Jump Through Hoops to Grok the Loops

Pathfinder's Purely Relational Account of XQuery-style Iteration Semantics

Torsten Grust Jan Rittinger

Technische Universität München
Munich, Germany

torsten.grust | jan.rittinger@in.tum.de

ABSTRACT

What remains if you remove XML trees and node construction, XPath location path traversal, atomization, and all other XML-inflicted concepts from XQuery? — We describe how the design of the database-supported XQuery processor *Pathfinder* has been centered around the compilation of the language's core side effect-free iteration construct, `for`, a concept that XQuery shares with many (data-intensive) languages, *e.g.*, SQL, Linq, Links, Ruby, and Haskell's or Python's list comprehensions. The compiler implements *loop lifting*, a compilation technique that lets a relational database back-end fully realize—and benefit from—the independence of the individual iterations of a `for` loop. We explore useful extensions and special cases of loop lifting and will see how XQuery 1.1's proposed windowed iteration construct `forseq` may be grokked this way—this also uncovers, however, `forseq`'s lurking cost.

1. XQUERY MINUS THE 'X'

In retrospect, the design, architecture, and performance of the relational XQuery compiler *Pathfinder* [11] have been primarily determined by the compiler's specific approach to `for` *loop compilation*. On the other hand, the XML-related aspects of XQuery's data model—the representation and construction of nodes of several kinds or traversal along all XPath axes, for example—have been secondary design choices that do not reach particularly deep. In fact, *Pathfinder*'s relational XML node encoding has been replaced during the lifetime of the project. The system currently relies on a variant of the now pervasive pre|post range encoding, but this detail is "pluggable" and the compiler will accept any encoding that preserves node identity and document order (*e.g.*, ORDPATH labels [20]).

To acknowledge this key role of XQuery's `for` in *Pathfinder*, in the pages to come we will *not* discuss the "*X-ish*" aspects of the language. Instead, we focus on the efficient compilation of (nested) iteration over ordered sequences of arbitrary items. To illustrate, consider Query Q_1 in Figure 1. Q_1 searches an XMark [24] instance `auction.xml` for corrupted auction data (those open auctions for which the `current` bid does not properly reflect the `increase`s of the `initial` price set by the seller). The query appears to be dominated by XPath location path traversal and predicate evaluation, element and attribute construction, and implicit node atomization (the latter caused by occurrences of `sum(·)` and the general comparison operator `=` in Q_1). Here, we take the efficient evaluation of these aspects for granted—that is why they are grayed out in Figure 1—and instead zoom in on the iterative core of the query, depicted in Figure 2. In this XQuery Core equivalent Q_2 of the original Q_1, the tables have turned: nested `for` loops and conditional expressions dominate and clearly call for adequate treatment during query compilation (the e_i are placeholders for Q_1's secondary XML-specifics, such that $e_2 \equiv$ `$v1/(initial | bidder/increase)` and $e_6 \equiv$ `text { 'no' }`), for example.

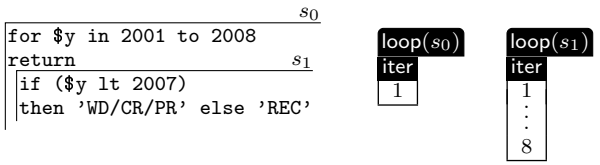
```
let $auction := doc('auction.xml')
for $o in $auction/open_auctions/open_auction
return
 <corrupt id='{ $o/@id }'>
  { if (sum($o/(initial | bidder/increase)) = $o/current)
    then text { 'no' }
    else $auction//people/person[@id = $o//@person]/name }
 </corrupt>
```

Figure 1: Search XMark document `auction.xml` for corrupted bid data (Query Q_1).

```
for $v1 in e1
return
 if (exists(for $v2 in sum(e2)
            return for $v3 in e3
                   return if (e4) then e5 else ()))
 then e6
 else for $v4 in e7
      return
       if (exists(for $v5 in e8
                  return for $v6 in e9
                         return if (e10) then e11 else ()))
       then e12 else ()
```

Figure 2: The core of Q_1 minus the XML-specific baggage: `for` iterations and conditionals (Query Q_2).

XQuery and its horde of companion languages. Once we remove the XML-inflicted baggage, it becomes apparent that the XQuery language has numerous close companions.



