MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine

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

[1]CWI Amsterdam, The Netherlands {boncz,manegold}@cwi.nl
[2]Technische Universität München, Germany {grust,rittinge,teubnerj}@in.tum.de
[3]University of Twente, The Netherlands m.vankeulen@utwente.nl

ABSTRACT

Relational XQuery systems try to re-use mature relational data management infrastructures to create fast and scalable XML database technology. This paper describes the main features, key contributions, and lessons learned while implementing such a system. Its architecture consists of *(i)* a range-based encoding of XML documents into relational tables, *(ii)* a compilation technique that translates XQuery into a basic relational algebra, *(iii)* a restricted (order) property-aware peephole relational query optimization strategy, and *(iv)* a mapping from XML update statements into relational updates. Thus, this system implements all essential XML database functionalities (rather than a single feature) such that we can learn from the full consequences of our architectural decisions. While implementing this system, we had to extend the state-of-the-art with a number of new technical contributions, such as *loop-lifted staircase join* and efficient relational query evaluation strategies for XQuery theta-joins with existential semantics. These contributions as well as the architectural lessons learned are also deemed valuable for other relational back-end engines. The performance and scalability of the resulting system is evaluated on the XMark benchmark up to data sizes of 11 GB. The performance section also provides an extensive comparison of all major XMark results published previously, which confirm that the goal of purely relational XQuery processing, namely speed and scalability, was met.

1. INTRODUCTION

We describe *MonetDB/XQuery*, an XML database system that fully supports the W3C XQuery language. It consists of the *Pathfinder* XQuery compiler [17] on top of the *MonetDB* RDBMS [4], and is available now, both for real use and as a research and experimentation platform under a nonrestrictive open-source license[1]. To complete its functionality as an XML database, we also outline a mechanism for supporting efficient updates. This system is a *purely* relational engine in the sense that it does not strictly require any XML storage nor query execution extensions. In this sense, Pathfinder could be deployed on top of *any* RDBMS.

[1]See `http://monetdb-xquery.org/`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.
Copyright 2006 ACM 1-59593-256-9/06/0006 ...$5.00.

We evaluate and compare performance on the XMark benchmark and some synthetic tests to benchmark document shredding and serialization performance, and also include a survey of previously published XMark results.

Contributions. The main contribution of our system is to show that the relational XQuery paradigm can indeed leverage the power of mature relational database technology to deliver speed and scalability in the XML domain. Specifically, our system stores XML and manipulates XQuery sequence data *purely* using relational algebra on relational tables, without data type extensions or changes to the RDBMS storage manager. We do extend the relational query evaluator with a *staircase join* [19] operator, but this is not strictly needed; it only accelerates XPath location steps.

In building the system, we learned valuable lessons regarding RDBMS functionality that can help to improve the performance of relational XQuery. One prominent opportunity is the use of positional algorithms to support lookup into SQL autoincrement key columns. We also present here the two most significant technical innovations to the relational XQuery paradigm (*i.e.*, those that improve performance by more than an order of magnitude).

First, we discovered that *staircase join*, as a technique originally developed for XPath evaluation, falls short of adequately evaluating XQuery, as it cannot efficiently deal with XPath expressions embedded in nested `for`-loops. The new *loop-lifted staircase join* presented here addresses this problem as a fast execution algorithm suitable for an XQuery processor providing the *full axis feature*.

Second, we formulate *relational query optimization* strategies that are specifically suited to efficiently evaluate the query plans originating from XQuery compilation. We define a small number of column properties that are used to drive a peephole optimization stage just before relational code generation. It allows to recognize join patterns in a way that is immune to syntactic variance in XQuery queries, and also allows to avoid expensive sorting operations. We also formulate relational XQuery *join evaluation* strategies that exploit the existential semantics of general comparisons in XQuery (in contrast to plain relational joins).

Outline. Section 2 introduces the basic concepts of relational XML storage and the XQuery compilation scheme we use. Sections 3 and 4 discuss our contributions in the area of loop-lifted staircase join and join optimization, respectively. In Section 5, we fill in a number of details of our implementation regarding the storage scheme used, and the way query optimization and updates are handled. Section 6 focuses on XMark and provides performance results up to XML documents of 11 GB. We also summarize all previously published XMark results to put our system in its proper perspective. We wrap up by discussing related work in Section 7 and outlining our conclusions in Section 8.

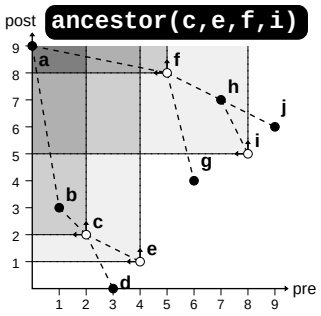


Figure 1: Pruning: we can prune context nodes c, f as they are inside the `ancestor` regions of e, i and thus only generate duplicate results that would need to be eliminated later.

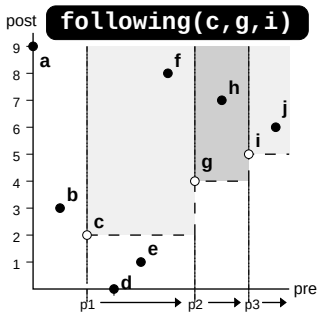


Figure 2: Partitioning: the overlapping `following` regions covered by c, g, i are partitioned along the *pre* axis at p_1, p_2, p_3. Thus, result generation for c is cut-off at p_2, avoiding any remaining duplicate generation.

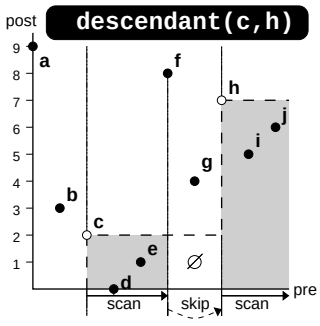


Figure 3: Skipping: after hitting f, the `descendant` staircase join infers that no results can occur until h and thus skips a potentially large part of the pre|size|level table.

2. RELATIONAL XML

In [19], Grust et al. described a relational encoding of XML fragments that is a true isomorphism with respect to the tree structure. The encoding is based on *preorder* and *postorder traversal ranks* to encode the XML tree structure. Here, we use an equivalent encoding variant in which the location of a node v in the document tree is represented as the triple $\langle pre(v), size(v), level(v)\rangle$, recording v's preorder rank, the number of nodes in the subtree below v, and the distance from the tree's root, respectively. (From this, the postorder rank may be recovered via $post(v) = pre(v) + size(v) - level(v)$.) The preorder rank $pre(v)$ simultaneously serves as a node identifier. Figure 4 depicts an XML fragment and the encoding we assign to this fragment. The system maintains further tables to capture more node properties (*e.g.*, tag name, node kind, text content), as described in Section 5.

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

	pre	size	level	post
a	0	9	0	9
b	1	3	1	3
c	2	2	2	2
d	3	0	3	0
e	4	0	3	1
f	5	4	1	8
g	6	0	2	4
h	7	2	2	7
i	8	0	3	5
j	9	0	3	6

Figure 4: XML encoding.

Our tree representation exhibits a number of useful characteristics, among which element (or tree) construction through pasting of encodings and highly efficient XPath processing with *staircase join* are especially relevant in the XQuery context.

Staircase Join. While the pre|size|level table storage already allows accelerated (B-tree powered) relational XPath evaluation, additional performance can be won if the relational join operator is aware of the XPath semantics and the tree properties encoded in the pre|size|level table. Staircase join scans the pre|size|level table sequentially at most once, never delivers duplicate nodes, and produces result nodes in document order, so no post-processing is needed to comply with XPath semantics. Figures 1, 2, and 3 show the three techniques that distinguish it from similar approaches:

(i) Pruning: given a context node, the main XPath axes `descendant`, `ancestor`, `following` and `preceding` correspond to the four quadrants in the *pre/post* plane. Any context node c_i that is inside the quadrant covered by another context node c_j will generate a subset of c_j's result nodes. In order to avoid these duplicates, we can simply omit (prune) those covered context nodes.

(ii) Partitioning: when covered regions overlap partially, duplicate nodes may still be generated. This can be avoided by partitioning the regions along the *pre* axis. This eliminates all duplicate result generation.

(iii) Skipping: sometimes we may infer from the properties of the pre|size|level encoding and the particular XPath step at hand, that certain regions cannot contain results. The effectiveness of skipping in the four main XPath axes is high. For each node in the context, we either hit a node to be copied into the result, or 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 [18].

The XPath `child` axis provides similar skipping opportunities. 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: $v_{i+1} = v_i + size(v_i) + 1$. We will further exploit this observation in Section 3.

2.1 From XQuery To Relational Algebra

Apart from XML trees, *item sequences* constitute the principal data structure manipulated by XQuery. It will turn out that once the relational representation of item sequences is fixed, much of the relational XQuery processing strategy may be rather straightforwardly derived.

