Accelerating XPath Location Steps

Torsten Grust
University of Konstanz
Department of Computer and Information Science
P.O. Box D 188, D-78457 Konstanz, Germany
Torsten.Grust@uni-konstanz.de

ABSTRACT

This work is a proposal for a database index structure that has been specifically designed to support the evaluation of XPath queries. As such, the index is capable to support *all* XPath axes (including `ancestor`, `following`, `preceding-sibling`, `descendant-or-self`, *etc.*). This feature lets the index stand out among related work on XML indexing structures which had a focus on regular path expressions (which correspond to the XPath axes `children` and `descendant-or-self` plus name tests). Its ability to start traversals from arbitrary context nodes in an XML document additionally enables the index to support the evaluation of path traversals embedded in XQuery expressions. Despite its flexibility, the new index can be implemented and queried using purely relational techniques, but it performs especially well if the underlying database host provides support for R-trees. A performance assessment which shows quite promising results completes this proposal.

1. INTRODUCTION

It is hard to find a proper answer to the question of why XML has been so successful in being adopted as a universal data exchange format, but a piece of the truth might be the following: the data type underlying the XML paradigm, namely the *tree*, is expressive enough to capture the structure of diverse data sources, yet simple enough to lend itself to efficient as well as elegant (*esp.* recursive) algorithms operating on such data.

Essentially, XML provides an unlimited number of *tree dialects*, some of which have been formally described by DTDs or XML Schema types, some of which are used in a one-time or ad-hoc manner (schema-less instances), however. The elegance and simplicity of the XML approach made hundreds of dialects emerge, among these the most widely used dialect HTML (or XHTML, to be precise). Other dialects include the NITF standard (data exchange in the news industry), the weather markup language WeatherML, CellML (computer-based biological models), or XMLPay, whose instances describe Internet-based payments.

As more and more sources switch over and express their contents using XML dialects, the sheer volume of data calls for XML-aware data management solutions build on database technology.

The database community is well underway to adapt its technology to host large XML stores and to query these stores efficiently, preferably using query languages developed in the XML domain: XPath [1] and XQuery [4].

In line with the tree-centric nature of XML, XPath provides operators to describe path traversals in a tree-shaped document. Path traversals evaluate to a collection of subtrees (*forests*), which may then, recursively, be subject to further traversal. Starting from a so-called *context node*, an XPath query traverses its input document using a number of *location steps*. For each step, an *axis* describes which document nodes (and the subtrees below these nodes) form the intermediate result forest for this step. The XPath specification [1] lists a family of 13 axes (among these the `children` and `descendant-or-self` axes, probably more widely known by their mnemonic abbreviations `/` and `//`, respectively).

The recursion inherent in tree-shaped data types as well as in operations over these types turns out to be a challenge for database-based approaches to XML storage and querying. This is especially true for relational database technology whose native data model (tables of tuples) and native query language SQL have originally not been designed to deal with recursion.

Recently, a whole host of efficient storage structures and indexing schemes that summarize an XML document so that these problems can be dealt with have been developed [14, 6, 9]. Almost exclusively, these techniques put their focus on providing efficient support for sequences of `/` and `//` location steps, the *regular path expressions*, however. This is hardly adequate support for the XPath language (or XQuery for that matter, whose expression syntax allows for embedded path traversals). Additionally, these proposals quite often rely on query processing algorithms which call for implementation techniques that lie outside the relational domain, with all the related drawbacks (software layers in addition to the database host, transactional issues, performance implications, *etc.*)

This work proposes an index structure, the *XPath accelerator*, that can completely live inside a relational database system, *i.e.*, it is a *relational storage structure* in the sense of [13]: the index can be constructed and queried using relational idioms only. Its implementation, however, can benefit

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACM SIGMOD 2002, June 4–6, Madison, Wisconsin, USA
Copyright 2002 ACM 1-58113-497-5/02/06 ...$5.00.

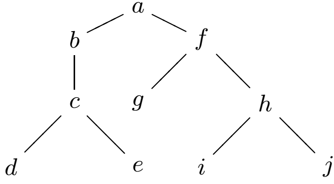


Figure 1: Tree representation of a small XML document instance.

from advanced index technology, *esp.* the R-tree, that has by now found its way into mainstream relational database systems. The index has been developed with a close eye on the XPath semantics and is thus able to support *all* XPath axes. Loading as well as querying the index is simple, yet its performance comes close to or beats measurements published in recent related work.

The paper proceeds as follows. The next section provides a closer look at the XPath axes and their semantics. This will yield the notion of *document regions*. An efficient encoding for these is then described in Section 3. Such an encoding can be generated on the fly during XML document loading. This will be detailed in Section 4. If the underlying database host does not provide R-tree support, it is particularly important to further optimize index lookups (Section 5). Section 6 reports on performance measurements and comparisons. The two final sections review related work and then conclude.

2. XPATH AXES AND XML DOCUMENT REGIONS

The core of the XPath language, the *path expression*, directly reflects the recursive nature of tree-shaped data. To be more precise, XPath expressions operate on trees of *element* or *attribute* nodes, a small example of which is depicted in Figure 1 (details of the XPath data model can be found in [1]).

In this tree, the inner nodes a, b, c, f, g, h represent XML element nodes, the leaf nodes d, e, g, i, and j represent either element or attribute nodes (later, we will care about this distinction and also add element content). A corresponding XML fragment would be:

```
<a>
    <b>
        <c>
            <d> </d><e> </e>
        </c>
    </b>
    <f>
        <g> </g>
        <h>
            <i> </i><j> </j>
        </h>
    </f>
</a>
```

(To synchronize some terminology: node a is the *root* of the tree; $height(v)$ is the length of the longest path from v to a leaf in the subtree rooted at v, *e.g.*, $height(a) = 3$; $level(a) = 0$, while $level(v) = n$ if the path from the root to v has length n.)

XPath expressions specify a tree traversal via two parameters: (1) a *context node* (not necessarily the root) which is the starting point of the traversal, (2) and a sequence of *location steps* syntactically separated by `/`, evaluated from left to right. Given a context node, a step's *axis* establishes a subset of document nodes (a *document region*). This set of nodes, or forest[1], provides the context nodes for the next step which is evaluated for each node of the forest in turn. The results are unioned together and sorted in document order.

Axis α	Result Forest
`child`	direct element child nodes of v
`descendant`	recursive closure of `child`
`descendant-or-self`	like `descendant`, plus v
`parent`	direct parent node of v
`ancestor`	recursive closure of `parent`
`ancestor-or-self`	like `ancestor`, plus v
`following`	nodes following v in doc. order
`preceding`	nodes preceding v in doc. order
`following-sibling`	like `following`, same parent as v
`preceding-sibling`	like `preceding`, same parent as v
`attribute`	attribute nodes of node v
`self`	v
`namespace`	namespace nodes of node v

Table 1: Semantics of axes α supported by XPath (step v/α).

To illustrate the semantics of the XPath axes, Figure 2 depicts the result forests for three steps along different axes taken from context node f (note that the `preceding` axis does not include the ancestors of the context node). Table 1 lists all XPath axes and verbally sketches their semantics. We will provide a precise specification soon.

2.1 XML Document Partitions

There are four axes which are of primary interest to us, namely: `descendant`, `ancestor`, `following`, and `preceding`. For the sole purpose of easy identification, we will call these *major axes* from now on.

