Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

INS

Information Systems

INformation Systems

Pathfinder: relational XQuery over multi-gigabyte XML inputs in interactive time

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

REPORT INS-E0503 MARCH 2005

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-3681

Pathfinder: relational XQuery over multi-gigabyte XML inputs in interactive time

ABSTRACT
Using a relational DBMS as back-end engine for an XQuery processing system leverages relational query optimization and scalable query processing strategies provided by mature DBMS engines in the XML domain. Though a lot of theoretical work has been done in this area and various solutions have been proposed, no complete systems have been made available so far to give the practical evidence that this is a viable approach. In this paper, we describe the ourely relational XQuery processor Pathfinder that has been built on top of the extensible RDBMS MonetDB. Performance results indicate that the system is capable of evaluating XQuery queries efficiently, even if the input XML documents become huge. We additionally present further contributions such as loop-lifted staircase join, techniques to derive order properties and to reduce sorting effort in the generated relational algebra plans, as well as methods for optimizing XQuery joins, which, taken together, enabled us to reach our performance and scalability goals.

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

Pathfinder: Relational XQuery Over Multi-Gigabyte XML Inputs In Interactive Time

Peter Boncz[△] Torsten Grust[□] Stefan Manegold[△] Jan Rittinger[◇*] Jens Teubner[◇]

[△]CWI Amsterdam, Netherlands {boncz, manegold}@cwi.nl [□]Tech. Univ. of Clausthal, Germany grust@in.tu-clausthal.de [◇]Univ. of Konstanz, Germany {rittinge, teubner}@inf.uni-konstanz.de

ABSTRACT

Using a relational DBMS as back-end engine for an XQuery processing system leverages relational query optimization and scalable query processing strategies provided by mature DBMS engines in the XML domain. Though a lot of theoretical work has been done in this area and various solutions have been proposed, no complete systems have been made available so far to give the practical evidence that this is a viable approach. In this paper, we describe the *purely relational XQuery processor Pathfinder* that has been built on top of the extensible RDBMS *MonetDB*. Performance results indicate that the system is capable of evaluating XQuery queries efficiently, even if the input XML documents become huge. We additionally present further contributions such as loop-lifted staircase join, techniques to derive order properties and to reduce sorting effort in the generated relational algebra plans, as well as methods for optimizing XQuery joins, which, taken together, enabled us to reach our performance and scalability goals.

1998 ACM Computing Classification System: H.2.4, H.2.3, H.2.2, E.1

Keywords and Phrases: XML, XQuery, XPath Accelerator, relational XQuery evaluation, loop-lifted staircase join, order-aware physical relational algebra, RDBMS, MonetDB

Note: Work carried out under projects MultimediaN N3 "Ambient Multimedia Databases" and Bricks IS2 "Petabyte Data Mining".

1. INTRODUCTION

Since XQuery has been proposed as the W3C standard query language for XML data, both the database research community and industry have been trying to create systems that efficiently implement XQuery on large XML databases. This poses a non-trivial challenge due to the inherent tree structure of XML data coupled with the iterative and recursive nature of the XQuery language.

Roughly, there are two approaches being pursued. The bottom-up variant of the "native" approach to XQuery processing advocates the use of the tree data structure as a basic building block deep in the database storage system. These tree data structures are then processed with tree algebras. The top layer of such systems translates XQuery into algebraic plans [JLST01]. A top-down variant of the native approach are "stream" oriented systems that enrich an XML parser with on-the-fly query processing functionality, which may be supported by tree algebra operators and tree data structures [KSSS04, FHK+04].

In contrast, we follow the alternative approach of supporting XQuery using relational database technology, in order to leverage mature relational query optimization techniques as well as efficient and scalable relational query processing operators. XML documents are stored in "shredded" form in relational tables in a schema-oblivious fashion and XQuery expressions are translated into relational algebra queries.

In this paper, we report on *Pathfinder powered by MonetDB*, a full-fledged implementation of this approach. We build on earlier work on relational storage schemes for XML that allow for efficient evaluation of all XPath axes ("staircase join" [GvKT03]) and a mapping of XQuery language constructs onto relational alge-

*Work was done while visiting CWI.

bra [GST04]. The final outcome is an open-source system[1] that provides a serious implementation of XQuery, currently using the MonetDB open-source RDBMS as its back-end [BK99].

Contributions. Our main contribution is to provide empirical evidence that the "relational approach" works. The performance results presented here greatly surpass those of all other known XQuery systems. Also, unsurpassed scalability is demonstrated by the interactive time in which all XMark queries can be run on XML input documents of up to 11 GB data size. Few other systems support this size, let alone in interactive time.

In order to get these results, we have extended the state of the art of "relational XQuery" with the following additional contributions:

Loop-Lifted Staircase Join. The *loop-lifted* staircase join is an extension of the staircase join, which is necessary to be able to efficiently apply this technique—originally developed in the XPath domain—in XQuery. Briefly, staircase join is fast because it can evaluate XPath axis steps in a single sequential pass over the shredded document. However, in XQuery, path expressions may appear nested in `for` -loops, which ultimately leads to many sequential passes (one for each loop iteration). The loop-lifted staircase join allows execution of XPath steps for multiple sequences of context nodes in a single pass by keeping more state and exploiting XPath axis properties.

Join Optimization. In XQuery, a join may be expressed using a variety of equivalent syntactic forms. We introduce a simple yet effective join recognition logic which operates on the level of normalized XQuery Core expressions and uses information on free query variables to emit efficient relational joins for such queries. XQuery's general comparison operators (`<`, `=`, ...) have existential semantics. This is exploited by Pathfinder if these operators are used in join predicates.

Physical Algebra and Order Awareness. In contrast to relational algebra plans that evaluate SQL queries, plans for XQuery tend to impose tight control over tuple ordering *throughout* query processing. We build on previous work for denoting and deriving sorting properties over physical relational algebra operators [SSM96, WC03] with a generalized secondary ordering criterion $A|B$, meaning "for all equal values in column B, the values in column A are sorted". This turns out to be the minimal property over relational tuples required to retain XQuery sequence order. Also, we identify a new pipelined relational sorting operator called `RefineSort` that extends an existing ordering with additional attributes by sorting in a chunk-like fashion. Taken together, these contributions allow us to reduce the cost of sorting in the generated relational plans.

This paper is structured as follows. Section 2 is a refresher of staircase join and the XQuery-to-relational-algebra compilation approach we employ. Sections 3, 4, and 5 discuss our contributions in the area of loop-lifted staircase join, join optimization, and physical algebra (and order awareness), respectively. In Section 6, we delve into some MonetDB-specific details of our implementation, before discussing performance and scalability experiments in Section 7. We wrap up by discussing related work in Section 8, and outlining our conclusions and future work in Section 9.

2. RELATIONAL XQUERY

Obviously, there is a gap between the tabular data model supported by the back-end relational database system and the two principal data types which form the backbone of the XQuery data model, namely *ordered, unranked trees of nodes* and *ordered, finite sequences of items*. To bridge this gap, we introduce *relational encodings* of both data types and then describe how the relational back-end may act as an XQuery processor by manipulating these encodings. 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.

Encoding Item Sequences. 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 identified. We will use a relational sequence encoding that explicitly reflects sequence order by means of

[1] Pathfinder will be released with MonetDB/XQuery.

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* either is 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

Nodes are represented by surrogates that can reflect *document order* and *node identity*: for two nodes v_1, v_2 and their surrogates $\gamma_{v_1}, \gamma_{v_2}$, we require $v_1 \ll v_2 \Leftrightarrow \gamma_{v_1} < \gamma_{v_2}$ and v_1 `is` $v_2 \Leftrightarrow \gamma_{v_1} = \gamma_{v_2}$. The database community has devised a variety of ways to implement such node representations [LM01, OOP+04], below we will discuss one alternative, *preorder ranks*, in more detail.