Representing Sequences as Tables. The evaluation of any XQuery expression yields an *ordered sequence* of $n \geqslant 0$ items x_i, denoted $(x_1, x_2, \ldots, x_n)$. In the XQuery data model, a single item x and the singleton sequence (x) are identical.

We will use a relational sequence encoding that explicitly reflects sequence order by means of a pos column as depicted here. Item x is represented as the singleton relation of type pos|item containing the tuple $\langle 1, x\rangle$, the empty relation of type pos|item encodes the empty sequence `()`. An XQuery *item* is either of an *atomic type* or of type node. To represent the former, we choose an implementation type t supported by the relational back-end such that the domain of t either *(i)* can represent the corresponding XQuery type directly (*e.g.*, integer, string), or *(ii)* allows encoding the domain of the XQuery type (*e.g.*, a string of the form `"--MM-DD"` can encode values of the XQuery type gMonthDay).

pos	item
1	x_1
2	x_2
⋮	⋮
n	x_n

We represent nodes by their preorder rank $pre(v)$ which reflects *document order* and *node identity*, *i.e.*, for two nodes v_1, v_2, we have that $v_1 \texttt{<<} v_2 \Leftrightarrow pre(v_1) < pre(v_2)$ and $v_1 \texttt{is} v_2 \Leftrightarrow pre(v_1) =$

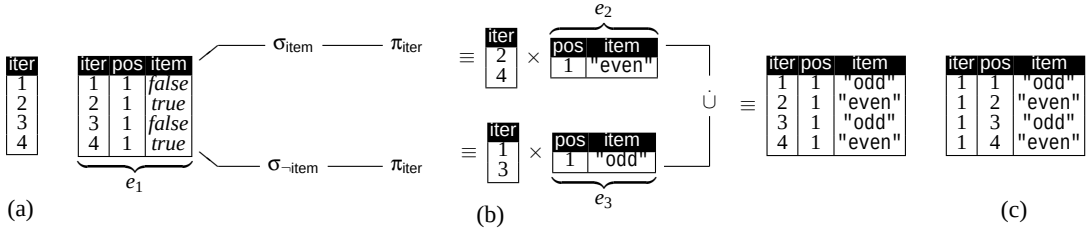


Figure 5: Relational XQuery evaluation: (a) loop relation in scope s_v, (b) evaluation of conditional, (c) final result in outermost scope.

$pre(v_2)$. Any node surrogate that satisfies these requirements may serve as a node representation equally well, and the database community has devised a variety of alternative ways to implement them [25, 30].

In XQuery, sequences host items of arbitrary type. The sequence `(2,"x",<a/>)` leads to the depicted relational encoding (where γ_a denotes the surrogate of the XML node constructed by the node constructor `<a/>`) with a polymorphic item column. For simplicity, we stick to this representation here, see [5] for more details on how a back-end that implements monomorphic columns only may support this encoding nevertheless.

pos	item
1	2
2	"x"
3	γ_a

XQuery to Relational Algebra Compilation. Since the XQuery processor is hosted by a relational back-end, relational algebra is the target language of XQuery compilation. A number of research teams have investigated such relational XQuery compilers [12, 13], but here we will adopt the approach developed in [17].

Relational algebra is a *combinator style* language and thus lacks *variables*, a core XQuery concept. We will thus discuss compilation of variables bound in `for`-loops first. Consider the following XQuery `for`-loop:

$$\texttt{for \$}v\texttt{ in (}x_1\texttt{,}x_2\texttt{,}\ldots\texttt{,}x_n\texttt{) return }e\ .$$

This expression successively binds each x_i to variable $\$v$ and evaluates the loop body e in each iteration. We have already derived a relational representation for the sequence $(x_1, x_2, \ldots, x_n)$ above.

iter	pos	item
1	1	x_1
2	1	x_2
.	.	.
.	.	.
n	1	x_n

The relation shown here suitably encodes all bindings of $\$v$ in a single relation. This iter|pos|item encoding will be pervasive in the following: a tuple $\langle i, p, x\rangle$ indicates that in the i-th iteration, the item at position p in the represented sequence has value x. Note that the database system can easily derive the representation of a variable from the representation of the sequence it gets bound to: *(i)* attach a new iter column, densely numbered from $1 \ldots n$ in the *order* given by the pos column, *(ii)* then set the pos column to constant 1.

The *row-numbering* in step *(i)* is characteristic for this approach and we assume the availability of a respective relational operator $\rho_{A:\langle C_1,\ldots,C_n\rangle/C_g}(R)$ that, for each tuple group defined by column C_g, extends relation R with a densely numbered column A respecting the ordering specified by the columns C_i.[2]

iter
1
2
.
.
n

Note how the iter column encodes the iteration performed by the `for`-loop. The principal idea of this compilation approach is that *each query subexpression is compiled in dependence of all enclosing `for`-loops*, the latter being represented by a unary iter relation (this unary relation is referred to as the loop relation in the sequel). In the query above, loop body e is in the scope of the n-fold iteration encoded by the relation loop depicted here on the left.

[2] As observed in [17], ρ exactly embodies the functionality of SQL:1999's OLAP extension `DENSE_RANK() OVER (PARTITION BY` C_g `ORDER BY` $C_1,\ldots,C_n$`)`.

When a constant subexpression e' of loop body e is compiled into its relational algebra equivalent, q' say, q' is *lifted* according to the current loop to give $\text{loop} \times q'$. To exemplify, the XQuery constant 42 represented by the pos|item tuple $\langle 1, 42\rangle$, is lifted to give the relation on the right (to be read as: in each of the n iterations, the constant assumes the value 42).

iter	pos	item
1	1	42
2	1	42
.	.	.
.	.	.
n	1	42

It is both a consequence of the XQuery semantics and this approach, that in a nested iteration like[3]

$$\texttt{for \$}v_1\texttt{ in (}x_1\texttt{,}x_2\texttt{,}\ldots\texttt{,}x_n\texttt{) return}\quad s_{v_1}\ \left\{\begin{array}{l}\texttt{for \$}v_2\texttt{ in (}y_1\texttt{,}y_2\texttt{,}\ldots\texttt{,}y_m\texttt{) return}\\ \quad s_{v_1\cdot v_2}\ \{\ e\end{array}\right. ,$$

the representation of the sequence $(y_1, y_2, \ldots, y_m)$ in scope s_{v_1} as well as the representation of variable $\$v_2$ in the innermost scope $s_{v_1\cdot v_2}$ contain $n * m$ tuples due to the necessary loop-lifting (each y_i occurs n times). The avoidance of the computation of such "Cartesian products" will be discussed in Section 4.

In order to provide an intuition of the typical algebraic plans emitted by the XQuery compiler, let us briefly review the compilation and execution of the XQuery expression

$$\texttt{for \$}v\texttt{ in (3,4,5,6) return}\quad s_v\ \{\ \texttt{if (}\underbrace{\$v\texttt{ mod 2 eq 0}}_{e_1}\texttt{) then }\underbrace{\texttt{"even"}}_{e_2}\texttt{ else }\underbrace{\texttt{"odd"}}_{e_3}\ .$$

The current loop relation in scope s_v is shown in Figure 5(a). For brevity, we already show the intermediate result obtained through the evaluation of the predicate subexpression e_1 on the very left of Figure 5(b). In the third iteration, for example, the predicate evaluates to the single item *false*. Depending on the outcome of the predicate, we need to either evaluate the `then` branch e_2 or the `else` branch e_3. Two independent selections compute the respective set of iter values ($\sigma_A(R)$ selects all tuples with value *true* in column A, $\sigma_{\neg A}(R)$ selects the complement). Figure 5(b) shows the resulting loop relations which are used for loop-lifting in the `then` and `else` branches, respectively. The evaluation of the conditional is completed by forming the *disjoint* union of the intermediate results in both branches. Note that this result is still represented with respect to scope s_v (each iteration contributes one item to the result). A back-mapping step (a single equi-join of the intermediate result with a so-called scope map relation [17]) then yields the final result sequence of length 4 (Figure 5(c)).

To wrap up, note that this type of XQuery compiler targets a rather standard logical relational algebra—in addition to ρ mentioned above, we require σ, π, ⋈, ×, \, ∪̇ (disjoint union) as well as a means to evaluate arithmetic and comparison operators.

[3] We denote a variable scope by $s_{v_1\cdots v_n}$ if variables $\$v_1, \ldots, \$v_n$ are visible in that scope.

3. FROM XPATH TO XQUERY: LOOP-LIFTING STAIRCASE JOIN

It is the gist of the original staircase join algorithm that it evaluates an XPath location step for an entire context sequence in a single sequential scan over a pre|size|level encoded XML document.

In XQuery, however, expressions occur in nested iteration scopes. The same is true for XPath sub-expressions. Consider the query

$$\texttt{for \$}v\texttt{ in }(x_1,x_2,\ldots,x_n)\texttt{ return } e(\$v)\texttt{/child::t} \qquad (Q_1)$$

The evaluation of the loop body of queries such as (Q_1) requires n invocations of staircase join and thus as many sequential scans over the document encoding table. This surely seems wasteful. To avoid the overhead of repetitive scans, we may *loop-lift* (see Section 2.1) the staircase join algorithm, so that it can evaluate the axis step for all context node sequences of all iterations with one sequential scan over the document encoding table.

