

This will turn out to be useful as one can choose either condition to check as convenient. For the preceding nodes of context node  $c$ , they are those nodes  $v$  whose opening tag comes before that of  $c$  except when they are the ancestors of  $c$  ([6]). That means, if we perform a depth-first traversal, we will finish the traversal of the subtree rooted at  $v$  before we start traversing the subtree rooted at  $c$ . Hence we have the condition

$$l(v) < f(c). \quad (3)$$

Symmetrically, the following nodes of  $c$  are those nodes  $v$  whose opening tag comes after that of  $c$ , except those of  $c$ 's descendants. That is, we completed traversing the subtree of  $c$  before starting the traversal of  $v$ . Hence the nodes  $v$  can be characterised as:

$$l(c) < f(v). \quad (4)$$

Note that such characterization should be quite obvious. For example, it has been mentioned in [25, 28].

As all three axes involve range queries, we store the first and last-order number of the nodes as attributes in a relational table and build a B<sup>+</sup> tree index on it. More specifically, we will have the following two tables:

```

XTfirst(first, last, data)
XTlast(last, first, data).
    
```

The attributes `first` and `last` store  $f(u)$  and  $l(u)$  of a node  $u$  respectively. The attribute `data` contains other information of  $u$ . An underlined attribute indicates that it is (part of) the primary key for indexing and the tuples are sorted on the primary key.

To compute the following axis of  $c$ , we select those tuples  $t$  from `XTfirst` where  $l(c) < t.first$ . The preceding axis of  $c$  is given by selecting those tuples  $t$  from `XTlast` where  $t.last < f(c)$ . Descendants of  $c$  can be found by selecting tuples  $t$  from `XTlast` where  $f(c) < t.last < l(c)$ . Since the tuples are sorted according to the primary key, the search will take only  $O(\log_B n + k/B)$  time where  $k$  is the output size and  $B$  is a parameter depending on the block size.

## 4 The Ancestor Axis

Using Proposition 1, the ancestors of a context node  $c$  are those nodes  $v$  such that  $[f(v), l(v)]$  contains  $f(c)$  (or equivalently  $l(c)$ ). To find those intervals enclosing  $f(c)$ , we make use of an RI-tree index.

### 4.1 RI-Tree

The original RI-tree ([18]) of height  $h$  consists of an implicit complete binary tree  $U$  of height  $h$  (with the root at height  $h - 1$  and leaves at height 0). Each node will get an ID (i.e., an integer) and for convenience, we identify a node with

its ID. The root is  $2^{h-1}$  and its left and right children are  $2^{h-1} - 2^{h-2} = 1 \cdot 2^{h-2}$  and  $2^{h-1} + 2^{h-2} = 3 \cdot 2^{h-2}$  respectively. In general, for a node  $x = x'2^\ell$  where  $2^\ell$  is the largest power of two that divides  $x$ , its left and right sons are  $x'2^\ell - 2^{\ell-1} = (2x' - 1)2^{\ell-1}$  and  $x'2^\ell + 2^{\ell-1} = (2x' + 1)2^{\ell-1}$ , respectively.

**Construction.** Suppose we are to store a set of  $n$  intervals  $[l_i, r_i]$ ,  $1 \leq i \leq n$ , whose boundaries are taken from a bounded universe  $\{1, \dots, 2^h - 1\}$ , i.e., each boundary can be represented as an  $h$ -bit binary number. The main idea is to associate each interval  $[l_i, r_i]$  with the highest node  $x$  such that  $l_i \leq x \leq r_i$ .

To compute such an  $x$ , we can search from the root. Alternatively, one can observe that  $x$  is nothing but the lowest common ancestor of  $l_i$  and  $r_i$  in  $U$ . Thus  $x$  can be computed easily by considering the binary representation of  $l_i$  and  $r_i$  and finding the most significant bit on which they differ. Let's say this is the  $(l-1)$ -st bit counting from bit 0. Then  $x = \lfloor l_i/2^l \rfloor \cdot 2^l$  which is the same as  $\lfloor r_i/2^l \rfloor \cdot 2^l$ .

