INformation Systems

Loop-lifted staircase join: from XPath to XQuery

P.A. Boncz, T. Grust, M. van Keulen, S. Manegold,
J. Rittinger, J. Teubner

REPORT INS-E0510 MAY 2005

ISSN 1386-3681

Loop-lifted staircase join: from XPath to XQuery

ABSTRACT
Various techniques have been proposed for efficient evaluation of XPath expressions, where the XPath location steps are rooted in a *single* sequence of context nodes. Among these techniques, the *staircase join* allows to evaluate XPath location steps along arbitrary axes in at most one scan over the XML document, exploiting the *XPath accelerator* encoding (aka. *pre/post* encoding).

In XQuery, however, embedded XPath sub-expressions occur in arbitrarily nested `for`-loops. Thus, they are rooted in *multiple* sequences of context nodes (one per iteration). Consequently, the previously proposed algorithms need to be applied repeatedly, requiring multiple scans over the XML document encoding.

In this work, we present *loop-lifted staircase join*, an extension of the staircase join that allows to efficiently evaluate XPath sub-expressions in arbitrarily nested XQuery iteration scopes with only a single scan over the document. We implemented the loop-lifted staircase join in *MonetDB/XQuery*, that uses the XQuery-to-Relational Algebra compiler *Pathfinder* on top of the extensible RDBMS *MonetDB*. Performance results indicate that the proposed technique allows to build a system that is capable of efficiently evaluating XQuery queries including embedded XPath expressions, obtaining interactive query execution times for all XMark queries even on multi-gigabyte XML documents.

1998 ACM Computing Classification System: H.2.4, H.2.3, H.2.2, E.1
Keywords and Phrases: XML; XPath; XQuery; XPath Accelerator; relational XQuery evaluation; loop-lifted staircase join; RDBMS; MonetDB
Note: Work carried out under projects MultimediaN N3 ``Ambient Multimedia Databases'' and Bricks IS2 ``Petabyte Data Mining''.

Loop-lifted Staircase Join: from XPath to XQuery

Peter Boncz[1] Torsten Grust[2] Maurice van Keulen[3]
Stefan Manegold[1] Jan Rittinger[4] Jens Teubner[4]

[1]CWI Amsterdam, The Netherlands
{boncz ,manegold }@cwi.nl

[2]Technical University of Munich, Germany
grust@in.tum.de

[3]University of Twente, The Netherlands
keulen@cs.utwente.nl

[4]University of Konstanz, Germany
{rittinge ,teubner }@inf.uni- konstanz.de

ABSTRACT

Various techniques have been proposed for efficient evaluation of XPath expressions, where the XPath location steps are rooted in a *single* sequence of context nodes. Among these techniques, the *staircase join* allows to evaluate XPath location steps along arbitrary axes in at most one scan over the XML document, exploiting the *XPath accelerator* encoding (aka. *pre/post* encoding).

In XQuery, however, embedded XPath sub-expressions occur in arbitrarily nested `for` -loops. Thus, they are rooted in *multiple* sequences of context nodes (one per iteration). Consequently, the previously proposed algorithms need to be applied repeatedly, requiring multiple scans over the XML document encoding.

In this work, we present *loop-lifted staircase join*, an extension of the staircase join that allows to efficiently evaluate XPath sub-expressions in arbitrarily nested XQuery iteration scopes with only a single scan over the document. We implemented the loop-lifted staircase join in *MonetDB/XQuery*, that uses the XQuery-to-Relational Algebra compiler *Pathfinder* on top of the extensible RDBMS *MonetDB*. Performance results indicate that the proposed technique allows to build a system that is capable of efficiently evaluating XQuery queries including embedded XPath expressions, obtaining interactive query execution times for all XMark queries even on multi-gigabyte XML documents.

1998 ACM Computing Classification System: H.2.4, H.2.3, H.2.2, E.1
Keywords and Phrases: XML, XPath, XQuery, XPath Accelerator, relational XQuery evaluation, loop-lifted staircase join, RDBMS, MonetDB
Note: Work carried out under projects MultimediaN N3 “Ambient Multimedia Databases” and Bricks IS2 “Petabyte Data Mining”.

1. Introduction

Relational XML databases aim to re-use mature relational DBMS technology to provide scalability and efficiency. Our Pathfinder compiler translates XQuery queries into relational algebra, using an RDBMS as execution back-end [9]. Our relational approach also encompasses relational XML document validation [8], and relational XML document updates [5]. We implemented these techniques in the MonetDB/XQuery system, where the MonetDB main-memory DBMS [2] serves as database back-end.[1]

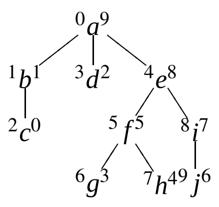
Here, we report on the new *loop-lifted staircase join* algorithm used inside MonetDB/XQuery for the efficient execution of XPath expressions that occur embedded in XQuery queries. This work builds on the staircase join algorithm [10], which allows a single (non-XQuery) XPath expression to be executed efficiently in a sequential pass on the relational table that stores the encoded XML document.

If staircase join is applied directly in XQuery, *embedded* path expressions (*e.g.*, inside `for-` loops) lead to repeated scans of the document table, severely impacting performance. The loop-lifted staircase join exploits tree properties of the document table to reduce common work between the “repeated” execution of XPath

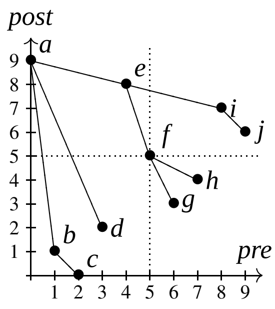
[1]MonetDB/XQuery is open source software, see `http://monetdb.cwi.nl/` and [4].

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

(a) XML document.



(b) Tree skeleton with *pre*/*post* ranks.



(c) Resulting *pre*/*post* plane.

post = pre + size - level

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

(d) Actual relational encoding.

Figure 1: Relational Storage With *pre/size/level* To Support Efficient XPath Axis Traversal

expressions, evaluating those using a single scan over the document table. This strongly improves response time over repeated execution of simple staircase join.