The loop-lifted staircase join performs a *single* sequential scan over both its inputs: the relation with all context node sequences and the document encoding table. To achieve this, it expects the former to be sorted on pre|iter, so the context nodes appear in document order and for each context node all related iterations appear clustered and in order. At runtime, the relational encoding of the n context node sequences (resulting from the n evaluations of $e(\$v)$ in (Q_1)) will take the form depicted here: 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 $(1 \leqslant i \leqslant n)$. We might have $s_i = 0$ for some i: in this case, no tuple with iter value i will occur. This is the input of the loop-lifted staircase join.

iter	pos	item
1	1	$\gamma_{1,1}$
1	2	$\gamma_{1,2}$
⋮	⋮	⋮
1	s_1	γ_{1,s_1}
⋮	⋮	⋮
n	1	$\gamma_{n,1}$
⋮	⋮	⋮
n	s_n	γ_{n,s_n}

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. Therefore, our algorithm ignores the pos column.

The general idea of loop-lifted staircase join is best explained using the three main techniques that exploit XPath semantics and the tree properties of the relational encoding:

(i) Pruning: now that context sequences of *multiple* iterations are evaluated together, the standard pruning method would eliminate too many context nodes. To be precise, the context set now consists of pre|iter tuples, and each pre value may appear for multiple iter values. The same pruning rules from plain staircase join are used on pre|iter context values, but only delete context values from the same iter.

(ii) Partitioning: while in plain staircase join only a single context node could be active, in the loop-lifted version each context node may be active for a set of iterations, and stays active for that iteration until we hit another context node that belongs to that iteration. Implementation-wise, this means that the algorithm now manages a *stack* of active context values pre|iter, with at most one value on the stack for each different iter. In this sense, loop-lifted staircase join becomes more similar to alternative stack-based approaches such as structural join [1] and holistic twig join [7], though these algorithms lack pruning and skipping, and support the `descendant` axis only (and `child` relatively inefficiently, as a post-filter).

(iii) Skipping: the skipping logic in general remains the same. In particular, the algorithm will touch at most $|\mathsf{result}|+|\mathsf{context}|$ nodes from the document relation to produce its result. Furthermore, we retain the sequential access pattern on the pre|size|level table for highly efficient cache usage.

Note that loop-lifted staircase join is capable of evaluating *all* XPath axes (not only `descendant` or `child`). As we lack space to provide detailed algorithms for all XPath steps, we focus in our detailed explanation on `child`.

3.1 Detailed Algorithm: The `child` Axis

The way the algorithm of Figure 6 processes both input relations is illustrated in Figure 7. Suppose we evaluate a child step in a query where there are two iterations. In the first, the context node sequence is (c_1) and in the second it is (c_1,c_2). Because of partitioning, the algorithm skips any nodes preceding c_1 ①. Context node c_1 is pushed on the stack with some associated information: the end of the associated partition (*eos*), the next child to be processed (*nxtChld*), and in which iterations it is active (*fstIter*, *lstIter*). Then, `inner_loop_child` scans the document encoding table to produce all children up to the preorder rank of c_2 ②. Note that all children are produced twice, because c_1 is active in both iterations.

c_2 is pushed on the stack ③ and its children are produced in the same way ④. When the end of the partition associated with c_2 is encountered, it is popped from the stack ⑤ and production of the remaining children of c_1 resumes ⑥ until the end of its partition is encountered as well ⑦.

Note that for each context node c, procedure `inner_loop_child` directly accesses a child in *doc* using its preorder rank for positional (or index) lookup. By exploiting the knowledge of sub-tree size, it effectively *skips* any document nodes that do not appear in the result ②. *Pruning* does not apply to the `child` axis and, hence, is not mentioned in our example. Other axes benefit from the technique nevertheless. This `child` algorithm guarantees document order, avoids the generation of duplicate result nodes within iterations, and ensures that result nodes that belong to multiple iterations occur in iteration order. The imposed access pattern is forward-only and hence I/O and CPU cache-friendly.

3.2 Predicate Pushdown

The basic version of our loop-lifted staircase join ignores name tests and/or arbitrary predicates following the XPath step. Such selections 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. 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 taken 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 cannot perform skipping with positional lookup.

We developed a version of our loop-lifted staircase join that allows predicate pushdown. It expects a list of candidate nodes (in document order for the forward axes, and in reverse order for the reverse axes); which is typically delivered by an index structure. Where the normal loop-lifted staircase join generates results directly, the predicate pushdown version checks whether a result node is in the candidate list, and only then generates it. As results are generated in (reverse) document order for the (reverse) XPath axes, this relies on an efficient two-way merge operator between emitted results and candidate list. Also, the *skipping* logic for several axes such as `child` and `descendant` is extended to *skip* context nodes that given the next candidate can never yield a result.

```
ll_scj_child (doc : TABLE(pre,size), ctx : TABLE(iter,pre))
 BEGIN
    ASSERT (doc.pre IS DENSE AND ASCENDING);     // for positional lookup
    ASSERT (ctx IS SORTED ON (pre,iter));        // document order
    result ← NEW TABLE(iter,pre);                // the result
    active ← NEW STACK(eos,nxtChld,fstIter,lstIter); // stack of active context nodes
    nxtCtx ← 0;      lstCtx ← SIZE(ctx);         // first & last context node
    WHILE (nxtCtx ≤ lstCtx ) DO                  // iterate over all context nodes
      IF (active IS EMPTY) THEN                  // stack is empty
①       nxtCtx ← push_ctx(nxtCtx);               // 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 ← push_ctx(nxtCtx);               // 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               // fini sh 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)
 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,fstIter,lstIter⟩ ON active; // push current context on stack
    RETURN nxtCtx;                               // return next context
 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.

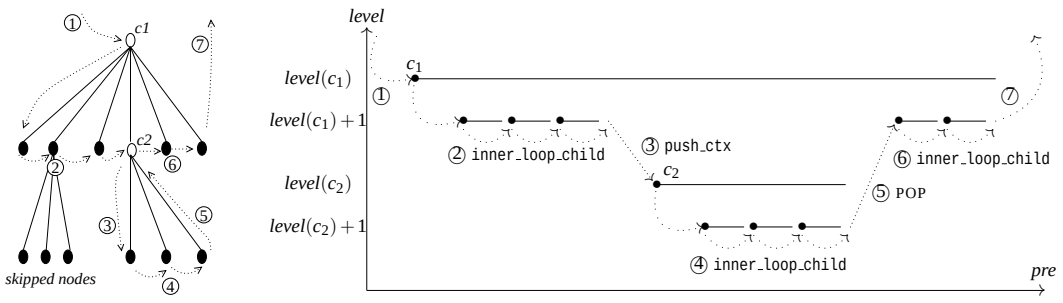


Figure 7: Example `child` traversal (left), illustrated with calls in the algorithm of Figure 6 (right).

4. RELATIONAL QUERY OPTIMIZATION

Over the entire history of database research, query optimization has been one of the most active topics, and relational XQuery processing just adds new challenges to it. The purely relational compilation approach used here, creates query plans of considerable size, due to the representation of sequences, iterations, and maps via tables. Therefore, the relational optimizer should exploit the particular properties exhibited by the generated plans. As we currently do not expect relational back-ends to properly do this, our current XQuery compiler performs algebraic rewrites itself, and generates physical relational algebra plans. This approach in a sense defeats the aim of re-using mature relational query optimization techniques. On the other hand, we think the purely relational query optimization techniques outlined here are quite general, and should be a fine addition to the current generation of relational query optimizers that will benefit other relational applications as well.

4.1 Peephole Optimization

In case of the XMark benchmark, the generated query plans contain 86 relational algebra operators on average, of which 9 are joins. These joins are the result of *loop-lifting* input sequences over `for` loops and map old iteration numbers into new ones. The only "real" value-based joins actually appear in the expected places in join queries Q8–Q12. The other joins are lookups into ordered and even *dense* integer iter and pos key columns in the temporary tables that we use to represent item sequences.

The latest version of the SQL standard introduces a number of features for *sequence-generating* or *identity* columns. Such columns containing densely increasing sequences $\langle 1,2,3,\ldots\rangle$ are used as key in almost *all* our temporary result tables, which in addition are materialized always, as they tend to be re-used multiple times in the query plan. Our RDBMS back-end already provides efficient support for these sequence columns in read-only materialized intermediate results, in that it can use *positional* join and selection algorithms when equality clauses are used on such key columns.

A key observation for these joins on dense sequence-number keys is that they have a fixed join hit rate of 1, and the optimal physical algorithm to choose is obvious (positional join). Thus, the optimizer should not waste effort considering various join ordering permutations for them. In our system, we therefore restrict the ability of the RDBMS optimizer to change the query plan order, and instead perform *property*-driven *peephole optimization* of linear complexity on the intermediate relational plans. The plan emitted after this optimization is in order-aware physical relational algebra, and currently enforces that all intermediate iter|pos|item temporary tables representing item sequences are ordered on [iter, pos]. This order-awareness avoids the need for expensive re-sorting.

The peep-hole optimizer maintains the following column properties on intermediate table results:

$dense(c)$	column c is a sequence 1,2,3,..
$key(c^D)$	column c is unique from domain D
$const(c^v)$	column c has constant value v
$ord([c_i^D])$	tuples are lexicographically ordered on columns $[c_i]$
$grpord([c_i^D], g^D)$	all sub-sequences of the table formed by tuples with the same value in g, are $ord([c_i])$
$indep(\{c_i^D\})$	table is independent of columns $[c_i]$

The *dense* and *key* properties are used to recognize the opportunity for positional join and selection algorithms, and to exclude such joins from join-order selection. The *const* property is used to omit columns from the generated intermediate tables if their value is constant. The *ord* and *grpord* are used to prune sort operators from the physical plan (if we can conclude that the required order is already present) and to choose more efficient merge algorithms when possible. The *grpord* property is a generalization of the secondary sorting property [39], with the difference that it does not require the groups within which the secondary ordering exists, to appear clustered in the input (i.e. as consecutive tuples). This property is especially useful for efficiently executing the SQL:1999 `DENSE_RANK() OVER (PARTITION BY` g `ORDER BY` $c_1..c_n$ `)` operator in a *streaming fashion*, by using a hash-based numbering algorithm that increments a counter stored in a hash table for each active group value from g, rather than the default re-numbering algorithm that requires a full sort on $[g, c_1, .., c_n]$.

Finally, the *indep* property is used to detect *independence* between sub-expressions e_1 and e_2. Typically, the XQuery compiler emits plans where e_1 is *loop-lifted* over e_2 using a join over a scope map relation. If independence is detected, the loop-lifting is substituted by a Cartesian product between e_1 and e_2. If the XQuery contains some subsequent comparison expression between e_1 and e_2; this resulted in the presence of a selection operator in the relational plan. The combination of Cartesian product and selection can then easily be rewritten into an algebraic join. The beauty of this approach is that it is guaranteed to detect XQuery joins in *any* syntactic form. More details on this join recognition method and the property propagation rules used can be found in [16].

4.2 Exploiting Existential Join Semantics

In XQuery, a *general comparison* e_1 `=` e_2 (`<`, `<=`, ...) uses *existential semantics*: if any item in sequence e_1 is equal to any item in sequence e_2 the comparison yields *true*. The W3C XQuery Working Draft specifies that an implementation may process a general comparison using *shortcut* evaluation: as soon as a matching pair of items is found, the processor may return *true*.

Since general comparisons are pervasive in XQuery, the XQuery compiler generates the corresponding relational plans with care, especially if such comparisons are used in join predicates. Consider