For any given context node v, the four major axes specify a *partitioning* of the document containing v (this is our main motivation for calling the respective result forests document *regions*). The node set

$$v/\texttt{descendant} \cup v/\texttt{ancestor} \cup \\ v/\texttt{following} \cup v/\texttt{preceding} \cup \{v\}$$

contains each document node exactly once. Figure 2 illustrates this property for context node f (note: f`/following` yields the empty forest for this document instance). We have

$$\begin{pmatrix} f/\texttt{descendant} \cup f/\texttt{ancestor} \cup \\ f/\texttt{following} \cup f/\texttt{preceding} \cup \{f\} \end{pmatrix} = \{a \dots j\}.$$

The evaluation of an XPath step sequence thus amounts to the repeated computation of partitions of XML document trees.

The key idea of this work is to find an index structure such that, for any given context node, we can efficiently determine the set of nodes in the four document partitions specified by the major axes. The further XPath axes (`parent`, `child`, `descendant-or-self`, `ancestor-or-self`, `following-sib-`

[1]In the following we will frequently identify a node and the subtree rooted at that node.

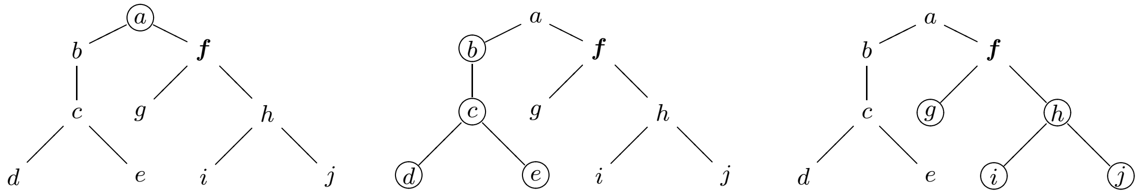


Figure 2: XPath semantics: circled nodes are elements of the result forest if an `ancestor::*` (`preceding::*`, `descendant::*`) step is taken from context node f (shown from left to right).

`ling`, and `preceding-sibling`) determine specific supersets or subsets of these node sets which are easy to characterize. Attribute nodes will be marked as such during document loading to support the `attribute` axis.

To complete this section, let us note that an XPath step along axis α can be augmented by a *name test* for element tag or attribute name n (the syntactic form is α`::`n). Again, this specifies a subset of the region indicated by axis α, containing those nodes with element tag or attribute name equal to n. The name test α`::*` succeeds for any element tag or attribute name.[2]

3. ENCODING XML DOCUMENT REGIONS

We are now left with the challenge to find an encoding of the tree-shaped node hierarchy in an XML document that

1. retains the region notion induced by the four major XPath axes, and
2. can be efficiently supported by existing database technology.

Here, *efficiency* means that the encoding has to map the input tree-shape into a domain in which a node's region membership may be tested by a simple relational query.

The problem is that the XPath semantics are far from simple. To quote the XPath 2.0 specification, "*... the* `preceding` *axis contains all nodes in the same document as the context node that are before the context node in document order, excluding any ancestors and excluding attributes nodes and namespace nodes.*" [1]

Informally, the *document order* in an XML instance orders its nodes corresponding to the order in which a sequential read of the XML (textual) representation of the instance would encounter the nodes. A much more useful characterization of document order in our context is that this order is determined by a *preorder traversal* of the document tree. In a preorder traversal, a tree node v is visited and assigned its *preorder rank* $pre(v)$ before its children are recursively traversed from left to right.

For the example instance shown in Figure 1, the document order is $a < b < c < d < e < f < g < h < i < j$, and thus $pre(a) = 0$, $pre(b) = 1$, ..., $pre(j) = 9$.

A *postorder traversal* is the dual of preorder traversal: a node v is assigned its *postorder rank* $post(v)$ after all its children have been traversed from left to right. Again, for the example we get $post(d) = 0$, $post(e) = 1$, ..., $post(a) = 9$.

[2]XPath furthermore includes a generic predicate test α`[`p`]` to constrain the result forest of a step but this is not the focus of this paper.

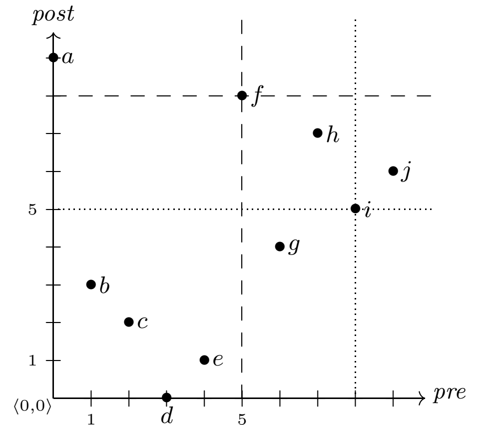


Figure 3: Node distribution in the ***pre/post*** **plane and XML document regions as seen from context nodes** f **(– –) and** i **(·······).**

As others have noted [7, 14, 18], one can use $pre(v)$ and $post(v)$ to efficiently characterize the descendants v' of v. We have that

$$v' \text{ is a descendant of } v \Leftrightarrow pre(v) < pre(v') \wedge post(v') < post(v) \ .$$

Intuitively, this may be read as: during a sequential read of the XML document, we have seen the opening tag `<v>` *before* `<v'>` and the closing tag `</v>` *after* `</v'>`. In other words, the element corresponding to v' is contained in the element corresponding to v.

This characterizes the `descendant` axis of context node v, but we can use $pre(v)$ and $post(v)$ to characterize all four major axes equally simple.

Figure 3 illustrates the node distribution of the example document after its nodes have been mapped into the *pre/post* plane (*e.g.*, document root a is located at coordinates $\langle pre = 0, post = 9\rangle$) like its preorder and postorder ranks determine).

As indicated, node f induces a partition of the plane into four disjoint regions (*cf.* Figure 2):

1. the lower-right partition contains all `descendant`s of f,
2. in the upper-left partition we find the `ancestor`s of f, *i.e.*, node a only,
3. the lower-left region hosts the nodes `preceding` f in document order, and finally

4. the upper-right partition represents the nodes `following` f in document order (as we have noted earlier, this region is empty for this example instance).

This characterization of document regions applies to all nodes in the plane (note that the `descendant` axis of node i is empty, since i is a leaf node). This means that we may pick any node v and use its location in the plane to start an XPath traversal, *i.e.*, make v the context node. This turns out to be an important feature when it comes to the implementation of XQuery, where iteration constructs (`for`, `every`, `some`, ...) arbitrarily bind context nodes and then traverse from there (which is different from the evaluation of standalone XPath queries, say, where the context node preferably is the document root).

3.1 Axes and Query Windows

Evaluating a step along a major axis thus amounts to respond to a rectangular region query in the *pre*/*post* plane. Database indices, *esp.* R-trees but also B-trees, are highly optimized to support this kind of query.

To support the further XPath axes and name tests, we need only little extra bookkeeping for each node.

For context node v, axes `ancestor-or-self` and `descendant-or-self` simply add v to the `ancestor` or `descendant` regions, respectively. Node v is easily identified in the plane since its preorder rank $pre(v)$ is unique. For axes `following-sibling` and `preceding-sibling` it is sufficient to keep track of the parent's preorder rank $par(v)$ for each node v (siblings share the same parent). $par(v)$ readily characterizes axes `child` and `parent`, too. To support the `attribute` axes and, in line with the XPath semantics, to exclude attribute nodes from all other axes, we maintain the boolean attribute $att(v)$ for each node v. Finally, name tests are supported by attribute $tag(v)$ which stores the element tag or attribute name for node v.

