First publ. in: ACM Transactions on Database Systems (TODS), number 1, ACM, March 2004

Accelerating XPath Evaluation in Any RDBMS

TORSTEN GRUST
University of Konstanz
and
MAURICE VAN KEULEN
University of Twente
and
JENS TEUBNER
University of Konstanz

This article is a proposal for a database index structure, the *XPath accelerator*, that has been specifically designed to support the evaluation of XPath path expressions. 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 the `child` and `descendant` axes only. The index has been designed with a close eye on the XPath semantics as well as the desire to engineer its internals so that it can be supported well by *existing* relational database query processing technology: the index (a) permits set-oriented (or, rather, sequence-oriented) path evaluation, and (b) can be implemented and queried using well-established relational index structures, notably B-trees and R-trees.

We discuss the implementation of the XPath accelerator on top of different database backends and show that the index performs well on all levels of the memory hierarchy, including disk-based and main-memory based database systems.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*relational databases*; *query processing*; E.1.0 [**Data Structures**]: Trees

General Terms: Performance, Theory

Additional Key Words and Phrases: Main-memory databases, XML, XML indexing, XPath

Authors' addresses: Torsten Grust and Jens Teubner, University of Konstanz, Department of Computer and Information Science, P.O. Box D188, 78457 Konstanz, Germany, `{grust,teubner}@inf.uni-konstanz.de`. Maurice van Keulen, University of Twente, Faculty of Computer Science, P.O. Box 217, 7500 AE Enschede, The Netherlands, `keulen@cs.utwente.nl`.

This is a preliminary release of an article accepted by the ACM Transactions on Database Systems. The definitive version is currently in production at ACM and, when released, will supersede this version.

1. INTRODUCTION

It is hard to find an answer to the question of why XML has been so successful in being adopted as a universal data exchange format. One possible answer might be that the data type underlying the XML, the *tree*, is expressive enough to capture the structure of diverse data sources, yet simple enough to permit efficient as well as elegant (especially recursive) algorithms to process 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* schema-less manner. The 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 mark-up language WeatherML, CellML (computer-based biological models), or XMLPay, whose instances describe Internet-based payments.

As more sources switch over and express their contents using XML dialects, the sheer volume of data calls for XML-aware data management solutions built 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 [Berglund et al. 2002] and XQuery [Boag et al. 2002; Fernandez et al. 2002].

In line with the tree-centric nature of XML, XPath provides operators to describe path traversals in tree-shaped documents. Starting from a *context node*, an XPath query traverses its input document using a number of *steps*. A step's *axis* indicates which tree nodes are reachable from the context node, the step's *node test* then filters the reachable nodes by tag name or node kind. These intermediary nodes are then, recursively, interpreted as context nodes for subsequent steps, and so forth. The XPath specification [Berglund et al. 2002] lists a family of 13 axes, among these the `child` 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 have been developed that summarize an XML document so that these problems can be dealt with [Cooper et al. 2001; Li and Moon 2001; Suciu and Milo 1999; Goldman and Widom 1997]. Almost exclusively, however, these techniques put their focus on support for step evaluation along the `child` and `descendant-or-self` axes. This is hardly adequate support for the XPath language. 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 index structure* in the sense of Kriegel et al. [2000]. Its implementation can benefit from well-established indexing technology, notably the B-tree but also the R-tree, which 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. The XPath accelerator maintains the *document order* among nodes and supports XPath path traversals which resume from *arbitrary context nodes* (*i.e.*, the document root node is not special). Loading as well as querying the index is simple, yet its performance beats measurements published in recent related work.

It is possible to squeeze even more out of the XPath accelerator idea, if it is carefully implemented and tuned for a specific database back-end. We describe such refinements tailored for the relational disk-based database system IBM DB2 as well as the main-memory database system Monet [Boncz 2002]. In the case of IBM DB2, we pursue a purely relational implementation using SQL as the implementation language. In the case of Monet, with its open and extensible database kernel, we take advantage of properties of XPath accelerator internals, *e.g.*, the exploitation of *document order*, which a traditional RDBMS cannot. Instead, we use Monet's versatile programmable algebraic kernel interface that enables a number of most useful optimizations. We additionally describe logical, *i.e.*, back-end independent optimization hooks, which we believe to be relevant in other implementation scenarios as well.

The article 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. Section 4 exploits the fact that we are operating with *tree-shaped* data and derives a series of improvements to the original XPath accelerator idea. We then explore three possible XPath accelerator back-ends—IBM DB2, Monet, and an R-tree based file interface—and discuss a number of implementation details as well as back-end specific issues in Section 5. Section 6 assesses and compares the performance of the resulting XPath engines. Section 7 reviews related work before we conclude in Section 8. Two *electronic appendices* additionally shed light on how the XPath accelerator supports XML document loading and serialization.

We assume that the reader is familiar with the XPath 2.0 specification [Berglund et al. 2002]. In particular, we assume that an XPath location step yields a *node sequence* (in document order) rather than a *nodeset*.

2. XPATH AXES AND XML DOCUMENT REGIONS

XML documents represent tree-shaped data, and the XPath language is built around a core feature, the *path expression*, that has been designed to traverse such trees. Each XML tree node assumes one of several node kinds (*e.g.*, *element*, *attribute*, *text*, *comment*, *processing instruction*). Leaving these node kinds aside for a minute, the gist of a well-formed XML document always describes a tree whose shape is encoded via the proper nesting of *start* and *end tags* (details of the XPath data model can be found in [Berglund et al. 2002]).

Figure 1 depicts two XML fragments and the tree shape shared by both. In this tree, the inner nodes a, b, c, f, g, h represent XML element nodes, the leaf nodes

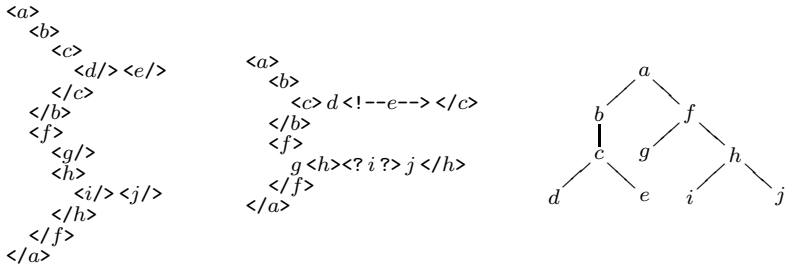
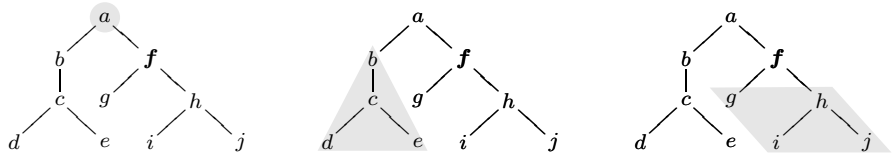


Fig. 1. Two XML fragments and their common tree shape.



(a) (b) (c)

Fig. 2. XPath semantics: Shaded nodes are reachable from context node f via a step along the (a) `ancestor`, (b) `preceding`, and (c) `descendant` axes.

d, e, g, i, and j represent either (empty) elements, attributes, text, comments, or processing instructions.

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.*, $height(a) = 3$; $level(v) = n$ if the path from the root to v has length n, *e.g.*, $level(a) = 0$ and $level(e) = 3$.

XPath path expressions specify a tree traversal via two parameters:

(1) a *sequence of context nodes* which provides the starting point of the traversal,

(2) a list of *steps*, syntactically separated by `/`, evaluated from left to right. For each context node in turn, a step's *axis* establishes a subset of document nodes (a *document region*). The subsets are unioned together and then sorted in document order to form the sequence of context nodes for the subsequent step, if any.

Note that these sequence-oriented semantics bear some resemblance with the relational algebra in which operators consume and produce sets of tuples rather than single tuples. Section 4.4 discusses optimizations we can derive from sequence-orientation.

To illustrate the XPath axes and the document regions they establish, Figure 2 depicts the resulting nodes for three steps along different axes taken from context node f (observe that the `preceding` axis does not include the ancestors of the context node). Table I lists all XPath axes and verbally sketches their semantics. We will provide a precise specification in Section 3.1.

Table I. Semantics of axes α supported by XPath (step v/α`::node()`).

Axis α	Result Nodes
`child`	child nodes of v
`descendant`	closure of `child`
`descendant-or-self`	like `descendant`, plus v
`parent`	parent node of v
`ancestor`	closure of `parent`
`ancestor-or-self`	like `ancestor`, plus v
`following`	nodes following v in the tree (excluding descendants)
`preceding`	nodes preceding v in the tree (excluding ancestors)
`following-sibling`	like `following`, same parent as v
`preceding-sibling`	like `preceding`, same parent as v
`attribute`	attribute nodes owned by v
`self`	v
`namespace`	namespace nodes owned by v

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 node sets document *regions*). Regardless of choice of v, the node set[1]

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

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

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

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-sibling`, and `preceding-sibling`) determine specific supersets or subsets of these node sets which are easy to characterize.

Note that an index designed along these lines will contain each document node exactly once, due to the partitioning property of the four major axes. We may thus use such an index as the *only* representation of the XML document inside the database.

To complete our review of the XPath core, let us note that a step along axis α is accompanied by a *node test* τ (the syntactic form is α`::`τ), which restricts the selected node set to either

[1] In line with the XPath specification, we identify a node v and the singleton node sequence (v). In the XPath expression $v/\alpha = (v)/\alpha$, a step along axis α is taken from the single context node v.

(1) those element or attributes nodes having name τ (*name test*), or
(2) those nodes having kind τ (*kind test*, with $\tau \in \{$`node()`, `text()`, `comment()`, `processing-instruction()`$\}$).

Without an explicit kind test, an axis exclusively delivers nodes of its *principal node kind* which is *element* for all but the `attribute` axis which yields attribute nodes only. A `node()` kind test accepts nodes of arbitrary kind, a `*` name test accepts nodes with arbitrary name.

In the following, we primarily focus on this XPath core, *i.e.*, on axis steps and accompanying node tests as these are what the XPath accelerator is designed to accelerate. In appropriate places, however, we will make short remarks on other features of XPath and how their evaluation can be combined with axis step acceleration.

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 relational 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 or document fragment as the context node, that are before the context node in document order, excluding any ancestors and excluding attributes nodes and namespace nodes.*" [Berglund et al. 2002]

Informally, the *document order* of the nodes of an XML instance corresponds 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 instances 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 $post(d) = 0$, $post(e) = 1$, ..., $post(a) = 9$ (see Figure 3 for the complete pre- and postorder rank assignment).

As others have noted [Dietz and Sleator 1987; Li and Moon 2001; Zhang et al. 2001], one can use $pre(v)$ and $post(v)$ to efficiently characterize the descendants v' of node v. We have that

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

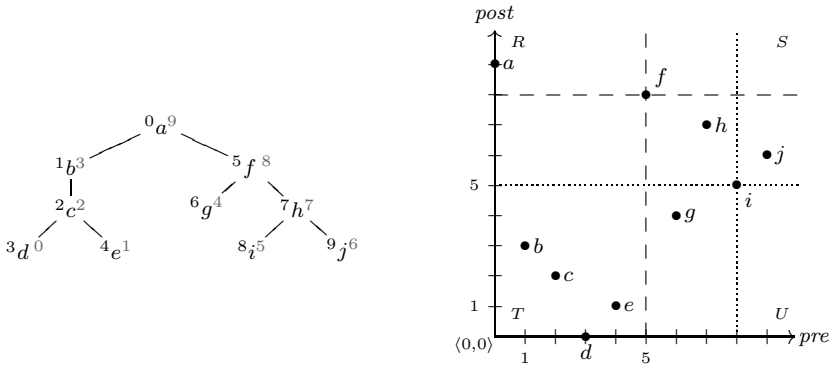


Fig. 3. Preorder/postorder rank assignment and node distribution in the resulting *pre*/*post* plane. Also indicated are the XML document regions as seen from context nodes f (− −) and i (·····).

Intuitively, this may be read as: During a sequential read of the XML document, we have seen the start tag <*v*> *before* <*v′*> and the end tag </*v*> *after* </*v′*>. In other words, the element corresponding to v' is part of the contents of 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 in an equally simple manner.

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

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

(1) the lower-right partition U contains all `descendant`s of f,

(2) in the upper-left partition R, we find the `ancestor`s of f, *i.e.*, node a only,

(3) the lower-left partition T hosts the nodes `preceding` f, and finally

(4) the upper-right partition S represents the nodes `following` f (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. The index has no bias towards a specific context node set, *e.g.*, the document root element, or a specific set of queries. This turns out to be an important feature when it comes to the implementation of XQuery. XQuery is a fully compositional query language: Arbitrary expressions (*e.g.*, variables bound in iteration constructs like `for` and `every`, calls to user-defined functions, element nodes constructed at runtime) yield arbitrary context node sequences from which an XPath path traversal may resume. This is different from the evaluation of *ad*

hoc XPath queries, say, where the context node for the first axis step preferably is the document root.

3.1 Axes and Query Windows

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

To support the remaining XPath axes and node 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, because siblings share the same parent. $par(v)$ readily characterizes axes `child` and `parent`, too.