To facilitate the searching of intervals associated with a node  $x$ , the intervals are stored twice, once in sorted order of left boundaries and once in right boundaries. Thus, there will be two database tables,  $L(\underline{\text{node}}, \underline{\text{left}}, \text{right})$  and  $R(\underline{\text{node}}, \underline{\text{right}}, \text{left})$  where the **node** attribute stores the ID of a node in the binary tree  $U$  and the **left**, **right** attributes store the left and right boundaries of an interval associated with the node specified in the **node** attribute. Clearly, the two tables  $L$  and  $R$  require  $O(n)$  space in total. They can be constructed in  $O(n \log_B n)$  time by scanning through the  $n$  intervals once. For each interval  $[l, r]$ , we compute the associated node  $x$  as described above and then insert the record  $(x, l, r)$  in  $L$  and  $(x, r, l)$  in  $R$ . Note that  $U$  is never explicitly stored. In particular, if no intervals are associated with a node  $x$  in  $U$ , the tables  $L$  and  $R$  will not have an entry with the **node** attribute storing value  $x$ .

**Querying.** To search for the intervals enclosing a query point  $q \in \{1, \dots, 2^h - 1\}$ , we start from the root  $2^{h-1}$  and search down the path to  $q$ . Suppose we are at a node  $x$  along this path. If  $q \leq x$ , then we report those interval  $[l, r]$  with  $l \leq q$  since its right endpoint  $r \geq x \geq q$ . (Those interval  $[l, r]$  such that  $l > q$  need not be reported because  $q$  is outside the interval.) This is done by searching the table  $L$ . Similarly, if  $q \geq x$ , then we need only report those interval  $[l, r]$  with  $r \geq q$ . This is done by searching table  $R$ . If  $q < x$  or  $q > x$ , we search down  $x$ 's left or right son respectively. Otherwise, we can stop.

In general, a query requires accessing  $\leq h$  nodes in the RI-tree. For  $0 \leq i \leq h-1$ , if the node at height  $i$  contains  $k_i$  intervals enclosing the query point, retrieving these intervals assuming a  $B^+$ -tree index requires  $O(\log_B n + k_i/B)$  time. Summing up all the  $h$  levels, the complexity for a query is  $O(h \log_B n + k/B)$  where  $k = \sum_i k_i$  is the total number of intervals to be reported.

## 4.2 Finding Ancestors

By Proposition 2, our intervals are nested. Thus, for those intervals associated a node in  $U$ , if they are already sorted in order of the left endpoints, they must

also be sorted in (reverse) order of their right endpoints. So, we will only keep one table, say, L.

To find the ancestors of a context node  $c$ , we search the RI-tree with the query point  $f(c)$ . We start from the root  $2^{h-1}$ . If  $f(c) \leq 2^{h-1}$ , we examine the table L and report all records  $t$  such that  $t.\text{node} = 2^{h-1}$  and  $t.\text{left} < f(c)$ . If  $f(c) \geq 2^{h-1}$ , we examine the table L and report all records  $t$  such that  $t.\text{node} = 2^{h-1}$  and  $t.\text{right} > f(c)$ . After that, we move down the tree one level to examine either the node  $1 \cdot 2^{h-2}$  or  $3 \cdot 2^{h-2}$  depending on  $f(c) < 2^{h-1}$  or  $f(c) > 2^{h-1}$ . The process is then repeated until we reached a node  $x$  where  $f(c) = x$ . To bound the query complexity, note that our interval boundaries lie within the range  $[1, \dots, 2n]$ . Thus, we have  $h = O(\log n)$  and the query time becomes  $O(\log n \log_B n + k/B)$ .

We remark that the RI-tree can also be used to find the parent of a node though it is not as efficient as the method we mention in the next section. Observe that the intervals enclosing the query point  $f(c)$  are nested and the innermost one corresponds to the parent of node  $c$ . We can make use of the RI-tree to find the innermost interval enclosing the query point. This is done by computing the interval with the largest left endpoint among those enclosing  $f(c)$ . Thus the query time is the same as that of finding ancestors.

## 5 The Local Axes

In this section, we modify our previous table XTfirst to support the child, parent and preceding/following-sibling axes.