This completes the encoding. Each node v is represented by its 5-dimensional *descriptor*

$$desc(v) = \langle pre(v), post(v), par(v), att(v), tag(v) \rangle \ .$$

An XPath axis corresponds to a specific query window in the space of node descriptors. Table 2 summarizes the windows together with the corresponding axes they implement. A node v' is inside the query window if its descriptor $desc(v')$ matches the query window component by component (for the first two components, *pre* and *post*, $pre(v')$ and $post(v')$ have to lie inside the respective ranges). A $*$ entry indicates a *don't care* match which always succeeds. The query window for the name test α`::`n is $window(\alpha, v)$ with its *tag* entry set to n.

Note that we try to be specific in the definition of the query windows. For a node v' to be a child of context node v it is sufficient to test the condition $par(v') = pre(v)$, thus we could have defined

$$window(\texttt{child}, v) = \langle *, *, pre(v), \mathit{false}, * \rangle \ .$$

However, a child v' of v is clearly contained in the `descendant` region of v, so we additionally know that $pre(v) < pre(v') \wedge post(v') < post(v)$. Similar remarks apply to the windows assigned to the `parent` and `attribute` axes.

We will have to say more about essential opportunities to shrink window sizes in Section 5.

3.2 XPath Evaluation Scheme

We have now collected all the necessary pieces to specify a first relational SQL-based evaluation scheme for an XPath traversal.

Assume that we have loaded the node descriptors of a document into a 5-column table *accel* $\underline{pre \mid post \mid par \mid att \mid tag}$ (loading is discussed in the following section).

We specify the evaluation scheme inductively: if e denotes an XPath path expression and α denotes an axis, we define

$$query(e/\alpha) \quad = \quad \begin{array}{rl} \texttt{SELECT} & v'\texttt{.*} \\ \texttt{FROM} & query(e)\ v, accel\ v' \\ \texttt{WHERE} & v'\ \texttt{INSIDE}\ window(\alpha, v) \ . \end{array}$$

(The hypothetical SQL keyword `INSIDE` symbolizes the window test, *i.e.*, a conjunction of simple comparison operations on the descriptor components of v and v'.)

The SQL query binds v to the node descriptors which provide the context nodes for the next step along axis α. (Note that we can obtain a translation for a step of the form $e/\alpha[p]$ if we rewrite the `WHERE` clause into the obvious $(v'\ \texttt{INSIDE}\ window(\alpha, v)\ \texttt{AND}\ p(v'))$.)

The base case for this recursive translation scheme may be provided by any subset of node descriptors in table *accel* or, specifically if e is an *absolute* path expression, by the document root, *i.e.*, the only node v with $pre(v) = 0$.

As given, the translation scheme generates an SQL query of nesting depth n for a path expression of n steps. Straightforward query unnesting, however, may transform the original query into a flat n-ary self-join. For the XPath expression `/descendant::`n_1`/preceding-sibling::`n_2, for example, we obtain

$$\begin{array}{rl} \texttt{SELECT} & v_2\texttt{.*} \\ \texttt{FROM} & accel\ v_1, accel\ v_2 \\ \texttt{WHERE} & 0 < v_1.pre \\ \texttt{AND} & v_1.tag = n_1 \\ \texttt{AND} & v_2.pre < v_1.pre\ \texttt{AND}\ v_2.post < v_1.post \\ \texttt{AND} & v_2.par = v_1.par \\ \texttt{AND} & v_2.tag = n_2 \ . \end{array}$$

4. XML INSTANCE LOADING

Now that we are this far, we know that *loading* an XML document instance into the database essentially means to map its nodes into the 5-dimensional descriptor space. Each document node makes for exactly one node in the descriptor space so that the size of the loaded index will be linear in the size of the input instance.

All five components of the node descriptors can be computed during a single sequential parsing pass over the input XML instance. If we use an event-based parsing XML framework, like SAX [15], we are guaranteed to need only very limited scratch space during loading: the size of temporary memory needed is bounded by the instance's *height* (not by its size).

In a nutshell, the instance loader is implemented by two simple SAX callback procedures: $startElement(t, a, atts)$ and $endElement(t)$. The SAX parser backend calls procedure $startElement(t, a, atts)$ whenever it encounters an XML opening element tag. Parameter t then holds the tag name, boolean parameter a is set to *false* (indicating that the parser has encountered an element, not an attribute), and *atts* is bound to a list of attribute names if the element contains attributes or *nil* otherwise. Procedure $endElement(t)$ is invoked whenever a closing tag for element t is encoun-

Axis α	Query window $window(\alpha, v)$				
	pre	*post*	*par*	*att*	*tag*
`child`	$\langle(pre(v), \infty)$	, $[0, post(v))$	, $pre(v)$	, *false* ,	$*$ $\rangle$
`descendant`	$\langle(pre(v), \infty)$	, $[0, post(v))$	, $*$	, *false* ,	$*$ $\rangle$
`descendant-or-self`	$\langle[pre(v), \infty)$	, $[0, post(v)]$	, $*$	, *false* ,	$*$ $\rangle$
`parent`	$\langle[par(v), par(v)]$	, $(post(v), \infty)$	, $*$	, *false* ,	$*$ $\rangle$
`ancestor`	$\langle[0, pre(v))$	, $(post(v), \infty)$	, $*$	, *false* ,	$*$ $\rangle$
`ancestor-or-self`	$\langle[0, pre(v)]$	, $[post(v), \infty)$	, $*$	, *false* ,	$*$ $\rangle$
`following`	$\langle(pre(v), \infty)$	, $(post(v), \infty)$	, $*$	, *false* ,	$*$ $\rangle$
`preceding`	$\langle(0, pre(v))$	, $(0, post(v))$	, $*$	, *false* ,	$*$ $\rangle$
`following-sibling`	$\langle(pre(v), \infty)$	, $(post(v), \infty)$	, $par(v)$	, *false* ,	$*$ $\rangle$
`preceding-sibling`	$\langle(0, pre(v))$	, $(0, post(v))$	, $par(v)$	, *false* ,	$*$ $\rangle$
`attribute`	$\langle(pre(v), \infty)$	, $[0, post(v))$	, $pre(v)$	, *true* ,	$*$ $\rangle$

Table 2: XPath axes α and their corresponding query windows $window(\alpha, v)$ (context node v).

tered.

We display the two callback procedures below.[3] To keep track of elements whose opening tag we have already seen but whose closing tag is still to come, we maintain a stack S of yet incomplete node descriptors. (The stack operations *push*, *pop*, *top*, and *empty* should be self-explaining.) Whenever we encounter an element's closing tag, we are ready to fix up its yet unspecified *post* component and then insert the node into the database table. Obviously, the size of S is bounded by the input instance's height. No additional temporary memory space is needed.

```
startElement(t, a, atts)
  v ← ⟨pre = gpre, post = ␣,
        par = (S.top()).pre, att = a, tag = t⟩;
  S.push(v);
  gpre ← gpre + 1;
  FOR v' IN atts DO
    startElement(v', true, nil);
    endElement(v');
```

```
endElement(t)
  v ← S.pop();
  v.post ← gpost;
  gpost ← gpost + 1;
  insert v into table accel;
```

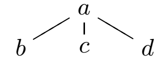
Loading is initiated as follows:

```
gpre ← 0;   gpost ← 0;
S.empty();
S.push(⟨pre = −1, post = ␣, par = ␣, att = ␣, tag = ␣⟩);
SAXparseFile();
S.pop();
```

Note how procedure $startElement(t, a, atts)$ itself generates events for all attributes v' in *atts* associated with element t. This ensures that attribute nodes are inserted with correct document order ($pre(v')$ value). An XML element like

```
<a b="" c="">
    <d> </d>
</a>
```

will thus be treated like the document tree



which is in line with XML document order semantics (the attributes of a node v appear before v's children in document order).