To support node tests, *i.e.*, name tests as well as kind tests, we additionally maintain

—$name(v)$, storing the element tag name or attribute name of node v if v is of the respective kind, otherwise $name(v) = \text{␣}$ (undefined), and

—$kind(v) \in \{node, elem, attr, text, comment, processing\text{-}instruction\}$.

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

$$desc(v) = \langle pre(v), post(v), par(v), kind(v), name(v) \rangle \ .$$

An XPath axis corresponds to a specific *query window* in the space of node descriptors. Table II 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 *elem* and *attr* entries under *kind* in Table II reflect the *principal node kinds* [Berglund et al. 2002] of the respective axes. If a name or kind test τ is applied to the step, the *name* or *kind* entry in $window(\alpha, v)$ is set to τ, respectively. We thus have, for example,

$$window(\texttt{preceding::text()}, v) = \langle [0, pre(v)), [0, post(v)), *, text, * \rangle \ .$$

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

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

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 say more about essential opportunities to shrink window sizes in Section 4.1.

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

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

The above encoding is presented as if there were only one document, whereas in general a system may store many. Observe, however, that multiple documents can be gathered into one *global document* by introducing a *global root node* that has the root nodes of the various documents as its children. By encoding the global document in the aforementioned way, all one-document mechanisms readily carry over to a multi-document setting. Query windows should stay within document boundaries, which can easily be achieved by respecting separately stored minimum and maximum preorder rank values.

4. ENHANCING TREE AND XPATH AWARENESS

In what follows, we will explore four refinements of the original XPath accelerator idea. These optimizations aim to make better use of the fact that the *pre*/*post* plane encodes tree-shaped data rather than an arbitrary point set. In a sense, the optimizations enhance the *tree awareness* of the index:

(1) The first exploits a dependency between the $pre(v)$ and $post(v)$ ranks for any node v in the document tree to substantially shrink the size of the query windows we need to consider.

(2) We may also choose to "stretch" the *pre*/*post* plane such that the node subset associated with the `descendant` axis is characterized by a *single pre* or *post* range (*i.e.*, not both). This especially boosts B-tree based XPath accelerators.

(3) The XPath language specification exhibits symmetries between axes. We can exploit these to reorder XPath expressions such that the portion of the *pre*/*post* plane we need to consider is reduced for the reordered expression.

(4) Finally, remember that an XPath axis step is computed for a *sequence* of context nodes. An analysis of the context nodes and their placement in the *pre*/*post* plane can be used to avoid pointless and duplicate query window evaluation beforehand.

4.1 Staking Out Subtrees

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. There are additional dependencies between $pre(v)$, $post(v)$, as well as the tree

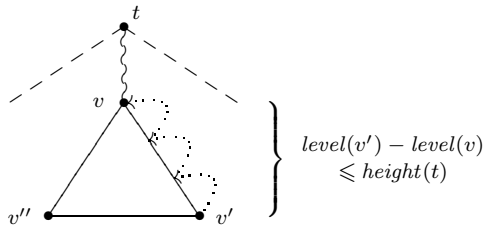


Fig. 4. Identifying the nodes with minimum *post* (v'') and maximum *pre* (v') ranks if a `descendant` step is taken from v.

height $height(t)$, which we can use to efficiently characterize the subtree below node v and thus the nodes returned by v`/descendant::`τ.

The following observation justifies the optimization: for any node v in a tree t, we have that

$$pre(v) - post(v) + size(v) = level(v) \tag{1}$$

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

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

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

For a specific leaf below v, namely the rightmost leaf v' (Figure 4), we additionally know that

$$post(v) = post(v') + \underbrace{\big(level(v') - level(v)\big)}_{\leqslant height(t)} \tag{3}$$

since a postorder traversal of t consecutively ranks the $level(v') - level(v)$ ancestors of v' until it finally visits node v (*cf.* the traversal steps $\cdots\searrow$ in Figure 4).

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 thus has a preorder rank less than $pre(v')$.

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:

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

Note that both bounds are exclusively expressed in terms of the descriptor of the context node v and the overall height of the XML document. Given only the

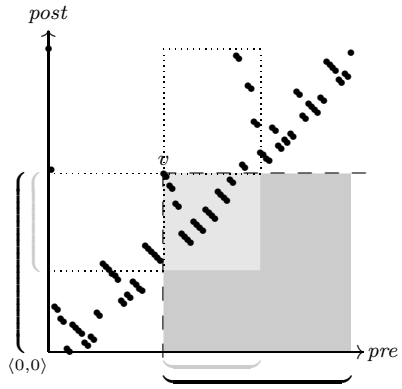


Fig. 5. Original (dark) and shrunk (light) *pre* and *post* scan ranges for a `descendant` step to be taken from v.

context node v, this enables us to shrink the associated `descendant` window as shown below:

$$\begin{aligned} &\mathit{window}(\texttt{descendant}, v) = \\ &\qquad \big\langle \big(\mathit{pre}(v), \mathit{post}(v) + \mathit{height}(t)\big], \\ &\qquad\ \big[\mathit{pre}(v) - \mathit{height}(t), \mathit{post}(v)\big), \\ &\qquad\ *, \mathit{elem}, * \ \big\rangle \ . \end{aligned} \tag{4}$$

As a result, the size of the `descendant` window of v is now solely dependent on the size of the actual subtree below v, regardless of the size of the overall document. Due to the approximation of *level* in Equation 3, this estimation of $\mathit{size}(v)$ may be off by maximally $\mathit{height}(t)$. This is insignificantly small, however, since in practice, XML document trees t are often rather shallow; a typical $\mathit{height}(t)$ is below one hundred, even for multi-million node documents.

While this optimization is tailored to improve steps along the `descendant` axis, the original definitions for $\mathit{window}(\texttt{descendant-or-self}, v)$, $\mathit{window}(\texttt{child}, v)$, and $\mathit{window}(\texttt{attribute}, v)$ can be altered in the same manner and will benefit as well. Figure 5 illustrates the original as well as the improved query window and scan ranges for a `descendant` step.

4.2 A Stretched *pre*/*post* Plane

All axis query windows in the two-dimensional *pre*/*post* plane depend on a range selection in the *pre* as well as the *post* dimension. If the nodes in the window are determined via two *independent* range queries in both the *pre* and *post* dimensions, large query windows generally lead to numerous false hits during either scan: In Figure 5, the two dotted regions enclose the false hit nodes encountered during the scans along the *pre* and *post* dimensions. These nodes have to be filtered out during a subsequent intersection.

A simple modification to the construction of the *pre*/*post* plane allows us to take a step along the pervasive `descendant` axis with a single range scan over the *pre* *or* the *post* dimension.

Note that the document regions with respect to a context node v, as displayed in Table II, are defined relative to $\mathit{pre}(v)$ and $\mathit{post}(v)$. The absolute *pre* and *post*

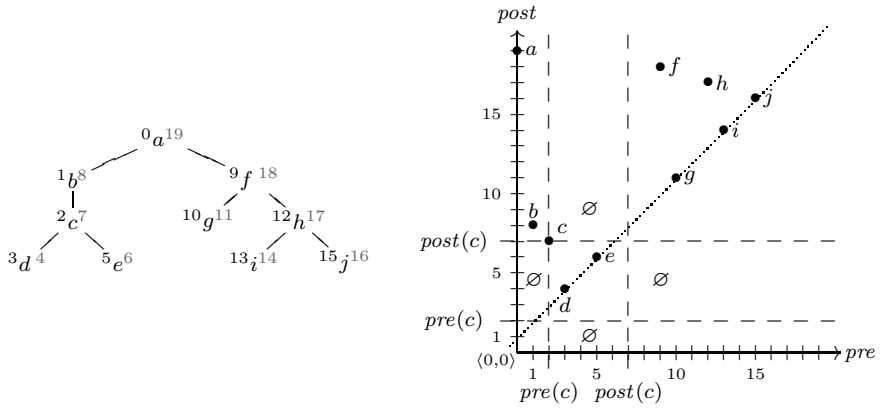


Fig. 6. Stretched preorder/postorder rank assignment and node distribution in the resulting *pre*/*post* plane. The dashed lines (− −) mark a *pre* and a *post* range, *any* of which characterizes the descendants d, e of context node c.

values, however, are insignificant. We can exploit this observation and modify the computation of $pre(v)$ and $post(v)$: *Couple* the preorder and postorder ranks such that whenever *pre* is incremented, *post* is as well and *vice versa*.

In the resulting preorder and postorder rank assignment (depicted in Figure 6) for all descendants v of node c, say, we thus have

$$pre(c) < pre(v) < post(c) \quad \text{as well as} \quad pre(c) < post(v) < post(c) \ . \tag{5}$$

No other nodes v fulfill the inequalities in (5) since we continue to monotonically increment *pre* and *post* once we are done traversing the subtree below c (see the empty *pre*/*post* plane regions marked ∅ in Figure 6). The evaluation of a `descendant` window query in the stretched *pre*/*post* plane consequently never encounters any false hits.

Additionally, we lose no other valuable properties of the *pre*/*post* plane:

(1) all axis query windows continue to work as before,
(2) the $<$ order on *pre* still reflects document order,
(3) both $pre(v)$ and $post(v)$ still uniquely identify document node v, and
(4) the estimation of the subtree size below node v is now completely accurate:

$$size(v) = {}^1\!/_2 \ (post(v) - pre(v) - 1) \ , \tag{6}$$

i.e., the maximal error of $height(t)$ is gone.

From the query evaluation perspective, Equation (5) gives us the freedom to choose one of the following query windows to evaluate a `descendant` step from v (note the $*$ entries in the *pre* and *post* positions, respectively):

$$\begin{aligned} & window(\texttt{descendant}, v) \quad = \quad \big\langle (pre(v), post(v)), *, *, elem, * \big\rangle \\ \textit{or} \\ & window(\texttt{descendant}, v) \quad = \quad \big\langle *, (pre(v), post(v)), *, elem, * \big\rangle \end{aligned}$$

As already mentioned, this simplification applies to the `descendant-or-self`, `child`, and `attribute` windows, too.

Note that, for any implementation using a fixed bit width representation for the coupled $pre(v), post(v)$ ranks, stretching the *pre*/*post* plane implies that the number of representable nodes is effectively divided by two if compared with the non-stretched case (*e.g.*, if $pre(v)$ and $post(v)$ are mapped into a 32 bits wide integer domain, the resulting stretched *pre*/*post* plane can host a maximum of 2^{31} nodes). Section 5.2.4 discusses further implications for implementations that operate with a stretched *pre*/*post* plane.

4.2.1 *Leaf Node Access.* For a certain class of XPath steps we can tell at query compile time that all nodes in the result set will be leaves. This is specifically so for steps along the `attribute` axis, any step with a kind test `text()`, `comment()`, or `processing-instruction()`, as well as XPath predicate queries of the general form e`[not(child::node())]`.

Due to the coupling of the preorder and postorder rank assignments in the stretched *pre*/*post* plane, for any leaf node l we know that

$$post(l) \;=\; pre(l) + 1 \;\;.$$

Cast into terms of the *pre*/*post* plane, document leaf nodes are to be found on the dotted diagonal (·····) in Figure 6. This knowledge is easily incorporated into query evaluation schemes (see Section 5.1.2).

It is interesting to note that the presence of this "leaves diagonal" enables the XPath accelerator to process certain types of path expressions in a *backwards* fashion. This blends elegantly with symmetry properties of XPath which have been extensively explored in [Olteanu et al. 2001]. Suppose we are to process the XPath query

`/descendant::`n`/child::text()` .

We could trade the original query for the *symmetric equivalent*

$\underbrace{\texttt{/descendant::text()}}$`[parent::`$n$`]`

in which the braced subquery selects a subset of nodes on the leaves diagonal. The remaining predicate then simply calls for a $window($`parent::`$n, l)$ evaluation for all matching nodes l found on the diagonal. We will use similar symmetry arguments in Section 4.3.

4.2.2 *Exploiting Schema Information.* The presence of a DTD (or XML Schema information) for a pre-/postorder ranked document tree may be used to generalize the last observation about the leaves diagonal. From a DTD, we can derive *minimum* and *maximum subtree sizes* for any element with tag t, *e.g.*, by counting path lengths in its corresponding DTD/element graph [Shanmugasundaram et al. 1999]. Together with Equation (6), this is sufficient to establish a *diagonal stripe* in the *pre*/*post* plane which is guaranteed to contain all elements with tag t.

For tag b in the DTD of Figure 7, for example, we can statically derive $1 \leqslant size(v) \leqslant 2$ for any node v with tag b in any valid instance. With Equation (6), we

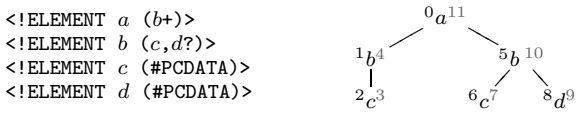


Fig. 7. A DTD and a valid pre-/postorder ranked XML document tree.