```
for $u in e1, $v in e2
  where $u/p1/@a1 = $v/p2/@a2          (Q2)
  return $u ,
```

Figure 8(a) exemplifies the evaluation of the corresponding relational join plan (in this example, the atomization of the path expression $\$u/p_1/@a_1$ shall evaluate to the sequences (20) and $(30,20)$ for the bindings of $\$u$ to the two items of e_1, respectively). MonetDB/XQuery executes a theta-join using the corresponding *value comparison* (here: `eq`) in the predicate. In general, this leads to

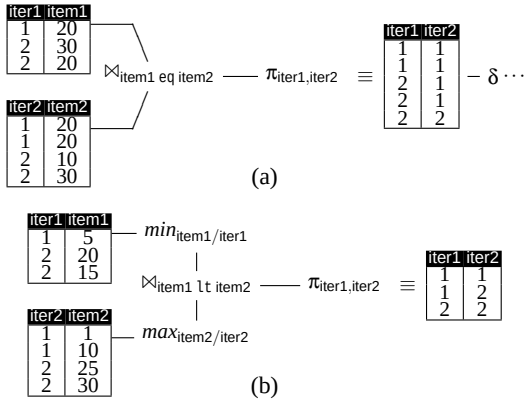


Figure 8: Implementing the existential semantics of XQuery's general comparison operators: (a) duplicate elimination after join, (b) join pushed beyond aggregates.

duplicate $\langle \text{iter1}, \text{iter2} \rangle$ pairs in the join result. A subsequent duplicate elimination (operator δ) reduces this to unique pairs and thus implements the required existential semantics.

As mentioned, the physical algebra emitted by MonetDB/XQuery is order-aware and ensures that intermediate result relations are sorted on $[\text{iter}, \text{pos}]$ (and thus iter). The hash-join used by MonetDB/XQuery for eq theta-predicates respects the $[\text{iter1}]$ order of its left input and due to the the $[\text{iter2}]$ order of its right input will construct the hash-table in such a way that multiple matches on the equal inner iters occur consecutively. Thus, the δ can use a merge algorithm to eliminate duplicates, making this step highly efficient.

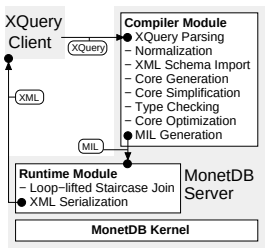
One should note that the use of efficient ordered duplicate elimination we mention here only concerns elimination of duplicate iter combinations generated by the join. MonetDB/XQuery always first materializes both join input sequences. As the join needs $[\text{iter}, \text{pos}]$ order for its inputs, such materialization is inevitable for results of XPath expressions, because those are delivered by loop-lifted staircase join in document order (*i.e.*, in item column order), and thus need to be re-sorted first.

If a theta-join predicate employs one of XQuery's general comparison operators (<, <=, >=, >) the generation of duplicates may be avoided a priori. Consider Figure 8(b) in which < occurs in the predicate. Since, in each iteration, it suffices to find *any* pair of items in the lt-relationship, we might as well only compare the smallest and largest items of the sequences. To exploit this observation, the system applies *min* and *max* aggregates in each iter-group of the left and right inputs, respectively. Note that the grouping is for free due to the $[\text{iter}]$ order of both inputs. After aggregation, the iter values are unique per join input. The theta-join thus delivers unique $\langle \text{iter1}, \text{iter2} \rangle$ pairs directly.

The choice of algorithm for non-equi theta-join can either fall on an index-lookup based algorithm (that creates a temporary index on-the-fly) or nested-loop join. Only if the join hit-ratio is moderate, the former approach is better, otherwise result construction cost dominates and the performance of both is about the same. In that case, however, nested-loop join is to be preferred, as it delivers its result ordered on $[\text{iter1}, \text{iter2}]$, as required. In contrast, the output of the index-lookup join is only ordered on $[\text{iter1}]$ and must be *refine-sorted* on iter2 within equal chunks of iter1 (the MonetDB RDBMS already provides such an incremental, pipelinable refine-sorting algorithm). Given that the join inputs are materialized anyway, MonetDB/XQuery generates a "choose-plan" that at run-time decides between nested-loop and index-lookup join, by estimating the actual join hit-rate by computing a small join sample first.

5. SYSTEM IMPLEMENTATION

MonetDB/XQuery consists of the *Pathfinder* compiler [17] on top of the an extensible open-source *MonetDB* RDBMS [4]. MonetDB has been extended with an XQuery *runtime module* that adds the loop-lifted staircase join (Section 3) as a physical operator.



5.1 Document Representation

For any XQuery Core construct, the compiler emits a physical relational algebra sub-plan. The translation follows the ideas in [17] followed by the peephole optimization described in Section 4.

The backbone of any XQuery processor is the storage of XML trees and fragments. The pre|size|level document table contains additional columns storing node properties. Relevant properties

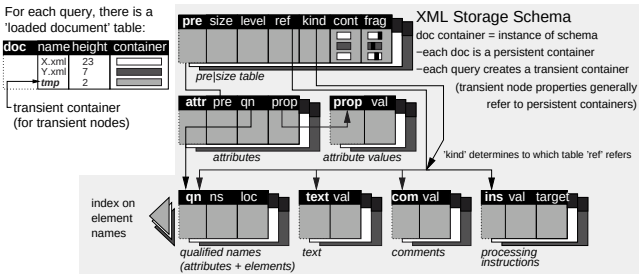


Figure 9: Partitioned XML storage in document containers.

depend on the node's *kind* (*e.g.*, element, text node,...). We use a set of *property containers* for the different node kinds. Figure 9 lists the property containers and their schemata (name space and local name for XML elements, textual content for text and comment nodes, and a target/value pair for processing instructions).

With possibly multiple documents contributing to the query, we hold a separate instance of this storage layout (that we refer to as a *document container*) for any referenced document. An additional document container hosts all transient nodes computed during query evaluation (*e.g.*, the result of XQuery's element construction operator). To keep nodes from disjoint tree fragments apart in this container, we introduce the frag column that uniquely identifies each XML tree fragment.[4] A *loaded document* table keeps track of any document container currently active.

The pre|size encoding allows for a particularly efficient implementation of subtree copying: the corresponding region may simply be copied verbatim from the pre|size table to capture the *structural properties* of the subtree. Our implementation extends this idea and provides shallow copying for the further node properties. We introduce the cont column that references the document container in which each node's properties are to be found. We copy ref, kind, and cont along with the structural part, retaining cont as a reference to its original container.

5.2 XML Updates

Only recently, W3C has formulated its first proposal for an XML update language [10]. Here, we implemented the same update functionality by means of a series of new XQuery operators with side effects [6], minimizing the required changes in our XQuery parser.

XML updates can be classified as: *(i) value updates*, which include node value changes (be it text, comment or processing instructions), and any change concerning attributes (attribute value changes, attribute deletion and insertion). Other modifications are *(ii) structural updates*, that insert or delete nodes in an XML document. With the pre|size|level encoding, value updates map quite trivially to updates in the underlying relational tables. Therefore, we focus on *structural updates* in the remainder.

Structural Update Problems. Figure 10 illustrates how the pre|size|level document encoding is affected by a subtree insert (cf. delete): all pre values of the nodes following the insert point require renumbering, as well as the size property of all ancestor nodes. The former issue imposes an update cost of $O(N)$, with N the document size, because on average half of the document are following nodes. The latter issue is not so much a problem in terms of update volume (renumbering affects at most $height(t)$ nodes, which remains small even for huge XML instances) but rather one

[4] Column frag provides a natural means to implement document order across multiple XML fragments in line with the XQuery semantics. MonetDB/XQuery sorts tuples on the $[\text{frag}, \text{pre}]$ combination.