Up to now we have not discussed how to store the actual element content (CDATA sections) of an XML document. Two alternatives suggest themselves:

1. Handle the content *inline*, *i.e.*, treat element content like an additional child of its containing element (much like the attribute treatment sketched above). As a consequence, CDATA content is stored right next to the containing node.
2. Maintain a *separate* table $\underline{pre \mid cdata}$, save the element content in the *cdata* column and establish *pre* as a foreign key referencing the *accel* table.

The latter variant has been identified as superior by previous work [8].

Finally, remember that name tests are implemented as equality tests on the *tag* component of the node descriptors. It is sufficient to store hash values rather than the actual element names in the *tag* component. If the DTD of the input document is known *a priori*, we can even set up a simple translation table to map element names to numerical values before loading starts.

A word on updating the *accel* table. Due to the order in which the preorder and postorder traversal visit tree nodes, it is necessary to renumber all nodes in the `following` and `ancestor` axes of a newly inserted document node. To delete a node, however, it is sufficient to remove its descriptor entry from *accel*.

4.1 Node Descriptor Indexing

Node descriptors are elements of a 5-dimensional space. Domains of such dimensionality have been found to be efficiently supported by R-trees [2]. Our experiments indeed indicate that XPath step evaluation with the help of R-trees performs well (Section 6).

Our approach to model document regions via query windows is, of course, directly tailored to be supported by a multi-dimensional index structure like the R-tree [10]. R-trees are well suited to accelerate XPath location steps for a number of other reasons.

[3] For the sake of clarity, note that we slightly simplify the actual implementation. The real loader code, however, is only marginally different.

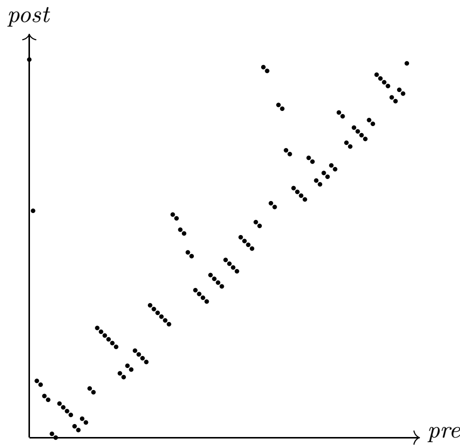


Figure 4: Example of a *pre/post* rank distribution for an XML document instance of 100 nodes.

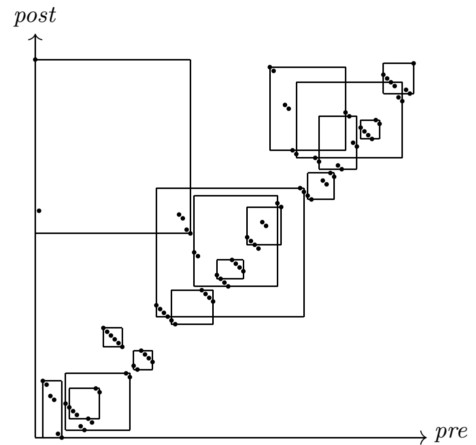


Figure 5: Leaf level of a *preorder packed* R-tree after loading an XML instance of 100 nodes (leaf capacity 6 nodes).

4.2 R-trees

Figure 4 shows a typical node distribution in the *pre/post* plane for an XML instance of 100 nodes. The diagonal of this plane is tightly packed with nodes, while the upper left is only sparsely populated. The lower right half is completely empty. (This is due to dependencies between the tree height and the preorder as well as postorder ranks in a tree—Section 5 will investigate this more closely to optimize index lookups.) R-trees adapt well to such distributions because of their *incomplete partitioning* of the space (as opposed to *space partitioning* trees like the quad tree). The data-driven R-tree remains balanced even in the presence of skewed distributions.

If R-trees are indeed supported by the database host, we can further optimize the XML bulk-loading process and use *R-tree packing* [12] techniques: at the cost of using temporary storage for sorting, we insert node descriptors in increasing order of *pre* values. This insertion order leads to a 100 % storage utilization in the R-tree leaves and additionally improves query performance considerably as coverage and overlapping of the leaves are minimized (Figure 5).

Note how the R-tree leaf level reflects the typical shape of an XML document tree in which the upper levels contain significantly fewer nodes than the lower levels: the upper three levels of the example instance are completely covered by two R-tree leaves only.

Preorder packing the R-trees had an additional beneficial effect in our implementation of the XPath accelerator: R-tree window queries returned the result nodes in increasing order of *pre* values, *i.e.*, in document order.[4] Since the XQuery specification demands document order on forests resulting from path expressions, this saved the implementation from extra sorting effort. For XPath, the preorder packing facilitates the implementation of *context positions* [1, Section 2.3.3] which are used in XPath predicates of the form α`[position() = `i`]`.

4.3 B-trees

Should R-trees be unavailable, a combination of B-tree indices can lead to good performance figures, too. We created two ascending B-tree indices on the *pre* and *post* columns of the *accel* table, respectively (note that both *pre* and *post* are unique). Additionally, we requested to cluster the *accel* table with respect to the *pre* index. This led, just like in the R-tree variant, to query results that were sorted in document order. (This time, sorting is guaranteed.)

In the B-tree case, an XPath axis query window is searched using two independent B-tree range scans on both the *pre* and *post* indices. The SQL query optimizer of the relational database system we were using in our experiments, IBM DB2 V7.1, recognized the opportunity to exploit index intersection (plan operator `IXAND`) to efficiently compute the window contents. All other node descriptor components (*par*, *att*, *tag*) simply require equality comparisons which we accelerated via hash indices.

Section 6 reports on the results of performance experiments for both the R-tree and the B-tree variants.

5. SHRINK-WRAPPING THE `//`-AXIS

It should be obvious that the *area* covered by the query window corresponding to an XPath axis has an impact on the performance of step evaluation along this axis. Especially in the case of B-trees, where two independent scans over the *pre* and *post* indices yield potential result nodes but in general also yield false hits (see the previous section), query window size plays a major role.

As we have mentioned in Section 3.1, we already tried to be restrictive in defining the extent of the query windows, but specific properties of the preorder and postorder ranks in a tree allows us to further shrink the windows substantially. Specifically, we will discuss how to reduce the query window corresponding to the `descendant` (and `descendant-or-self` or `//`) axis. As the `child` and `attribute` axes select subsets in the `descendant` document region, these will also benefit from this optimization.

The following observation justifies the optimization: for

[4]This behavior is, of course, not part of the R-tree specification. Nevertheless, all R-tree implementations we were using observed this order.