To find the children of a context node  $c$ , enumerating all descendants of  $c$  and picking the maximal ones, i.e., the ones not contained in any other descendant of  $c$ , would be slow. Instead, we change one of the tables, XTfirst, to XTfirst(parent, first, last, data) with tuples sorted in the order of the key (parent, first). With this change, the children of context node  $c$  can be obtained by selecting the tuples  $t$  where  $t.\text{parent} = f(c)$ . Querying on the child axis will take  $O(\log_B n + k/B)$  time where  $k$  is the number of children in the result set.

In addition, finding parent is also easy: Suppose the parent attribute of (the record in XTfirst for) the context node  $c$  contains the value  $f(p)$ . Then we search for the record  $t$  where  $t.\text{first} = f(p)$ . This takes  $O(\log_B n)$  time assuming a  $B^+$ -tree index is built on attribute first. Finding the preceding or following siblings of  $c$  can be done by finding the parent  $p$  of  $c$  and then the children  $v$  of  $p$  with  $f(v) < f(c)$  (for preceding-siblings) or  $f(v) > f(c)$  (for following-siblings). Thus, these axes take  $O(\log_B n + k/B)$  time as well, where  $k$  is, again, the output size.

The drawback of introducing the parent attribute to table XTfirst is that the complexity for the following axis will be larger as the records are now clustered around the parent attribute instead of the first attribute. However, as we have a  $B^+$ -tree index on the first attribute, we can still bound the complexity by the depth  $d$  of the document tree as follows. Recall that the tuples are arranged in the order of their parents' first-order number. Consider an arbitrary context node  $c$  and let its ancestors be  $u_1, u_2, \dots, u_{d'}$ ,  $d' \leq d$ , as we walk up the path from  $c$  to the root of the document. For convenience, define  $u_0 = c$ . Then for  $1 \leq i \leq d'$ ,

$u_{i-1}$  is a child of  $u_i$ . Let  $w_{i1}, w_{i2}, \dots$  be the children of  $u_i$  that follows  $u_{i-1}$ ; and let  $W_{i1}, W_{i2}, \dots$  be the subtrees rooted at these nodes respectively. The following axis of  $c$  is then the union of those nodes in  $W_{i1}, W_{i2}, \dots$ , for  $1 \leq i \leq d'$ . Fix an  $i$  and observe that the first-order numbers of the nodes in  $W_{i1}, W_{i2}, \dots$  are larger than that of  $u_i$  but smaller than those of the nodes in  $W_{i+1,1}, W_{i+1,2}, \dots$ . The first-order number of the parent of a node in  $W_{i1}, W_{i2}, \dots$  is either the first-order number of  $u_i$  or that of some node in  $W_{i1}, W_{i2}, \dots$ . Hence, we can conclude that the following nodes of  $c$  are partitioned into at most  $2d'$  segments in the physical storage. Hence retrieving all the segments require  $O(d \log_B n + k/B)$  time. In practice,  $d$  is often a small value.

## 6 Handling Name Tests

We are now ready to present our final design of the index structure which supports name tests. This is important because XPath expressions often contain tests on the tag names. To support such expressions efficiently, we introduce a tag attribute in the database table. Thus the modified tables are:  $\text{XTfirst}(\text{tag}, \text{parent}, \text{first}, \text{last}, \text{data})$  and  $\text{XTlast}(\text{tag}, \text{last}, \text{first}, \text{data})$ . The  $B^+$ -tree index for attribute `first` in  $\text{XTfirst}$  is now extended to  $(\text{tag}, \text{first})$ . Then with name test, the descendant, preceding, child, preceding- and following-sibling axes are now supported in  $O(\log_B n + k/B)$  time. The following axis with name test is supported in  $O(d \log_B n + k/B)$  time. The ancestor axis, with name test, takes  $O(\log n \log_B n + d/B)$  (recall  $d$  is the document tree height) while taking  $O(\log n \log_B n + k/B)$  time without name test. The parent axis remains to be  $O(\log_B n)$  with or without name test.