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, but Section 6 will discuss how a back-end that implements monomorphic columns only may support this encoding nevertheless.

pos	item
1	2
2	"x"
3	γ_a

Encoding XML Fragments. In [GvKT03], we 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* and yields node surrogates exhibiting the required properties. Here, we use an equivalent encoding variant in which the location of a node v in the document tree is represented as the 3-tuple $\langle pre(v), size(v), level(v)\rangle$, 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 $post(v) = pre(v) + size(v) - level(v)$.) Since the unique preorder traversal rank $pre(v)$ reflects document order, we set $\gamma_v \equiv pre(v)$. Figure 1 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 6. This tree representation exhibits a number of salient characteristics, among which element (or tree) construction through pasting of encodings [GT04] and highly efficient XPath processing are especially relevant in the XQuery context. The *staircase join* [GvKT03] evaluates XPath location steps for a given sequence of context nodes by means of a single sequential scan over the tree encoding table. During the scan, staircase join exploits the isomorphism between the XML tree and its tabular encoding to *(i)* reduce the context sequence size, *(ii)* completely avoid the generation of duplicate nodes in the result, and *(iii)* skip considerable parts of the encoded tree. An off-the-shelf B-tree index on columns pre|size supports the operation of staircase join perfectly and also guarantees that the resulting output node sequence may directly be emitted in document order as required by the XPath semantics.

```
<a>
  <b/>
  <c>
    <d/><e/>
  </c>
</a>
```

pre	size	level
0	4	0
1	0	1
2	2	1
3	0	2
4	0	2

Figure 1: Tree encoding.

Relational XQuery Evaluation

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 [DTCÖ03, DT03], but here we will adopt the approach developed in [GST04, GT04]. The resulting system can compile the XQuery Core dialect sketched in Figure 2. We will briefly review the fundamental ideas of this compilation strategy here—refer to [GST04, GT04] for details.

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:

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

We have already seen the representation of the sequence $(x_1, x_2, \ldots, x_n)$ above. Loop body e is evaluated in n iterations, with variable $\$v$ successively bound to exactly one x_i in each iteration. A relational representation

atomic literals	document order ($e_1 \ll e_2$)
sequences (e_1, e_2)	node identity (e_1 is e_2)
variables ($\$v$)	arithmetics (+,-,*,idiv ,...)
let $\$v$:= e_1 return e_2	comparisons (eq, lt ,...)
for $\$v_1$ [at $\$v_2$] in e_1 return e_2	Boolean connectives (and , or)
if e_1 then e_2 else e_3	fn:doc(e)
typeswitch e_1 case e_2 default e_3	fn:root(e)
element $\{e_1\}\{e_2\}$	fn:data(e)
text $\{e_1\}\{e_2\}$	fn:distinct-doc-order(e)
e_1 order by $e_2, \ldots, e_n$	fn:count(e), fn:sum(e)
XPath ($e/\alpha[[e]]$)	fn:empty(e)
function application	fn:position(e), fn:last(e)

Figure 2: Supported XQuery Core dialect, expressions may be composed arbitrarily; α denotes an XPath axis.

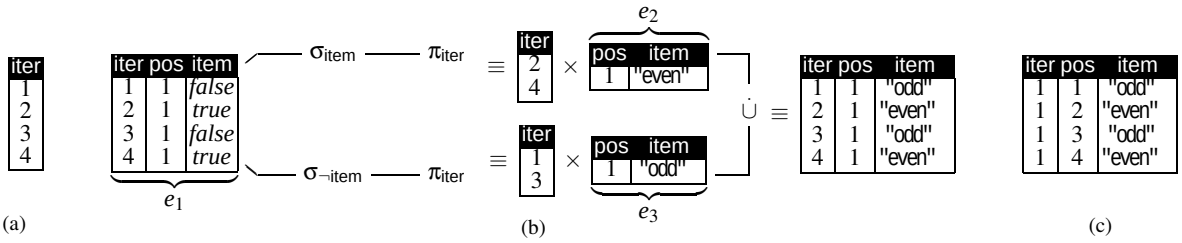

Figure 3: Relational XQuery evaluation: (a) loop relation in scope s_v, (b) evaluation of conditional, (c) final result in outermost scope.

of $\$v$ would thus be the relation shown here. This iter|pos|item encoding will be pervasive in the following: a tuple $\langle i,p,x\rangle$ in this representation 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.

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

The *row-numbering* in step *(i)* is characteristic for this approach and we assume the availability of a relational operator $\rho_{A:C_1,\ldots,C_n/C_g}(R)$ that, for each 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] Section 5 will discuss measures to implement this operation efficiently (or to avoid it, if possible).

Note how the iter column encodes the iteration performed by the for -loop. The most important 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 right. 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 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
1
2
⋮
n

iter	pos	item
1	1	42
2	1	42
⋮	⋮	⋮
n	1	42

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

$$\texttt{for } \$v_1 \texttt{ in } (x_1, x_2, \ldots, x_n) \texttt{ return } \begin{cases} \texttt{for } \$v_2 \texttt{ in } (y_1, y_2, \ldots, y_m) \texttt{ return} \\ \quad s_{v_1 \cdot v_2} \{ e \end{cases}_{s_{v_1}},$$

[2] As observed in [GST04], ρ exactly embodies the functionality of SQL:1999's OLAP extension DENSE_RANK() OVER (ORDER BY $C_1, \ldots, C_n$ PARTITION BY C_g) AS A.

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

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 the subject of 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

```
for $v in (3,4,5,6)   return
  s_v { if ($v mod 2 eq 0) then "even" else "odd" }
          \_____e1_____/      \__e2__/   \__e3__/
```

The current loop relation in scope s_v is shown in Figure 3(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 3(b). In the third iteration, for example, the predicate evaluates to the single item *false*. Dependent on the outcome of the predicate, in one iteration of the loop 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 3(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 [GST04]) then yields the final result sequence of length 4 (Figure 3(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, $\bowtie$, $\times$, $\setminus$, $\dot{\cup}$ (disjoint union) as well as a means to evaluate arithmetic and comparison operators.

3. Embedded XPath Evaluation

The efficient evaluation of XPath location steps—rooted in arbitrarily located context nodes and along arbitrary axes—proved to be quite a challenging research topic in itself. Evaluating XPath sub-expressions embedded in XQuery expressions adds yet another twist.

In general, XQuery expressions occur in nested iteration scopes (Section 2) and the same is true for XPath sub-expressions. Consider the query

```
for $v in (x1, x2, ..., xn) return
  e($v)/descendant::t                    ,        (Q1)
```

in which $e(\$v)$ denotes an expression that yields a context node sequence for each of the n bindings of variable $\$v$. This initiates n XPath traversals along the descendant 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.

Rephrased in the terminology of our relational XQuery processor, the evaluation of expression $e(\$v)$ will yield a relation like the one depicted here: in each of the n iterations, $e(\$v)$ evaluates to a sequence of node preorder ranks $(\gamma_{i,1}, \ldots, \gamma_{i,s_i})$ of length s_i $(1 \leqslant i \leqslant n)$. (Note that we might have $s_i = 0$ for some i: in this case, no tuple with iter value i will occur.)

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}

As already mentioned in Section 2, we decided to implement XPath location step evaluation based on *staircase join*, a carefully tuned join algorithm that can evaluate an XPath location step for a whole 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 as many sequential scans over the document encoding table. This surely seems wasteful.

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*, a variant of staircase join [GvKT03]. 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. Much of the efficiency of loop-lifted staircase join is due to its strict sequential table access discipline. Section 7 reports on performance improvements of approximately a factor of 5. For space reasons, we limit our discussion to the XPath `descendant` axis. Similar, if not identical, adaptions apply to the remaining XPath axes supported by staircase join [GvKT03].

First, note that the sequence order of the input context node sequences is *irrelevant* for XPath location step evaluation: the XPath semantics [BBC+04] require the result of any location step to be in document order regardless of the context node sequence order. Column pos in the context sequence encoding is thus completely ignored (neither inspected nor generated) by the loop-lifted staircase join algorithm `ll_scj_desc` shown in Figure 4: the join algorithm receives the tabular XML representation *doc* as well as an encoding of the *n* context sequences in a binary iter|item table (column pos projected away) and emits an iter|item table, say *R*. Once the join completes, the required explicit sequence order information, *i.e.*, the pos column, in the final result may then be derived from the document order encoded in the node surrogates stored in the item column (in other words: document order determines sequence order in XPath results):

$$\rho_{\mathsf{pos:item/iter}}\,(R)\ .$$

Since the loop-lifted staircase join algorithm emits the result table in [item] | [iter] order (in each iter group, the item column is sorted in document order, see Section 5 for a detailed discussion of such order properties), this final row-numbering step may be implemented by means of the cheap `MergeRowNumber` physical algebra operator (Section 5).