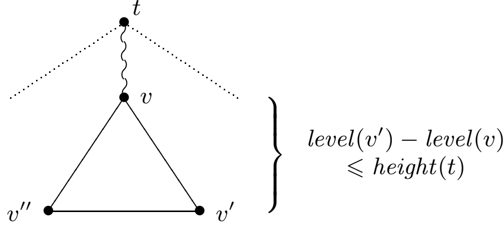


Figure 6: Identifying the nodes with minimum $post(v'')$ and maximum $pre(v')$ ranks if a //-step is taken from v.

any node v in a tree t, we have that[5]

$$pre(v) - post(v) + size(v) = level(v) \quad (1)$$

(where $size(v)$ denotes the size of subtree rooted at v). In Figure 1, for example, we know that $pre(b) = 1$, $post(b) = 3$, and $size(b) = 3$, so that $1 - 3 + 3 = 1$ equals $level(b)$.

Consequently, for a leaf v' of the tree, we have $size(v') = 0$ by definition, so that the above becomes

$$pre(v') - post(v') = level(v') \leqslant height(t) \ . \quad (2)$$

For a specific leaf below v, namely the rightmost leaf v' (Figure 6), we additionally know that

$$post(v) = post(v') + \underbrace{\left(level(v') - level(v)\right)}_{\leqslant height(t)} \quad (3)$$

since a postorder traversal of tree t consecutively ranks the $level(v') - level(v)$ ancestors of v' until it finally visits node v.

Now suppose that we are about to take a step along the `descendant` axis from context node v. In the subtree below v, the rightmost leaf node v' clearly is the node with the maximum preorder rank (any other node in the subtree has been visited prior to v' and consequently has a preorder rank $< pre(v')$).

Equations (2) and (3) provide us with an upper bound for $pre(v')$ and thus for all nodes in the subtree, namely

$$pre(v') \leqslant post(v) + height(t) \ .$$

A dual argument applies to the leftmost leaf node v'' below v. Its postorder rank $post(v'')$ is minimal in the subtree. Again, (2) and (3) characterize a lower bound for $post(v'')$ and therefore for all other nodes in subtree:

$$post(v'') \geq pre(v) - height(t) \ .$$

Note that both bounds are exclusively expressed in terms of the context node's descriptor and the overall height of the XML document. This enables us, given only the context node v, to shrink the associated `descendant` window as shown below:

$$\begin{aligned} &window(\texttt{descendant}, v) = \\ &\quad \langle \ \left(pre(v), post(v) + height(t)\right], \\ &\quad\ \ \left[pre(v) - height(t), post(v)\right), \\ &\quad\ \ *, false, * \ \rangle \ . \end{aligned}$$

[5] Note, how (1) relates our work to the *order-size* scheme of [14].

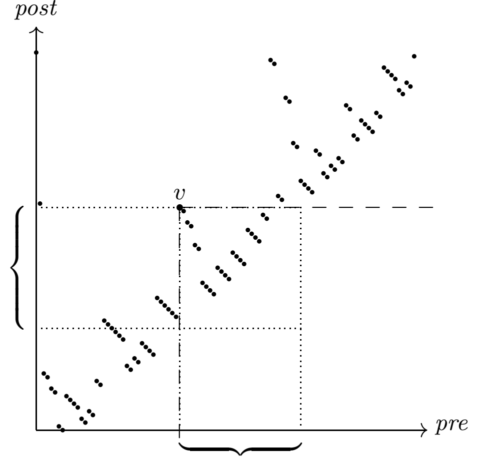


Figure 7: Original (– –) and shrunk (⋯⋯) *pre* and *post* scan ranges for a //-step to be taken from v.

The definitions for axes $window($`descendant-or-self`$, v)$, $window($`child`$, v)$, and $window($`attribute`$, v)$ can be improved in the same manner. Figure 7 illustrates the original as well as the improved query window and scan ranges.

Especially for B-tree based XPath acceleration we have found this optimization to make a substantial difference. In anticipation of the performance figures and experimental setup in the next section, we ran a series of queries stressing the //-axis against an B-tree based XPath accelerator built on top of IBM DB2 loaded with an XML instance of 1.1 MB size (21051 document nodes). Table 3 shows the timing results as well as the size of the result forests (see Figure 9 in Section 6 for a sketch of the document type we were querying against). Shrinking the `descendant` window resulted in a speed-up of up to three orders of magnitude.

5.1 Attribute (Leaf) Access

For a certain class of XPath steps we can tell at query compile time that all nodes in the result forest will be leaves. This is specifically so for any step along the `attribute` axis as well as for explicit leaf queries, like p/n`[not(*)]` or $p//n$`[not(*)]` (with n denoting a name test including `*`).

If the indexing scheme underlying the XPath accelerator can evaluate non-rectangular queries (*e.g.*, as in the B-tree case), we can shrink the windows further with the help of (2).

Due to (2), in the *pre*/*post* plane for document tree t, we know that for any leaf node l, $pre(l)$ and $post(l)$ differ by at most $height(t)$. This means that leaf nodes are to be found in a strip of width $height(t)$ above the diagonal through the *pre*/*post* plane given by

$$post = pre - height(t) \ .$$

Figure 8 illustrates the resulting query window if this observation is combined with a shrunk `descendant` axis window. This is a query window as narrow as we can hope for if the context node's descriptor is all we have in our hands.

6. PERFORMANCE CHARACTERISTICS

Query	t^{shrunk} [s]	t [s]	# Nodes
`//open_auction//description`	0.2	53	120
`//open_auction//description//listitem`	0.32	55.5	126
`//open_auction//description//listitem//keyword`	0.34	124	90

Table 3: XPath traversals with and without shrunk query window sizes.

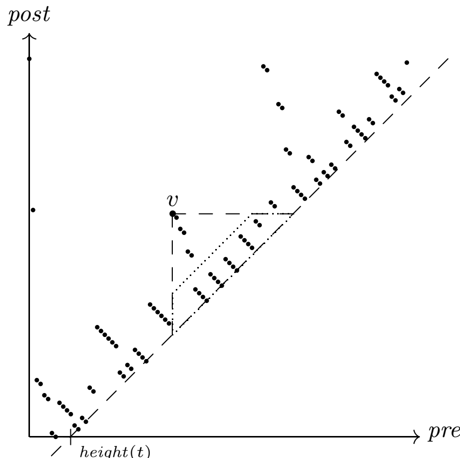


Figure 8: XML document leaves below v are to be found in this strip (⋯⋯) of width $height(t)$.

To assess the performance of the XPath accelerator, we implemented a SAX-based document loader (Section 4). We fed its output, the node descriptor tables, into two different setups:

1. A relational database management system, namely IBM DB2 V7.1. We created the two *pre* and *post* B-tree indices as described in Section 4.3 and additionally maintained hash indices on the *par* and *tag* columns. During the experiments, the only client connected to the database server was the XPath accelerator.
2. An implementation of the XPath evaluation scheme that lives on top of a GiST-based backend (a C++ library providing a number of generalized search tree variants) [11]. This setup used a 5-dimensional point R-tree to store the nodes.

Axis query windows were implemented as they are shown in Table 2 with the optimizations of Section 5 applied.

Both setups were hosted on an Intel i586 PC (clocked at ≈ 1 GHz), using a version 2.4 Linux kernel, and running off a standard file system (`ext2`) on an EIDE hard disk. The host was equipped with 256 MB RAM (no swapping occurred) and the system load average was near zero (no other processes were active besides a small number of sleeping system daemons).

(For the sake of comparability with related work we also temporarily moved the R-tree based setup to another host. Details are given below.)

To ensure the test runs to be reproducible, we used an easily accessible source of XML documents, namely the XML generator XMLgen, developed for the *Xmark benchmark project* [16]. For a fixed DTD (modeling an Internet auction site, see the element hierarchy depicted in Figure 9), this generator produces instances of controllable size. Table 4 lists the document sizes we were using for our experiments. All documents were of height 11.