For the descendant, preceding and following axes without name test, we make use of the RI-tree, exploiting the nesting properties of the intervals. To compute the preceding axis of context node  $c$ , we need to find all document nodes  $v$  such that  $1 \leq l(v) < f(c)$ . Consider an RI-tree node  $x$ . If  $x \geq f(c)$ , then it follows from the construction of RI-tree that any interval associated with  $x$  must have right endpoint at least  $f(c)$  and hence not in the answer set. Hence we need only consider those node  $x < f(c)$ . For each such node, we examine its associated intervals and report only those with right endpoint less than  $f(c)$ . It can be shown that there are at most  $\log n$  RI-tree nodes which contain intervals not in the answer set. (The main idea is that each level of the RI-tree  $U$  can have at most one node containing intervals not in the answer set. Formal proof will be given in the full paper.) The query complexity is then  $O(\log n \log_B n + k/B)$  where  $k$  is the output size. The following axis is handled similarly.

To compute the descendant axis of  $c$ , we can do even better by using the fact that the query range  $[f(c), l(c)]$  is itself one of the nested intervals. As before, we need not consider those RI-tree node  $x$  outside  $(f(c), l(c))$ . Let  $[f(c), l(c)]$  be associated with node  $z$  in the RI-tree. We claim that for any RI-tree node  $x$  in  $(f(c), l(c))$  except for  $z$ , all its associated intervals  $[l, r]$  must satisfy  $f(c) < l < r < l(c)$ . Otherwise, if  $l < f(c) < l(c) < r$ , then  $[l, r]$  would have been associated with a node  $y$  at least as high as  $z$  in the RI-tree. So,  $y$  is higher than

$x$ , a contradiction. Hence the claim follows. Thus, the query complexity for the descendant axis is  $O(\log_B n + k/B)$ .

Now, the only axes for which our structure does not guarantee a worst case time bound are the child, preceding- and following-siblings without name test.

## 7 Experimental Evaluation

In this section, we present our experimental results on the query performance of each axis. Our XML documents are generated by XMLgen of the XMark benchmark project [24]. The tool produces XML documents of a specific structure but allows us to specify the document size.

We implemented our index structure as well as the B<sup>+</sup>-tree and R-tree version of Grust et al.’s design for comparison. All programs are written in Java. All the experiments were performed on an Intel Pentium 4 2.6 GHz PC with 1GB RAM running Windows XP. Our purpose here is not to compare the different designs on absolute terms but to elicit their behaviours as the document sizes and/or output sizes grow.

We used the PostgreSQL database since it is equipped with GiST that supports both of the indexes. For the B<sup>+</sup>-tree version, we implemented its stretched version (Section 4.2 [13]). For the R-tree version, we use the optimization described in Section 4.1 of [13] that minimizes the query window size using the subtree size as additional information. However, we only use the R-tree for indexing the two dimensional *pre/post* plane rather than making the whole tuple as a 5-dimensional descriptor. It is well-known that the performance of an R-tree deteriorates as the number of dimensions increases. The indexes and clustering they mentioned are also applied.

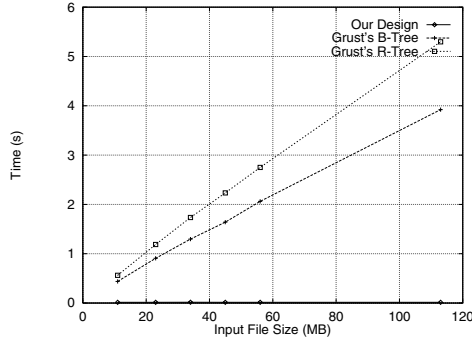
### 7.1 Range Searching on B<sup>+</sup>-Tree

Our descendant, preceding, child, preceding- and following-sibling axes with name test requires range searching on B<sup>+</sup>-tree, taking  $O(\log_B n + k/B)$  time. The following axis requires  $O(d \log_B n + k/B)$  time. To verify the formula, we design the experiments so that in each set, either only the input size or the output size varies.

**Input Size.** To show the effect on varying the input size  $n$ , we choose a path that returns only one node in whatever input size.