Outline In Section 2, we first recap the *pre/post* relational document encoding, and the basic ideas behind the staircase join algorithm. In Section 3, we then explain the XQuery challenge of embedded XPath execution and provide a detailed description of the new loop-lifted staircase join algorithm. In Section 4, we outline the MonetDB/XQuery system in which both algorithms were implemented, and evaluate its performance on the XMark benchmark, scaling from 11 MB to 11 GB documents. Here we show that loop-lifted staircase join accelerates performance by a factor of 10, making MonetDB/XQuery one of the fastest and most scalable XQuery processors currently available. Finally, in Section 5 we discuss related work, before describing our conclusions and future work in Section 6.

2. Staircase Join Re-Cap

The four parts of Figure 1 shortly explain the *pre/post* and *pre/size/level* relational XML document encodings. Part *(i)* shows the example document. In part *(ii)*, nodes of the XML tree are assigned *pre* and *post* ranks, which count how many tags have been opened and closed, respectively, as seen when parsing the document sequentially. In part *(iii)*, which plots all document nodes in a *pre/post* plane, we clearly recognize the tilted XML tree. It also shows that for each node (in this case the context node is `f`), the quadrants of the *pre/post* plane correspond to the major XPath axes: `descendant`, `following`, `ancestor` and `preceding`. As such, this representation allows to express all XPath axes as simple comparisons on the *pre* and *post* columns, which can be evaluated efficiently in an SQL-speaking RDBMS [7]. Finally, part *(iv)* of Figure 1 shows the actual relational XML representation used in MonetDB/XQuery, which instead of the *post* column stores two columns holding the subtree *size* and a tree *level*. This *pre/size/level* encoding is equivalent to *pre/post* since *post = pre + size - level*.

Making the query optimizer of an RDBMS more "tree-aware" allows it to improve its query plans concerning XPath evaluation. However, incorporating knowledge of the *pre/post* plane should, ideally, not clutter the entire query optimizer with XML-specific adaptations. In [10], Grust et al. proposed a special join operator, the *staircase join*, that exploits and encapsulates all "tree knowledge" present in the *pre/post* plane. It behaves to the query optimizer in many ways as an ordinary join, *e.g.*, by admitting selection pushdown. In the remainder of this section, we will briefly recap the basic principles and features of the staircase join and the tree-aware optimizations it encapsulates.

The staircase join requires at most a single scan over the document and the context set to produce a result that complies to the XPath semantics: duplicate free and sorted in document order. To achieve this, the staircase join applies three techniques that distinguish it from similar approaches: *(i) pruning* to reduce the set of context nodes to the minimal set for producing the result nodes, *(ii) partitioning* to consider each document node

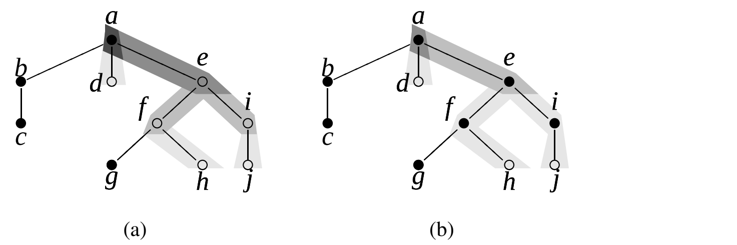


Figure 2: (a) Intersection and inclusion of the `ancestor-or-self` paths of a context node sequence. (b) The pruned context node sequence covers the same `ancestor-or-self` region and produces less duplicates (3 rather than 11).

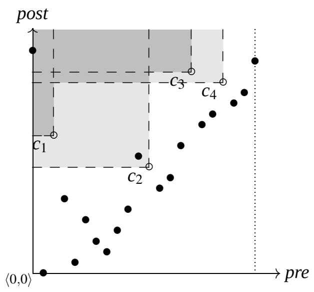


Figure 3: Overlapping regions in *pre/post* plane (encoding of a larger XML document; context nodes c_i).

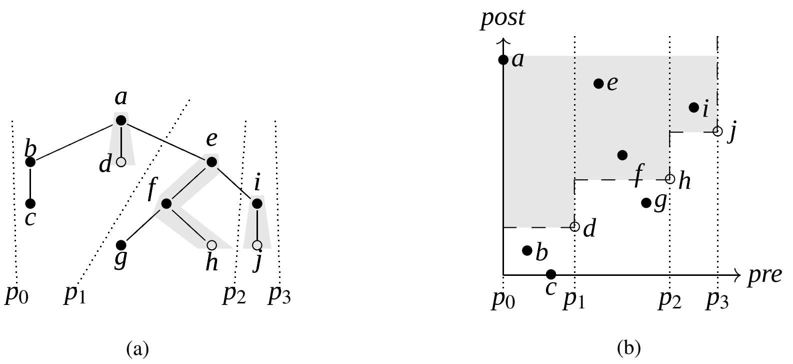


Figure 4: The partitions $[p_0,p_1)$, $[p_1,p_2)$, $[p_2,p_3)$ of the `ancestor` staircase separate the `ancestor-or-self` paths in the document tree.

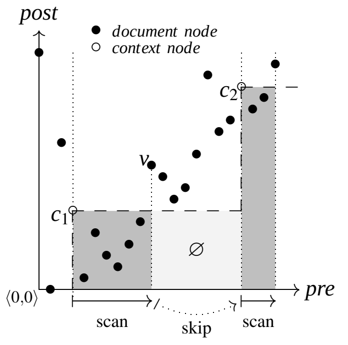


Figure 5: Skipping technique used for accelerating the `descendant` axis.

only once, and *(iii) skipping* to avoid considering nodes for which the tree and XPath axis properties tell us beforehand that they can never be in the result.

Pruning The evaluation of an axis step for a certain context node boils down to selecting all document nodes in the corresponding region. In XPath, however, an axis step is generally evaluated on an *entire sequence* of context nodes. This leads to duplication of work if the *pre/post* plane regions associated with the step are independently evaluated for each context node. Figure 2 (a) depicts the situation if we are about to evaluate an `ancestor-or-self` step for context sequence (d,e,f,h,i,j). The darker the path's shade, the more often are its nodes produced in the resulting node sequence—which ultimately leads to the need for duplicate removal operator `unique` in the query plan to meet the XPath semantics. Obviously, we could remove nodes e,f,i—which are located along a path from some other context node up to the root—from the context node sequence without any effect on the final result (a,d,e,f,h,i,j) (Figure 2 (b)). Such opportunities for the simplification of the context node sequence arise for all axes.