Document size [MB]	# Nodes	XMLgen factor
0.11	2086	0.001
0.55	10492	0.005
1.1	21051	0.01
11.0	206130	0.1
55.0	1024073	0.5
111.0	2048193	1.0

Table 4: XML document sizes and number of (element and attribute) nodes in document trees. Entries in the last column were given as a size factor to XMLgen (switch `-f`) to control document sizes.

6.1 Relational XPath Evaluation

The first experiments were exclusively run on top of the relational platform. A wide variety of proposals to map XML instances onto relational tables exist. Out of these, we picked the *edge mapping* to compare its performance with the XPath accelerator.

The edge mapping, just like our XPath acceleration scheme, stores the XML document structure in a single relational table and thus does not flood the database with table definitions (unlike mapping schemes that introduce a separate table for each element tag name encountered in an instance). This mapping scheme has been shown to effectively support the evaluation of regular path expressions, *esp.* if these include selective name tests. We have measured the running times of such queries and report on the results below.

In a nutshell, the edge mapping maintains a table *edge* $\underline{node \mid par \mid att \mid ord \mid tag}$ in which for each *node* (id) its parent *par* node is listed (each edge in the document tree is represented by a tuple). The *ord* attribute keeps track of a node's order among the nodes below a common parent. This is sufficient to restore the overall document order of nodes although this is an expensive operation. Attributes *att* and *tag* indicate the type of the node and its element tag (or attribute name), respectively. Element content is maintained in a separate $\underline{node \mid cdata}$ table so that the edge mapping table represents the document structure without content overhead. As recommended in the literature, we created indices on the *node* and *par* attributes to speed up closure computation as well as an index on *tag* to effectively support name tests.

The measurements shown in Figure 10 report the timing results for the XPath query `//open_auction//description` against Xmark document instances of increasing size. All queries were run multiple times (the average timings re-

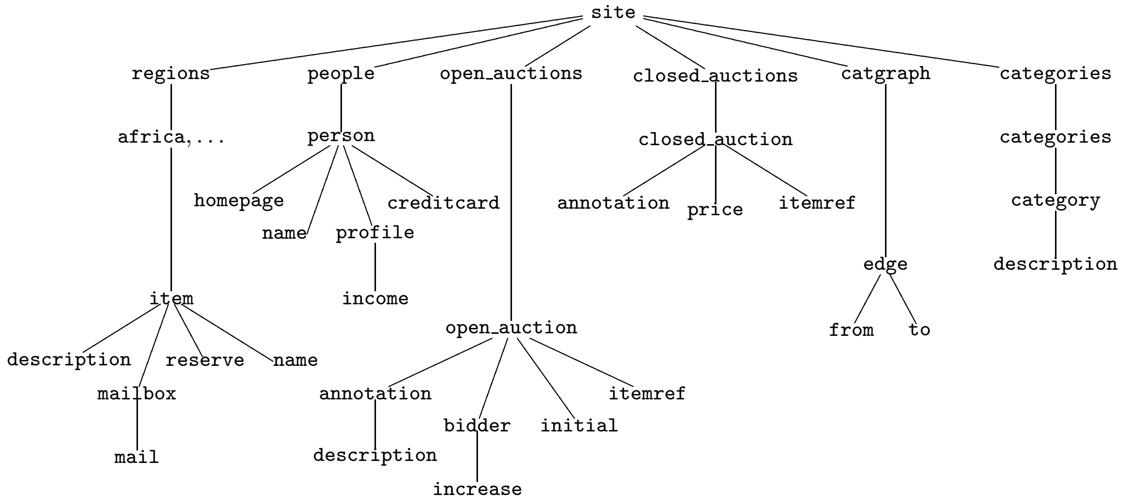


Figure 9: Element hierarchy (top-level) of Xmark XML benchmark document instances.

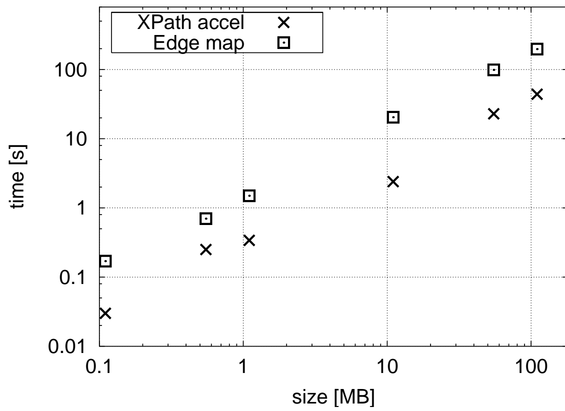


Figure 10: Performance of the relational implementation: XPath accelerator *vs.* edge mapping, log-log scale. (Query: `//open_auction//description`**)**

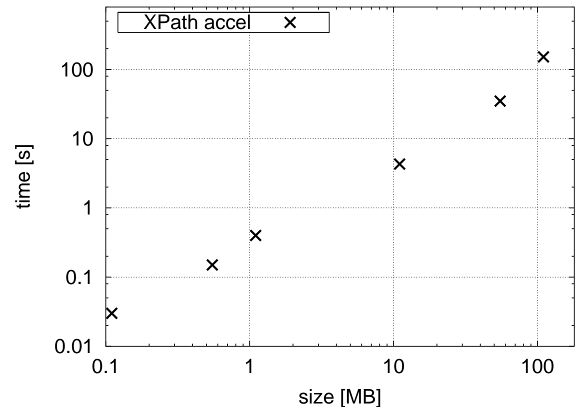


Figure 11: Linear scale-up for the XPath accelerator with respect to document size, log-log scale. Result sizes grow from 257 up to 271992 nodes. (Query: `//name/following-sibling::*`**)**

ported here were measured when the database buffer cache was hot). Query result size grew linearly with the document size from 12 to up to 12000 nodes.

Note that for the edge mapping, steps along the XPath `//`-axis have to be evaluated by a recursive computation of the closure of the edge table with respect to attributes *node* and *par*. We applied name tests as early as possible to reduce closure size.

The XPath accelerator turned out to be at least 5 times faster than the edge mapping alternative. In a multi-step XPath query, the optimized axis query windows appear to restrict the node set to be examined for subsequent steps quite effectively.

The time taken for an XPath accelerated query grows linearly with the instance size. We measured this linear scale-up for queries along other axes as well (with the exception of the `parent` and `ancestor` axes, see below). The performance figures for `following-sibling` steps illustrated in Figure 11 are exemplary.

6.2 R-tree Supported XPath Acceleration

Since our index proposal has been originally designed to be supported by a multi-dimensional access method, we expected XPath performance to be significantly better if an R-tree based backend would be available. Although the R-tree variant ran off a standard file system without any further buffering support, we indeed found it to clearly outperform the B-tree alternative.