Figure 4 shows that the times for Grust et al’s design grows linearly instead of logarithmically in the input size  $n$ . The reason is that they need to filter out nodes that do not match the node test (false hits). Our design (near the horizontal axis in the graph) takes negligible time even for different input sizes. This indicates that the  $\log_B n$ -term is very small (at least for input size ranging from 11 to 113 MB).

**Output Size.** To show the effect on varying output size  $k$ , we use a large enough input file (56MB) and provide different name tests that yield different output sizes.



**Fig. 4.** Effect of Input Size on Descendant Axis with Name Test (Query: /descendant::open\_auctions)

Figure 5 shows the result for descendant, child, preceding and following axes. In all cases, our design gives linear growth with respect to the output size  $k$  while Grust’s design may sometimes give irregular growth, especially for their R-tree version. This is possibly due to the overhead of false hits, and the fact that R-tree’s performance has no worst case guarantee.

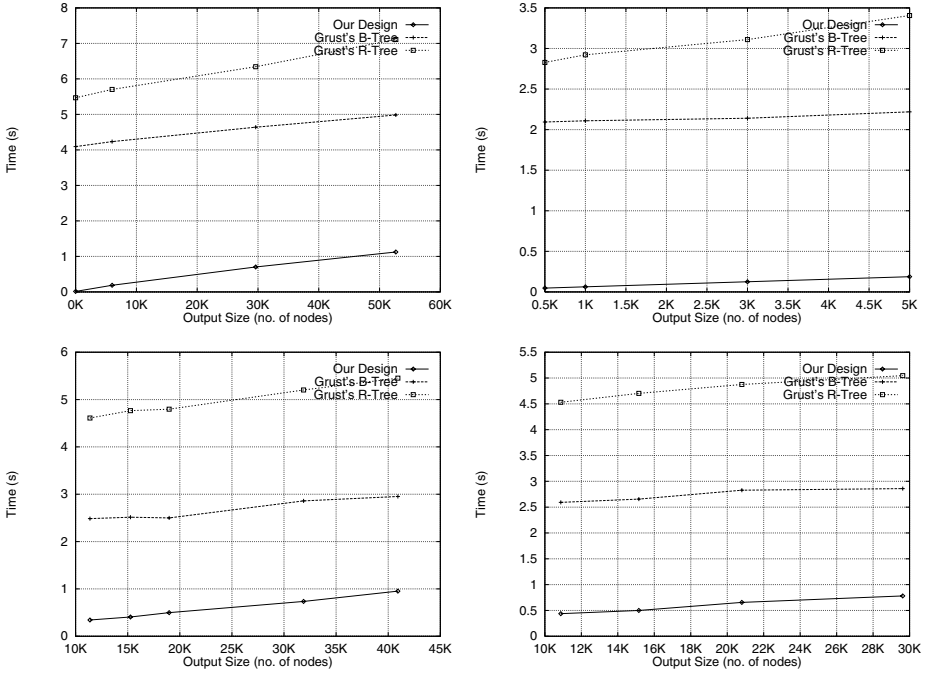
### 7.2 Interval Queries of RI-Tree

Our ancestor axis (without name test) is done by an interval query on the RI-tree using  $O(\log_2 n \log_B n + k/B)$  time. Due to the given DTD of the XMLgen, the generated XMLs have a depth of  $d = 11$  which is too small for showing the effect of output size. Therefore we focus on the effect of the input size in this subsection. Here, we first use a descendant axis to select a single element node called ‘africa’ that has  $k = 2$  ancestors for any input size. We can then vary the input size  $n$ . In measuring the time, we only count the second step (i.e., the ancestor axis).

The result in Figure 6 (left) shows that Grust et al.’s B<sup>+</sup>-tree takes linear time in  $n$  while their R-tree and our design have negligible time. To magnify the processing time to observable magnitude, we perform the previous experiment again but for each point we repeat the operation 1,000 times. Then we plot the graph with x-axis in log-scale. Figure 6 (right) verified our claims that the time grows in  $\log n \log_B n$  time where  $\log_B n$  can be treated as constant for our range of input sizes.

### 7.3 Range Query on RI-Tree

Our descendant axis (without name test) is handled by range query on the RI-Tree and takes  $O(\log_B n + k/B)$  time. Our preceding and following axes without name test are handled by range query on RI-Tree as well and take  $O(\log n \log_B n + k/B)$  time.