Figure 3 depicts the situation in the *pre/post* plane as this is the RDBMS's view of the problem. Result nodes can be found in the shaded areas. In general, regions determined by context nodes can *include* one another or *partially overlap* (dark areas). Nodes in these areas generate duplicates.

The removal of nodes e,f,i earlier is a case of *inclusion*. Inclusion can be dealt with by removing the covered nodes (c_1,c_3) from the context. The process of identifying the context nodes at the cover's boundary is referred to as *pruning* and is easily implemented for a *pre/post* encoded context node sequence [10].

Partitioning While pruning leads to a significant reduction of duplicate work, Figure 2 (b) exemplifies that duplicates still remain due to intersecting `ancestor-or-self` paths originating in different context nodes. A much better approach results if we *separate* the paths in the document tree and evaluate the axis step for each context node in its own partition (Figure 4 (a)).

Such a separation of the document tree is easily derived from the staircase induced by the context node sequence in the *pre/post* plane (Figure 4 (b)): each of the partitions $[p_0, p_1)$, $[p_1, p_2)$, and $[p_2, p_3)$ define a region of the plane containing all nodes needed to compute the axis step result for context nodes d, h, and j, respectively. Note that pruning reduces the number of these partitions.

The basic approach to evaluating a staircase join between a document and a context node sequence thus is to sequentially scan the *pre/post* plane once from left to right selecting those nodes in the current partition that lie within the boundary established by the context node sequence. Since the XPath accelerator maintains the nodes of the *pre/post* plane in the *pre*-sorted table doc, the result nodes are encountered and written in document order.

This basic algorithm has several important characteristics:
(1) it scans the doc and context tables sequentially,
(2) it scans both tables only once for an entire context sequence,
(3) it never delivers duplicate nodes, and
(4) result nodes are produced in document order, so no post-processing is needed to comply with XPath semantics.

Skipping Sometimes we may infer from the properties of the *pre/post* encoding and the particular XPath step at hand, that certain regions cannot contain results. Figure 5 illustrates this for the XPath axis step (c_1, c_2)/descendant . The staircase join is evaluated by scanning the *pre/post* plane from left to right starting from context node c_1. During the scan of c_1's partition, v is the first node encountered outside the descendant boundary and thus not part of the result.

Note that no node beyond v in the current partition contributes to the result (the light grey area is empty). This is, again, a consequence of the fact that we scan the encoding of a tree data structure: node v is following c_1 in document order so that both cannot have common descendants.

Staircase join uses this observation to terminate the scan of the current partition early which effectively means that the portion of the scan between $pre(v)$ and the successive context node $pre(c_2)$ is *skipped*. As a result, staircase join actually scans (*i.e.*, accesses) only those portions of the document relation that contain results.

The effectiveness of skipping is high. For each node in the context, we either (1) hit a node to be copied into the result, or (2) encounter a node of type v which leads to a skip. To produce the result, we thus never touch more than $|\text{result}| + |\text{context}|$ nodes in the *pre/post* plane while the basic algorithm would scan along the entire plane starting from the context node with minimum preorder rank.

Other XPath axes lead to similar skipping opportunities. If in the ancestor axis inside the partition of context node c, we encounter a node v outside the ancestor boundary, we know that v and all its descendants are in the *preceding* axis of c and thus can be skipped. In the child axis, we know that if a context node c has children (*i.e.*, $size(c) > 0$), node $v_1 = c + 1$ is the first child, and the other children can be found iteratively by skipping over all their descendants: $v_{i+1} = v_i + size(v_i) + 1$. The latter example illustrates that separating *post* in *size* and *level* sometimes provides extra skipping opportunities; which is why MonetDB/XQuery uses the *pre/size/level* encoding.

3. Embedded XPath Evaluation

In XPath, a location step e/α along some axis α is taken from a *single* sequence e of arbitrary context nodes and staircase join as well as other structural join algorithms [1, 6] have been devised to efficiently evaluate such steps. XQuery has been designed as a host language that embeds XPath. Due to XQuery's orthogonality and compositionality, XPath expressions thus potentially occur in the scope of surrounding for -iterations[2], possibly at deep nesting levels. Query Q_1 illustrates a possible XQuery expression embedding an XPath child location step:

```
for $v in (x1, x2, ..., xn) return
  e($v)/child::t                    ,
```
(Q_1)

[2]Note that, in normalized XQuery Core, for is the only iteration primitive.

Here, $e(\$v)$ denotes an XQuery expression in which variable $\$v$ occurs free. If this query is correctly typed, each binding of variable $\$v$ will yield a separate context node sequence $e(\$v)$. Q_1 thus will, in effect, initiate n location along the `child` axis, each rooted in a different context node sequence. Any XQuery implementation will face the challenge to evaluate XPath traversals embedded in possibly deeply nested iterations.

Pathfinder and MonetDB/XQuery internally operate in a purely relational fashion. At runtime, the relational encoding of the n context node sequences (resulting from the n evaluations of $e(\$v)$) will take the form depicted here [9]: in each of the n iterations, encoded in column iter, $e(\$v)$ evaluates to a sequence of node preorder ranks $(\gamma_{i,1}, \ldots, \gamma_{i,s_i})$ of length s_i $(1 \leqslant i \leqslant n)$. These preorder ranks are maintained in column pre. We might have $s_i = 0$ for some i: in this case, no tuple with iter value i will occur. Note that, at this point, the system does *not* need to maintain the actual sequence order of the $\gamma_{i,j}$: the semantics of an XPath location step is unaffected by the context node sequence order and staircase join will process the context nodes in document order anyway. This relational encoding of the result of evaluating $e(\$v)$ is indifferent to the actual `for`-iteration nesting depth [9].

iter	pre
1	$\gamma_{1,1}$
1	$\gamma_{1,2}$
$\vdots$	$\vdots$
1	γ_{1,s_1}
$\vdots$	$\vdots$
n	$\gamma_{n,1}$
$\vdots$	$\vdots$
n	γ_{n,s_n}

Basic staircase join can evaluate an XPath location step for a single context node sequence (*i.e.*, for one of the above sequences $(\gamma_{i,1}, \ldots, \gamma_{i,s_i})$) during a single scan over a pre|size|level encoded XML document. Nevertheless, the evaluation of the loop body of query (Q_1) requires n invocations of staircase join (one for each iter group) and thus as many sequential scans over the document encoding table. This surely seems wasteful.