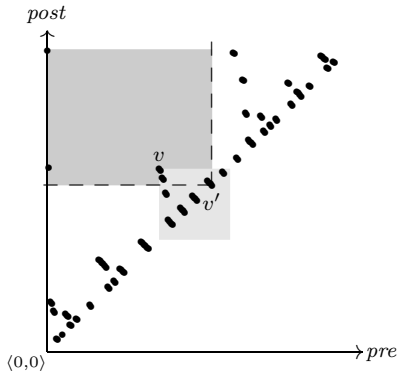


Fig. 8. A comparison of window areas in the stretched *pre/post* plane: Taking an `ancestor` step from v' (dark) *vs.* taking a `descendant` step from v (light).

thus can identify the stripe defined by $3 \leqslant post(v) - pre(v) \leqslant 5$ as the region of the *pre/post* plane that holds elements with tag b.

Note, however, that for tags t whose content models contain the regular expression constructors `+` and `*`, useful subtree size bounds cannot be established. For XML instances that have been validated against a given XML Schema, on the other hand, subtree size bounds might even be explicitly given by the occurrence attributes `minOccurs` and `maxOccurs`.

4.3 XPath Symmetries

Axis window size indeed is the dominating performance factor for the XPath accelerator. The correlation of window size and query response time is so evident that the simple *window size* notion could form the basis of a cost model for accelerated XPath evaluation.

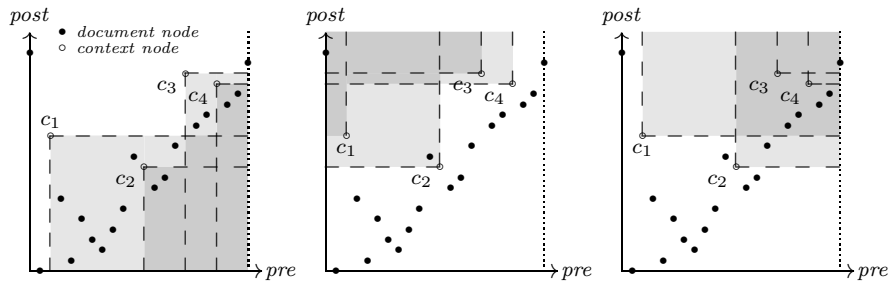
Suppose that we are processing the XPath expression below to retrieve all elements with tag name m containing at least one element named n:

$$\texttt{/descendant::}n\texttt{/ancestor::}m \ .$$

With the XPath accelerator we may, literally, follow two different paths to respond to the query (Figure 8 depicts the scenario in the *pre/post* plane):

(1) Establish the intermediary context node sequence containing element nodes with tag n, then, for each node v' in this sequence, evaluate the axis step $window(\texttt{ancestor::}m, v')$ to find the result element nodes v.

(2) Establish the result context node sequence containing all elements with tag m, then, for each node v in this sequence, evaluate $window(\texttt{descendant::}n, v)$ to check if v has an n element descendant v'; if no such v' is found, reject v.



(a) `descendant` axis (b) `ancestor` axis (c) `following` axis

Fig. 9. Overlapping query windows (context nodes c_i).

Observe that the second alternative corresponds to the XPath expression

$$\texttt{/descendant-or-self::}m\texttt{[descendant::}n\texttt{]}\ ,$$

the *symmetric equivalent* of the original query.

With the optimizations of Sections 4.1 or 4.2 applied, we know that we can reduce the area covered by *window*(`descendant`, v). For the 100-nodes document depicted in Figure 8 the benefit is clearly recognizable, but for real-world XML instances the reduced index scan effort of alternative (2) is substantial. This is even more so since *window*(`ancestor`, v') contains few document nodes only but the *pre* and *post* index range scans cover large portions of the document. They thus yield numerous false hits before index intersection determines the actual ancestors of v'. With alternative (2), if the system employs a stretched *pre*/*post* plane, the number of false hits can be reduced to zero: The two necessary `descendant-or-self` and `descendant` window queries to evaluate the symmetric equivalent never generate any false hits.

4.4 Context Node Sequences and Empty *pre*/*post* Plane Regions

Relational database engines derive much of their efficiency from a *set-oriented* mode of operation: Rather than operating on a tuple-by-tuple basis, query operators are applied to sets of tuples, generating set-valued results in general. We would give up a lot of this efficiency, if we did not adopt this execution model for database-supported XPath evaluation. Actually, set-orientation fits well with the sequence-oriented semantics of XPath (see Section 2): Axis steps are always evaluated for a *sequence of context nodes*.

In general, evaluating an axis step for a sequence of context nodes c_i leads to *pre*/*post* plane query regions that either include each other or partially overlap (dark areas in Figure 9). Nodes in these areas generate *duplicate* nodes in the final query result. To comply with the XPath semantics, a subsequent duplicate elimination phase is required if we evaluate the step for the context node sequence as is.

The *pre*/*post* encoding provides a simple means to avoid the generation of duplicate nodes altogether, however.

Recall that in the *pre*/*post* plane the four partitions contain the nodes of the four

major axes. When determining the combined descendants of two distinct context nodes v and v' (v and v' in document order), there are two possible cases:

(1) v' is a descendant of v, or
(2) v' follows v.

The two nodes partition the *pre*/*post* plane into nine regions as shown in Figure 10. Each region determines nodes which are in relationship with both context nodes, *e.g.*, in case (1), region V contains those nodes that are descendants of v *and* ancestors of v'.

Since we are working with tree data, certain regions are guaranteed to contain no nodes at all. In case (1), regions U and S are empty, because an ancestor of v' cannot precede or follow v if v' is a descendant of v. In case (2), region Z is empty, because v and v' cannot have a common descendant if v' follows v.

Consequently, in case (1) the combined descendants of v and v' are equal to the regions V, W, Y, and Z, *i.e.*, equal to the descendants of v alone. In case (2), it is equal to the regions Y and W. Because Z is empty, we can combine the descendants of both nodes without generating duplicates. These observations readily carry over to context node sequences of more than two nodes.

Now, if the database engine asserts to process the context node sequence in document order, we can optimize query window evaluation for all four major axes.

First, during `descendant` axis evaluation we can avoid unnecessary processing as follows:

—If the context node v' is a descendant of the previous one, we can skip v' resulting in less query windows actually being evaluated (case (1)).
—If the context node follows the previous one in document order, the descendants of both can be combined without generating duplicates (case (2)).
—Both observations together guarantee that no duplicates are generated at all, so duplicate removal is unnecessary.

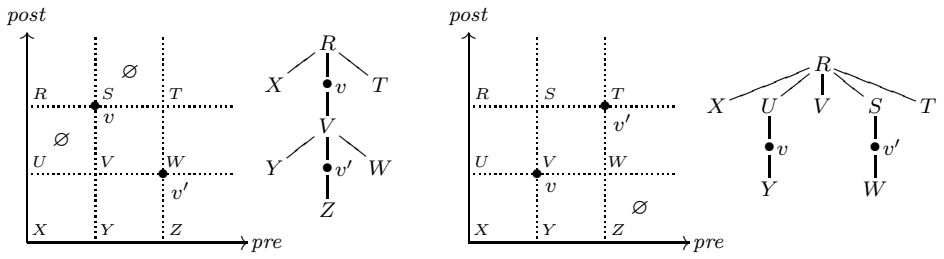
The test whether or not a context node v' is a descendant of the previous context node v is simple: Evaluate $post(v') < post(v)$. Iteration in document order already ensures that $pre(v') > pre(v)$.

Second, for the `ancestor` axis, we can derive an *incremental* evaluation strategy: The lower *pre* limit for the `ancestor` window of context node v' may be determined by $pre(v)$, with v being the context node processed just prior to v', *i.e.*:

$$window(\texttt{ancestor}, v') = \langle (pre(v), pre(v')), (post(v'), \infty), *, elem, * \rangle \ .$$

Figure 11 illustrates the resulting incremental `ancestor` window evaluation: The `ancestor` window for v' does not extend to the left of the dashed line marking the *pre* rank of the prior context node v. As we process the context node sequence in document order, the dashed line "sweeps" the *pre*/*post* plane from left to right. In case (1), this avoids the duplication of nodes in region R, in case (2), this only produces nodes in region S, again avoiding duplication of the nodes in regions R.

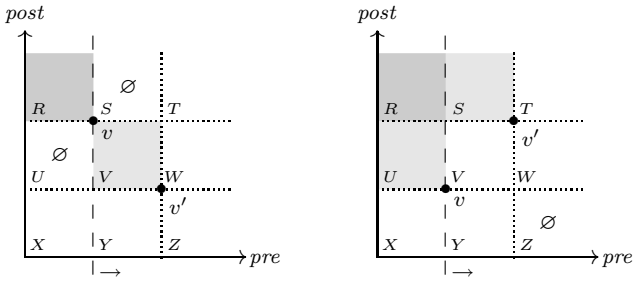
Finally, the nine-fold partitioning in Figure 10 also reveals optimization opportunities for the `preceding` and `following` axis. The combined preceding nodes of



Case (1): Context nodes v and v' relate to each other on the `ancestor`/`descendant` axis.

Case (2): context nodes v and v' relate to each other on the `preceding`/`following` axis.

Fig. 10. Identifying empty regions in the *pre*/*post* plane.



Case (1): context nodes v and v' are in `descendant`/`ancestor` relationship.

Case (2): context nodes v and v' are in `preceding`/`following` relationship.

Fig. 11. Redundant work saved during optimized set-oriented `ancestor` evaluation: in either case, region R is scanned once only (context node set $\{v, v'\}$ processed in document order). The arrows ($\rightarrow$) indicate the direction in which the dashed lines "sweep" the plane while the context node set is being processed.

v and v' are regions X and Y in case (1), *i.e.*, the preceding nodes of v', because region U is empty. In case (2), it is equal to regions U, V, X, and Y, *i.e.*, again the preceding nodes of v'. Consequently, the combined preceding nodes of a sequence of context nodes is equal to the preceding nodes of the context node with maximum preorder rank. Analogously, the nodes in the `following` axis of a sequence of context nodes is equal to the following nodes of the context node with minimum preorder rank: for both the `preceding` and `following` axes, the system can always reduce the context node sequence to a singleton.

5. BACK-ENDS FOR THE XPATH ACCELERATOR

The implementation prerequisites for the XPath accelerator are rather light. We implemented a SAX-based document loader (see the electronic appendix) and fed its output into implementations on top of two different back-ends:

(1) a purely relational implementation on top of IBM DB2 V7.1, and

(2) a main-memory DBMS implementation on top of Monet [Boncz 2002].

In Sections 5.1 and 5.2, we describe both in more detail. We explain the storage structure, the XPath evaluation scheme used, how to apply the tree and XPath specific optimizations of Section 4, and back-end specific issues. Additionally, we describe in Section 5.3 some specifics of indexing the node descriptor space with R-trees.

5.1 A Purely Relational Implementation

5.1.1 *Storage Structure.* The XPath accelerator has been designed to enable efficient *relational* XPath support: XML documents are represented via relational data structures (*i.e.*, tables), and XPath queries are evaluated by mapping such queries to relational equivalents (*i.e.*, SQL). Such an implementation is *purely relational* in the sense that we do not require to invade the relational database kernel to implement XPath support.

The most straightforward way to represent a node v's 5-dimensional descriptor $desc(v)$ (see Section 3.1) inside the RDBMS is to load it into a 5-column table *accel* with schema $\underline{\mathit{pre} \mid \mathit{post} \mid \mathit{par} \mid \mathit{kind} \mid \mathit{name}}$.

Non-element content, *e.g.*, the actual characters associated with a text node, attribute values, comment content, or the target and instruction of an XML processing instruction, is held outside the main table *accel*. With each document node v being uniquely identified by its preorder rank $pre(v)$, we maintain separate *content relations* $\underline{\mathit{pre} \mid \mathit{text}}$, $\underline{\mathit{pre} \mid \mathit{attr}}$, $\underline{\mathit{pre} \mid \mathit{comment}}$, $\underline{\mathit{pre} \mid \mathit{p\text{-}i}}$ instead, save the document content into the appropriate relation and establish the *pre* columns as foreign keys referencing the *accel* table.

We have found this table layout to come with advantages: the evaluation of the actual XPath axes and node tests exclusively touches the *accel* table with the content relations only accessed when absolutely necessary, for example, during value atomization [Berglund et al. 2002, Section 2.4.3.1] or result serialization after query processing has finished.

The electronic appendices contain details about the XML loading process to populate these tables and how to reconstruct the original XML document, respectively.