Loop-lifted staircase join performs a *single* sequential forward scan over the document encoding as well as the context sequences, regardless of the number of iterations. To achieve this, the algorithm of Figure 4 reorders the input iter|item table on [item, iter], *i.e.*, brings it into document 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 `descendant` axis, an iter|item 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. If a descendant node v of c is found, $\langle i, v\rangle$ is appended to the *result* table for all distinct active iterations i, sorted by i (procedure `inner_loop_desc`). This avoids the generation of duplicate result nodes. Finally, loop-lifted staircase join generates the requested ordering by sorting *result* on [iter, item].

When processing a multi-step path, we do skip this final sort for all but the last step in the path, leaving *result* in [item, iter] order. Thus, the subsequent step gets its input already in [item, iter] order, *i.e.,* we also save the initial sort in all but the first step in the path.

4. XQuery Join Processing

Much like in the relational setting, efficient join evaluation remains a tough problem for XQuery processors. In reports on XMark benchmark [SWK+02] performance, for example, XQuery systems typically exhibit quadratic complexity for queries with join predicates (Q8–Q12, see Table 2).

4.1 Join Recognition

In Pathfinder, the problem surfaces as a direct consequence of the loop-lifting technique introduced in Section 2. The canonical algebraic plan for query

$$s_0\left\{\begin{array}{l}\texttt{for \$}u\texttt{ in (30, 20)}\\ s_u\left\{\begin{array}{l}\texttt{for \$}v\texttt{ in (1, 2, 3)}\\ s_{u\cdot v}\left\{\begin{array}{l}\texttt{where \$}u\texttt{ eq \$}v\texttt{ * 10}\\ \texttt{return "match"}\end{array}\right.\end{array}\right.\end{array}\right. \qquad (Q_2)$$

is shown in Figure 5. The loop body in scope $s_{u\cdot v}$ is evaluated 6 times and the loop-lifted relational representations of the subexpressions `$u` and `$v * 10` on the very left of Figure 5 reflect this explicitly. In effect, the plan

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

```
ll_scj_desc (doc : TABLE(pre, size), iter_item : TABLE(iter, item))
 BEGIN
  iter_item ← SORT iter_item ON (item, iter);  /* doc. order */
  result ← NEW TABLE(iter, item);  /* the result */
  active ← NEW STACK(iter, eos);  /* stack of active iters */
  nxt ← FIRST TUPLE FROM iter_item;
  lst ← LAST TUPLE FROM iter_item;
  WHILE (nxt ≤ lst) DO  /* iterate over all context nodes */
   cur_item ← nxt.item;
   nxt ← NEXT TUPLE FROM iter_item;
   /* push all not yet active iters of the current item on stack */
   WHILE (nxt ≤ lst AND nxt.item = cur_item) DO
    IF (nxt.iter NOT ON active) THEN
     eos ← nxt.item + doc[nxt.item].size;
     PUSH ⟨nxt.iter, eos⟩ ON active;
    nxt ← NEXT TUPLE FROM iter_item;
   IF (nxt ≤ lst) THEN
    IF (nxt.item ≤ TOP(active).eos) THEN
     /* next node is descendant of current node
        (TOP(active)); find all results til nxt */
     cur_item ← inner_loop_desc(cur_item, nxt);
    ELSE
     /* next node is no descendant of current node,
        finish all active scopes that end before nxt */
     WHILE (TOP(active).eos < nxt.item) DO
      cur_item ← finish_scope_desc(cur_item);
  /* no context node left; finish all remaining active scopes */
  WHILE (active IS NOT EMPTY) DO
   cur_item ← finish_scope_desc(cur_item);
  result ← SORT result ON (iter, item);  /* result order */
  RETURN result;
 END

finish_scope_desc (first)
 BEGIN
  eos ← TOP(active).eos;
  /* find all results in the current scope */
  first ← inner_loop_desc(first, eos);
  /* back to enclosing scope: remove all iters that are done */
  WHILE (TOP(active).eos ≤ eos) DO
   POP(active);
  RETURN first;
 END

inner_loop_desc (first, last)
 BEGIN
  /* iterate over all doc nodes in the given preorder range */
  FOR v FROM first TO last DO
   /* add a result tuple to each active iter */
   FOREACH DISTINCT ⟨i, _⟩ ON active DO
    APPEND ⟨i, v⟩ TO result;
  RETURN last + 1;
 END
```

Figure 4: Loop-lifted staircase join: descendant axis.

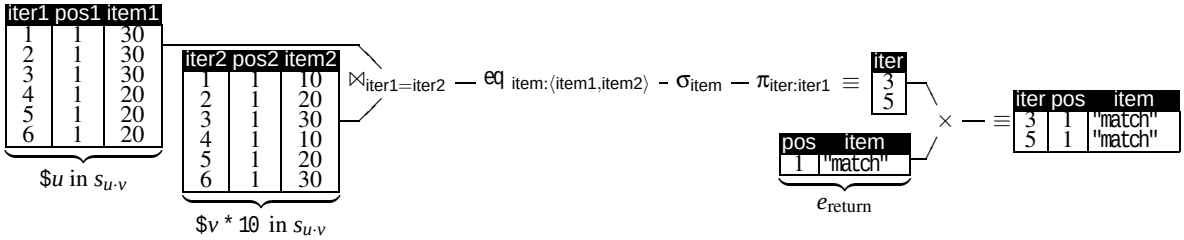

Figure 5: Relational evaluation of XQuery example Q_2. Evaluation of predicate `eq` requires both operands (`$u` and `$v * 10`) to be represented loop-lifted with respect to the inner `for` loop. `iter` values that satisfy the `eq` predicate form the loop relation for the loop-lifted evaluation of the `return` clause.

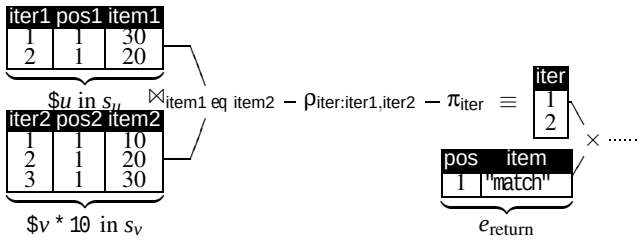

Figure 6: Join plan for the evaluation of example Q_2. Both join operands are computed independently. The join result ultimately serves as relation `loop` to compile the `return` part.

computes the Cartesian product of the two input sequences `(30,20)` and `(1,2,3)` participating in the join, ultimately leading to the scalability problems mentioned above.

The key observation here is that if the inputs to such an XQuery join may be evaluated *independently* of each other—for Q_2 this is trivial since the inputs are constant item sequences—loop-lifting may be avoided. This independence observation is then used by the compiler to emit a relational join between the two inputs instead. Pathfinder recognizes such join scenarios on the level of normalized and simplified XQuery Core expressions such that the syntactic diversity of XQuery does not impact join recognition. For example, this detects all joins in the XMark benchmark set [SWK+02] as well as those joins listed in Appendix G.1 ("Joins") of the W3C XQuery Working Draft [BCF+04].

The join recognition logic is triggered whenever a XQuery Core pattern

```
for $v in e_in
  return if (p(e1,e2)) then e_return else () .        (P1)
```

is found at arbitrary query nesting depth. Independence is then detected based on the presence of free variables and their binding sites. If *(i)* variable `$v` does not appear free in e_1[5], *(ii)* variables occurring free in e_2 and e_{in} are bound in any enclosing scope, except for the scope that *directly* encloses P_1, and *(iii)* predicate p is supported by the theta-join implementation of the relational back-end (*i.e.*, typically, p will be `eq`, `lt`, ..., or one of the XQuery general comparison operators with existential semantics like `=`, `<`; see Section 4.2), then e_{in}, e_1, e_2, e_{return} are *not* loop-lifted. Instead, the compiler directly emits a p-theta-join. Figure 6 depicts the resulting plan for the example query Q_2. Section 7 sheds light on the effectiveness of this approach.

4.2 Existential Semantics of Join Predicates

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, Pathfinder generates the corresponding relational plans

[5]The roles of e_1, e_2 may be arbitrarily swapped.

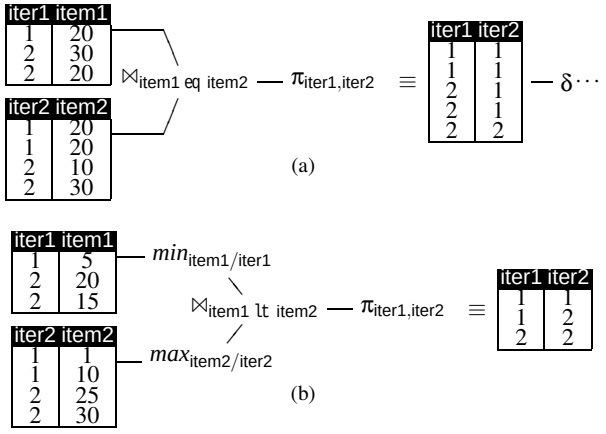

Figure 7: Implementing the existential semantics of XQuery's general comparison operators: (a) duplicate elimination after join, (b) join pushed beyond aggregates.