3.1 Loop-lifted Staircase Join

To save the relational back-end from performing this significant amount of work repeatedly (for huge XML input documents, the encoding tables will be huge as well), we propose *loop-lifted staircase join*. Loop-lifted staircase join inherits many beneficial features of staircase join, like the guarantee to produce duplicate-free node sequences in document order as required by the XPath semantics, but the new variant is a considerably better fit for the XQuery compilation approach pursued here.

With the original staircase join, partitioning ensures that only a single context node is *active* (and hence "producing" result nodes) during the scan over the document. If the result regions of two context nodes overlap, each result node in the overlapping region must appear only once in the result according to the XPath semantics.

With embedded XPath, however, the same node(s) might occur more than once in the result, each occurrence belonging to a different iteration. Obviously, if the context sequences of two iterations share context nodes, the respective result node must occur in the result sequence with both iterations. But even if two context sequences do not have common context nodes, they might have common result nodes. Assume we have a document instance encoded as depicted in Figure 3, and we want to evaluate an `ancestor-or-self` step for iteration 1 with context node c_2 and for iteration 2 with context node c_4. In this case, document nodes in those parts of the result regions that do not overlap (light shading) must occur only once in the result (assigned to the respective iteration). Document nodes in the overlapping parts of the result region (dark shading) need to occur twice in the result, once for each iteration. The naive application of the original staircase join by ignoring the iteration and simply treating the set of context sequences as a single one would hence create a wrong result, as partitioning would avoid all duplicates in the result.

To overcome this, we create the loop-lifted staircase join, a variant of the original staircase join that is "iteration-aware" and allows multiple context nodes to be active at a time. Basically, the algorithm keeps a stack of active context nodes accompanied with the iterations that they occur in. While scanning/skipping over the document, the partition boundaries determine when to push the next context node on the stack and when to pop a finished context node from the stack. When a result node is encountered, the stack is analyzed to determine the active nodes and their iteration in order to produce the correct result.

In the following, we will explain our loop-lifted staircase join in detail, using the `child` step as example. Similar adaptations apply to the remaining XPath axes, although they might differ in details.

```
ll_scj_child (doc : TABLE(pre, size), ctx : TABLE(iter, pre), cand : TABLE(pre))
 BEGIN
    ASSERT (doc.pre IS DENSE AND ASCENDING);              // for positional lookup
    ASSERT (ctx IS SORTED ON (pre, iter));                // document order
    ASSERT (cand IS SORTED ON (pre));                     // document order
    result ← NEW TABLE(iter, pre);                        // the result
    active ← NEW STACK(eos, nxtChld, nxtCand, fstIter, lstIter); // stack of active context nodes
    nxtCtx ← 0;     lstCtx ← SIZE(ctx);                   // first & last context node
    nxtCand ← 0;    lstCand ← SIZE(cand);                 // first & last candidate node
    nxtCtx, nxtCand ← match_ctx_cand(nxtCtx, nxtCand);// skip non-matching contexts and candidates
⓪   WHILE (nxtCtx ≤ lstCtx AND nxtCand ≤ lstCand) DO// iterate over all context nodes
      IF (active IS EMPTY) THEN                           // stack is empty
①       nxtCtx, lstCand ← push_ctx(nxtCtx, nxtCand);     // push current context on stack
      ELIF (TOP(active).eos ≥ ctx[nxtCtx].pre) THEN       // next context is descendant of current context
②       inner_loop_child(ctx[nxtCtx].pre);               // process children of current context until next context
③       nxtCtx, lstCand ← push_ctx(nxtCtx, nxtCand);     // push next context on stack
      ELSE                                                // next context is not descendant of current context
④       inner_loop_child(TOP(active).eos);               // process all children of current context
⑤       POP(active);                                     // pop finished context from stack
    WHILE (active IS NOT EMPTY) DO                        // finish all remaining active scopes
⑥     inner_loop_child(TOP(active).eos);                  // process all remaining children of current context
⑦     POP(active);                                        // pop finished context from stack
    RETURN result;                                        // return result
 END

push_ctx (nxtCtx, nxtCand)
 BEGIN
    curPre ← ctx[nxtCtx].pre;                             // preorder rank of current context
    eos ← curPre + doc[curPre].size;                      // end of current scope
    nxtChld ← curPre + 1;                                 // first child of current context
    fstIter ← nxtCtx;                                     // first iter of current context
    WHILE (ctx[nxtCtx].pre = curPre) DO                   // iterate of all iters of current context
      nxtCtx ← nxtCtx + 1;                                // next iter of current context
    lstIter ← nxtCtx − 1;                                 // last iter of current context
    PUSH ⟨eos, nxtChld, nxtCand, fstIter, lstIter⟩ ON active;  // push current context on stack
    nxtCtx, nxtCand ← match_ctx_cand(nxtCtx, nxtCand);// skip non-matching contexts and candidates
    RETURN nxtCtx, lstCand;                               // return next context and candidate
 END

inner_loop_child (eos)
 BEGIN
    nxtChld ← TOP(active).nxtChld;                        // next child of current context
    fstIter ← TOP(active).fstIter;                        // first iter of current context
    lstIter ← TOP(active).lstIter;                        // last iter of current context
    WHILE (nxtChld ≤ eos) DO                              // iterate of all children in current scope
      FOR iter FROM fstIter TO lstIter DO                 // iterate over all iters of current context
        APPEND ⟨ctx[iter].iter, nxtChld⟩ TO result;       // append (iter,pre) to result
    IF (nxtChld ≤ TOP(active).eos) DO                     // current context not yet finished
      TOP(active).nxtChld ← nxtChld;                      // recall where to proceed
    RETURN;                                               // return
 END
```

Figure 6: Loop-lifted staircase join: child axis (underlined parts are only used in Section 3.3).