5.1.2 *XPath Evaluation Scheme.* The evaluation of an XPath path expression $p = s_1/s_2/\cdots/s_n$ leads to a series of n region queries where the node sequence output by step s_i is the context sequence for the subsequent step s_{i+1}. The context node sequence for step s_1 is held in table *context*. If p is an absolute path, *i.e.*, $p = /s_1/\cdots$, *context* holds a single tuple, namely the encoding of the document root. For the XML fragments of Figure 1 we would thus have

$$\mathit{context} \quad = \quad \begin{array}{|c|c|c|c|c|} \hline \mathit{pre} & \mathit{post} & \mathit{par} & \mathit{kind} & \mathit{name} \\ \hline 0 & 9 & \text{␣} & \mathit{elem} & a \\ \hline \end{array} \ .$$

We arrive at the plain SQL implementation shown in Figure 12. XPath requires the resulting node sequence to be *duplicate free* as well as being sorted in *document order* [Berglund et al. 2002] which explains the presence of the `DISTINCT` and `ORDER BY` clauses in lines 1 and 4, respectively. Function `INSIDE`$(\cdot)$ implements the actual

```
1   SELECT DISTINCT v_n.*
2     FROM context c, accel v_1, ..., accel v_n
3    WHERE INSIDE(window(s_1, c), v_1) AND ··· AND INSIDE(window(s_n, v_{n-1}), v_n)
4 ORDER BY v_n.pre ASC
```

Fig. 12. XPath to SQL translation scheme for the XPath expression $p = [/]s_1/s_2/\cdots/s_n$.

query window test, *e.g.*:

$$\texttt{INSIDE}(\langle [pre_l, pre_h], [post_l, post_h], p, k, n\rangle, v) \equiv$$
$$pre_l < v.pre \texttt{ AND } pre_h > v.pre \texttt{ AND } post_l < v.post \texttt{ AND } post_h > v.post \texttt{ AND}$$
$$v.par = p \texttt{ AND } v.kind = k \texttt{ AND } v.name = n \ .$$

The existential semantics of XPath predicates are naturally expressed by a simple exchange of correlation variables in the translation scheme of Figure 12. The XPath expression $s_1\texttt{[}s_2\texttt{]/}s_3$[2] is evaluated by the RDBMS via the SQL query (note the exchange of v_1 for v_2 in $\texttt{INSIDE}(window(s_3, v_1), v_3)$, line 4):

```
1   SELECT DISTINCT v_3.*
2     FROM context c, accel v_1, accel v_2, accel v_3
3    WHERE INSIDE(window(s_1, c), v_1) AND INSIDE(window(s_2, v_1), v_2) AND
4          INSIDE(window(s_3, v_1), v_3)
5 ORDER BY v_3.pre ASC
```

For query $p = \texttt{descendant-or-self::}n\texttt{/preceding-sibling::text()}$, we obtain the SQL query below in which simplifications like the removal of comparisons with ∞ or $*$ have already been made:

```
SELECT DISTINCT v_2.*
  FROM context c, accel v_1, accel v_2
 WHERE c.pre ⩽ v_1.pre AND v_1.post ⩽ c.post
   AND v_1.name = n
   AND v_2.pre < v_1.pre AND v_2.post < v_1.post
   AND v_2.par = v_1.par
   AND v_2.kind = text
ORDER BY v_2.pre ASC .
```

5.1.3 *Index selection.* Each $\texttt{INSIDE}(\cdot)$ query window test generates a conjunction of two range predicates plus up to three equality comparisons. The range predicates are efficiently supported by regular B-trees.

With two separate B-trees on the *pre* and *post* columns, the system needs two separate B-tree range scans whose results are then intersected. In case of IBM DB2, however, the RDBMS's optimizer detected the opportunity to use the `IDXAND` index intersection operator to efficiently compute window contents. Alternatively, concatenated *pre-post* B-trees can be used to support the XPath accelerator.

For the experiments of Section 6, we created two ascending B-tree indexes on the *pre* and *post* columns of the *accel* table, respectively (note that both *pre* and *post*

[2]Here, s_2 is assumed to be a path expression again—a treatment of the translation of the general XPath predicate syntax is beyond the scope of this text.

are unique). Additionally, we requested to cluster the *accel* table with respect to the *pre* index. This enabled IBM DB2 to generate query plans which avoided the extra sort for document order.

All other node descriptor components (*par*, *kind*, *name*) simply require equality comparisons which we accelerated via hash indexes.

5.1.4 *Enhancing Tree and XPath Awareness.* To conclude our discussion of the purely relational back-end, let us briefly review to which extent an RDBMS can actually benefit from the XPath accelerator optimizations developed in Section 4.

Staking Out Subtrees. This optimization makes the `descendant` query window size a function of the actual subtree size (or the tree height, respectively, see Equation 4). In Section 4.1, we modified the window definition $window(\texttt{descendant}, v)$ such that the XPath evaluation scheme of Figure 12 need not be changed to benefit: The `INSIDE`$(\cdot)$ query window test will generate significantly tighter range query bounds in both the *pre* and *post* dimensions.

A Stretched pre/post Plane. We may, again, stick to the evaluation scheme of Figure 12. For a `descendant` axis step, however, the `INSIDE`$(\cdot)$ window test will generate a *single* range predicate on either the *pre* or *post* dimension. A single B-tree range scan suffices to evaluate such a step, and index intersection becomes superfluous.

XPath Symmetries. Transforming an XPath expression into its symmetric equivalent is perfectly applicable to the relational implementation. Transformations should try to trade an `ancestor` for a `descendant` step, *e.g.*, apply rewrite rules like (taken from [Olteanu et al. 2001])

$$\begin{array}{rcl} \texttt{/descendant::}n\texttt{/ancestor::}m & \rightarrow & \texttt{/descendant-or-self::}m\texttt{[descendant::}n\texttt{]} \\ \texttt{/descendant::}n\texttt{[ancestor::}m\texttt{]} & \rightarrow & \texttt{/descendant-or-self::}m\texttt{/descendant::}n \ , \end{array}$$

since the `descendant` steps on the right-hand sides benefit from query window reduction and *pre*/*post* plane stretching.

Context Node Sequences and Empty pre/post Plane Regions. The empty region analysis of Section 4.4 introduces dependencies between the nodes in *context*. To preprocess and reduce the context sequence for the `descendant` and `ancestor` axes we thus require a rather expensive self-join of *context* with itself, the cost of which is generally not paid back by making the duplicate elimination in the original evaluation scheme unnecessary. Context node skipping "on the fly", *i.e.*, during processing of the context node sequence, would require the RDBMS to remember already seen context nodes to compare postorder ranks. Such memory can be added to the implementation of the RDBMS's join operators [Grust and van Keulen 2003]. This would, however, violate our goal of developing a purely relational implementation for the XPath accelerator.

5.2 Monet: A Main-Memory DBMS Implementation

Monet is a DBMS specifically targeted at query-intensive applications such as OLAP. It uses a fully fragmented data model that consists of tables with only

two columns, called *BATs* (Binary Association Table). Monet's query processing infrastructure is optimized towards main-memory execution. Monet was designed to work in a front-end/back-end system architecture where the back-end provides a kernel of DBMS facilities, and several co-existing front-ends provide SQL, ODMG, and application-specific support. These front-ends communicate with the back-end using the *Monet Interpreter Language (MIL)*. The core of MIL is a query algebra on its single bulk data type BAT supplemented with control structures, aggregators, iterators, and accelerators. Other features are ADT-based extensibility, module-support, and much more. For details, we refer to [Boncz and Kersten 1999]. Within this architecture, our implementation amounts to an XPath front-end that maps axis operations onto MIL procedures.

Some of the observations that motivated the choice for Monet are:

—The *pre/post* plane is an encoding of the XML document tree based on a binary relationship. This fits well with the *binary relational data model* of Monet, where all tables have exactly two columns.

—Monet, being a main-memory DBMS, can benefit from certain properties of the node ranking schemes which a traditional RDBMS cannot. Its programming language-like query interface, called MIL, additionally enables tuning techniques that are not available in an SQL interface (or an R-tree API for that matter).

—The full vertical fragmentation induced by the binary relational approach in a natural way avoids loading of data that is not needed to answer a query (*e.g.*, text content or tags).

—And finally, Monet, being a full-fledged DBMS, offers readily available functionality that facilitates experimentation and tuning, such as native support for enumerated types.

5.2.1 *Storage Structure.* The two columns of a BAT are called *head* and *tail*, respectively. Any (multi-column) table structure can be represented with BATs. The commonly used technique is to introduce a separate BAT for each column in the original table, put the values of the columns in the tail of each BAT, and use an identifier in the head to relate them. For the table *accel* of Section 5.1.1, the preorder rank is used as identifier. We systematically name BATs as *tablename_columnname*, such as *accel_name* for the BAT containing all element and attribute names. Furthermore, non-element content can simply be stored in separate BATs, such as text content of text nodes in *accel_text*. The presence of the unique preorder rank $pre(v)$ in BATs such as *accel_text* implies that node v actually is a node of that kind (*i.e.*, of kind *text*). Therefore, we have refrained from using a *accel_kind* BAT. Table III gives an overview of the storage structure in Monet.

There are two major advantages to keep these BATs sorted on preorder rank, *i.e.*, store the nodes in document order. First, Monet may exploit this property to select more efficient algorithms for operations on such BATs. Secondly, a feature that plays a prominent role is Monet's support for *void columns* (*void* stands for *virtual oid*). If a column contains a range of consecutive numbers, then there is no need to store these numbers individually. It suffices to store the offset. Such a column is called *dense*. As *accel_prepost* and *accel_par* contain all nodes of a document, their heads contain a consecutive sequence of numbers, and, thus, can

Table III. Representation of the (multi-column) table *accel* with BATs.

accel_prepost	*pre* | *post*	Preorder and postorder ranks of all nodes.
accel_par	*pre* | *par*	Preorder rank of parents of all nodes except root.
accel_attr	*pre* | *attr*	Attribute values of all attribute nodes.
accel_name	*pre* | *name*	Names of element tags and attributes.
accel_text	*pre* | *text*	Text content of text nodes.

be stored as *void* columns. This reduces the storage requirements of these BATs by 50% and may result in selection of even more efficient algorithms like selections by offset. Realize that this means that the XML document tree structure has simply been encoded by an ordered sequence of postorder rank values.

It is likely that the number of different element tags and attribute names in a given document is rather small. Therefore, it is beneficial to set up a translation table to map these to numerical values. Monet provides support for enumerated types that does exactly this in a transparent way. As a consequence, almost all our BATs are numerical in both head and tail.

5.2.2 *The MIL Algebra.* A typical example of a BAT algebra operator is `select`. The MIL-expression *accel_prepost* · `select`(n_1, n_2) selects all rows of the BAT *accel_prepost* with a tail-value (*i.e.*, a postorder rank) between n_1 and n_2.[3] There is no distinct operator for selection on the head, because instead, there is an operator `reverse` which swaps the head and tail columns. The BAT internal data structure consists of a few descriptor records containing pointers into a heap that contains the actual data. Because of this implementation, `reverse` is an operation on the descriptors only, which does not touch the data inside the BAT. Hence, this operator executes in a negligible and data volume independent amount of time. The BAT data structure supports more such "tricks." The operator `slice`, for example, selects rows positionally (*e.g.*, the first one hundred rows). This operator is also executed without the need to copy the selected rows into a new BAT. The MIL operators that are used in this paper are listed in Table IV.

MIL operators are defined in an algebraic way, independent of the algorithms implementing them. Query-optimizing front-ends produce MIL programs deciding the execution order (strategic optimization), and the MIL operator implementation chooses at run-time the appropriate algorithm based on properties and statistics of the operands involved (tactical optimization). For the one operator `select`, there are 10 algorithms implemented, which are expanded to 55 type-optimized routines.[4] One of the algorithms is obviously a sequential scan over the entire BAT. If, however, the tail column is sorted (one of the many properties a BAT

[3]The dot-notation is interpreted as a function application where the operand on the left is the first parameter, *e.g.*, *accel_prepost* · `select`$(n_1, n_2) \equiv$ `select`$(accel\_prepost, n_1, n_2)$. Operator · is left-associative.

[4]These numbers are taken from [Boncz and Kersten 1999] and, hence, valid for the then used version of Monet. Other (newer) versions may, obviously, have a different number of algorithms and routines.

Table IV. Some MIL operators and their semantics. Although the set notation below suggests otherwise, BATs are actually sequences, *i.e.*, they may contain duplicates and order matters. This also makes `find` and `kunique` deterministic as they will choose the first row encountered if there are multiple choices. `find` may fail, though, if no such (a, b) exists.

Operator	Semantics	Comment
`reverse`(A)	$\{(b, a) \mid (a, b) \in A\}$	zero cost
`mirror`(A)	$\{(a, a) \mid (a, b) \in A\}$	zero cost
`select`(A, n)	$\{(a, b) \mid (a, b) \in A \wedge b = n\}$	
`select`(A, n_1, n_2)	$\{(a, b) \mid (a, b) \in A \wedge n_1 \leqslant b \leqslant n_2\}$	zero cost if tail dense, $n_1 =$ `nil` and $n_2 =$ `nil` represent $-\infty$ and ∞, respectively
`project`(A, n)	$\{(a, n) \mid (a, b) \in A\}$	$n =$ `nil` effectively projects out tail values
`join`(A, B)	$\{(a, d) \mid (a, b) \in A \wedge (c, d) \in B \wedge b = c\}$	
`semijoin`(A, B)	$\{(a, b) \mid (a, b) \in A \wedge (c, d) \in B \wedge a = c\}$	
`kdiff`(A, B)	$\{(a, b) \mid (a, b) \in A \wedge \nexists (c, d) \in B \,.\, a = c\}$	
`find`(A, n)	b such that $\exists (a, b) \in A \,.\, a = n$	zero cost if head dense
`kunique`(A)	$B \subseteq A$ such that $\forall (a, b) \in A \, \exists! (c, b) \in B \,.\, c = a$	
`count`(A)	$\lvert A \rvert$	number of rows in A
`insert`(A, B)	$A \leftarrow A \cdot$ `append`(B)	destructive update

can have), an algorithm is chosen that does a binary search-based lookup to locate the first and last positions and then produces a slice. The latter is evidently much more efficient. A selection on a dense column need not even do a lookup, but can calculate the first and last positions directly using the offset. Execution of a select on such a BAT, hence, only evaluates a slice, which takes only a negligible amount of time.