Figure 12 repeats the timings we have found for the relational XPath acceleration of the path `//open_auction//description` and compares these figures with the measurements for the R-tree based implementation. Query response times once more improved by a factor of 10, approximately. Just like for the relational implementation, the R-tree based

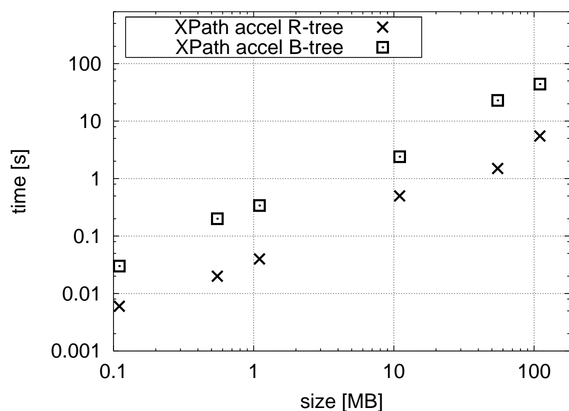


Figure 12: Performance comparison of the B-tree based implementation (inside an RDBMS) *vs.* the R-tree based variant (GiST), log-log scale. (Query: `//open_auction//description`)

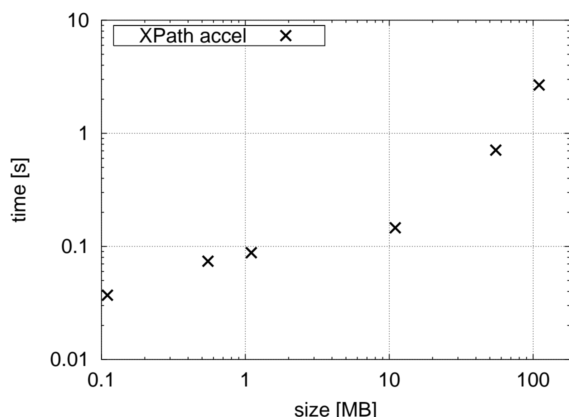


Figure 13: Average time needed to evaluate `following-sibling::*` steps for randomly selected context nodes, log-log scale.

timing results grew linearly with the document size.

This observation was reinforced when we measured the average traversal times along other axes (Figure 13 contains the exemplary plot for the `following-sibling` axis). We randomly selected context nodes in the documents to account for the typical situation in which an XQuery implementation iterates the evaluation of a path traversal, *e.g.*,

```
for $v in e
    return $v//bidder/../initial
```

(in general, the elements in forest e are arbitrarily computed nodes, and thus scattered over the whole document tree).

However, for two axes, namely **`parent`** and **`ancestor`**, we observed that the query response time was indifferent to the document size as well as its height: stepping along these axes up to the document root required almost no time, regardless of the level of the context node, *i.e.*, regardless of the length of the path that was traversed. Figures 14 and 15 display the corresponding timings (both queries completed in about 6 ms).

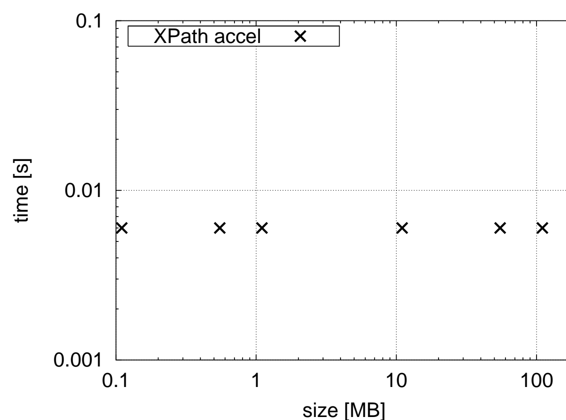


Figure 14: The performance of traversals along the `ancestor` axis is indifferent to the XML document size, log-log scale. (Query: *leaf*`/ancestor::*`)

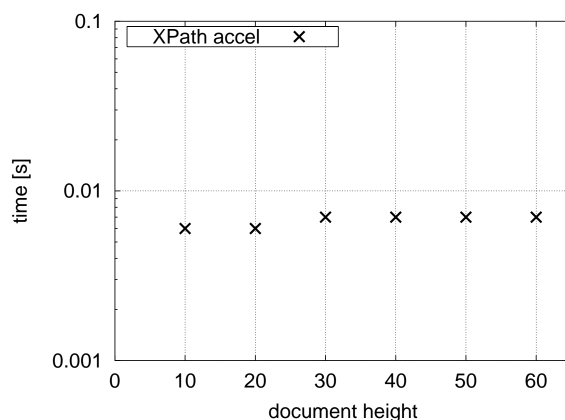


Figure 15: Document height does not seriously affect the performance of `ancestor`-axis traversals: stepping from a leaf l with $level(l) = height(t)$ up to the root. (Query: *leaf*`/ancestor::*`)

Figure 16 depicts the query window for a step from leaf l in the direction of the **`ancestor`** axis to provide an intuition for the situation. Completely unaffected by document size and regardless of the choice of l, the window will never contain more than $height(t)$ nodes, *i.e.*, typically one R-tree leaf access will suffice to answer the query (*cf.* Figure 16).

We finally moved the R-tree setup to a different system, namely a Sun Ultra Sparc II, running Solaris 2.6, equipped with 256 MB RAM. This almost exactly reproduced the experimental setup of the work recently reported in [14] by Li and Moon. There, B-tree indices are created to support regular path queries (*i.e.*, queries along the **`/`** and **`//`** axes) against XML documents. Interestingly, this work (1) used variants of the *pre*/*post* ranking to represent document structure, and (2) was also implemented on top of GiST. All in all, this provided for a rather unique opportunity to directly compare the XPath accelerator with the work of [14].

In [14], three separate B-tree indices, the *element (tag) index*, the *structure index*, and the *attribute* index are created to maintain an XML instance. The structure index

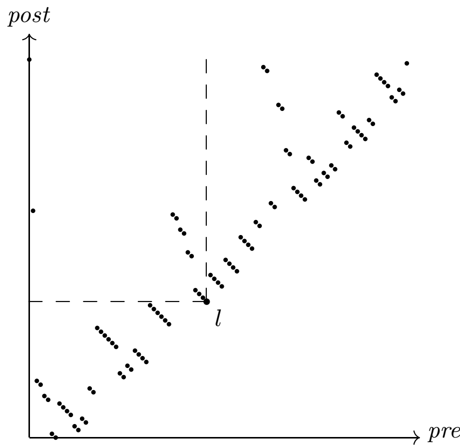


Figure 16: Nodes inside the ancestor query window (shown for a leaf node l.)

represents the *parent–child* relationship only, so that for the `//`-axis a separate procedure is provided that computes the closure of this relationship on demand. Two additional join algorithms, the *element–element* ($\mathcal{EE}$) and *element–attribute* ($\mathcal{EA}$) joins, implement steps along the `child` and `attribute` axes, respectively.

We performed the experiments reported in [14] which were ran against (1) documents containing XML markup of Shakespeare's plays [3], a rather shallow instance (height 5) of 7.3 MB (179619 nodes), and (2) an 8 MB XML document conforming to the NITF DTD (330860 nodes).

Table 5 displays the timing results for these experiments. The XPath accelerator comes close for the `//`-axis traversal but improves access times for the `attribute` axis. For the latter query, it obviously paid off that the XPath accelerator uniformly maintains element and attribute nodes in a single index (with attributes stored next to their containing elements, see Section 4). Li and Moon, however, pay a rather high price to restore the correct document order when their $\mathcal{EA}$-join assembles elements and attributes from two separate indices.

We have found these results quite motivating, especially since the XPath accelerator provides support for all XPath axes and thus goes beyond regular path expressions. Additionally, the accelerator relies on a XPath evaluation scheme simple enough to be implemented relationally (which has not been investigated in [14]).