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                    (Q3)
  return  $u ,
```

which qualifies as a join in the sense of Section 4.1, *i.e.*, $u does not occur free in e_2. Figure 7(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). Pathfinder executes a theta-join using the corresponding *value comparison* (here: eq) in the predicate. In general, this leads to duplicate $\langle$iter1, iter2$\rangle$ pairs in the join result since we were comparing sequences. A subsequent duplicate elimination (operator δ) reduces this to unique pairs and thus implements the required existential semantics. In anticipation of the following section, note that the physical algebra used by Pathfinder (on MonetDB) is order-aware and ensures that intermediate result relations are sorted on [iter, pos] (and thus iter). The theta-join respects the order of its inputs such that δ will be applied to a sorted relation, making this step particularly efficient.

Further, if one of the general comparison operators $\{<, \leq, \geq, >\}$ is used in a join predicate, the generation of duplicates may be avoided a priori. Consider Figure 7(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 iter order of both input relations. After aggregation, the iter values in both join inputs will be unique. The theta-join will thus deliver unique $\langle$iter1, iter2$\rangle$ pairs directly.

5. Physical Algebra and Order Awareness

The mapping of XQuery onto relational query plans, as described in Section 2, maintains order information in the relations that encode the input XML documents and in the (intermediate) results of XQuery expressions. Document order is reflected by the node surrogates (preorder ranks). Likewise, sequence order is maintained in the iter and pos columns of the intermediate relational results.

This mapping would allow keeping track of order information without the need for explicit sorting in the physical relational query plans. However, it turns out that all conceivable implementations of the row-numbering operator $\rho_{A:C_1,\ldots,C_n/C_g}(R_{in})$ impose some ordering on their input relation as a precondition. This may thus lead to a large number of sorts in the physical plans.

RDBMS query optimizers often decorate physical query plans with order properties. This idea stems from System R [SAC+79] ("interesting orders") and was significantly extended to annotate each physical operator with rules for propagation of multi-column order and also grouping information in [SSM96, WC03]. The

authors show that, when known functional (key) dependencies and order propagation are combined, the DBMS query optimizer can detect that certain required orderings are already present. This may lead to significant sort pruning.

5.1 Order Propagation Framework

A lexicographical ordering O on columns C_1 (major) and C_2 (minor) of a relation R is denoted $O = [C_1, C_2]$. The *refinement* of an ordering is expressed with $+$:

$$[C_1, C_2] + [C_3] = [C_1, C_2, C_3] \quad .$$

All orderings that hold on a relation R are denoted $orders(R) = \{O \mid O \text{ holds for } R\}$. This ordering framework is similar to [WC03]. However, we introduce $O_1|O_2$ as a generalization of the "secondary ordering" condition to hold for *all* tuples rather than for a particular tuple subset. Its meaning hence becomes "all tuples with the same value in the columns of O_2 appear in an order that respects O_1". One can see that the example relation R (shown here) is neither sorted on [iter] nor [pos], but nevertheless respects [ord, pos] | [iter]. Intermediate results with similar ordering properties are characteristic for the relational XQuery compilation approach pursued by Pathfinder.

iter	pos	item	ord
1	1	"one"	*a*
1	2	"two"	*a*
2	1	"four"	*a*
1	1	"three"	*b*
2	1	"five"	*b*

Relation R.

Order Propagation in Physical Operators Table 1 specifies the propagation of order properties as pre- and post-conditions for a number of physical implementations of the (logical) relational algebra used in our XQuery translation (Section 2, but extended with *sort*).

Select and Join. Table 1 specifies that the ScanSelect implementation of σ propagates all order properties from the input relation to the output. In the case of $\bowtie$, the HashJoin (resp. IndexJoin) iterates over an outer relation, looking up matches in permuted order in the inner relation via a hash table (or a search tree). Thus, it propagates *only* the order properties of the outer relation to the result. In contrast, the NestedLoopJoin generates matches for each outer tuple in the inner relation order, thus concatenating the order properties of the inner relation as minor ordering columns to the result as well.

Disjoint Union and Row-Numbering. Disjoint union ($\dot\cup$) can be implemented by a MergeUnion that conserves orderings that are common to both input relations in the output. The alternative AppendUnion, which just appends relations, is less CPU intensive as it does not need to merge. Also, in a pipelined execution model, the latter requires fewer memory resources, because the query subtrees that produce the input relations may run sequentially rather than concurrently.

Disjoint union is used for sequence construction (*i.e.*, XQuery's comma operator $\cdot,\cdot$) in relational XQuery. In the example of relation R, a sequence lifted over a loop with two iterations was constructed from two subsequences using AppendUnion. The sequence construction compilation rule [GST04] uses projection to add a constant ord column to each subsequence first, in order to ensure that elements from the second sequence come after those of the first (in the example relation R, we have $a < b$). The next step in sequence construction is to create a new pos column and eliminate the ord column. This is done by the row-numbering operator(ρ):

$$\pi_{\text{iter,pos}',\text{item}}\left(\rho_{\text{pos}':\text{ord,pos}/\text{iter}}(R)\right) \quad ,$$

iter	pos'	item
1	1	"one"
1	2	"two"
2	1	"four"
1	3	"three"
2	1	"five"

yielding the relation on the side that encodes the sequences ("one","two","three") and ("four","five"). In case relation R would already have been sorted on [iter, ord, pos], the MergeRowNumber implementation could have been be used. It just fills the new pos′ column with a counter that starts at 1, increments it at each tuple, and resets it to 1 whenever a new iter value is seen.