The algorithms and data structures in Monet are optimized for main-memory access. With vertical fragmentation successfully avoiding unnecessary I/O, the balance of query processing cost shifts from I/O to CPU cycles and memory access time. This calls for different optimization criteria. As advances in CPU speed far outpace advances in DRAM latency, the effect of optimal use of memory caches becomes ever more important. Pipelined memory transfer capabilities of modern CPUs make sequential access significantly faster than random access. As a result, many elaborate index structures as well as search and join algorithms in traditional RDBMSs, are defeated by simple (sequential scan) algorithms. The XPath accelerator indeed uses sequential index scans over preorder and postorder ranges to evaluate a window query.

Note, however, that Monet is not an all-or-nothing main-memory system. If the database hot set exceeds main memory, the system relies on operating system support for managing virtual memory. Modern operating systems increasingly respond well to memory access advice, which has made the main-memory approach indeed feasible.

5.2.3 *XPath Evaluation Scheme.* Evaluating an axis step in the XPath accelerator amounts to evaluating simple range predicates on the preorder and postorder ranks. For example, the query v`/descendant-or-self::node()` results in all nodes

accel_prepost · `reverse` · `select`($pre(c)$, `nil`) · `reverse` · `select`(0, $post(c)$)

pre	*post*
0	9
1	3
2	2
3	0
4	1
5	8
6	4
7	7
8	5
9	6

post	*pre*
9	0
3	1
2	2
0	3
1	4
8	5
4	6
7	7
5	8
6	9

post	*pre*
2	2
0	3
1	4
8	5
4	6
7	7
5	8
6	9

pre	*post*
2	2
3	0
4	1
5	8
6	4
7	7
8	5
9	6

pre	*post*
2	2
3	0
4	1

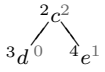


Fig. 13. MIL execution and resulting XML fragment for *c*`/descendant-or-self::node()` (with $pre(c) = 2$ and $post(c) = 2$).

in the *pre*/*post* plane window $\langle[pre(v), \infty), [0, post(v)]\rangle$. In MIL:

$$v\texttt{/descendant-or-self::node()} \equiv \begin{array}{l} \mathit{accel\_prepost} \\ \quad \cdot \texttt{reverse} \\ \quad \cdot \texttt{select}(pre(v), \texttt{nil}) \\ \quad \cdot \texttt{reverse} \\ \quad \cdot \texttt{select}(0, post(v)) \end{array} \tag{7}$$

Figure 13 illustrates how Monet executes this MIL-program step-by-step for context node *c* in the example document of Figure 3. The other axis steps can be implemented analogously.

Before we proceed, a small intermezzo on the performance of these MIL expressions. Recall that '*accel_prepost* · `reverse`' is dense in its tail and that a `select` on such a BAT amounts to executing a positional slice. Hence, the first three operators are executed in a negligible amount of time. The clock starts ticking at the second `select` which operates on the unsorted column with postorder ranks.

Moreover, Monet chooses an algorithm for `select` that produces a sorted BAT, such that subsequent operators benefit from our decision to store the original BATs in document order as well. In conclusion, even without other optimizations that can be applied as we will see later on, this is already a rather efficient algorithm.

An axis step results in a set of nodes. When there are subsequent axis steps in the query, these intermediary nodes are recursively interpreted as context nodes for the subsequent axis steps according to the XPath specification [Berglund et al. 2002]. In the database field, we have learned that it is fruitful to think in a set-oriented way. Analogously, it is fruitful to regard an axis step as determining in one go the resulting nodes for an entire sequence of context nodes instead of for a single context node.

Therefore, we developed, for each axis step and node test, MIL procedures that have an input BAT parameter *context* representing the context node sequence. An example of such a procedure is given for `descendant-or-self` in Figure 14. The algorithm simply iterates over the sequence of context nodes, determines for each the descendants, and gathers them in *result*. As mentioned earlier, the XPath semantics requires results to be duplicate free and in document order. In MIL, this is implemented as a post-processing step *result* · `kunique` · `sort`.

```
proc_descendant_or_self(context)
  result ← NEW BAT;
  FOR v IN context DO
  |  desc ←
  |    accel_prepost
  |      · reverse
  |      · select(pre(v), nil)
  |      · reverse
  |      · select(nil, post(v));
  |  result · insert(desc);
  RETURN result · kunique · sort;
```

Fig. 14. Set-oriented algorithm for `descendant-or-self`.

The evaluation of an XPath path expression $p = \alpha_1::\tau_1/\alpha_2::\tau_2/\cdots/\alpha_n::\tau_n$ leads to a series of MIL procedure calls where the node sequence output of the one is the context sequence for the next MIL procedure:

$$context \cdot \texttt{proc}_{\alpha_1} \cdot \texttt{proc}_{\tau_1} \cdot \texttt{proc}_{\alpha_2} \cdot \texttt{proc}_{\tau_2} \cdots \texttt{proc}_{\alpha_n} \cdot \texttt{proc}_{\tau_n}$$

where $\texttt{proc}_\alpha$ and $\texttt{proc}_\tau$ stand for the MIL procedure corresponding with axis step α and node test τ, respectively. If p is an absolute path, *context* is the MIL procedure `root`(*doc*) which returns a context node sequence containing the root node of (current) document *doc*.

For query `/descendant-or-self::n/preceding-sibling::text()`, we obtain the MIL expression

```
root(doc)
  · descendant_or_self · nametest(n)
  · preceding_sibling · nodetest(text)
```

In MIL, XPath predicate evaluation requires more than in the pure relational case, where a simple exchange of correlation variables was sufficient. Apart from an iterative approach which evaluates the predicate on a per context node basis, XPath symmetries can be used to remove predicates containing path expressions. In general, an expression s_1`[`s_2`]`—with s_2 being a path expression again—can be rewritten into the symmetrical equivalent

$$s_1 \texttt{ intersect } s_1\texttt{/}s_2\texttt{/ancestor-or-self::node()}$$

XPath's `intersect` can simply be evaluated by MIL's `semijoin`.

5.2.4 *Enhancing Tree and XPath Awareness.*

Staking Out Subtrees. The technique of Section 4.1 shrinks the query window of an axis step. Incorporating this in the XPath evaluation scheme for Monet requires only an adaptation of the window boundaries. MIL expression (7) on page 24

becomes

$$
\begin{aligned}
&v\texttt{/descendant-or-self::node()} \quad \equiv \\
&\quad \mathit{accel\_prepost} \\
&\qquad \cdot\, \texttt{reverse} \\
&\qquad \cdot\, \texttt{select}(\mathit{pre}(v), \mathit{post}(v) + \mathit{height}(t)) \\
&\qquad \cdot\, \texttt{reverse} \\
&\qquad \cdot\, \texttt{select}(\mathit{pre}(v) - \mathit{height}(t), \mathit{post}(v))
\end{aligned}
$$

The Monet implementation benefits primarily from the reduced window on the preorder rank, because it results in a small intermediary result. Therefore, a smaller BAT needs to be scanned by the second `select`.

A Stretched pre/post Plane. Stretching the *pre/post* plane has the benefit of evaluating an axis step using a single `select` on either preorder *or* postorder rank. Because Monet can exploit more efficient implementations for `select` if the corresponding column is sorted, the best choice is to evaluate the window on the *pre*-column which is sorted to reflect document order.

Unfortunately, a stretched *pre/post* plane results in preorder ranks that are not consecutive anymore. Therefore, the *pre*-column is no longer *dense*. Consequently, preorder ranks need to be stored as well requiring more bytes per node. This has an immediate effect on query performance as well, because the volume of data to be accessed in main-memory doubles. Furthermore, several efficient MIL operator implementations exploiting the dense property become unavailable.

Therefore, a trade-off occurs between the advantage of being able to evaluate an axis step using a single `select` and the disadvantage of loosing the dense property. Experiments have shown that, in general, the disadvantage is more severe than the advantage. We therefore propose the original non-stretched ranking scheme for the Monet back-end.

XPath Symmetries. Rewriting an XPath expression into an equivalent results in a different order of execution using the XPath evaluation scheme for Monet. Since each XPath axis step is evaluated using a number of algebraic operations in MIL, optimization by reordering operations is also possible on MIL-level. Here, it may be beneficial to exchange MIL-operations within one axis step or exchange an operation from one axis step with operations from another. The "Node Test Push Down" technique below is an example of such an optimization.

Empty pre/post Plane Regions. Using the iterative capabilities of Monet, optimizations related to empty *pre/post* plane regions can be fully exploited. One of the most prominent examples of such an optimization, is being able to reduce the context node sequence to one single node in case of evaluating `preceding` or `following` axis steps. Figure 15 shows how this is done in MIL for `preceding`.

Besides the benefit of reducing the number of query window evaluations, the post-processing phases of duplicate removal and sorting in document order can also be avoided. This holds for the MIL procedures for all axis steps. Another example can be found in Figure 16, which incorporates some other optimizations as well. For `descendant-or-self`, some context nodes can be skipped. The necessary memory of the postorder ranks of previously encountered context nodes is implemented by means of the MIL variable *max*.

```
proc_preceding(context)
  maxpre ← context · reverse · max;
  post ← accel_prepost · find(maxpre);
  RETURN accel_prepost · reverse
                       · select(nil, maxpre − 1)
                       · reverse
                       · select(nil, post − 1);
```

Fig. 15. The combined preceding nodes of a sequence of context nodes are the preceding nodes of the context node with maximum preorder rank.

Node Test Push Down. An axis step $c/\alpha\texttt{::}\tau$ results in all nodes that are in the *pre*/*post*-plane region corresponding with α *and* qualify for node test τ. It does not matter if one

(1) first determines all nodes in the appropriate *pre*/*post*-plane region and then selects among those the ones qualifying for the node test, or
(2) first determines all nodes that qualify for the node test and then selects among those the ones in the appropriate *pre*/*post*-plane region.

In a pure relational implementation using SQL, the query optimizer of the RDBMS will make such decisions, *i.e.*, pushing highly selective selections down in the query plan. In the query optimization approach of Monet, which distinguishes strategic and tactical optimization, this kind of optimization is strategic, because it concerns the order of execution. Hence, it is the responsibility of the XPath front-end to explicitly choose the most efficient execution order in MIL.

We will illustrate node test push down by means of the example path expression v`/descendant-or-self::`n. Combining the axis step and the subsequent name test results in the following MIL expression:

$$
\begin{aligned}
v\texttt{/descendant-or-self::}n \quad \equiv \quad & \mathit{accel\_prepost} \\
& \cdot \texttt{reverse} \\
& \cdot \texttt{select}(\mathit{pre}(v), \mathit{post}(v) + \mathit{height}(t)) \\
& \cdot \texttt{reverse} \\
& \cdot \texttt{select}(\mathit{pre}(v) - \mathit{height}(t), \mathit{post}(v)) \\
& \cdot \texttt{mirror} \\
& \cdot \texttt{join}(\mathit{accel\_tag}) \\
& \cdot \texttt{select}(n)
\end{aligned}
$$

The alternative of doing the name test first can be expressed in MIL as follows:

$$
\begin{aligned}
v\texttt{/descendant-or-self::}n \quad \equiv \quad & \mathit{accel\_prepost} \\
& \cdot \texttt{semijoin}(\mathit{accel\_tag} \cdot \texttt{select}(n)) \\
& \cdot \texttt{reverse} \\
& \cdot \texttt{select}(\mathit{pre}(v), \mathit{post}(v) + \mathit{height}(t)) \\
& \cdot \texttt{reverse} \\
& \cdot \texttt{select}(\mathit{pre}(v) - \mathit{height}(t), \mathit{post}(v))
\end{aligned}
$$

The general point here is that the properties of the *pre*/*post* plane and axis windows remain valid if one "deletes" some nodes in the plane (*e.g.*, via an early node test). For a set-oriented algorithm, this even means that we can do a node test as early as even outside the loop that iterates over the context node sequence.

```
proc_descendant_or_self_nametest(context, name)
  result ← NEW BAT;
  max ← 0;
  nametest ← accel_tag · select(name);
  restrict ← accel_prepost · semijoin(nametest);
  FOR v IN context DO
    IF post(v) > max THEN
      desc ←
        restrict
          · reverse
          · select(pre(v), post(v) + height(t))
          · reverse
          · select(pre(v) − height(t), post(v));
      result · insert(desc);
      max ← post(v);
  RETURN result;
```

Fig. 16. Algorithm for a combined `descendant-or-self` and name test showing staking out subtrees, node test push down, and the ability to skip over context node which either produce empty or duplicate results.