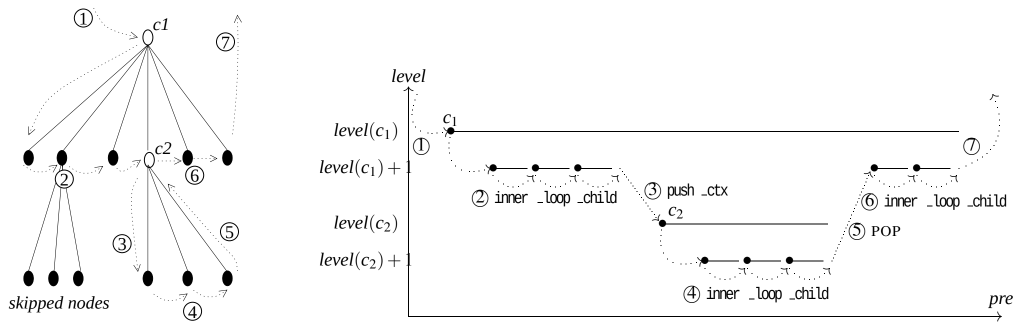


Figure 7: Example child traversal (left), illustrated with calls in the algorithm of Figure 6 (right).

3.2 Pure Loop-lifted Staircase Join

Our loop-lifted staircase join performs a *single* sequential forward scan over the context sequences and requires at most one sequential traversal of the document encoding, regardless of the number of iterations. To achieve this, the algorithm of Figure 6[3] expects the document encoding *doc* to be sorted on the preorder ranks (trivial) and the set of context node sequences *ctx* to be sorted on $(\mathsf{pre}, \mathsf{iter})$, *i.e.*, the context nodes appear in document order and for each context node all related iterations appear clustered and in order. In contrast to the original staircase join algorithm in which a single context node was "active" at a time, in the loop-lifted variant up to n iterations may be active. In the case of the child axis, an iter|pre tuple $\langle i, c\rangle$ in the context sequence defines iteration i to be active if the current document node's preorder rank lies within the interval $(pre(c), pre(c) + size(c)]$.[4] The algorithm maintains the currently active iterations on the *active* stack. While scanning over the context nodes in *ctx* ⓪, three situations can occur (see the algorithm in Figure 6, and the processing example in Figure 7):

1. The stack is empty. Then, a 4-tuple that represents the current context node (*cf.*, procedure `push_ctx`) is pushed on the stack ①:

   eos: holds the preorder rank of the last descendant of the current context node. We require this to determine when processing of this context node will be finished and can be popped from the stack.

   nxtChld: holds the preorder rank of the next child to be processed. Initially, this is the first child. When processing nested context nodes, nxtChld holds the child where to proceed processing for the outer node, once an inner node has finished.

   fstIter, lstIter: hold pointers to the first and the last iteration of the current node in the context node sequence. To find lstIter, the routine `push_ctx` advances over the input table to find all subsequent occurrences of the current context node in different iter-s.

2. The current context node is a descendant of the top of the stack. In this case, we first process all children of the current top of the stack that are predecessors of the current context node ②, before pushing the current context node on the stack ③.

3. The current context node is a successor of the top of the stack. In this case, we process all remaining children of the top of the stack ④ before removing it from the stack ⑤.

Once *ctx* has been scanned entirely, we finish off by successively processing the remaining children of each node that is still on the stack ⑥ and removing the node from the stack ⑦.

For each context node c, procedure `inner_loop_child` directly accesses the first child (or next child to be processed after finishing a descendant context node) in *doc* using its preorder rank for positional (or index)

[3]The underlined parts are only used for the variant described in Section 3.3.

[4]Note that loop-lifted staircase join for the child axis does not depend on $level(c)$. This is different for some other axes, *e.g.*, following-sibling.

```
match_ctx_cand ($nxtCtx, nxtCand$)
 BEGIN
  // advance $nxtCtx$ & $nxtCand$ until $nxtCand$ is descendant of $nxtCtx$
  $curPre \leftarrow ctx[nxtCtx].\mathsf{pre}$;
  $eos \leftarrow curPre + doc[curPre].\mathsf{size}$;
  WHILE ($nxtCtx < lstCtx$ AND $nxtCand < lstCand$ AND
        ($cand[nxtCand].\mathsf{pre} \leq ctx[lstCtx].\mathsf{pre}$ OR $eos < cand[nxtCand].\mathsf{pre}$)) DO
   IF ($cand[nxtCand].\mathsf{pre} \leq ctx[nxtCtx].\mathsf{pre}$) THEN
    // scan or binary search
    $nxtCand \leftarrow$ FIRST $c > nxtCand$ WITH $cand[c].\mathsf{pre} > ctx[nxtCtx].\mathsf{pre}$;
   ELIF ($eos \leq cand[nxtCand].\mathsf{pre}$) THEN
    // scan or binary search
    $nxtCtx \leftarrow$ FIRST $c > nxtCtx$ WITH $ctx[c].\mathsf{pre} > eos$;
    // scan
    $nxtCtx \leftarrow$ FIRST $c > nxtCtx$ WITH $ctx[c].\mathsf{pre} + doc[ctx[c].\mathsf{pre}].size >= cand[nxtCand].\mathsf{pre}$;
  RETURN $nxtCtx, nxtCand$;
 END

inner_loop_child ($eos$)
 BEGIN
  $nxtChld \leftarrow$ TOP$(active).\mathsf{nxtChld}$;                      // next child of current context
  $nxtCand \leftarrow$ TOP$(active).\mathsf{nxtCand}$;                      // next candidate for current context
  $fstIter \leftarrow$ TOP$(active).\mathsf{fstIter}$;                      // first iter of current context
  $lstIter \leftarrow$ TOP$(active).\mathsf{lstIter}$;                      // last iter of current context
  WHILE ($nxtChld \leq eos$ AND $nxtCand \leq lstCand$) DO// iterate of all children in current scope
   IF ($nxtChld < cand[nxtCand].\mathsf{pre}$) THEN
    $nxtChld \leftarrow nxtChld + doc[nxtChld].\mathsf{size} + 1$;          // skip directly to next child
   ELIF ($cand[nxtCand].\mathsf{pre} < nxtChld$) THEN
    $nxtCand \leftarrow$ FIRST $c > nxtCand$ WITH $cand[c].\mathsf{pre} >= nxtChld$; // scan or binary search
   ELSE  // ($nxtChld = nxtCand.\mathsf{pre}$)
    FOR $iter$ FROM $fstIter$ TO $lstIter$ DO                              // iterate over all iters of current context
     APPEND $\langle ctx[iter].\mathsf{iter}, nxtChld \rangle$ TO $result$;  // append (iter,pre) to result
  IF ($nxtChld \leq$ TOP$(active).\mathsf{eos}$) DO                         // current context not yet finished
   TOP$(active).\mathsf{nxtChld} \leftarrow nxtChld$;                       // recall where to proceed
   TOP$(active).\mathsf{nxtCand} \leftarrow nxtCand$;                       // recall where to proceed
  RETURN;                                                                    // return
 END
```

Figure 8: Loop-lifted staircase join: `child` axis with candidate list.

lookup. From there on, `inner_loop_child` subsequently *skips* to the next child (exploiting the knowledge of the sub-tree size) until the end of the current scope (*i.e.*, partition) is reached. With each child node v of c, $\langle i, v \rangle$ is appended to the *result* table for all iterations i related to c, sorted by i. This guarantees document order, avoids the generation of duplicate result nodes within iterations, and ensures that result nodes that belong to multiple iteration occur in iteration order.