(a) Query Q_3. **(b) Associated loop tables.**

Figure 3: XQuery and its W3C Recommendation track maturity level (Query Q_3). Annotations $s_{0,1}$ denote iteration scopes.

```
(Linq)     Enumerable.Range(2001,8).Select(
             y => y < 2007 ? 'WD/CR/PR' : 'REC' )

(Links)    for (y <- [2001,2002,...,2008])
             [if (y < 2007) then "WD/CR/PR" else "REC"]

(Ruby)     (2001..2008).collect {
             |y| y < 2007 ? 'WD/CR/PR' : 'REC' }

(Haskell)  [ if y < 2007 then "WD/CR/PR" else "REC" |
             y <- [2001..2008] ]

(Python)   [ 'WD/CR/PR' if y < 2007 else 'REC'
             for y in range(2001,2008) ]
```

Figure 4: A sample of iterative constructs found in XQuery's companion languages (paraphrases of Q_3).

Figure 3(a) shows Query Q_3, one instance of an XQuery `for` iteration (ignore the $s_{0,1}$ annotations for now). Despite the syntactic diversity, this XQuery construct shares a common semantic ground—*monad comprehensions* [26]—with SQL as well as the iteration primitives in, *e.g.*, Microsoft®'s Linq [19], Wadler's three-tier language Links [6], the purely functional language Haskell [21], and the dynamic languages Ruby [23] and Python [22] (see Figure 4).[1] All of these language constructs describe the iterative evaluation of expressions under bindings of an unmodifiable *loop* or *iteration variable*. Even for the non-pure languages Linq, Ruby (as of Version 1.9), and Python, an assignment to a variable named `y` in the loop body will shadow the iteration variable and thus not influence the behavior of the loop.

Here, we are especially interested in loops in which the iterated expression does *not* perform side-effecting computation such that the individual iterations may be evaluated *independently*. For XQuery, SQL, Links, and Haskell this is a given. For Linq, Ruby, and Python this requires programming discipline. As the individual iterations cannot interfere, the language processor may evaluate the iterations in arbitrary order—or even in parallel.

These common semantic roots create a playground in which database queries and the mentioned companion languages may closely interact. We currently study

(1) the construction of systems in which two (or more) companion languages *share* a single database-supported runtime—this can lead to a truly integrated SQL/XML processor, for example, in which the typical mix of SQL and XQuery query fragments is uniformly compiled to yield a homogeneous executable form, and

(2) an even deeper integration of database query functionality into programming languages (as exemplified by ActiveRecord or Ambition in the Ruby ecosystem [1, 2], Linq, and Links), in which selected iterative host programming language fragments may be translated into set-oriented algebraic programs. This lays the groundwork for database-supported language runtimes that do not stumble if programs consume huge input data instances [13].

[1] Until June 2001, the W3C XQuery Formal Semantics Draft [9] explicitly discussed monads and associated laws.

Operator	Semantics
$\pi_{\mathsf{a:b}}$	project onto column b (and rename into a)
σ_{a}	select rows with column a = *true*
$\times$	Cartesian product
$\bowtie_{\mathsf{a=b}}$, $\ltimes_{\mathsf{a=b}}$	equi-join, equi-semijoin
$\dot\cup$, $\cup$, $\setminus$	(disjoint) union, difference
δ	eliminate duplicate rows
$@_{\mathsf{a}:c}$	attach column a containing constant value c
$\#_{\mathsf{a}}$	attach arbitrary key (row id) in column a
$\varrho_{\mathsf{a}:\langle \mathsf{b}_1,..,\mathsf{b}_n\rangle}$	attach row rank in a (in b_i order)
$\vec{\#}_{\mathsf{a}:\langle \mathsf{b}_1,..,\mathsf{b}_n\rangle/\mathsf{c}}$	attach $1, 2, ..$ in a (in b_i order per c-group)
$\circledast_{\mathsf{a}:\langle \mathsf{b}_1,..,\mathsf{b}_n\rangle}$	attach result of n-ary op. $* \in \{+, <, \neg, ..\}$ in a
$\mathrm{AGG}_{\mathsf{a:b/c}}$	attach aggregate of b in a (per c-group)

Table 1: Excerpt of Pathfinder's target table algebra (with AGG $\in$ {COUNT, MIN, ...}).

Algebraic code. With *Pathfinder* we designed and implemented a compiler that translates such loop-centric programs in a fully compositional manner [16]. The compiler emits plans over a table algebra whose operators (Table 1) have been selected to reflect the capabilities and execution model of modern SQL-based RDBMS. The few non-textbook operators include the family of $\#$, ϱ, and $\vec{\#}$ which perform variants of row numbering and correspond to SQL:1999's `ROW_NUMBER` clause. As a consequence of the algebra's purist RISC-like style (*e.g.*, selection σ_{a} does not evaluate predicates on its own but relies on the presence of a Boolean column a), the resulting plans tend to be somewhat verbose—typical *Pathfinder* plans feature 100s, not 10s, of operators—but can be implemented on a wide variety of back-end systems. Currently this includes code generators that target the SQL processors of IBM® DB2 and Microsoft SQL Server as well as the algebraic MIL language of MonetDB [4] and the APL-like language Q of kdb+.