In situations where the intermediary results between axis steps are large, this is a considerable performance gain. Furthermore, since a node test may be quite selective, the operations inside the loop are performed on considerably smaller BATs.

Figure 16 shows an example of an optimized MIL procedure, namely one for a combined `descendant-or-self` and name test. It incorporates the shrunk query windows of staking out subtrees, node test push down, and skipping context nodes based on empty *pre*/*post* plane regions.

5.3 Indexing the XML Node Descriptor Space with R-Trees

Although the hierarchical structure of XML document trees is already fully captured by the two-dimensional *pre*/*post* plane, the evaluation of an XPath step with an associated kind or name test needs to inspect all five dimensions of the node descriptor space in general (Section 3.1). Domains of such dimensionality have been found to be efficiently supported by R-trees [Guttman 1984; Böhm et al. 2000]. A 5-dimensional point R-tree is able to evaluate all dimensions of an axis window "in parallel". This saves the database host from the otherwise necessary post-processing to remove those nodes found in a *pre*/*post* plane region that fail to satisfy a given node test.

The data organization may be further improved by *R-tree packing* techniques [Roussopoulos and Leifker 1985; Kamel and Faloutsos 1993]: At the cost of using temporary storage for sorting, 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.

For the particular R-tree implementation we used for the experiments in Section 6, the GiST library providing a family of generalized search tree variants [Hellerstein et al. 1995], leaf packing had the additional beneficial effect of R-tree window queries returning result nodes in increasing order of *pre* values, *i.e.*, in document order.

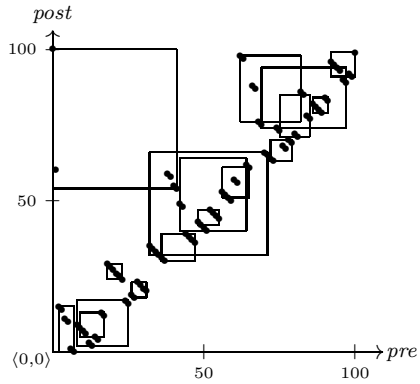


Fig. 17. Leaf level of a *preorder packed* R-tree after loading an XML instance of 100 nodes (R-tree leaf page capacity 6 nodes).

5.3.1 *Enhancing Tree and XPath Awareness.* While, besides *preorder packing*, there remain only few handles to make an R-tree based implementation more tree and XPath aware, it is interesting to see how the R-tree data structure implicitly realizes improvements which we have discussed for the XPath accelerator. This is primarily due to the R-tree's *incomplete partitioning* of the space (as opposed to *space partitioning* trees like the quad tree). Note how the R-tree leaf level reflects the typical shape of an XML document tree in the *pre*/*post* plane (Figure 17):

(1) The lower-right half below the diagonal is empty; no R-tree page has its center beyond the diagonal which corresponds to the `descendant` window optimization of Section 4.1.
(2) The sparsely populated upper-left half above the diagonal is covered by few R-tree leaves, such that the evaluation of `ancestor` steps touches few leaf pages.

The data-driven R-tree adapts well to the irregularly populated *pre*/*post* plane. It remains balanced even in the presence of XPath accelerator's skewed node distribution, and thus implements a notion of empty region analysis although by different means than those discussed in Section 4.4.

6. PERFORMANCE CHARACTERISTICS

This section intends to illustrate that the XPath accelerator can turn RDBMSs into efficient XPath processors. In Section 6.1, in particular, we will assess the effects of an enhanced tree and XPath awareness for IBM DB2 and Monet. Section 6.2 concludes with a performance comparison of the three XPath accelerator back-ends—IBM DB2, Monet, R-tree (GiST)—as well as two additional alternative database-supported XPath processors.

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* [Schmidt et al. 2002]. For a fixed DTD modeling an Internet auction site (see the element hierarchy depicted in Figure 18), this generator produces instances of controllable size. Table V lists the document sizes we were using for our experiments. All documents were of height 11.

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

Document size [MB]	# Nodes	XMLgen factor
0.11	5 257	0.001
0.55	25 951	0.005
1.1	52 180	0.01
11.0	511 474	0.1
55.0	2 538 027	0.5
111.0	5 077 531	1.0

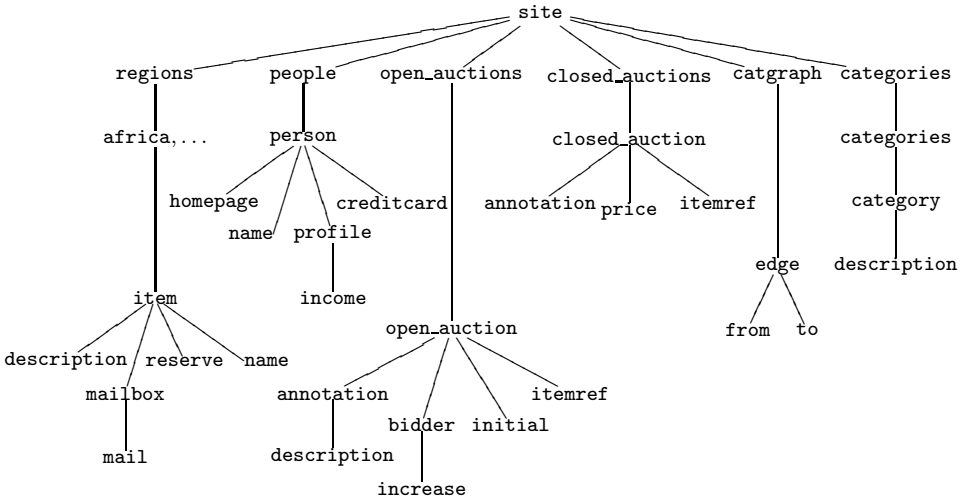


Fig. 18. Element hierarchy (top-level) of XMark XML benchmark document instances.

Against these instances, we ran a selection of three queries:

Q_1 `/descendant::open_auction/descendant::description`

Q_2 `/descendant::age/ancestor::person`

Q_3 `/descendant::open_auction/child::privacy/preceding-sibling::bidder`

Note that these queries stress the navigational (or structural) aspects of XPath queries, *i.e.*, we mainly measured the performance of raw path navigation, because this is what the XPath accelerator has been designed for.

The experimental setup was hosted on an Intel Pentium III PC, clocked at ≈ 1.2 GHz, using a version 2.4 Linux kernel, and running off a standard `ext2` file system on a SCSI hard disk. The host was equipped with 2 GB of RAM (no swapping occurred), and the system load average was near zero (no other processes were present besides a small number of sleeping system daemons).

6.1 Effects of Enhanced Tree and XPath Awareness

6.1.1 *IBM DB2 and Staking Out Subtrees.* Query Q_1 stresses the `descendant` axis and thus should benefit significantly from a reduction of axis window size as

Table VI. XPath traversals with and without staked out query window sizes. (Platform IBM DB2, Query Q_1)

Document size [MB]	# Result nodes	t [ms]	t^{stake} [ms]
11.0	1 200	855	70
55.0	6 000	27 860	246
111.0	12 000	291 731	470

Table VII. XPath traversals with and without stretched *pre*/*post* plane. (Platform IBM DB2, Query Q_1)

Document size [MB]	# Result nodes	t^{stake} [ms]	$t^{stretch}$ [ms]
11.0	1 200	70	45
55.0	6 000	246	222
111.0	12 000	470	444

discussed in Section 4.1. We ran query Q_1 against a B-tree based XPath accelerator on top of IBM DB2 (Section 5.1). Table VI shows the timing results as well as the size of the result node sets. The shrunk *pre* and *post* B-tree scan ranges for the staked out `descendant` window result in a speed-up of up to three orders of magnitude. Note how the processing time in the unoptimized case grows proportionally with the squared document size because the translation scheme of Figure 12 yields a join of table *accel* with itself for the two-step query Q_1. Actually, processing time increases even slightly worse than quadratic: The wide B-tree scans generate a significant number of false hits (Figure 5). With the staked out `descendant` window, however, we achieve a linear dependency on subtree size.

6.1.2 *IBM DB2 and a Stretched pre/post Plane.* The presence of a stretched *pre*/*post* plane removes two range predicates, on either the *pre* or *post* dimensions, from the SQL query generated for XPath query Q_1. To evaluate the two `descendant` steps in query Q_1, IBM DB2 now generates two B-tree index scans only. The otherwise necessary index intersection to evaluate the original two-dimensional `descendant` window is removed by IBM DB2's query optimizer. As expected, processing times go down again (Table VII). Note how column $t^{stretch}$ in Table VII reflects the fact that we can compute subtree sizes without error in the stretched *pre*/*post* plane: The processing time grows *perfectly* linear with the XML instance size.

6.1.3 *IBM DB2 and XPath Symmetries.* Trading `ancestor` for `descendant` steps by means of XPath symmetries makes queries like Q_2 amenable to the optimizations we have just discussed. In the *pre*/*post* plane, the small number of nodes in a context node's `ancestor` axis—equal or less to the document tree's height—are scattered all over the upper left of the plane. The resulting wide index scans and large number of false hits significantly affect performance. Table VIII indeed shows the expected performance gain of up to three orders of magnitude for the larger XML instances: t^{symm} reports on the processing time for the symmetric equivalent

$$Q_2^{symm} \quad = \quad \texttt{/descendant-or-self::person[descendant::age]}$$

of Q_2. Note that a cost model based on axis window areas would clearly identify Q_2^{symm} as superior to Q_2 (*cf.* Section 4.3).

Table VIII. XPath traversals before and after application of XPath symmetry rewrites. (Platform IBM DB2, Query Q_2)

Document size [MB]	# Result nodes	t [ms]	t^{symm} [ms]
11.0	631	4 430	202
55.0	3 163	85 183	875
111.0	6 409	340 759	1 876

Table IX. XPath traversals with and without empty region analysis. (Platform Monet, Query Q_2)

Document size [MB]	# Result nodes	t [ms]	t^{empty} [ms]
11.0	631	3 300	69
55.0	3 163	74 671	336
111.0	6 409	296 913	680

6.1.4 *Monet and Empty pre/post Plane Regions.* Remember how Section 4.4 used ordered processing of the context node sequence plus an analysis of empty regions in the *pre/post* plane to avoid the generation of duplicate result nodes. For reasons explained in Sections 5.1.4, these optimizations are not immediately expressible in plain SQL.

For Monet, however, the performance gain can be substantial. For example, the root node of a document lies in the `ancestor` axis of any node in that document except itself. In query Q_2, the initial step `descendant::age` produces a node sequence of 6 409 nodes for the 111 MB XML document. This means that for the subsequent `ancestor::person` step, a naive algorithm produces the root node 6 409 times leaving it up to duplicate removal to remove 6 408 copies of the root node. Similar remarks apply to all nodes in the upper tree levels.

Table IX illustrates the contribution of avoiding duplicates for query Q_2. The incremental `ancestor` evaluation strategy discussed in Sections 4.4, which immediately yields a duplicate free node set in document order, significantly reduces Monet's processing time.

6.2 Comparison of Back-Ends

Despite their conceptual simplicity and light implementation requirements, the techniques discussed in this article lead to performance figures which beat current implementation strategies for database-supported XPath engines. To provide points of reference we compared the XPath accelerator back-ends with two further XPath processors:

(1) We loaded the XML instances into a commercial native XML database system.[5] Prior to document loading, we provided the system with a DTD to enable the system to index its native XML storage.

(2) We also implemented an alternative relational encoding of XML documents, the *edge mapping*. We picked the edge mapping because
(a) it is, just like the XPath accelerator, a purely relational storage structure,

[5]The system's XML batch load client failed to process the two larger document instances of 55 MB and 111 MB so that we had to skip those in the subsequent measurements. Note that, in general, this specific XML DBMS has been engineered to efficiently process collections of small XML instances rather than large monolithic documents.

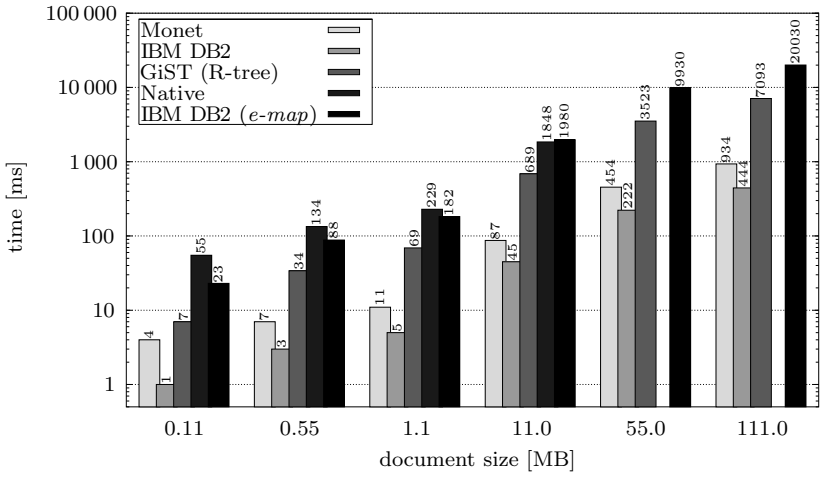


Fig. 19. XPath evaluation performance (Query Q_1, result size grows linearly from 12 to $\approx$ 12 000).