In fact, the algorithm in Figure 6 does not even scan the whole document, but rather accesses only the children of each context node, *i.e.*, exactly those document nodes that make up the result of the given axis step. The imposed access pattern is forward-only and hence cache-friendly. To achieve this, the algorithm exploits the characteristics of the pre|size|level encoding. For the preorder rank $pre(c)$ of each context node c, we can easily derive the preorder rank of first child k as $pre(k) = pre(c) + 1$. The other children can than be directly reached by skipping to the next sibling $pre(k) + size(k) + 1$, until the last node in c is reached ($pre(c) + size(c)$).

3.3 Loop-lifted Staircase Join With Selection

The basic version of our loop-lifted staircase join as described above focuses — just like the original staircase join — on the pure XPath location step, ignoring other aspects of XPath like name tests and/or arbitrary predicates. Name tests and predicates can be applied as post-filters on the result of the loop-lifted staircase join.

In practice, however, predicates are often more selective than the pure location steps. Thus, the naive approach of applying predicates as post-filters might first generate a large result of the location step, only to reduce it in the predicate. Due to the commutativity of both operations, it is possible to first evaluate the predicates on the whole document, and then execute the location step only on the reduced document, avoiding the creation of possibly large intermediate results. Obviously, the decision whether to *push-down* predicates underneath a location step should be take by the query compiler/optimizer, based on estimations of the respective selectivities. On a reduced document (*i.e.*, a reduced pre|size|level table), however, we can no longer benefit from positional lookup and skipping.

The selection-enabled version of our loop-lifted staircase join is based on the algorithm depicted in Figure 6 with the underlined parts now enabled. Additionally, we replace the procedure `inner_loop_child` by the one given in Figure 8 and add procedure `match_ctx_cand` as shown in Figure 8.

Selection-enabled loop-lifted staircase join still operates on the complete pre|size document table, but expects an additional third input table that represents the result of the predicate evaluation as a list of the preorder ranks of the qualifying nodes, sorted in document order. Hence, this list contains the potential result *cand*idates. With both the context sequence and the candidate list sorted in document order, the algorithm applies a merge-like synchronized sweep over both inputs, which allows to efficiently skip all context nodes whose children do not appear in the candidate list. For the child step, a candidate can only be a child of a given context node, if it is a descendant of that context node. Procedure `match_ctx_cand` hence performs a merge-like synchronized sequential scan over the context nodes and the candidates to skip non-matching combinations that cannot produce any result. With both *ctx* and *cand* sorted on pre, we can even replace scans of (possibly long) regions of non-matching nodes by binary search and thus reduce the complexity of this merge-like operation from linear to logarithmic. Similarly, the modified `inner_loop_child` skips over all children until the next candidate note is reached, and skips over candidates until the next child is reached.

Except from these changes, the selection-enabled loop-lifted staircase join works just like to basic one. In particular, it performs only a single concurrent forward-only traversal across its inputs (*doc*, *ctx*, *cand*), applying skipping to avoid actual data access where ever possible. Thus, the overall data access pattern is just as cache-friendly as with the basic loop-lifted staircase join and the original staircase join.

4. Quantitative Assessment in MonetDB/XQuery

We implemented loop-lifted staircase join in our open-source XQuery processor MonetDB/XQuery [4]. Figure 9 shows the MonetDB/XQuery system to consist of the XQuery-to-Relational Algebra compiler *Pathfinder* [9] running on-top of the open-source RDBMS *MonetDB* [2]. MonetDB is an extensible system and this feature has been used to introduce an XQuery *runtime module*. It extends the Monet Interpreter Language (MIL) with a few additional operators, mainly the loop-lifted staircase join, plus support for XML input (document shredding) and output (serialization). Queries are parsed, normalized and statically typed (including XML Schema support) and represented as XQuery Core trees, closely following the W3C Formal Semantics. Then, several optimization techniques are applied (including join recognition and order optimizations [3]). As a final stage, physical algebra plans, formulated in MIL are generated for execution on MonetDB.

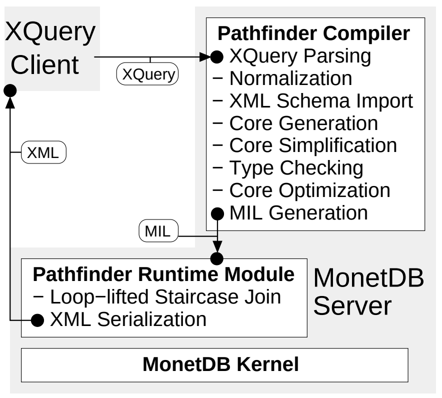


Figure 9: MonetDB/XQuery Architecture

Experiments We evaluated the performance of our algorithms on XMark [12], with scaling factors from 0.1 up to 100, yielding documents from 11 MB up to 11 GB. We are unaware of any other systems that can han-