However, in the example just given, R is *not* ordered on [iter, ord, pos], so MergeRowNumber is not applicable. Still, in this case we can use HashRowNumber. It creates a hash-table on iter which maintains a counter that is initialized to 1 for each bucket. For each hashed tuple, this counter is looked up, emitted into the result and incremented. This requires that for each iter, the tuples in the HashRowNumber input have to appear in sequence order, *i.e.*, HashRowNumber indeed needs the [ord, pos] | [iter] order on R as a precondition (see Table 1).

$\sigma_C(R_{in})$
ScanSelect($R_{in}, R_{in}.C = true$) $\rightarrow R_{out}$
post: orders(R_{out}) $\supset$ orders(R_{in})
$R_l \bowtie_{C_1 \theta C_2} R_r \quad \theta \in \{<, >, =, \leqslant, \geqslant\}$
HashJoin($R_l, R_r, C_1 = C_2$) $\rightarrow R_{out}$
IndexJoin($R_l, R_r, C_1 \theta C_2$) $\rightarrow R_{out}$
post: orders(R_{out}) $\supset$ orders(R_l)
post: key($R_l.C_x$) $\Rightarrow$ orders(R_{out}) $\supset$ orders(R_l)+$[C_x]$
NestedLoopJoin($R_l, R_r, C_1 \theta C_2$) $\rightarrow R_{out}$
HashOrderJoin($R_l, R_r, C_1 = C_2$) $\rightarrow R_{out}$
post: orders(R_{out}) $\supset$ { $O_l+O_r \mid O_l \in$ orders(R_l), $O_r \in$ orders(R_r)}
post: key($R_l.C_x$) $\Rightarrow$ orders(R_{out}) $\supset$ { $O_l+[C_x]+O_r \mid O_l \in$ orders(R_l), $O_r \in$ orders(R_r)}
CartProd(R_l, R_r) $\rightarrow R_{out}$
post: orders(R_{out}) $\supset$ { $O_l+O_r \mid O_l \in$ orders(R_l), $O_r \in$ orders(R_r) }
$R_l \dot{\cup}_{C_1,\ldots,C_n} R_r$
MergeUnion($R_l, R_r, [C_1, \ldots, C_n]$) $\rightarrow R_{out}$
pre: $[C_1, \ldots, C_n] \in$ (orders(R_l) $\cap$ orders(R_r))
post: $[C_1, \ldots, C_n] \in$ orders(R_{out})
AppendUnion(R_l, R_r) $\rightarrow R_{out}$
post: $[C_{m+1}, \ldots, C_n] \mid\mid [C_1, \ldots, C_{m-1}] \in$ (orders(R_l) $\cap$ orders(R_r) $\wedge \max(R_l.C_m) < \min(R_r.C_m)$ $\Rightarrow [C_m, \ldots, C_n] \mid\mid [C_1, \ldots, C_{m-1}] \in$ orders(R_{out})
$\rho_{C_r:C_1,\ldots,C_n/C_g}(R_{in})$
MergeRowNumber($R_{in}, C_g, [C_1, \ldots, C_n]$) $\rightarrow R_{out}$
pre: $[C_g, C_1, \ldots, C_n] \in$ orders(R_{in})
post: $[C_g, C_r] \in$ orders(R_{out})
HashRowNumber($R_{in}, C_g, [C_1, \ldots, C_n]$) $\rightarrow R_{out}$
pre: $[C_1, \ldots, C_n] \mid\mid [C_g] \in$ orders(R_{in})
post: $[C_r] \mid\mid [C_g] \in$ orders(R_{out})
$sort_{C_1,\ldots,C_n}(R_{in})$
StandardSort($R_{in}, [C_1, \ldots, C_n]$) $\rightarrow R_{out}$
post: $[C_1, \ldots, C_n] \in$ orders(R_{out})
StableSort($R_{in}, [C_1, \ldots, C_n]$) $\rightarrow R_{out}$
pre: $[C_{n+1}, \ldots, C_{n+m}] \in$ orders(R_{in})
post: $[C_1, \ldots, C_{n+m}] \in$ orders(R_{out})
RefineSort($R_{in}, [C_1, \ldots, C_n], [C_{n+1}, \ldots, C_{n+m}]$) $\rightarrow R_{out}$
pre: $[C_1, \ldots, C_n] \in$ orders(R_{in})
post: $[C_1, \ldots, C_{n+m}] \in$ orders(R_{out})

Table 1: Physical algebra with pre– and postconditions on order properties.

Sorting. The StandardSort (QuickSort, in MonetDB) just introduces the sort order as the only new order of the result. StableSort algorithms (like RadixSort) conserve orderings on the input as minor orderings to the new sort order. Finally, the RefineSort algorithm exploits the knowledge that the input relation is already sorted on a major subset of the required order. This algorithm is pipelinable, as it just needs to merge through the input relation, identifying sub-ranges with equal values on the already sorted major ordering columns. Only the current subrange needs to be buffered and sorted on the minor columns, before new tuples are returned. To our knowledge, MonetDB is the only DBMS that currently employs RefineSort .

5.2 Order Optimization in Relational XQuery

Relational XQuery order optimization may take two forms: *(i)* in the logical plan, we may omit all effort to generate pos columns, if these columns are not used later on, *(ii)* on the physical level, we should exploit ordering propagation to avoid sorting of relations that are already sorted.

Although order is a pervasive concept in XQuery, there may indeed be sub-plans in the generated algebraic query which may not depend on order at all. Examples include the sub-plans which evaluate the arguments to XQuery aggregate functions (*e.g.*, fn:sum() , fn:count()), quantifiers (some , every) as well as general comparison operators. Additionally, the XQuery keyword unordered explicitly indicates that item order in its

argument sequence may be arbitrary. Such sub-plans need *not* produce or maintain pos information at all.

Pathfinder generates a physical algebra subquery for each XQuery translation rule [GT04]. Currently, it enforces that *all* generated sequence encodings are ordered on [iter, pos]. We do optimize the way in which this ordering is created: already sorted relations are not re-sorted and partially sorted relations are only refined with RefineSort. For instance, sequence construction is evaluated via MergeUnion with merging on [iter, ord, pos]:

$$\texttt{MergeRowNumber}\ \big(\mathsf{pos}', \texttt{MergeUnion}\ (R_1, R_2, [\mathsf{iter}, \mathsf{ord}, \mathsf{pos}]), \mathsf{iter}, [\mathsf{ord}, \mathsf{pos}]\big)\ .$$

Our ordering framework allows us to conclude that the result relation must be in $[\mathsf{iter}, \mathsf{pos}']$ order.

Similar optimized translations have been made for joins. The full sorting policy leads to both input relations R_1, R_2 being ordered on [iter] and the requirement for the result to be ordered on $[\mathsf{iter}_{R_1}, \mathsf{iter}_{R_2}]$. The standard operator chosen is now HashJoin producing a result in $[\mathsf{iter}_{R_1}]$ order, which means that we need a RefineSort on iter_{R_2}. Here, we exploit the *run-time* query optimization of MonetDB to check if the join selectivity is low and if so use a NestedLoopJoin, which has the advantage that it already yields a $[\mathsf{iter}_{R_1}, \mathsf{iter}_{R_2}]$ output without reordering. In other cases, the superior performance of HashJoin outweighs the cost of the extra RefineSort.

Future Optimizations. In Table 1 we already mention a HashOrderJoin operator. The idea behind this hash join is to enforce that the collision lists in the hash-table are in tuple order of the hashed relation. By adding such an operator to the relational back-end, we would get a join that produces results with the major ordering of the outer relation *and* the minor ordering of the inner, eliminating the need for further RefineSort.

6. Implementation on top of MonetDB

Pathfinder is backed by the open-source RDBMS MonetDB, following the architecture depicted in Figure 8. MonetDB is an extensible system and this feature has been used to introduce an XQuery *runtime module*. It extends the MIL[6] algebra of MonetDB with a few additional operators, mainly the loop-lifted staircase join (Section 3).