(b) the mapping has been shown to efficiently support the evaluation of XPath path expressions, and
(c) its internals have been described in sufficient detail in the publicly available literature, *e.g.*, by Florescu and Kossmann [1999].

In a nutshell, the edge mapping encodes XML document structure in a table $\underline{node \mid par \mid ord \mid kind \mid name}$ in which for each *node* (id) its parent *par* is listed. Hence, similar to the XPath accelerator, each edge in the document tree is represented by one tuple. The *ord* attribute keeps track of a node's sibling order among the nodes below a common parent. This is sufficient to restore the global document order although this is an expensive operation. Attributes *kind* and *name* indicate a node's kind and its tag (or attribute) name, respectively. Document content is maintained in separate content relations. The overall table layout thus resembles the XPath accelerator. Specifically, both schemes avoid to flood the database with table definitions, unlike mappings that introduce a separate table for each distinct element tag name encountered in an instance.
As recommended in the literature, we created indexes on the *node* and *par* attributes to speed up closure computation as well as an index on *name* to support name tests.

Figures 19, 20, and 21 report on the overall timing results for all XPath implementations against XMark instances of increasing size. All queries were run multiple times. The average timings reported here were measured when the database buffers (or, in the case of GiST, the file system cache) were hot. Note that elapsed times in all charts are given in milliseconds (ms) on logarithmically scaled y-axes.

On the relational database host (IBM DB2), the XPath accelerator turned out to be at least 20 times faster than the edge mapping, with a tendency in favour of the XPath accelerator with growing document sizes. The query evaluator based on the edge mapping clearly spends its time while stepping along the `descendant` axis

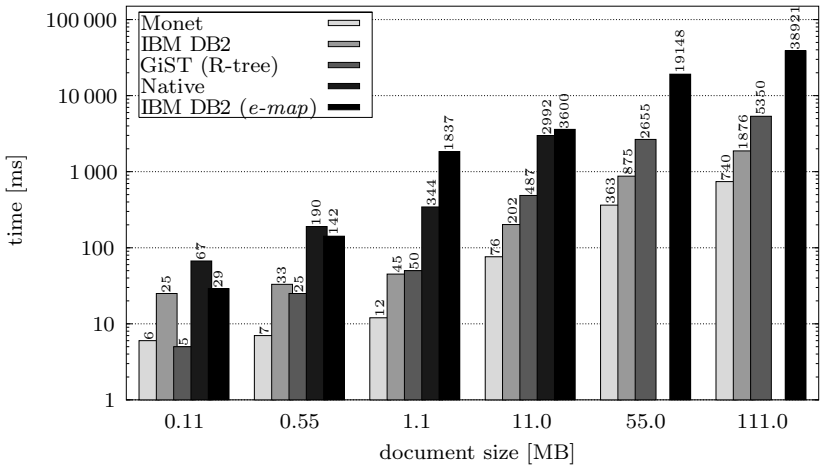


Fig. 20. XPath evaluation performance (Query Q_2, result size grows linearly from 8 to $\approx 6\,400$).

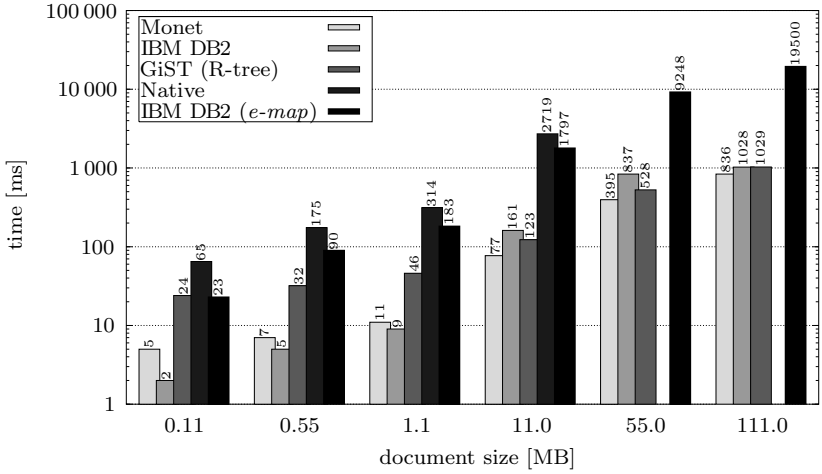


Fig. 21. XPath evaluation performance (Query Q_3, result size grows linearly from 31 to $\approx 31\,000$).

which inherently calls for a computation of the *par* closure rooted in the current context node. We applied name tests as early as possible to reduce closure size. For Q_1 (Figure 19, featuring two consecutive `descendant` steps) the XPath accelerator can gain a speedup of more than 40 for large XML instances. Recall that the XPath accelerator evaluates a step via two *index range scans* while the edge mapping needs multiple *individual index lookups* to perform the necessary *node–par* joins.

The evaluation of the `child` and `preceding-sibling` steps occurring in query Q_3 (Figure 21), however, is reasonably well supported by the edge mapping's *par* and *ord* attributes.

Since the computation of *pre/post* plane *window contents* lies at the very heart of the XPath accelerator, we expected the R-tree based variant to perform well. Although the implementation on top of GiST ran off a standard file system without any further buffering support, we indeed found it to clearly outperform the native XML DBMS as well as the relational edge mapping.

Finally, although the native XML DBMS was directed to index all document tree nodes including inner nodes, the disappointing performance figures we obtained for this system might stem from the fact that the indexes of this system have been tailored to efficiently access leaf node *contents*. The XPath accelerator, with its tree aware enhancements, can turn general purpose RDBMSs into XPath processors that perform up to two orders of magnitude faster than the native XML DBMS.

7. MORE RELATED WORK

This field of research is dominated by work that aims to support the XPath axes `child` and `descendant` [Cooper et al. 2001; Li and Moon 2001; Suciu and Milo 1999; Goldman and Widom 1997]. 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) embed XPath expressions in their syntax. Efficient XPath support will continue to be an important core building block in XML query processors.

[Cooper et al. 2001] 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 document order among a query's result nodes). In [Cooper et al. 2001], 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 with this scheme.

The T-index structure, proposed by Milo and Suciu in [Suciu and Milo 1999], 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 [Suciu and Milo 1999], an effective applicability test for a T-index is known for a restricted class of queries only).

A similar idea underlies the covering indexes of [Kaushik et al. 2002]. Like the T-index, covering indexes reduce the overall index size by collapsing nodes that are indistinguishable by a given set of path expressions. This is an approach that nicely complements the XPath accelerator idea: The XPath accelerator may be used to

index the *reduced* instead of the original document tree. A point in the *pre/post* plane then represents a representative for a whole equivalence class of nodes. It is, however, sufficient to operate with the representatives of these classes until the result node sequence is output.

In an earlier report on our work [Grust 2002] we compared the R-tree based XPath accelerator to the $\mathcal{EE}/\mathcal{EA}$ (*element-element/element-attribute*) join indexes of Li and Moon [2001]. Interestingly, this work (1) used a variant of the pre-order/postorder ranking to represent document structure, and (2) was also implemented on top of GiST via B-trees. Li and Moon [2001], however, used the traversal ranks to capture XML element *containment* (and attribute ownership) only. As a consequence, this work was restricted to provide support for the `child` and `descendant` axes. We have found the more generic XPath accelerator to be at least as fast as the $\mathcal{EE}/\mathcal{EA}$ based join algorithms, nevertheless.

There is further related work that is not directly targeted at the construction of index structures for XML. In [Zhang et al. 2001], the authors discuss relational support for *containment queries* of which our XPath axes window queries are instances. Especially the *multi-predicate merge join* (MPMG join) presented in [Zhang et al. 2001] would provide an almost perfect infrastructure for the XPath accelerator. MPMG join supports multiple equality and *inequality* tests (*cf.* the $window(\alpha, v)$ query windows). The *staircase join* [Grust and van Keulen 2003] applies a variant of the MPMG join idea to the XPath accelerator. The observed speed-up is about an order of magnitude with respect 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*) [Kriegel et al. 2000]. 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.

The SQL-based XPath evaluation (Section 5.1.2) bears some interesting resemblance with the treatment of the linear XPath fragment discussed by Gottlob et al. [2002]. The XPath accelerator provides an efficient database-supported implementation of the XPath axes, *i.e.*, function χ in Gottlob et al. [2002]. Furthermore, the translation function $\mathcal{S}_{\rightarrow}$ of Gottlob et al. and the evaluation scheme of Figure 12 coincide: Both schemes proceed top-down, with node tests transformed into separate intersections (*cf.*, `INSIDE`$(\cdot)$). The SQL correlation variable exchange effectively translates the existential semantics of XPath predicates into left self semijoins over table *accel* while Gottlob et al. equivalently use intersection. Additionally, as in $\mathcal{S}_{\rightarrow}$, we could extend the XPath to SQL mapping to translate XPath's `and`, `or`, and `not` boolean operators into `UNION`, `INTERSECT`, and `EXCEPT`, respectively.

The XPath accelerator is undemanding in the sense that schema-less XML data, *i.e.*, document instances without associated DTD or XML Schema types, may be processed. The internal representation, table *accel*, is completely uniform, regardless of the actual shape of the document tree. This uniformity

(1) overcomes the conflict between the relational two-level *table–attribute* view of data and XML's capability of arbitrary element nesting [Shanmugasundaram et al. 1999], and

(2) leads to query plans whose complexity is perfectly predictable: an XPath path expression of n steps leads to a n-fold self join regardless of the axes traversed.

While this predictability is obviously desirable, lack of *schema awareness* may also give away possible performance improvements. Condensed and pre-processed schema information, *e.g.*, in the form of DTD graphs as proposed by Shanmugasundaram et al. [1999] may be useful to further shrink query window sizes (Section 4.2.2) or discover empty *pre/post* plane regions at query compile time.

Others have, interestingly, closely investigated the symmetry properties of XPath [Olteanu et al. 2001] but with a completely different motivation than we had in Section 4.3: The XPath reverse axes, like `ancestor`, pose a problem for so-called *streaming XPath processors* [Altinel and Franklin 2000]. XPath engines of this type try to perform a single preorder traversal over the input document to evaluate a given path expression (*e.g.*, by receiving the events of a SAX parser). The big win is that only very limited memory space is necessary to perform the evaluation: Streaming XPath processors can, in principle, operate on XML documents of arbitrary size.

The evaluation of a reverse axis step in such a setup is problematic, because the XPath processor would need temporary space to remember *past* SAX events: A reverse axis selects documents nodes that are *before* the context node with respect to document order. The symmetry properties of XPath offer the possibility to get rid of reverse axes altogether and to restore the modest memory requirements.

Finally, let us briefly turn to *updates* for the XPath accelerator. After a new node v has been *inserted*, it is necessary—due to the order in which the preorder and postorder traversals visit document tree nodes—to renumber all nodes covered by $window(\texttt{following}, v)$ and $window(\texttt{ancestor}, v)$. Li and Moon [2001] suggest to assign non-consecutive node ranks, sufficiently spread out to accommodate for future insertions (recall that the absolute rank values are immaterial, *cf.* Section 4.2). Only recently, Cohen et al. [2002] studied *dynamic* node label assignments which could be suitable for preorder/postorder ranking. The resulting labeling scheme can systematically account for ordered insertion of new labels (whose length, nevertheless, may exceed those of already assigned labels) and is tunable if the system can anticipate insertions in specific subtrees.

Note that to *delete* a node, however, it is sufficient to remove its descriptor entry, because the properties of the *pre/post* plane are indifferent to node removal.

8. CONCLUSION AND OUTLOOK

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

(1) running on top of a relational back-end to leverage its stability, scalability, and performance,

(2) providing coverage for all nodes of an XML document tree (such that the index itself can serve as the only representation of the document inside the database),

(3) closely tracking the XPath semantics (especially with respect to adequate support for *all* XPath axes and document order), as well as

(4) rooting XPath traversals in arbitrary context nodes.

The latter requirement, specifically, did arise in the context of an ongoing project to construct an XQuery compiler: An XQuery expression like

```
for $v in e
    return $v/p
```

sequentially binds variable $\$v$ to the arbitrarily computed nodes of sequence e. The index thus needs to evaluate path p rooted in context nodes scattered over the whole document tree. For the XPath accelerator, any context node is as good as any other, in particular, the index has no bias towards the document root element like many related proposals.

While the above makes the XPath accelerator a promising target for XQuery compilation, another core feature of XQuery, *element construction*, needs to be addressed with care. An XQuery element constructor

```
element n { e1,e2 }
```

constructs a new node (with tag n) with left and right subtrees e_1 and e_2, respectively. To generate a valid *pre/post* encoding for the constructed tree means to renumber all nodes in the encoding of e_2. This is similar to the node insertion problem we have briefly discussed in Section 7, since the nodes of e_2 are in the `following` axis of the nodes in e_1. The compiler, however, has control about the order in which e_1 and e_2 are evaluated: since XQuery is a functional language without side effects, evaluation order of subexpressions does not matter semantically. Now, if e_1 is evaluated prior to e_2, the compiler can—in anticipation of the '`,`' sequence construction operator—encode the XML fragment e_2 with *pre/post* ranks that are immediately *following* those used for e_1. The encoding for e_2 will be consistent in itself (remember that the absolute *pre* and *post* values are insignificant), and the element construction above is merely a matter of *concatenating* the *pre/post* encodings for e_1 and e_2. This *threading* of *pre/post* ranks through the evaluation of an XQuery program is a technique currently under investigation by the authors.