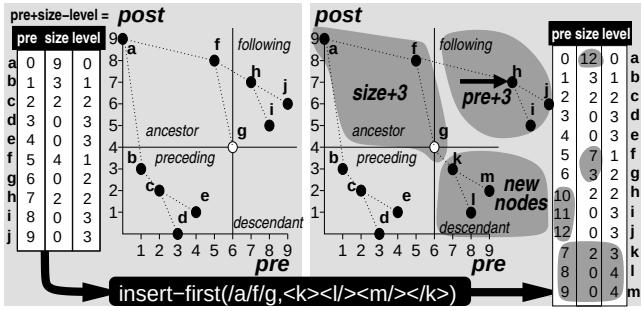


Figure 10: The impact of Structural Updates on pre|size|level

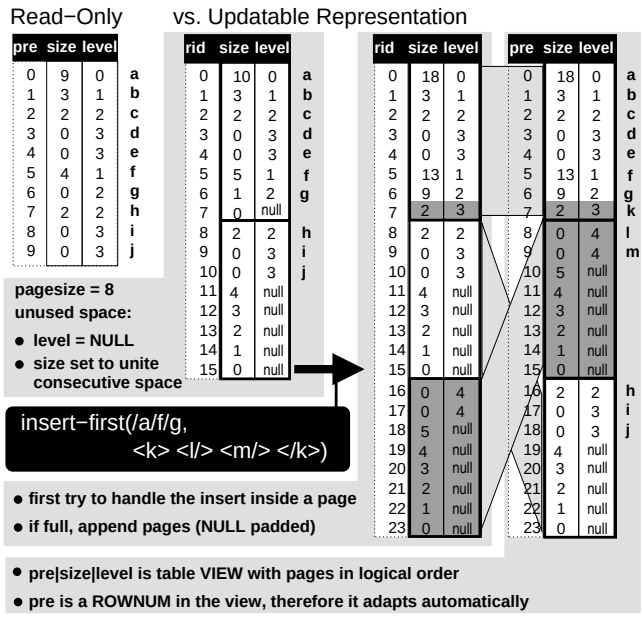


Figure 11: Updates With Logical Pages.

of locking: the document root is an `ancestor` of all nodes and thus must be locked by every update. This problem, however, can be circumvented by maintaining for each transaction a list of nodes of which the size is changed, together with the *delta* rather than the absolute changed value. This allows transactions to release locks on size immediately, and commit anytime later (even if the size of a node has been changed meanwhile by another committed transaction, we can just apply the delta to set it to a consistent state). With the problem of lock contention on size removed this way, in the sequel we concentrate on the problem of the shifts in pre here.

Page-Wise Remappable Pre-Numbers. Figure 11 shows the applied changes to handle structural updates in the pre|size|level table. The key observations are:

- the pre column has been replaced by the *row id* rid,
- the document table is divided into *logical pages*,
- each logical page may contain *unused tuples*,
- new logical pages are appended only (*i.e.*, at the end), and
- the pre|size|level table is a view on rid|size|level with all pages in logical order.

Figure 11 shows the example document being stored in two logical pages. The logical size is measured in an amount of tuples (here: 8) instead of bytes, and is always a power of two, such that pre-numbers can be quickly converted into rid numbers using a *swizzling* technique that uses the higher bits as a lookup value into a *page-mapping table*. This page-mapping table has a tuple for each logical page and contains both its sequence number in the rid table as well as the sequence number of its appearance in the pre view.

The document shredder already leaves a certain (configurable) percentage of tuples unused in each logical page. Initially, the unused tuples are located at the end of each page. Their level column is set to NULL, while the size column holds the amount of directly following consecutive unused tuples. This allows the staircase-join to skip over unused tuples quickly.

The advantage of unused tuples is that structural deletes just leave the tuples of the deleted nodes in place (they become unused tuples) without causing any shifts in pre numbers[5]. Also, inserts of subtrees whose size do not exceed the amount of unused tuples on the logical page, do not cause shifts on other logical pages. Larger inserts only use page-wise table appends. This is the main reason to replace pre by rid. The rid column is a densely increasing (0, 1, 2, ...) integer column, such as generated by a SQL `autoincrement` column. A key feature of the update scheme is that we only *append* tuples to the rid|size|level table, and modify only non-key values (obviously); even in case of XML deletes.

In our RDBMS back-end, the append-only nature of the table is expressed using access rights, and allows the RDBMS to conclude that the generated sequence key column rid always corresponds to physical tuple order, such that it can use efficient *positional* key lookup and foreign key join algorithms.

In the example of Figure 11, three new nodes `k`, `l` and `m` are inserted as children of context node `g`. This insert of three nodes does not fit the free space (the first page that holds `g` only has one unused tuple at rid = 7). Therefore, a new logical page must be inserted in-between. Thus, we append eight new tuples, of which only the first two represent real nodes (`l` and `m`), the latter six are unused.

We also introduced new functionality in our RDBMS back-end to create a table view that map the underlying disk pages of a table in a different non-sequential order, given a pre $\leftrightarrow$ rid mapping table. Thus, by re-arranging the logical pages of the rid|size|level table in logical page order, overflow pages that were appended to it, become visible "halfway" in the pre|size|level view. Alternative to this table view, the same effect could be obtained by making the staircase join operators aware of the pre $\leftrightarrow$ rid page-mapping table. Each time the staircase join during its forward or backwards scan over the rid|size|level table crosses a logical page boundary, it must *swizzle* the destination pre number into a rid number using the mapping table. The resulting rid key can then be located using a B-tree lookup, or better, using a positional lookup algorithm.

All in all, an XML insert leads to writing at least one new logical page (plus the volume of the insert). Depending on how the logical page size is configured (as a number of tuples) with respect to the disk page size, this leads to a constant number of I/Os (typically one or two per logical page). Thus this scheme keeps the I/O volume caused by updates limited to a minimum. Its weakness may stem from the fact that these page-wise inserts and updates cause the database back-end to employ page-wise locking. In this respect, relational systems that use (derivatives of) the Dewey encoding to represent XML node-IDs can achieve a higher degree of transaction concurrency. We expect on the other hand that the simpler pre integer representation that allows for highly efficient node order comparisons, and XPath evaluation in loop-lifted staircase join backed by positional algorithms gives our system the upper hand in raw query performance, which (depending on the application workload) may offset the concurrency limitations of page-wise locking.

[5]Deletes may lead to concatenation of text nodes, but these are stored outside the rid|size|level table, leaving its tuple order intact.

6. QUANTITATIVE ASSESSMENT

In our experiments, we focused on XMark [36], which is the most frequently used benchmark for evaluating XQuery efficiency and scalability. The experimentation platform was a 1.6 GHz AMD Opteron 242 (1 MB L2 cache) processor with 8 GB RAM and a RAID-5 disk subsystem (3ware 7810, configured with eight 250 GB IDE disks of 7200 RPM). The operating system was Linux 2.6.11, using a 64-bit address space. We ran MonetDB/XQuery version 0.10.2 in client-server mode, making use of its physical query plan caching feature. We used the XMark benchmark on documents ranging from 1.1 MB to 11 GB. For all experiments, we report the best of five runs. Once a document is loaded by MonetDB/XQuery, run-time variations tend to be small.

We tested the latest snapshot of eXist (January 2006) [28], a popular open-source XML DBMS. Precise query timings for eXist were extracted from its execution log. The eXist system failed to import the XMark documents larger than 11 MB, stating that the auction document is too large or irregular for its indexing scheme. We also tested Galax (0.5.0) [14], an open-source XQuery system developed in the research community. We tested the standard version without the Jungle storage back-end, such that Galax parses the XML file on each query. We thus used the Galax "monitor" feature to account for the separate query processing phases and —for all systems— excluded document loading as well as result serialization times. Galax failed to process the queries once the XMark documents were beyond a size of 110 MB.