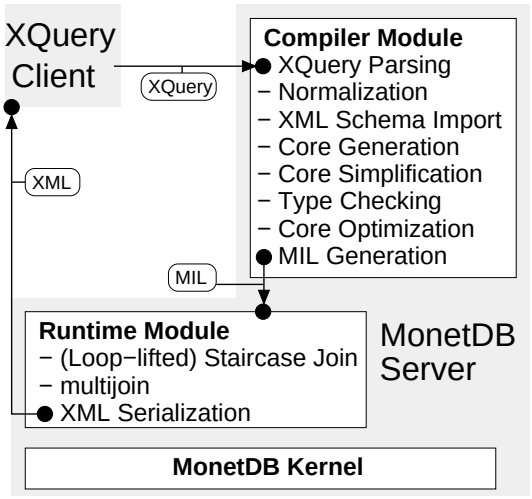

Figure 8: System architecture.

We implemented the front-end as an XQuery *compiler* that translates XQuery expressions into MIL code for execution on MonetDB. Pathfinder currently employs several XQuery Core optimization techniques (including the join optimization logic of Section 4), as well as XML Schema import and full static type checking. The compiler is designed to be *re-targetable*: the internal relational algebra may be implemented on any relational system that supports the operators mentioned in Section 2.

[6]MIL stands for MonetDB Interpreter Language.

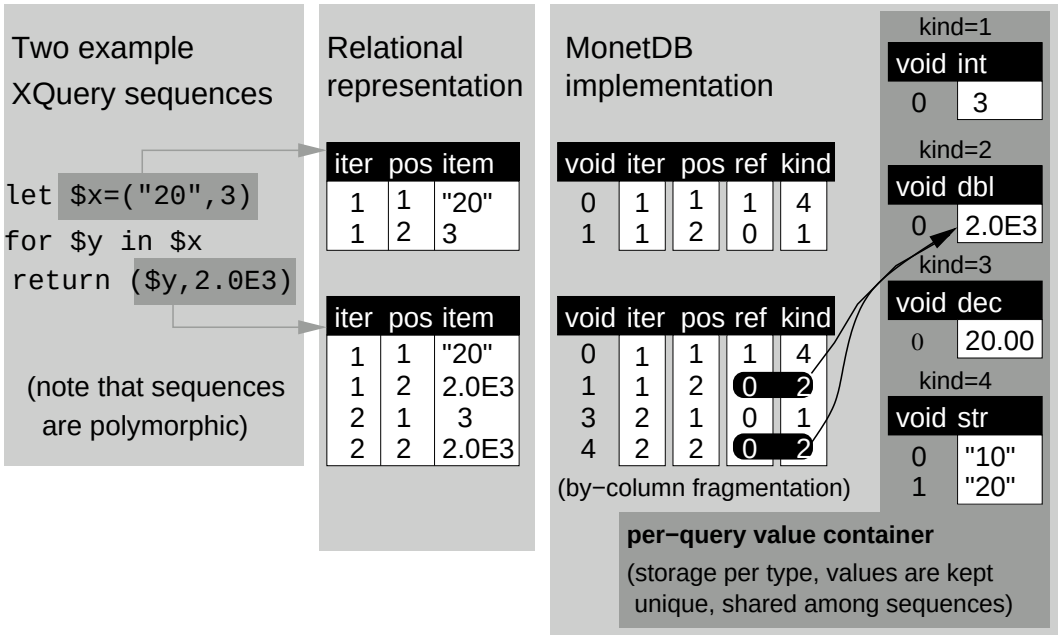

Figure 9: Storage of polymorphic XQuery item sequences in MonetDB.

Sequence and Document Representation

For any XQuery Core construct, the compiler emits a set of MIL commands that constitute our physical algebra. The translation follows the ideas in [GT04] with the refinement that any intermediate result is a relation of type iter|pos|item, fully ordered on [iter, pos].

In [GT04] we assumed the presence of a *polymorphic* item column, a feature that is typically not present in relational DBMSs (including MonetDB). We thus use a *boxed* representation of column item, as sketched in Figure 9. We substitute column item by the combination of the two columns ref and kind and represent a polymorphic value v as the pair $\langle ref(v), kind(v)\rangle$, where $kind(v)$ identifies v's implementation type, and $ref(v)$ is a foreign key (of MonetDB type oid). $ref(v)$ references the actual value of v in the *value container* associated with type $kind(v)$. Such a value container is maintained for each atomic type. In case of type node, $ref(v)$ is the preorder rank assigned to the represented node.

All value containers in our system use densely numbered key columns $(1, 2, 3, \ldots)$, a setup that is particularly well supported by MonetDB in terms of the void type ("virtual oid", [BK99]). A column c that is declared void may be read as "for row number i, attribute c takes the value i." MonetDB does not explicitly store void values, eliminating space overhead for the key column. Access by void keys is implemented using a positional lookup and thus highly efficient.

The backbone of any XQuery execution engine is the storage of XML trees and fragments. Our system uses the aforementioned pre|size|level encoding to encode both, persistently stored documents, as well as fragments constructed on-the-fly during query evaluation.

This pre|size|level table (see Section 2) is enriched with further columns to hold additional node properties. Relevant properties for XML tree nodes depend on the node's *kind* (*e.g.,* element, text node,...). We use a set of *property containers* for the different node kinds (much like the representation of the polymorphic item column in the foregoing). Figure 10 lists the property containers and their respective column schema (name space and local name for XML elements, textual content for text and comment nodes, and a target/value pair for XML processing instructions). Again, we benefit from MonetDB's void column to efficiently access node properties.

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

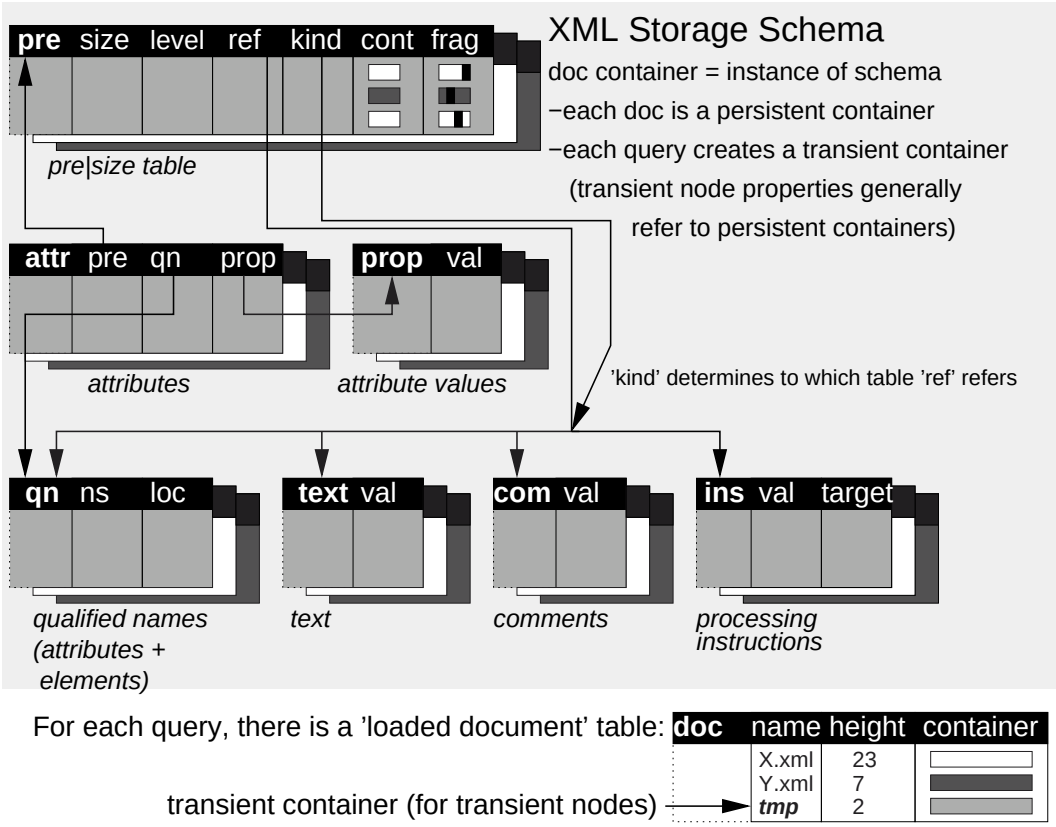

Figure 10: Horizontally partitioned XML storage in document containers.

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 (cf. [GST04]). A *loaded document* table keeps track of any document container currently active.

As already observed by [GT04], 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 ref as a reference to its original container.

7. Quantitative Assessment

We claim that, to date, Pathfinder powered by MonetDB provides one of the most scalable and fastest XQuery implementations available. This section reports on experiments which back up this claim.

In our experiments, we focused on XMark [SWK+02], 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 2.6.9, using a 64-bit address space. We used the XMark benchmark with scaling factors from 0.1 up to 100 (which yields documents from 11 MB up to 11 GB), using Pathfinder as well as the latest versions of Galax (0.5.0) [FSC+03] and X-Hive (6.0) [X-H].

Galax was used as it is the most popular XQuery engine available in open-source. From the performance results reported in [DTCÖ03], we concluded that X-Hive is one of the faster native XML database systems,

so we included it in our evaluation. We were able to significantly reduce the run time of X-Hive on a number of queries by creating value indices on the paths `buyer/@person` and `profile/@income`. The former index reduces the run time of XMark query Q8 from quadratic to linear, significantly surpassing all results reported for that query in [DTCÖ03].

Regrettably, Galax failed to process the queries once the XMark documents were beyond a size of 110 MB. Galax is a file-oriented system that parses the XML file on each query which often dominates run time. 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.

For Pathfinder, we ran the XQuery compiler to generate MIL scripts. The compiler itself used between 60 and 100 msec (this is excluded from the performance results).

Results

Loop-Lifted Staircase Join. Figure 11 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. Some queries (Q3,Q11-14), where path step cost is relatively small, in general benefit less (factor 2.5-5). 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 staircase join.

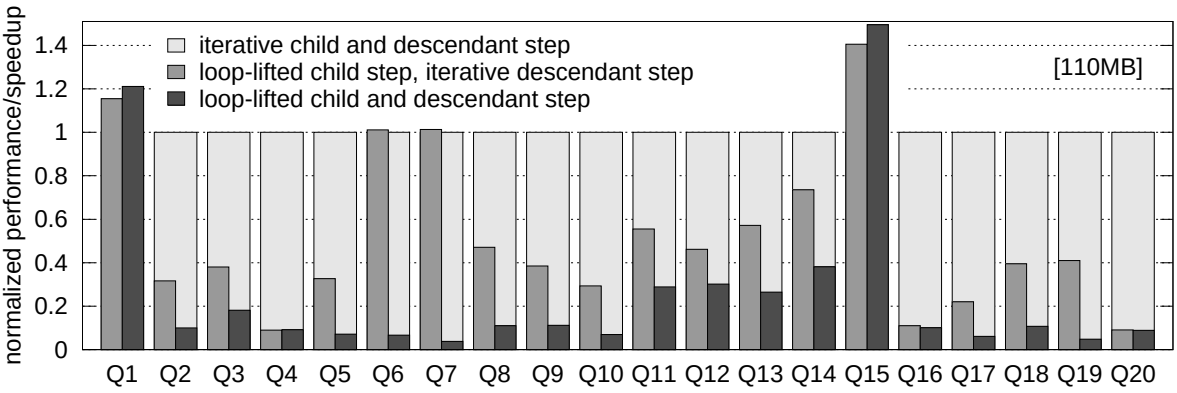

Figure 11: Benefits of loop-lifted staircase join.

Join Optimization. Before we installed the join optimizations of Section 4, Pathfinder 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 12 contrasts the results for the 11 MB document with the performance we obtained with join recognition enabled in our MIL generation. It is obvious that the execution of XQuery statements with join predicates simply *requires* join recognition when the query is run on significant XML document sizes.

Order Awareness. The next step is to analyse the performance improvements that are provided by the order aware physical algebra operators as presented in Section 5. These operators allow us to save sort operations which would otherwise be necessary to ensure that intermediate results are in document (respectively sequence) order, wherever this is required by the XQuery semantics. For the 110 MB document, Figure 13 compares the performance using the order aware operators to the normalized performance with the respective non-order aware operators, adding sort operations where required. The difference is quite significant. On average, we observe an improvement of about factor 2.