More on the theoretical side—geared towards the development of an *optimizing* XPath or XQuery compiler—we believe that the XPath accelerator provides the necessary hooks to incorporate an effective *cost estimation* for XPath queries.

As discussed in Section 4, it is the *axis window area* which dominantly influences the step evaluation cost. Since the *pre/post* plane allows us to very accurately estimate window sizes given only a context node's descriptor, this could yield a cost model that is sensitive to the actual *location* of the context nodes and not only to the query itself.

The XPath symmetry rewrites explored in [Olteanu et al. 2001] could then be used to establish the space of equivalent XPath queries out of which a cost-based optimizer would pick candidates based on (a function of) *pre/post* plane window areas providing the cost measure.

It will be interesting to compare this approach to more intricate cost models for XML queries as presented in [Chen et al. 2001] and [Wu et al. 2002].

ACKNOWLEDGMENTS

The authors would like to thank the Monet people at CWI (Amsterdam, The Netherlands) for their support and most useful feedback. Maurice van Keulen has been with the University of Konstanz as a DAAD INNOVATEC funded research fellow. The comments of the anonymous reviewers were of great help in improving the presentation of the material.

REFERENCES

Altinel, M. and Franklin, M. J. 2000. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. of the 26th Int'l Conference on Very Large Databases (VLDB)*. Morgan Kaufmann Publishers, Cairo, Egypt, 53–64.

Berglund, A., Boag, S., Chamberlin, D., Fernandez, M. F., Kay, M., Robie, J., and Siméon, J. 2002. XML Path Language (XPath) 2.0. Tech. Rep. W3C Working Draft, Version 2.0, World Wide Web Consortium. Aug. `http://www.w3.org/TR/xpath20/`.

Boag, S., Chamberlin, D., Fernandez, M., Florescu, D., Robie, J., and Siméon, J. 2002. XQuery 1.0: An XML Query Language. Tech. Rep. W3C Working Draft, World Wide Web Consortium. Aug. `http://www.w3.org/TR/xquery`.

Böhm, C., Berchtold, S., Kriegel, H.-P., and Michel, U. 2000. Multidimensional Index Structures in Relational Databases. *Journal of Intelligent Information Systems (JIIS) 15,* 1, 51–70.

Boncz, P. A. 2002. Monet: A Next-Generation DBMS Kernel for Query-Intensive Applications. Ph.D. thesis, University of Amsterdam, The Netherlands.

Boncz, P. A. and Kersten, M. L. 1999. MIL Primitives for Querying a Fragmented World. *The VLDB Journal 8,* 2, 101–119.

Chen, Z., Jagadish, H., Korn, F., Koudas, N., Muthukrishnan, S., Ng, R., and Srivastava, D. 2001. Counting Twig Matches in a Tree. In *Proc. of the 17th Int'l Conference on Data Engineering (ICDE)*. IEEE Computer Society, Heidelberg, Geramny, 595–604.

Cohen, E., Kaplan, H., and Milo, T. 2002. Labeling Dynamic XML Trees. In *Proc. of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. ACM Press, Madison, Wisconsin, 271–121.

Cooper, B. F., Sample, N., Franklin, M. J., Hjaltason, G. R., and Shadmon, M. 2001. A Fast Index for Semistructured Data. In *Proc. of the 27th Int'l Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann Publishers, Rome, Italy, 341–360.

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

Fernandez, M., Marsh, J., and Nagy, M. 2002. XQuery 1.0 and XPath 2.0 Data Model. Tech. Rep. W3C Working Draft, World Wide Web Consortium. Aug. `http://www.w3.org/TR/query-datamodel`.

Florescu, D. and Kossmann, D. 1999. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Tech. Rep. 3680, INRIA, Rocquencourt, France. May.

Goldman, R. and Widom, J. 1997. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. of the 23rd Int'l Conference on Very Large Databases (VLDB)*. Morgan Kaufmann Publishers, Athens, Greece, 436–445.

Gottlob, G., Koch, C., and Pichler, R. 2002. Efficient Algorithms for Processing XPath Queries. In *Proc. of the 28th Int'l Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann Publishers, Hong Kong, China, 95–106.

Grust, T. 2002. Accelerating XPath Location Steps. In *Proc. of the 21st Int'l ACM SIGMOD Conference on Management of Data*. ACM Press, Madison, Wisconsin, USA, 109–120.

Grust, T. and van Keulen, M. 2003. Tree Awareness for Relational Database Kernels: Staircase Join. In *Intelligent Search on XML*, H. Blanken, H.-J. Schek, and G. Weikum, Eds. Number 2818 in Lecture Notes in Computer Science. Springer Verlag, Heidelberg, Germany.

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

Hellerstein, J. M., Naughton, J. F., and Pfeffer, A. 1995. Generalized Search Trees for Database Systems. In *Proc. of the 21st Int'l Conference on Very Large Databases (VLDB)*. Morgan Kaufmann Publishers, Zurich, Switzerland, 562–573.

Kamel, I. and Faloutsos, C. 1993. On Packing R-Trees. In *Proc. of the 2nd Int'l Conference on Information and Knowledge Management (CIKM)*. ACM Press, Washington DC, USA, 490–499.

Kaushik, R., Bohannon, P., Naughton, J. F., and Korth, H. K. 2002. Covering Indexes for Branching Path Queries. In *Proc. of the 21st Int'l ACM SIGMOD Conference on Management of Data*. ACM Press, Madison, Wisconsin, USA, 133–144.

Kriegel, H.-P., Pötke, M., and Seidl, T. 2000. Managing Intervals Efficiently in Object-Relational Databases. In *Proc. of the 26th Int'l Conference on Very Large Databases (VLDB)*. Morgan Kaufmann Publishers, Cairo, Egypt, 407–418.

Li, Q. and Moon, B. 2001. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. of the 27th Int'l Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann Publishers, Rome, Italy, 361–370.

Olteanu, D., Meuss, H., Furche, T., and Bry, F. 2001. Symmetry in XPath. Tech. Rep. PMS-FB-2001-16, Institute of Computer Science, University of Munich, Germany.

Roussopoulos, N. and Leifker, D. 1985. Direct Spatial Search on Pictorial Databases Using Packed R-Trees. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*. ACM Press, Austin, Texas, 17–31.

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

Schmidt, A., Waas, F., Kersten, M., Carey, M. J., Manolescu, I., and Busse, R. 2002. XMark: A Benchmark for XML Data Management. In *Proc. of the 28th Int'l Conference on Very Large Databases (VLDB)*. Morgan Kaufmann Publishers, Honk Kong, China, 974–985.

Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., DeWitt, D., and Naughton, J. 1999. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. of the 25th Int'l Conference on Very Large Data Bases (VLDB)*. Morgan Kaufmann Publishers, Edinburgh, Scotland, 302–314.

Suciu, D. and Milo, T. 1999. 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). Springer Verlag, Jerusalem, Israel, 277–295.

Wu, Y., Patel, J. M., and Jagadish, H. 2002. Estimating Answer Sizes for XML Queries. In *Proc. of the 8th Int'l Conference on Extending Database Technology (EDBT)*. Springer Verlag, Prague, Czech Republic, 590–608.

Zhang, C., Naughton, J., DeWitt, D., Luo, Q., and Lohman, G. 2001. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*. ACM Press, Santa Barbara, California, 425–436.

Received M Y; revised M Y; accepted M Y

THIS DOCUMENT IS THE ONLINE-ONLY APPENDIX TO:

Accelerating XPath Evaluation in Any RDBMS

TORSTEN GRUST
University of Konstanz
and
MAURICE VAN KEULEN
University of Twente
and
JENS TEUBNER
University of Konstanz

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY, Pages 1–40.

A. XML DOCUMENT LOADING

Recall from Section 3, that *loading* an XML document instance into the database essentially means to map its nodes into a 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 XML parsing framework, like SAX [SAX], 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 a small number of SAX-specific callback procedures:[6]

(1) The SAX parser calls procedure *startElement*$(t, [a_1, a_2, \dots])$ whenever it encounters an XML start tag <t a_1=e_1 a_2=e_2 ...> in the serialized XML input. The parser supplies tag name t as well as the list of attributes $[a_1, a_2, \dots]$ as parameters.

(2) Procedure *endElement*(t) is invoked whenever the corresponding end tag </t> is encountered.

[6]For the sake of clarity, note that we slightly simplify the actual implementation as well as the description of the SAX API. The real loader code, however, is only marginally different.

startElement$(t, [a_1, a_2, \dots])$

```
v ← ⟨pre, ␣, pre(S.top()), elem, t⟩;
S.push(v);
pre ← pre + 1;
FOR v′ IN [a1, a2, ...] DO
  leaf(attr, v′);
```

leaf(k, n)

```
v ← ⟨pre, post, pre(S.top()), k, n⟩;
INSERT INTO accel VALUES v;
pre ← pre + 1;
post ← post + 1;
```

endElement(t)

```
v ← S.pop();
post(v) ← post;
INSERT INTO accel VALUES v;
post ← post + 1;
```

comment$(\cdots)$

```
leaf(comment, ␣);
```

Fig. 22. A family of SAX callback procedures (excerpt) which collaboratively load an XML document into node descriptor table *accel*. Auxiliary procedure *leaf*(k, n) inserts a document leaf node of kind k and associated name n. The omitted callbacks *characters*$(\cdots)$ and *processingInstruction*$(\cdots)$ are implemented analogously to *comment*$(\cdots)$.

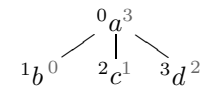
(3) The callbacks *characters*$(\cdots)$, *comment*$(\cdots)$, *processingInstruction*$(\cdots)$ are invoked whenever text content, XML comments, or processing instructions are seen in the input, respectively.

We display an excerpt of the callback procedures in Figure 22. Before loading starts, the rank variables *pre* and *post* are initialized to 0, and stack S contains the single dummy node descriptor $\langle ␣, ␣, ␣, ␣, ␣ \rangle$ (all entries undefined). To keep track of elements whose start tag we have already seen but whose end tag is still to come, we maintain a stack S of yet incomplete node descriptors. (The stack operations *push*, *pop*, and *top* should be self-explaining.) The procedures thus invariably find the current parent node on the stack top. 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 *accel* which constitutes the relational representation of the index. The size of S is obviously bounded by the input instance's height. No additional temporary memory space is needed.

Note how callback *startElement*$(t, [a_1, a_2, \dots])$ calls *leaf*$(attr, a_i)$ $(i = 1, \dots)$ to fit the attributes a_i immediately after element t into document order [Berglund et al. 2002, Section 2.3.1]. The document loader will thus encode an XML fragment of the form

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

as the document tree



.

The resulting tree shape coincides with the XPath semantics as well as the associated XPath data model semantics [Berglund et al. 2002; Fernandez et al. 2002]: attribute nodes

```
serialize(N)
FOR v IN N DO
    WHILE post(S.top()) < post(v) DO
        PRINT('</', name(S.top()), '>');
        S.pop();
    IF kind(v) = elem THEN
        PRINT('<', name(v), '>');
        S.push(v);
    ELSE
        {process other node kinds here}
WHILE not S.empty() DO
    PRINT('</', name(S.top()), '>');
    S.pop();
```

Fig. 23. XML serialization (only element handling shown here). The node descriptor table N is assumed to be sorted on its *pre* column.

(1) appear immediately after the owning element in document order, and,
(2) are recorded with the owning element as the parent node (*e.g.*, `c/parent::node()` will yield a).

It is worth noting that the XPath accelerator can be populated and then queried without any *a priori* knowledge of the DTD: the input document's structure is discovered during parsing.

B. XML SERIALIZATION

The "inverse" operation to document loading is XML *serialization*. Serialization is a prerequisite if the XPath accelerator tables—*accel* plus the content relations—indeed are the only document representation (*i.e.*, we discard the original XML input file after loading).

Remember that the preorder rank stored in *accel* determines the order of start tags in the associated XML document. We thus scan the nodes v in table *accel* in ascending order of *pre* values, emitting start tags as we go (Figure 23). We then push v onto a stack S to remember that we later need to print the corresponding closing tag. The *post* column, analogously, reflects the order of the document's closing tags. We therefore take care to pop and emit all closing tags of nodes v' with $post(v') < post(v)$ before we actually process v. The treatment of attributes and other XML node kinds should be obvious. Note that, again, the size of the node stack S is bounded by the instance's height.

To serialize the complete document, invoke *serialize*(*accel*). To serialize the document rooted in some element node v—*e.g.*, to visualize a query's result containing v—invoke *serialize* on the set of nodes inside *window*(`descendant-or-self::node()`, v). In both cases, a single *pre*-sorted scan of table *accel* is sufficient to perform the serialization.