**Fig. 5.** Effect of Output Size. Top-left: descendant axis with name test. Top-right: child axis with name test. Bottom-left: preceding axis with name test. Bottom-right: following axis with name test

**Output Size.** For conciseness, we only show the graph for the descendant and following axes. The queries used for the descendant axis are:

$$/\text{descendant}::?/\text{descendant}::*$$

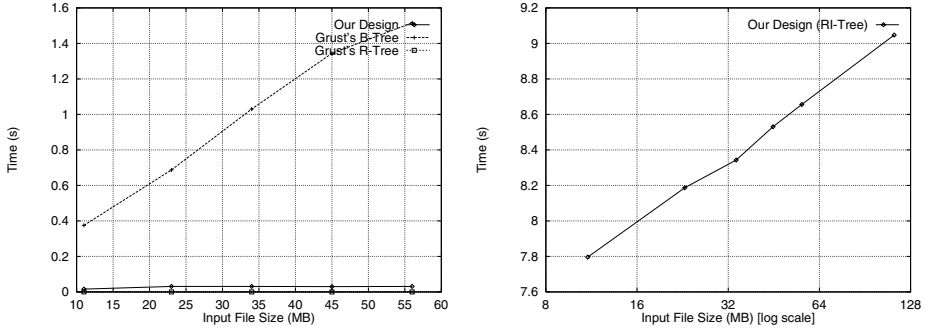
where ? is certain tag name. Those for the following axis are:

$$/\text{descendant}::?/\text{following}::*$$

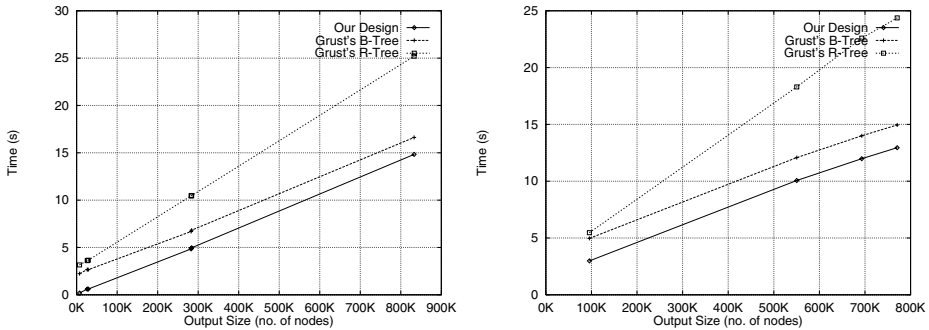
The graph for the preceding axis is similar to that of the following axes.

In these cases, Grust et al.'s B<sup>+</sup>-tree should have the best performance since it favors range query. Figure 7 shows that our RI-tree (originally designed for interval queries) has comparable performance with B<sup>+</sup>-tree on range queries. In comparison, the R-tree also grows linearly but at a somewhat larger slope.

**Input Size.** Without name test, it is difficult to control the output size when we vary the input size. So, we skip its evaluation in this paper.



**Fig. 6.** Effect of Input Size on Ancestor Axis without Name Test. Left: linear scale. Right: x-axis in log scale



**Fig. 7.** Effect of Output Size on Descendant Axis (left) and Following Axis (right) both without Name Test

## 8 Conclusion

We studied the problem of indexing an XML document with traditional relational database to support all the XPath axes. We consider the physical ordering of the data and derived worst case upper bounds on the execution time for most of the XPath axes. The only axes for which our structure does not provide a worst case guarantee on the performance bound are the child, preceding- and following-siblings when name tests are not present.

We verified experimentally our formulas by performing a series of experiments. The results show that in our design the input file size has very small effect compared to the output size. Thus, it is suitable for indexing large XML documents, even when they have varying structures. (Our design does not require a DTD of the documents.)

Our design requires only B<sup>+</sup>-tree index which is available in practically any relational database. Without the need of R-tree index, our index structure will have more predictable performance compared with the XPath Accelerator of Grust et al. This may be a desirable feature when it is used as a basic building block for other methods of XPath evaluation.