As for commercially developed systems, we tested BerkeleyDB XML 2.2 (BDB) [2] and X-Hive (6.0) [40]. For BDB, we used the timing feature of its administrative console, whereas for X-Hive we executed the XMark queries using a Java program that kept internal timings. For both system we tried to improve performance by creating structure indices as well as value indices (creating those on the XPath paths and text/attribute values used by the XMark queries). For both systems, however, it is hard to increase overall performance, as the effect of an index tends to be that one query becomes faster, while others become slower. Only for X-Hive, we were able to significantly improve a result reported in [12] without negative side-effects, by creating a value index on the path `buyer/@person`. This reduced the run time of XMark query Q8 from quadratic to linear. A similar result could not be achieved for BDB.

Loop-Lifted Staircase Join. Figure 12 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 10-30. Some queries (Q11-14), where path step cost is relatively small, in general benefit less (factor 3-6.5). 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 iterative version. Pushing the nametests below the respective location steps reduces the intermediate result sizes handled by the location steps and thus the overhead. Queries Q6 & 7 perform only one iteration, hence, loop-lifted staircase join does not yield any improvement. However, since both queries perform descendant steps from the document root (respectively a single top-level document node), nametest push-down becomes crucial: Without nametest push-down, location steps produce (almost) the whole document as intermediate result. Once pushed below the location steps, the (quite selective) nametests (now accelerated by indices on the element tables) reduce the intermediate result—and thus the total execution time—significantly.

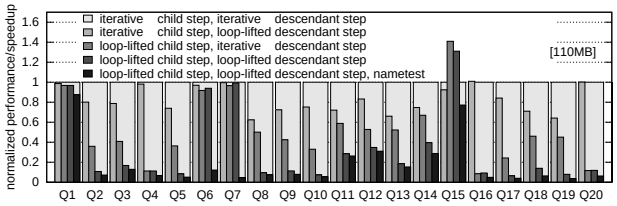


Figure 12: Benefits of loop-lifted staircase join.

Join Optimization. Before we developed the join recognition and evaluation strategies described in Section 4, MonetDB/XQuery was unable to evaluate the XMark join queries Q8–Q12 on document sizes beyond 110 MB. This turned out to be due to the generation of huge intermediate Cartesian products, a consequence of loop-lifting. Figure 13 contrasts the results for the 11 MB document with the performance we obtained with join recognition enabled in our relational algebra generation. This makes clear that queries with join predicates simply *require* join recognition when run on XML documents of significant size.

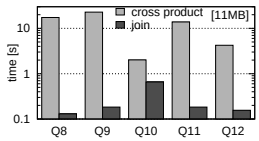


Figure 13: XQuery join optimization.

Sort Reduction. The next step is to analyze the performance improvements that are provided by our peephole optimization approach that maintains column order properties and thus is able to eliminate unnecessary sorting operators, converts full sorts into refine-sorts, and sorting *DENSE_RANK()* operators into a streaming ones. Figure 14 shows that XMark performance on the 110 MB document is boosted by a factor 2 using these order optimizations.

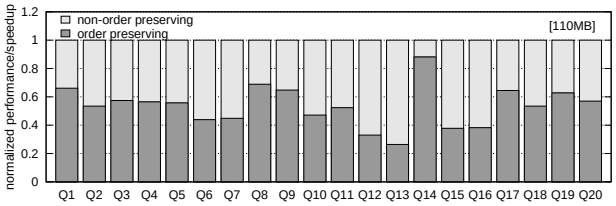


Figure 14: Benefits of Sort Reduction.

Scalability. Figure 15 shows the performance results of MonetDB/-XQuery, 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

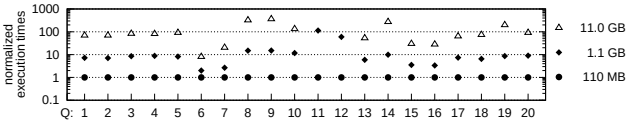


Figure 15: Scalability with respect to document size.

Q	1.1 MB					11 MB					110 MB				1.1 GB			11 GB
	MXQ	Galax	XHive	BDB	eXist	MXQ	Galax	XHive	BDB	eXist	MXQ	Galax	XHive	BDB	MXQ	XHive	BDB	MXQ
1	.013	.000	.170	.007	.011	0.01	0.06	0.37	0.05	0.10	0.12	0.72	1.29	0.51	1.3	9.9	5.9	14
2	.008	.002	.090	.014	.140	0.02	0.03	0.45	0.13	5.67	0.19	0.31	1.75	1.38	1.8	33.0	43.1	19
3	.029	.012	.120	.035	.176	0.14	0.14	0.65	0.34	6.61	1.20	1.76	5.66	3.55	11.5	25.1	37.1	176
4	.013	.026	.070	.042	.378	0.03	0.22	0.10	0.39	15.40	0.42	2.91	1.00	4.07	4.5	18.1	43.3	44
5	.006	.005	.040	.011	2.419	0.01	0.05	0.13	0.10	185.47	0.08	0.63	0.90	1.05	0.8	20.7	11.4	10
6	.003	.117	.100	.107	.002	0.00	1.30	1.07	1.14	0.01	0.00	13.29	10.17	13.23	0.0	178.1	*DNF*	.1
7	.003	.277	.110	.122	.007	0.00	2.68	1.57	1.31	0.01	0.01	30.01	24.84	14.70	0.1	278.4	*DNF*	.6
8	.014	.013	.220	.447	.660	0.04	0.16	0.85	51.21	429.89	0.47	2.12	3.51	9316.72	9.6	49.1	*DNF*	223
9	.022	.113	.580	.407	.783	0.05	113.23	32.25	47.03	333.47	0.52	*DNF*	12280.66	*DNF*	11.8	*DNF*	*DNF*	460
10	.163	.136	.500	.153	16.533	2.54	1.74	5.28	5.15	1559.17	5.18	18.61	442.37	*DNF*	62.8	*DNF*	*DNF*	2413
11	.018	.042	.160	1.26	2.064	0.11	2.62	98.91	121.75	374.46	3.62	*DNF*	19927.29	*DNF*	367.7	*DNF*	*DNF*	*DNF*
12	.044	.028	.310	.486	3.067	0.09	1.44	23.39	118.70	1584.91	2.11	*DNF*	5100.19	*DNF*	121.1	*DNF*	*DNF*	*DNF*
13	.022	.002	.010	.009	.008	0.03	0.03	0.10	0.08	0.03	0.10	0.66	1.03	0.79	0.9	12.9	8.1	8
14	.026	.109	.060	.106	.228	0.12	1.92	0.72	1.07	0.44	0.93	99.53	11.16	14.18	7.5	110.2	*DNF*	452
15	.026	.001	.010	.015	.015	0.03	0.02	0.03	0.13	0.05	0.07	0.20	0.49	1.37	0.4	10.6	28.5	3
16	.030	.003	.010	.016	.597	0.03	0.03	0.03	0.14	22.21	0.08	0.46	0.52	1.52	0.5	10.9	17.6	4
17	.022	.005	.010	.021	.018	0.03	0.06	0.09	0.20	0.18	0.15	0.82	0.85	2.08	1.4	11.8	34.1	31
18	.013	.007	.010	.020	.009	0.02	0.07	0.08	0.19	0.12	0.05	0.73	0.64	2.09	0.5	14.8	21.7	7
19	.029	.089	.070	.056	.037	0.06	1.17	0.67	0.57	0.51	0.38	14.73	12.15	6.74	7.0	254.5	135.6	128
20	.075	.030	.020	.037	.061	0.11	0.28	0.11	0.34	0.98	0.62	2.98	1.40	3.42	7.0	24.6	37.4	70

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

aggregation). *Any* XQuery system is bound to exhibit quadratic scaling with document size on Q11-12. Queries Q6, 7, 15, and 16 actually show sub-linear scaling: Here, the pushed-down nametests benefit from indices used in MonetDB/XQuery.

Other Systems. Table 1 lists our full experimental results. For the 1.1 MB size, average query run time is 150ms or lower on all systems except eXist. On the 11 MB dataset, eXist already becomes slow for Q4-5 and all join queries (Q8-12). On larger data sizes, those join queries also become very slow for BDB and X-Hive (except that, as mentioned, X-Hive can execute Q8 with linear complexity thanks to a value index). Queries that Did Not Finish within an hour are scored as "DNF". Galax is able to handle some joins (Q8,10) in linear time, but at 110 MB the other join queries crashed with *materialization out of bounds* errors.

Shredding and Serialization. To complete the performance assessment of MonetDB/XQuery, we also measured the times for shredding and serializing documents of various sizes. For the latter, we ran a query that constructs a copy of the entire input document. The results revealed that MonetDB/XQuery handles both tasks in interactive time and scales linearly as the documents grow larger, e.g., for the 11 MB and 1.1 GB documents, shredding took 0.84 and 89.69 seconds, and serialization took 1.88 and 190.48 seconds, respectively. Considering that the pre|size|level encoding stores tuples in the same order as they appear in the XML document, shredding causes sequential write access into the relational tables, and serialization sequential read access (the same holds for the property tables, which are either kept small due to duplicate elimination, or if not, are also accessed sequentially). This sequential access leads to relatively fast shredding/serialization.

	System	Source	CPU	MHz	SPEC	Factor
M	MonetDB/XQuery (MXQ)	Tab. 1	Opteron	1600	1068	1.00
E	eXist	[28] Tab. 1	Opteron	1600	1068	1.00
R	BerkeleyDB XML 2.2 (BDB)	[2] Tab. 1	Opteron	1600	1068	1.00
H	X-Hive 6.0	[40] Tab. 1	Opteron	1600	1068	1.00
G	Galax 0.5.0	[14] Tab. 1	Opteron	1600	1068	1.00
Y	Dynamic Interval Encoding	[12]	PentiumIII	1000	451	2.36
I	IPSI-XQ v1.1.1b	[20] [12]	PentiumIII	1000	451	2.36
K	Kweelt	[24] [12]	PentiumIII	1000	451	2.36
Q	QuiP	[34] [12]	PentiumIII	1000	451	2.36
D	Pathfi nder + IBM DB2 UDB V8.1	[17]	Pentium4	2200	780	1.37