Scalability. Figure 14 shows the performance results of Pathfinder, 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.

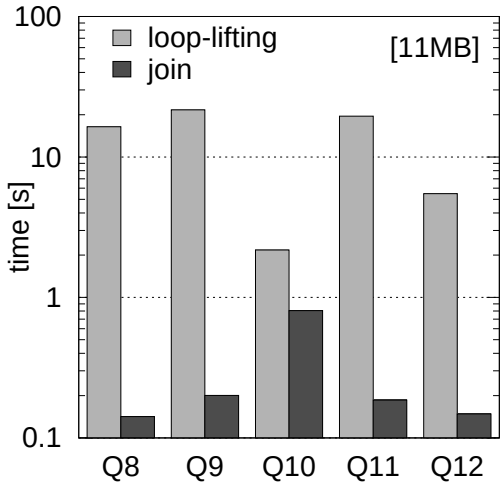

Figure 12: Benefits of XQuery join optimization.

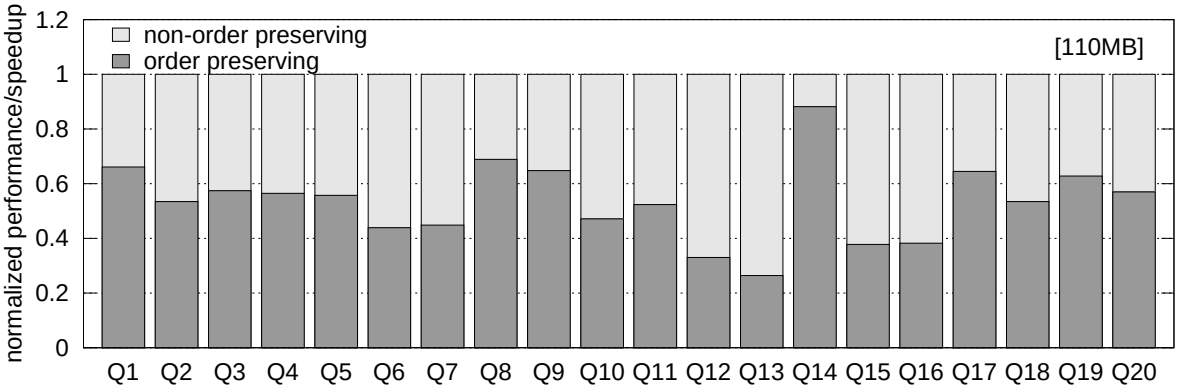

Figure 13: Benefits of order-preserving operators.

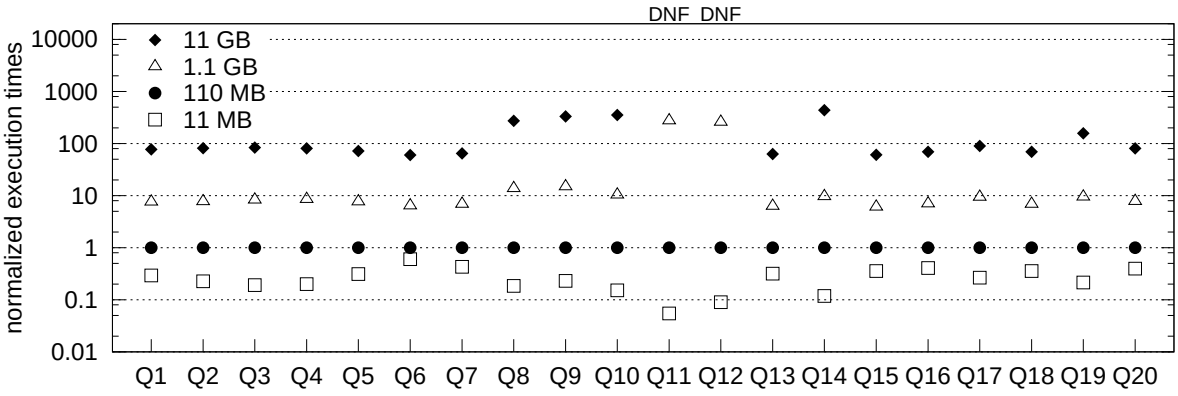

Figure 14: Scalability with respect to document size.

	11 MB			110 MB			1.1 GB		11 GB
Q	Galax	X-Hive	Pf/M	Galax	X-Hive	Pf/M	X-Hive	Pf/M	Pf/M
1	0.06	0.37	0.05	0.72	1.29	0.41	9.9	1.2	13
2	0.03	0.45	0.07	0.31	1.75	0.30	33.0	2.4	25
3	0.14	0.65	0.28	1.76	5.66	1.51	25.1	12.5	126
4	0.22	0.10	0.08	2.91	1.00	0.45	18.1	3.8	36
5	0.05	0.13	0.05	0.63	0.90	0.16	20.7	1.2	11
6	1.30	1.07	0.02	13.29	10.17	0.05	178.1	0.3	3
7	2.68	1.57	0.03	30.01	24.84	0.07	278.4	0.4	4
8	0.16	0.85	0.14	2.12	3.51	0.75	49.1	10.4	208
9	113.23	32.25	0.20	*DNF*	12280.66	0.87	*DNF*	12.9	289
10	1.74	5.28	0.80	18.61	442.37	5.31	*DNF*	55.0	1882
11	2.62	98.91	0.18	*DNF*	19927.29	3.48	*DNF*	960.9	*DNF*
12	1.44	23.39	0.14	*DNF*	5100.19	1.66	*DNF*	431.3	*DNF*
13	0.03	0.10	0.07	0.66	1.03	0.22	12.9	1.3	13
14	1.92	0.72	0.26	99.53	11.16	2.20	110.2	21.3	959
15	0.02	0.03	0.09	0.20	0.49	0.28	10.6	1.7	16
16	0.03	0.03	0.11	0.46	0.52	0.26	10.9	1.8	18
17	0.06	0.09	0.07	0.82	0.85	0.30	11.8	2.8	26
18	0.07	0.08	0.04	0.73	0.64	0.13	14.8	0.9	9
19	1.17	0.67	0.11	14.73	12.15	0.55	254.5	5.3	88
20	0.28	0.11	0.24	2.98	1.40	0.62	24.6	4.9	50

Table 2: Overview of XMark query evaluation times (elapsed time in seconds).

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.

Comparison with Galax. Table 2 lists our full experimental results. For the smaller document sizes (11 MB and 110 MB), Galax is on par with the other systems and sometimes performs fastest although by a small margin. For the join queries Q8–12, the optimizer in Galax 0.5.0 seems to spot the XQuery joins in Q8 and Q10 but fails to do so otherwise. The test runs crashed with *materialization out of bounds* errors, most probably due to the quadratic join complexity we have mentioned before.

Comparison with X-Hive. X-Hive [X-H] has no trouble importing the 1.1 GB XML document and allows the execution of non-join queries in reasonable time. As mentioned before, the join query Q8 also runs fast due to the value indices we created in X-Hive. However, the quadratic performance on Q9–Q12 indicates that such indices only help on a small class of queries. Table 2 shows that Pathfinder clearly outperforms X-Hive on these queries. If the queries join *intermediate* query results, indices cannot be used and performance degrades strongly. Another conclusion is that queries such as Q6 and Q7, which rely heavily on XPath traversals (`descendant` axis), appear to run significantly slower on X-Hive than on our system.

Comparison with Timber. To compare with Timber [JLST01], we use the results for the 110 MB XMark setup published in [PWLJ04][7]. Neglecting the hardware differences—1.833 GHz Athlon vs. 1.6 GHz Opteron—, Timber manages to outperform both Galax and X-Hive on most queries. On the non-join queries, Timber shows very similar performance as Pathfinder. On the join queries, however, Timber appears to be slower than Pathfinder by a factor 3 to 15. We are not aware of any reported Timber results for XMark documents larger than 110 MB.

The overall conclusion of the experiments is that Pathfinder powered by MonetDB 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.

[7] Timber is "only" available for Windows systems.

8. Related Research And Systems

The present work builds on both an XPath-aware relational encoding of XML trees [GvKT03] and a relational XQuery compiler [GST04] to turn a relational database back-end into an XQuery processor. To date, as suggested by a recent survey article [KKN03], 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.

In [DTCÖ03], the authors describe an 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 ([DTCÖ03] describes an XQuery join pattern that resembles the query pattern discussed in Section 4.1, but it largely remains unclear how to derive a relational join plan). 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 [DTCÖ03] by two orders of magnitude.

Quite timely, the order-aware optimization of relational queries has received renewed attention [WC03, MN04]. Inspired by the foundational work on "interesting orders" in System R [SAC$^+$79] 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 [SSM96], Wang and Cherniack describe order property inference rules [WC03]. 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 $O_1|O_2$, pervade in the algebraic plans emitted by our XQuery compiler.

Our work on loop-lifted staircase join was mainly driven by the insight that the relational back-end may devote a significant share of evaluation time to XPath navigation in nested `for` -loops. This renders the work on Nested XML Tableaux (NEXT) especially interesting: [DPX04] reports on an algorithm that minimizes redundant XPath axis traversals across nested subqueries. Ultimately, this can help minimizing the number of required loop-lifted staircase joins in compiled queries.

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 [OOP$^+$04, LM01, ZND$^+$01, TVB$^+$01]. Preorder ranks provide node surrogates of fixed byte-width, which greatly simplifies their storage and manipulation. Such fixed-size node encodings are known to incur significant inherent costs if general structural tree updates (*e.g.*, node insertions, subtree deletions) are to be processed [CKM02]. In contrast, variable-length surrogates such as, for example, ORDPATH labels [OOP$^+$04], 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 staircasejoin for efficient location step evaluation.

In this work, we focus on XQuery, where trees are solely built through the application of node *constructors*. The copy semantics of these constructors are efficiently supported by *pasting of subtree encodings* as described in [GT04].