Our observation that RI-Tree can handle range queries on nested intervals is also interesting. Perhaps more interesting properties of the RI-tree are waiting ahead for our discovery.

## References

1. S. Al-Khalifa, H. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: a primitive for efficient XML query pattern matching. In *18th International Conference on Data Engineering*, pages 141–152, 2002.
2. A. Berglund, S. Boag, D. Chamberlin, M. Fernandez, M. Kay, J. Robie, and J. Simeon. XML Path Language (XPath) 2.0. Technical Report W3C Working Draft, Version 2.0, World Wide Web Consortium, Aug. 2002.
3. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the 2002ACM SIGMOD Conference on the Management of Data*, pages 310–321, 2002.
4. Y. Chen, S. B. Davidson, and Y. Zheng. BLAS: an efficient xpath processing system. In *Proceedings of the 2004ACM SIGMOD Conference on the Management of Data*, pages 47–58, 2004.
5. S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 263–274, 2002.
6. W. W. W. Consortium. Extensible markup language (XML) 1.0 (second edition) – W3C recommendation. Available at <http://www.w3.org/TR/2000/WD-xml-2e-20000814>, 2000.
7. B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 341–350, 2001.
8. D. deHaan, D. Toman, M. P. Consens, and M. T. Ozsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proceedings of the 2003ACM SIGMOD Conference on the Management of Data*, pages 623–634, 2003.
9. R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23th International Conference on Very Large Data Bases*, pages 436–445, 1997.
10. G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 95–106, 2002.
11. G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Database Systems*, pages 179–190, 2003.
12. G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: improving time and space efficiency. In *19th International Conference on Data Engineering*, pages 379–390, 2003.
13. T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Transactions on Database Systems*, 29(1):91–131, 2004.
14. H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: indexing XML data for efficient structural joins. In *19th International Conference on Data Engineering*, pages 253–263, 2003.
15. R. Kaushik, P. Bohannon, J. Naughton, and H. Korth. Covering indexes for branching path queries. In *Proceedings of the 2002ACM SIGMOD Conference on the Management of Data*, pages 133–144, 2002.

16. R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *18th International Conference on Data Engineering*, pages 129–140, 2002.
17. D. D. Kha, M. Yoshikawa, and S. Uemura. A structural numbering scheme for XML data. In *EDBT Workshops*, pages 91–108, 2002.
18. H.-P. Kriegel, M. Potke, and T. Seidl. Managing intervals efficiently in object-relational databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 407–418, 2000.
19. Y. K. Lee, S. Yoo, K. Yoon, and P. B. Berra. Index structures for structured documents. In *Digital Libraries*, pages 91–99, 1996.
20. Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 361–370, 2001.
21. T. Milo and D. Suci. Index structures for path expressions. In *7th International Conference on Database Theory*, pages 277–295, 1999.
22. P. Rao and B. Moon. PRIX: indexing and query XML using Prüfer sequences. In *20th International Conference on Data Engineering*, pages 288–300, 2004.
23. A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of XML documents. In *Proceedings of the 3rd International Workshop on the Web and Databases*, pages 137–150, 2000.
24. A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: a benchmark for XML data management. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 974–985, 2002.
25. I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proceedings of the 2002ACM SIGMOD Conference on the Management of Data*, pages 204–215, 2002.
26. H. Wang, S. Park, W. Fan, and P. S. Yu. ViST: A dynamic index method for queryinh XML data by tree structures. In *Proceedings of the 2003ACM SIGMOD Conference on the Management of Data*, pages 110–121, 2003.
27. W. Wang, H. Jiang, H. Lu, and J. X. Yu. PBiTree coding and efficient processing of containment joins. In *19th International Conference on Data Engineering*, pages 391–, 2003.
28. M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology*, 1(1):110–141, 2001.
29. C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of the 2001ACM SIGMOD Conference on the Management of Data*, pages 425–436, 2001.
30. N. Zhang, V. Kacholia, and M. T. Ozsu. A succinct physical storage scheme for efficient evaluation of path queries in XML. In *20th International Conference on Data Engineering*, pages 56–65, 2004.