F	FluX	[22]	AthlonXP	1670	697	1.53
A	Anonymous commercial system	[22]	AthlonXP	1670	697	1.53
X	TurboXPath	[21]	PentiumIII	700	332	3.22
T	Timber	[32]	PentiumIII	866	411	2.60
L	Li	[26]	PentiumIII	933	421	2.53
Z	Qizx/Open (Version 0.4/p1)	[33] [26]	PentiumIII	933	421	2.53
S	Saxon (Version 8.0)	[35] [26]	PentiumIII	933	421	2.53
B	BEA/XQRL	[15]	Pentium4	1800	669	1.59
V	VX	[8]	Pentium4	1800	669	1.59

Table 2: Sources for XMark results in Figure 16.

Public Experimental Results. In addition to the above experimental comparison, we also collected representative XMark performance results from recent literature in order to perform a broader comparison in Table 2. These results exclude document parsing (shredding) and serialization (except system F, which did not provide separate timings). Since the various experiments were run on different hardware platforms, we divide the original results by the factor that the SPECINT-CPU2000 score of the respective CPU differs from the SPECINT-CPU2000 score of the CPU used in our experiments (see Table 2 for details). This method does assume that all systems ran CPU-bound, but it should bring all results in the right ballpark to enable a rough comparison in a log-scale plot. Figure 16 depicts the results for XMark documents of 11 MB, 110 MB, and 1.1 GB, respectively. For ease of comparison, we show the normalized performance relative to MonetDB/XQuery.

Four systems (B, S, K, Q) were reported to be able to handle only document sizes smaller than 11 MB, depicted by "DNF" in the plots. Next to our own measurements for systems M, E, R, H, and G, only [32] reports results for all 20 XMark queries (for system T on a 110 MB document); we use lines to depict these systems' results in Figure 16. For all other systems, we only found results for selected XMark queries, depicted by points in Figure 16.

This broader comparison confirms the results of our experiment reported above. With simple queries on small documents, the differences between the various systems are rather small. With more complex queries (e.g., the joins queries 8-12), the differences are more significant, even on small documents, and the purely relational approach of MonetDB/XQuery shows to be one of the most efficient solutions. For larger documents, the differences become more significant. For the documents of 1.1ĠB size, only very few results have been published, let alone larger data sizes.

The overall conclusion of the experiments is that MonetDB/XQuery is highly scalable, can handle XPath-intensive queries well (due to the loop-lifted staircase join), handles queries with theta-join predicates in linear time, and appears to outperform the current generation of XQuery systems by quite a big margin.

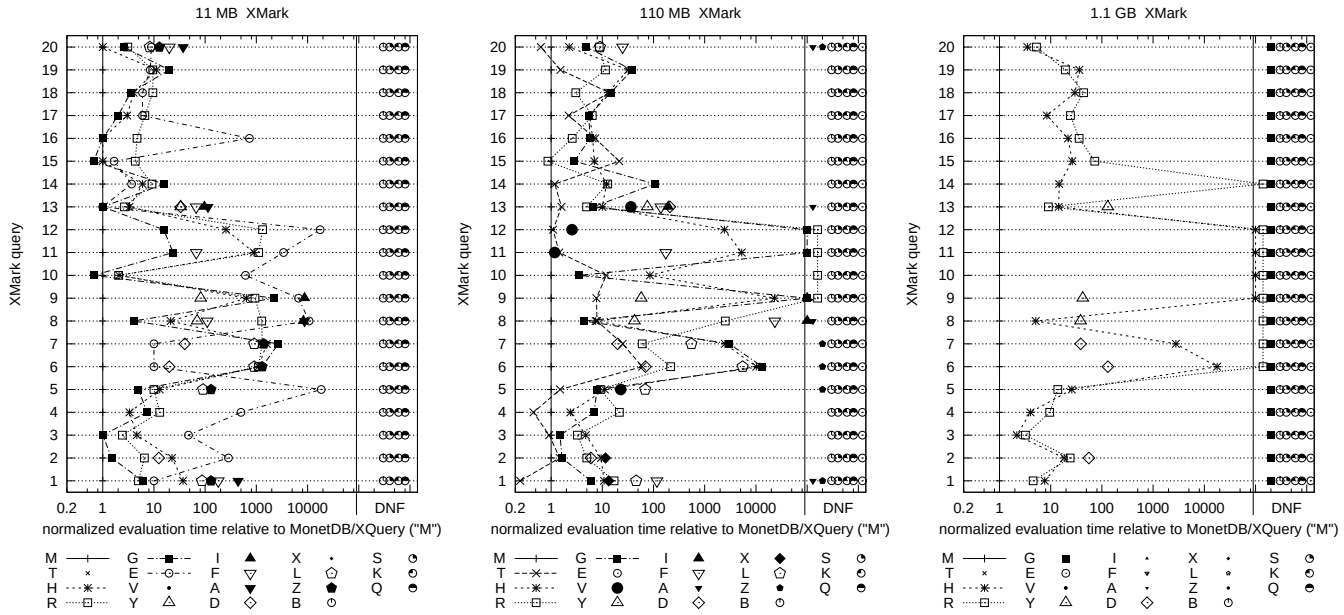


Figure 16: XMark performance comparison of the systems listed in Table 2.

7. RELATED RESEARCH AND SYSTEMS

The present work builds on both an XPath-aware relational encoding of XML trees [19] and relational XQuery compilation techniques [17] to turn a relational database back-end into an XQuery processor. To date, as suggested by a recent survey article [23], this work developed the first instance of a relational XQuery processor that really exhibits the efficiency and scalability needed to process XML input documents of up to 11 GB size in interactive time.

Note that the *node surrogates* γ (Section 2) constitute a rather generic concept: any XML tree encoding that is true to document order and node identity may be used in place of preorder ranks. The database research and industry communities have developed a variety of possible alternatives, among these [30, 25, 41, 38]. Preorder ranks provide node surrogates of fixed byte-width, which greatly simplifies their storage and manipulation. Such fixed-size node encodings are known to incur significant inherent costs if general structural tree updates (*e.g.*, node insertions, subtree deletions) are to be processed [11]. In contrast, variable-length surrogates such as, for example, ORDPATH labels [30], are designed to allow "low-cost" updates while still encoding document order. However, these features come at the expense of higher storage and manipulation costs, and, more importantly, non-denseness, prohibiting the application of staircase join for efficient location step evaluation.

In our work, we do not assume any knowledge of the schema of the XML documents stored in the database (as ,*e.g.*, in [9]). In [12], the authors describe a similar schema-free XQuery compiler that was originally designed to emit SQL code. Since the compiler is aware of the XQuery order semantics—at least partially: the system does not distinguish between sequence and document order—the generated SQL queries contain the expected necessary yet significant sorting overhead. Experiments with a prototypical relational engine led the authors to observations that match those we have made here: *(i)* built-in order-awareness and *(ii)* XQuery join recognition are key features if the system is to process XML documents of serious size. Performance-wise, we really reap the benefits of using an extensible RDBMS kernel as an XQuery runtime environment: the XMark benchmark figures obtained here surpass those reported in [12] by two orders of magnitude.

Recently, the major commercial RDBMS vendors have started to roll out XQuery capable extensions of their products SQL Server [31], Oracle [27] and DB2 [3]. All extend the SQL type system with a new XML type, that can be queried using XQuery expressions or mixed with SQL in SQL/XML. It is interesting to note the architectural diversity of these three implementations: DB2 chose to implement a separate XML-specific evaluation engine, Oracle followed a hybrid approach where some (/most) XQuery queries can be compiled to relational algebra and the rest is handled by a separate XQuery processor, whereas SQL Server compiles *all* XQuery expressions into relational algebra plans. In that sense, SQL Server most resembles our approach, though it uses the ORDPATH node encoding [30], a compressed variant of update-friendly Dewey codes. This represents in our eyes a trade-off that favors update performance over query performance; in the future we hope to obtain insight in the true performance characteristics of these systems.

Quite timely, the order-aware optimization of relational queries has received renewed attention [39, 29]. Inspired by the foundational work on "interesting orders" in System R and based on the idea to derive order properties of intermediate results from functional dependencies introduced by the application of operators of the relational algebra [37], Wang and Cherniack describe order property inference rules [39]. These rules are capable of inferring *secondary orderings*, *i.e.*, minor orderings respected in a group of tuples. As described, possibilities to exploit such orderings, here denoted by $grpord([O_1], O_2)$, pervade in the algebraic plans emitted by our XQuery compiler.

As mentioned at the end of Section 3, loop-lifted staircase join is related to the Structural Join [1] and Holistic Twig Joins [7] algorithms. Both Structural Join and Holistic Twig Join could be used in a loop-lifted scenario with the iter|pre table of context nodes as one of their input tables (sorted on pre). However, neither algorithm is aware of the different iterations and thus does not perform *(i) pruning* within each iteration. Consequently, duplicate nodes will be in their output, mandating the use of duplicate elimina-