In a future version of Pathfinder, we plan to also support document updates. Update basically require insertion or deletion of pages "somewhere" in the `pre|size|level` table that stores the encoded XML document. To maintain the dense preorder ranks, all the `pre` values in all subsequent pages have to be incremented or decremented accordingly. In MonetDB, however, this is not necessary, as Pathfinder stores preorder ranks using `void`-columns [BK99]. These `void` columns are virtual (they resolve to the tuple position in the table; see Section 6) and are never materialized on disk; thus they need not be updated.

As of today, Galax [FSC$^+$03] seems to be the primary XQuery "playground" and reference system for the database community. For good reason: Galax adheres closely to the W3C XQuery Working Drafts [BCF$^+$04] and implements the complete language semantics (with minor exceptions). In versions up to 0.4, Galax' ad-

herence to the XQuery Formal Semantics might have even been too close: all nested `for` -loops were actually evaluated in an iterative nested-loops fashion. Starting with version 0.5.0, the Galax compiler includes hooks to emit algebraic plans that exploit joins (as far as we are aware, these efforts were not published yet). We would be most interested in investigating how a Galax front-end could benefit from the relational XQuery runtime we are developing in this work.

9. Conclusions and Future Work

This work delivers the proof that relational databases may be turned into highly efficient XQuery runtime systems. We implemented an XQuery processor that is backed by the extensible MonetDB RDBMS. Mainly based on the ideas of [GST04, GT04], our system exploits a number of enhancements that make the resulting setup stand out among the XQuery implementations available today. The *loop-lifted staircase join* continues the work of [GvKT03] and executes XPath location steps for multiple sequences of context nodes in a single pass over an encoded XML document. A carefully designed *join processing* framework reduces the quadratic complexity of XQuery joins to scale linearly with the input document size. Exploitation of *order properties* in our physical algebra avoids the extensive use of sort operators in the execution plan.

At the final count, these contributions enabled us to assemble an XQuery processor that makes the *scalability* of relational back-ends available for XQuery processing. The system's code base currently undergoes a cleaning stage after which it will be released under an open-source license.

Apart from code-level work, we are now investigating possibilities for far-reaching algebraic optimization of XQuery. The generated algebraic plans exhibit a number of interesting properties (constant columns, key-joins over identical domains, multi-valued dependencies indicating expression independence, to name a few) which we will use to further enhance the quality of the emitted relational code.

References

[BBC+04] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernández, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) 2.0. World Wide Web Consortium, October 2004. http://www.w3.org/TR/xpath20/ .

[BCF+04] S. Boag, D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. World Wide Web Consortium, October 2004. http://www.w3.org/TR/xquery/ .

[BK99] P. Boncz and M.L. Kersten. MIL Primitives For Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, March 1999.

[CKM02] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Trees. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 271–281, Madison, WI, USA, June 2002.

[DPX04] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT Framework for Logical XQuery Optimization. In *Proc. VLDB Conf.*, pages 168–179, Toronto, Canada, September 2004.

[DT03] A. Deutsch and V. Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *Proc. VLDB Conf.*, pages 201–212, Berlin, Germany, September 2003.

[DTCÖ03] D. DeHaan, D. Toman, M.P. Consens, and M.T. Öszu. A Comprehensive XQuery to SQL Translation Using Dynamic Interval Encoding. In *Proc. SIGMOD Conf.*, pages 623–634, San Diego, CA, USA, June 2003.

[FHK+04] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, J. Carey, and A. Sundarajan. The BEA Streaming XQuery Processor. *The VLDB Journal*, 13(3):294–315, 2004.

[FSC+03] M.F. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax Experience. In *Proc. VLDB Conf.*, pages 1077–1080, Berlin, Germany, September 2003.

[GST04] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. VLDB Conf.*, pages 252–263, Toronto, Canada, September 2004.

[GT04] T. Grust and J. Teubner. Relational Algebra: Mother Tongue—XQuery: Fluent. In *Twente Data Management Workshop on XML Databases and Information Retrieval (TDM)*, pages 7–14, Enschede, The Netherlands, June 2004.

[GvKT03] 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, Berlin, Germany, September 2003.

[JLST01] H.V. Jagadish, L.V.S. Lakshamnan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *Proc. DBPL*, pages 149–164, Frascati, Italy, September 2001.

[KKN03] R. Krishnamurthy, R. Kaushik, and J. Naughton. XML-to-SQL Query Translation Literature: The State of the Art and Open Problems. In *Proc. XSym*, pages 1–18, Berlin, Germany, September 2003.

[KSSS04] 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 *Proc. VLDB Conf.*, pages 228–239, Toronto, Canada, September 2004.

[LM01] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. VLDB Conf.*, pages 361–370, Rome, Italy, September 2001.

[MN04] G. Moerkotte and T. Neumann. A Combined Framework for Grouping and Order Optimization. In *Proc. VLDB Conf.*, Toronto, Canada, September 2004.

[OOP+04] 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, Paris, France, June 2004.

[PWLJ04] S. Paparizos, Y. Wu, L.V.S. Lakshmanan, and H.V. Jagadish. Tree Logical Classes for Efficient Evaluation of XQuery. In *Proc. SIGMOD Conf.*, pages 71–82, Paris, France, June 2004.

[SAC+79] P. Selinger, A. Astrahan, A. Chamberlin, R. Lorie, and A. Price. Access Path Selection in a Relational Database Management System. In *Proc. SIGMOD Conf.*, pages 23–34, Boston, MA, USA, May 1979.

[SSM96] D. Simmen, E. Shekita, and T. Malkemus. Fundamental Techniques for Order Optimization. In *Proc. SIGMOD Conf.*, pages 57–67, Quebec, Canada, June 1996.

[SWK+02] 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, Hong Kong, China, August 2002.

[TVB+01] 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.*, Dallas, TX, USA, June 2001.

[WC03] X. Wang and M. Cherniack. Avoiding Sorting and Grouping in Processing Queries. In *Proc. VLDB Conf.*, pages 826–837, Berlin, Germany, September 2003.

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

[ZND+01] 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.*, Dallas, TX, USA, June 2001.