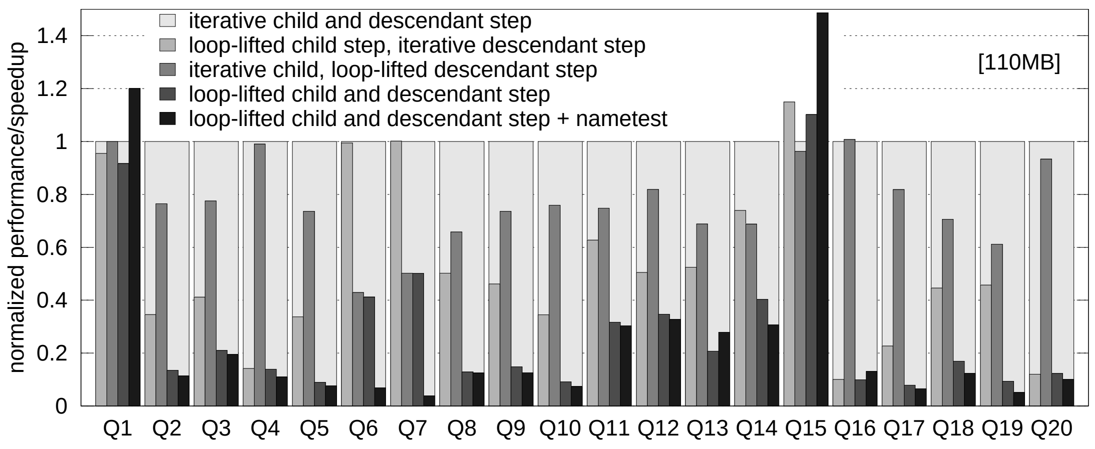


Figure 10: Benefits of loop-lifted staircase join.

dle the 1.1 GB and 11 GB sizes efficiently. The experimentation platform was a 1.6 GHz AMD Opteron 242 (1 MB L2 cache) processor with 8 GB RAM, running Linux 2.6.9 in a 64-bit address space. The following performance results focus on the pure query execution time, excluding the time for XQuery plan translation and optimization as well as result printing (typically < 50 ms).

Figure 10 shows the effect of using loop-lifted staircase join. Loop-lifted staircase join evaluates a path step in one sequential pass over the *pre/size* table for multiple sequences of context nodes in one go. The normal (*i.e.*, *iterative*) staircase join needs to make a sequential pass for each set. As we can see, on the 110 MB XMark document, query performance improves by a factor of 10 when the loop-lifted staircase join is used for both the `child` and the `descendant` axis (no other axis are relevant in the XMark queries). Some queries (Q3,Q11-14), where path step cost is relatively small, in general benefit less (factor 3-5).

MonetDB/XQuery maintains indices on the element names. Without using these indices for name tests, the descendant steps in Queries Q6 and Q7 produce quite large intermediate results that dominate the overall cost. Hence, the benefits of loop-lifted staircase join over iterative staircase join does not exceed a factor 2.5. Since the name test is quite selective for these two queries, using the loop-lifted staircase join variant that allows selection pushdown, yields another factor 6 and 12, respectively, thus resulting in an overall improvement of factor 15 and 25, respectively.

Query Q15 processes a particularly long path expression of 13 axis steps. In this case, loop-lifted staircase join suffers from the additional internal state keeping overhead (the *active* stack) and performs worse than the original staircase join. Furthermore, with both query Q1 and 15, the pure XPath steps are more selective than the respective name test, thus pushing the name test below the loop-lifted staircase join actually leads to a performance decrease.

Table 1 depicts the performance results of MonetDB/XQuery for all XMark queries on documents ranging from 11 MB to 11 GB in size. Figure 11 shows the respective relative performance, where all numbers are normalized to the elapsed time on the 110 MB document. The graph shows that our system scales linearly with document size. The only outliers are queries Q11/12. The bottleneck in both queries is a theta-join (comparison via >) that generates an intermediate result with about 120 K up to 120 G tuples for the 11 MB and 11 GB document sizes, respectively. Note that this concerns the query *result*, whose computation cannot be avoided (though the end result becomes small, due to subsequent aggregation). *Any* XQuery system must necessarily exhibit quadratic scaling with document size on Q11/12.

We should point out that are results are obtained on a memory-resident database and I/O plays no role. Only when intermediate result sizes become large, such as in join queries Q8-Q10 on the 11 GB document size, I/O due to swapping starts to impact performance. In absolute sense, the query execution times presented here for MonetDB/XQuery are quite fast and surpass all currently available XQuery systems such as TIMBER, X-Hive

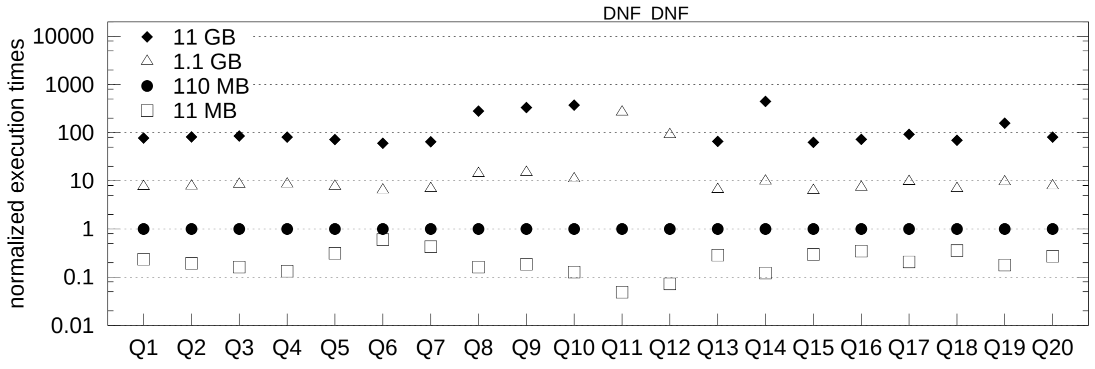


Figure 11: Scalability with respect to document size.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
11 MB	0.04	0.06	0.24	0.06	0.05	0.02	0.03	0.12	0.16	0.64
110 MB	0.18	0.30	1.48	0.45	0.16	0.05	0.07	0.74	0.87	5.05
1.1 GB	1.20	2.40	12.50	3.80	1.20	0.30	0.40	10.40	12.90	55.00
11 GB	13.00	25.00	126.00	36.00	11.00	3.00	4.00	208.00	289.00	1882.00

	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20
11 MB	0.16	0.12	0.06	0.17	0.08	0.09	0.06	0.04	0.10	0.17
110 MB	3.28	1.66	0.21	1.40	0.27	0.26	0.29	0.13	0.55	0.62
1.1 GB	872.50	150.70	1.30	13.70	1.70	1.80	2.60	0.90	5.30	4.90
11 GB	*DNF*	*DNF*	13.00	959.00	16.00	18.00	26.00	9.00	88.00	50.00

Table 1: Overview of XMark query evaluation times (elapsed time in seconds).

and Galax [3].