tion 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 Holistic algorithms 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 `child` step, staircase join skips over all descendants of a child to arrive at the next, whereas both Structural Join and Holistic Twig Join perform a child-test on all descendants. Similar skipping opportunities arise in the other XPath axes; which are all supported by loop-lifted staircase join.

8. CONCLUSIONS

In this work, we have described a new XML database system built purely on relational database technology. It provides a full XQuery implementation supporting both document and sequence order (including such features as XQuery modules and recursive user defined functions). We have also outlined an update scheme that addresses the difficult task of efficiently maintaining a range-based XML numbering scheme under structural updates. The main ideas behind this mechanism are a page-wise indirection scheme to limit the impact of node shifts to a single *logical* page, and the use of deltas to record changes in the *size* property of XML tree nodes, which avoids locking the root node during the entire transaction.

One of the striking features of this system is its outstanding query performance, especially when large XML documents are traversed with XPath location steps and when joins are involved. We compared this performance on the XMark benchmark both with some openly available XML database systems that can handle large single documents and with other (not openly available) systems that were described in literature. Our conclusion of this extensive performance evaluation is that MonetDB/XQuery is among the fastest and most scalable, thus showing that relational technology can indeed be leveraged in XML databases, without strict need for a native XML storage subsystem nor a native query processing algebra.

We described two main contributions that improved performance by an order of magnitude. First, the *loop-lifted staircase join* adapts the original staircase join as proposed in [19] for XPath location step evaluation, for use in XQuery. The crucial change is to add a stack-based mechanism to allow execution of XPath location steps for multiple sequences of context nodes—*e.g.*, when XPath expressions are nested in XQuery `for`-loops—in a single sequential pass. Second, a carefully designed *join processing* framework reduces the quadratic complexity of XQuery joins to scale linearly with the input document size, exploiting the existential semantics of XQuery joins. Join optimization is part of a wider *property-driven* peephole optimization framework that also is used to avoid unnecessary sorting, and to detect join patterns in XQuery queries in a way that is immune to any syntactic variation.

A MonetDB feature that we think would also benefit relational XQuery in other RDBMS's is *positional lookup*, which finds a record by address computation (without index access). This is applicable on (foreign) key access to a densely increasing (0, 1, 2, ...) integer column such as generated by a SQL `autoincrement` column, that is not updated. Such columns are found in intermediate result tables (pervasive in our XQuery translation) and in persistent append-only tables such as pre|size|level. This benefits foreign key joins, but also staircase join (positional skipping) as well as the page-wise update scheme (fast $pre - rid$ swizzling).

MonetDB/XQuery is available in open source as a fast and scalable XQuery system for application developers, but also as experimentation platform for future research on relational XQuery.

9. REFERENCES

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Effi cient XML Query Pattern Matching. In *ICDE*, pages 141–152, 2002.

[2] Berkeley DB XML. `http://www.sleepycat.com/products/bdbxml.html`.

[3] K. Beyer et al. System RX: One Part Relational, One Part XML. In *SIGMOD Conf.*, 2005.

[4] P. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, UVA, 2002.

[5] P. Boncz et al. Pathfi nder: Relational XQuery Over Multi-Gigabyte XML Inputs In Interactive Time. Technical Report INS-E0503, CWI, 2005.

[6] P. Boncz, S. Manegold, and J. Rittinger. Updating the Pre/Post Plane in MonetDB/XQuery. In *XIME-P*, 2005.

[7] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XMLPattern Matching. In *SIGMOD Conf.*, pages 310–321, 2002.

[8] P. Buneman, B. Choi, W. Fan, R. Hutchison, R. Mann, and S. Viglas. Vectorizing and Querying Large XML Repositories. In *ICDE*, 2005.

[9] M. Carey, D. Florescu, Z. Ives, Y. Lu, S. J., E. Shekita, and S. S. XPERANTO: Publishing Object-Relational Data as XML. In *WebDB*, 2000.

[10] D. Chamberlin, D. Florescu, and J. Robie. XQuery Update Facility. W3C, 2006. `http://www.w3.org/TR/xqupdate/`.

[11] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Trees. In *ACM Symposium on Principles of Database Systems*, 2002.

[12] D. DeHaan, D. Toman, M. Consens, and M. Öszu. A Comprehensive XQuery to SQL Translation Using Dynamic Interval Encoding. In *SIGMOD Conf.*, 2003.

[13] A. Deutsch and V. Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *VLDB Conf.*, 2003.

[14] M. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax Experience. In *VLDB Conf.*, 2003.

[15] D. Florescu et al. The BEA/XQRL Streaming XQuery Processor. In *VLDB Conf.*, 2003.

[16] T. Grust. Purely Relational FLWORs. In *Proc. XIME-P*, 2005.

[17] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *VLDB Conf.*, 2004.

[18] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *VLDB Conf.*, 2003.

[19] T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath evaluation in Any RDBMS. *ACM Trans. on Database Systems*, 29(1), 2004.

[20] IPSI-XQ. `http://ipsi.fhg.de/oasys/projects/ipsi-xq/`.

[21] V. Josifovski, M. Fontoura, and A. Barta. Querying XML streams. *The VLDB Journal*, 14(2), 2005.

[22] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams. In *VLDB Conf.*, 2004.

[23] R. Krishnamurthy, R. Kaushik, and J. Naughton. XML-to-SQL Query Translation Literature: The State of the Art and Open Problems. In *Proc. XSym*, 2003.

[24] Kweelt. `http://kweelt.sourceforge.net/`.

[25] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *VLDB Conf.*, 2001.

[26] X. Li and G. Agrawal. Effi cient Evaluation of XQuery over Streaming Data. In *VLDB Conf.*, 2005.

[27] Z. Liu, M. Krishnaprasad, and V. Arora. Native Xquery processing in Oracle XMLDB. In *SIGMOD Conf.*, 2005.

[28] W. Meier. eXist: An Open Source Native XML Database. In *Web, Web-Services, and Database Systems*, volume 2593 of *LNCS*, 2003.

[29] G. Moerkotte and T. Neumann. A Combined Framework for Grouping and Order Optimization. In *VLDB Conf.*, 2004.

[30] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATH: Insert-Friendly XML Node Labels. In *SIGMOD Conf.*, 2004.

[31] S. Pal et al. XQuery Implementation in a Relational Database System. In *VLDB Conf.*, 2005.

[32] S. Paparizos and H. Jagadish. Pattern tree algebras: sets or sequences? In *VLDB Conf.*, 2005.

[33] Qizx/open. `http://www.xfra.net/qizxopen/`.

[34] QuiP. `http://developer.softwareag.com/tamino/quip/`.

[35] Saxon. `http://saxon.sourceforge.net/`.

[36] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *VLDB Conf.*, 2002.

[37] D. Simmen, E. Shekita, and T. Malkemus. Fundamental Techniques for Order Optimization. In *SIGMOD Conf.*, 1996.

[38] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *SIGMOD Conf.*, 2001.

[39] X. Wang and M. Cherniack. Avoiding Sorting and Grouping in Processing Queries. In *VLDB Conf.*, 2003.

[40] X-Hive/DB. `http://www.x-hive.com/`.

[41] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *SIGMOD Conf.*, 2001.