7. MORE RELATED WORK

The concept of *regular path expressions* dominates this field of research by far [6, 14, 17, 9]: we have not yet learned about other real XPath-aware index structures until today. In some sense this comes as a surprise since the XPath 1.0 specification has been around since Winter 1999 and a number of other XML-related languages (*e.g.*, XSLT, XPointer, but most notably XQuery) embed XPath expressions in their syntax. Efficient XPath support will continue to be an important core building block in XML query processors.

Only recently, [6] presented an index over the *prefix-encoding* of the paths in an XML document tree (in a prefix-encoding, each leaf l of the document tree is prefixed by the sequence of element tags that one encounters during a path traversal from the document root to l). Since tag sequences obviously share common prefixes in such a scheme, a variant of the Patricia-tree is used to support lookups. Clearly, the index structure is tailored to respond to path queries that originate in the document root. Paths that do not have the root as the context node need multiple index lookups or require a post-processing phase (as does a restore of the document order in the result forest). In [6], so-called *refined paths* are proposed to remedy this drawback. Refined paths, however, have to be preselected before index loading time. Note that the prefix-encoding exclusively represents the `child` and `descendant` axes in a document—it remains unclear to us if support for other XPath axes blends well this scheme.

The T-index structure, proposed by Milo and Suciu in [17], maintains (approximate) equivalence classes of document nodes which are indistinguishable with respect to a given path template. In general, a T-index does not represent the whole document tree but only those document parts relevant to a specific path template. The more permissive and the larger the path template, the larger the resulting index size. This allows to trade space for generality, however, a specific T-index supports only those path traversals matching its path template (as reported in [17], an effective applicability test for a T-index is known for a restricted class of queries only).

There is other related work that is not directly targeted at the construction of index structures for XML. In [18], the authors discuss relational support for *containment queries* of which our XPath axes window queries are instances. Especially the *multi-predicate merge join* (MPMGJN) presented in [18] would provide an almost perfect infrastructure for the XPath accelerator. MPMGJN join supports multiple equality and *inequality* tests (*cf.* the $window(\alpha, v)$ query windows) and we thus expect it to perform exceptionally well for the *accel* table self-joins needed during XPath axes evaluation. The authors report an order of magnitude speed-up in comparison to standard join algorithms.

Another relational storage structure that seems to be well suited to support the XPath accelerator is the relational interval tree (*RI-tree*) [13]. Tailored to efficiently respond to interval queries of the form $[a, b]$, the RI-tree could be a promising candidate to index the *pre*/*post* plane. This option seems to be interesting especially if the database host lacks R-tree support: B-trees suffice to query the RI-tree efficiently.

8. CONCLUSION AND OUTLOOK

This work has been primarily motivated by the need for an XPath index structure that would be capable

1. to run on top of a relational backend to leverage its stability, scalability, and performance,
2. to support the whole family of XPath axes in an adequate manner, as well as
3. to originate XPath traversals in arbitrary context nodes.

The latter requirement, specifically, did arise in the context of an ongoing project to construct an XQuery runtime.

In the short term, we will assess how much the XPath accelerator could gain from advanced query processing techniques like the RI-tree, in the relational case, or the MPMGJN join which we plan to add to our GiST-based implementation.

XML instance	Query	$t^{accel}\,[s]$	$t^{\mathcal{EE}/\mathcal{EA}}\,[s]$	# Nodes
Shakespeare	`//ACT//SPEECH`	1.15	≈ 0.7	30951
NITF	`//block/attribute::dir`	5.41	≈ 7.0	3003

Table 5: A comparison of query response times for the XPath accelerator and the $\mathcal{EE}/\mathcal{EA}$-join based work of [14].

More on the theoretical side—geared towards the development of an optimizing XQuery runtime—we believe that the XPath accelerator provides the necessary hooks to incorporate an effective *cost estimation* for XPath queries: note, for example, that from Equation (1) and the obvious inequality $level(v) \leqslant height(t)$ for any node v and document tree t we can estimate the size of the subtree below v in rather tight bounds (since, for real XML documents, $height(t)$ is small). Observations of this kind, together with cost estimation procedures developed for packed R-trees [12], could lead to a rather pragmatic cost model for XPath queries. It will be interesting to compare this approach to more intricate cost models for XML queries as presented in [5].

9. REFERENCES

[1] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. Technical Report W3C Working Draft, Version 2.0, World Wide Web Consortium, December 2001. `http://www.w3.org/TR/xpath20/`.

[2] Christian Böhm, Stefan Berchtold, Hans-Peter Kriegel, and Urs Michel. Multidimensional Index Structures in Relational Databases. *Journal of Intelligent Information Systems (JIIS)*, 15(1):51–70, 2000.

[3] John Bosak. XML markup of Shakespeare's plays, January 1998. `http://www.ibiblio.org/pub/sun-info/standards/xml/eg/`.

[4] Don Chamberlin, James Clark, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. XQuery 1.0: An XML Query Language. Technical Report W3C Working Draft, World Wide Web Consortium, December 2001. `http://www.w3.org/TR/xquery`.

[5] Zhiyuan Chen, H.V. Jagadish, Flip Korn, Nick Koudas, S. Muthukrishnan, Raymond Ng, and Divesh Srivastava. Counting Twig Matches in a Tree. In *Proc. of the 17th Int'l Conference on Data Engineering (ICDE)*, pages 595–604, Heidelberg, Geramny, April 2001. IEEE Computer Society.

[6] Brain F. Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A Fast Index for Semistructured Data. In *Proc. of the 27th Int'l Conference on Very Large Data Bases (VLDB)*, pages 341–360, Rome, Italy, September 2001.

[7] Paul F. Dietz and Daniel D. Sleator. Two Algorithms for Maintaining Order in a List. In *Conference Record of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 365–372, New York City, May 1987. ACM Press.

[8] Daniela Florescu and Donald Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Technical Report 3680, INRIA, Rocquencourt, France, May 1999.

[9] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. of the 23rd Int'l Conference on Very Large Databases (VLDB)*, pages 436–445, Athens, Greece, August 1997.

[10] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD 1984, Proc. of Annual Meeting*, pages 47–57, Boston, Massachusetts, June 1984. ACM Press.

[11] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. In *Proc. of the 21st Int'l Conference on Very Large Databases (VLDB)*, pages 562–573, Zurich, Switzerland, September 1995.

[12] Ibrahim Kamel and Christos Faloutsos. On Packing R-Trees. In *Proc. of the 2nd Int'l Conference on Information and Knowledge Management (CIKM)*, pages 490–499, Washington DC, USA, November 1993.

[13] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. Managing Intervals Efficiently in Object-Relational Databases. In *Proc. of the 26th Int'l Conference on Very Large Databases (VLDB)*, pages 407–418, Cairo, Egypt, September 2000.

[14] Quanzhong Li and Bongki Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. of the 27th Int'l Conference on Very Large Data Bases (VLDB)*, pages 361–370, Rome, Italy, September 2001.

[15] SAX (Simple API for XML). `http://sax.sourceforge.net/`.

[16] Albrecht R. Schmidt, Florian Waas, Martin L. Kersten, Daniela Florescu, Ioana Manolescu, Michael J. Carey, and Ralph Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001.

[17] Dan Suciu and Tova Milo. Index Structures for Path Expressions. In *Proc. of the 7th Int'l Conference on Database Theory (ICDT)*, number 1540 in Lecture Notes in Computer Science (LNCS), pages 277–295, Jerusalem, Israel, January 1999. Springer Verlag.

[18] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 425–436, Santa Barbara, California, May 2001. ACM Press.