The overall conclusion of the experiments is that MonetDB/XQuery is a highly scalable XQuery processor that can handle XPath-intensive queries well, mainly thanks to the loop-lifted staircase join that accelerates performance by a factor 5-10.

5. Related Work

A wide variety of relational XML encodings has been proposed [11, 14, 13]. The *pre/size/level* representation provides node surrogates of fixed byte-width, which greatly simplifies storage and query execution. In particular, the dense *pre* numbers allow MonetDB to use *positional* lookup into its array-based storage of relational columns. This feature is maximally exploited in (loop-lifted) staircase join to implement skipping at a cost of < 10 CPU cycles per visited node. If a B-tree index lookup routine would have to be used, absolute performance would be impacted by an order of magnitude. In contrast, variable-length surrogates such as, ORDPATH labels [11], are designed to allow "low-cost" updates while still encoding document order. However, fast updates comes at the expense of higher storage and manipulation costs (positional skipping is not possible and index lookup must be used instead).

In a future version of MonetDB/XQuery, we plan to also support document updates. A scheme that virtualizes the pre-encodings in a page-wise manner to limit the number of *pre* numbers affected by a structural update to a single page, and uses commutative delta operations to reduce locking contention at the document root, promises to keep both update cost and read-only overheads low [5].

In the remainder, we compare loop-lifted staircase join to the related Structural Join [1] and Holistic Twig Joins [6] algorithms. First, we recall the three techniques behind the efficiency of staircase join: *(i) pruning* to reduce the set of context nodes to the minimal set for producing the result nodes *(ii) partitioning* to consider each document node only once, and *(iii) skipping* to avoid considering nodes for which the tree and XPath axis properties tell us beforehand that they can never be in the result.

Both Structural Join and Holistic Twig Join could be used with the iter|pre table of context nodes as one of their input tables (sorted on pre). However, both algorithms are not aware of the different iterations and thus do not perform *(i) pruning* within each iteration. Consequently, duplicate nodes will be in their output, mandating the use of duplicate elimination afterwards. In the case of Structural Join, pruning could be a preprocessing step, but in the case of Holistic Twig Join with *e.g.*, a `child` step followed by a `descendant` step (both with, *e.g.*, name tests), the most useful pruning (for descendants) cannot be done beforehand. We think that adapting the algorithm to allow pruning would be worthwhile. The stack-based nature of both Structural Join and Holistic Twig Join support *(ii) partitioning*, but they use less *(iii) skipping*. In the case of the `child` step, staircase join skips over all descendants of a child to arrive at the next, whereas both Structural Join and Holistic Twig Join consider all such descendants, filtering them out with a separate check. Similar skipping opportunities arise in the other XPath axes; all of which are supported by staircase join in MonetDB/XQuery.

Overall, we think that Holistic Twig Join will outperform loop-lifted staircase join especially on complex twig patterns with high selectivity predicates, that cause loop-lifted staircase join to generate unnecessary large intermediate results. On the other hand, loop-lifted staircase join is simpler and thus faster in terms of raw CPU speed, and uses more pruning and skipping, which can make it faster on simpler twig queries. In any case, since Holistic Twig Join only supports `child` and `descendant` steps, a full-fledged XQuery processor needs both kinds of algorithms, leaving it up to a query optimizer to decide which to use.

6. Conclusions and Future Work

The *staircase join* allows to efficiently evaluate XPath location steps along arbitrary axes in one sequential pass over the *pre/size/level* table. In XQuery, however, embedded XPath sub-expressions occur in arbitrarily nested `for`-loops. Thus, they are rooted in *multiple* sequences of context nodes (one per iteration), which leads to multiple scans over the XML document encoding, strongly decreasing performance. To address this, we presented *loop-lifted staircase join*, that conserves the pruning, partitioning and skipping techniques from staircase join but still manages to evaluate embedded XPath location steps in only a single sequential pass.

We implemented the loop-lifted staircase join in the open source system *MonetDB/XQuery*, that uses the XQuery-to-Relational Algebra compiler *Pathfinder* on top of the extensible RDBMS *MonetDB*. We evaluated our algorithms on the XMark benchmark for sizes up to 11 GB, showing that MonetDB/XQuery is a highly scalable XQuery processor that can handle XPath-intensive queries well, mainly thanks to the loop-lifted staircase join, that accelerates performance by a factor 5-10.

As for future work, we intend to integrate our pruning and skipping techniques in XML join algorithms that allow to evaluate multiple XPath steps simultaneously (*i.e.*, Holistic Twig Join). Other future work concerns improvements of MonetDB/XQuery in the area of document updates as well as algebraic and cost-based query optimization.

References

1. S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 141–152, 2002.
2. P. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, Univ. Amsterdam, 2002.
3. P. Boncz, T. Grust, S. Manegold, J. Rittinger, and J. Teubner. Pathfinder: Relational XQuery Over Multi-Gigabyte XML Inputs In Interactive Time. Technical Report INS-E0503, CWI, 2005.
4. P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. Pathfinder: XQuery—The Relational Way. In *Proc. VLDB Conf.*, 2005. Demo.
5. P. Boncz, S. Manegold, and J. Rittinger. Updating the Pre/Post Plane in MonetDB/XQuery. In *Proc. XIME-P*, 2005.
6. N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XMLPattern Matching. In *Proc. SIGMOD Conf.*, pages 310–321, 2002.
7. T. Grust. Accelerating XPath Location Steps. In *Proc. SIGMOD Conf.*, 2002.
8. T. Grust and S. Klinger. Efficient Validation and Type Annotation for Encoded Trees. In *Proc. XIME-P*, 2004.
9. T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. VLDB Conf.*, pages 252–263, 2004.
10. T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. VLDB Conf.*, pages 524–535, 2003.
11. P.E. O'Neil, E.J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATH: Insert-Friendly XML Node Labels. In *Proc. SIGMOD Conf.*, pages 903–908, 2004.
12. A. Schmidt, F. Waas, M.L. Kersten, M.J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB Conf.*, pages 974–985, 2002.
13. I. Tatarinov, S.D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proc. SIGMOD Conf.*, 2001.
14. C. Zhang, J.F. Naughton, D.J. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. SIGMOD Conf.*, 2001.