Loop lifting. In what follows we shed light on a compilation technique, coined *loop lifting* in [16], that has been designed to let a relational database back-end directly participate in the evaluation of programs (or queries) written in an iterative style. The loop-lifting compiler emits algebraic code for execution on the back-end which then realizes the semantics of the input program. Loop lifting fully realizes the welcome independence of the iterated evaluations and enables the relational query engine to take advantage of its set-oriented processing paradigm (Section 2).

Loop lifting is simple yet versatile. We sketch extensions as well as special cases that address features and peculiarities of XQuery (Section 3) and also cover the windowed iteration construct `forseq` proposed for XQuery 1.1 [18]. Strictly speaking, `forseq` violates iteration independence, though—this violation comes with inherent cost (Section 4).

2. LOOP LIFTING

With XML trees out of the picture, *ordered* sequences $(x_1, x_2, \ldots, x_n)$ of items x_i are the principal data structure in XQuery—the companion languages equivalently feature

pos	item
1	x_1
2	x_2
⋮	⋮
n	x_n

values of type `[`α`]` (HASKELL), `Array` (RUBY) or `IEnumerable<`T`>` (LINQ). To properly reflect this on the inherently unordered relational database back-end, we embed order in the data and use binary tables with columns pos|item (shown on the left) to represent such sequences. Note that the values in column pos need not be dense and not even be of type `integer`; any ordered domain will do. In specific cases the required order may already reflected by the items x_i themselves (think of a sequence of encoded nodes resulting from XPATH location step evaluation)—column item may then assume the role of pos.

Our view of the XQUERY dynamic semantics is principally determined by `for` as the core language construct: *any* subexpression is considered to be iteratively evaluated in the scope s_i of its innermost enclosing `for` loop. The top level of an expression e is assumed to be wrapped inside the scope s_0 of a pseudo single-iteration loop `for $_ in (0) return` e where `$_` does not occur free in e (*i.e.*, the choice of `0` is arbitrary). The fundamental idea behind *loop lifting* is to produce algebraic code that consumes and emits a "fully unrolled" tabular representation of e's value. Here, *unrolling* refers to the principle that

> a *single* ternary table with schema iter|pos|item holds the encoding of *all* values that e assumes during its iterative evaluation.

Generally, such a table has key ⟨iter, pos⟩ since e may yield a *sequence* of items in each distinct iteration—only if e's sequence type is a subtype of `item?` [7], iter by itself will be key. A row $\langle i, p, v\rangle$ in the table may invariably be read as "in iteration i, expression e yields item value v at the sequence position corresponding to p's rank in column pos."

In Query Q_3 (Figure 3(a)) we have made the iteration scopes explicit. The evaluation of the top-level expressions in the pseudo scope s_0 is iterated once. In the accompanying Figure 5, the bottommost iter|pos|item table shows the output of the algebraic code produced for the top-level expression `2001 to 2008`: all 8 items have been produced in the first and only iteration in scope s_0 (iter = 1 in all rows), in the order indicated by column pos. From this, the algebraic program derives that 8 iterations will be performed in the inner scope s_1: variable `$y` is bound to one integer in the sequence 2001, ..., 2008 in each iteration while the constant 2007 invariably evaluates to 2007.

Note how the bottom join $\bowtie_{\text{iter}=\text{iter}_1}$ in Figure 5 assembles the values of `$y` and 2007 in corresponding iterations such that the *single* invocation of $\ominus$ can compute the outcome of the comparison `$y lt 2007` for *all 8* iterations. In effect, the operator's internal row-by-row processing drives the iterative evaluation and no explicit iteration primitive or similar non-relational device is required. The back-end may autonomously decide about out-of-order row processing or the adequacy of parallel execution—its ⟨iter, pos⟩ key allows each item to be correctly positioned in the evaluation result. This is, in a sense, the algebraic embodiment of the iteration independence we underlined in Section 1.

There is, literally, no relational encoding of the empty sequence `()`: if subexpression e evaluates to `()` in iteration i, no row with iter = i will occur in the iter|pos|item table associated with e. The presence of `()` may be reconstructed, however: if required, the compiler emits code that produces

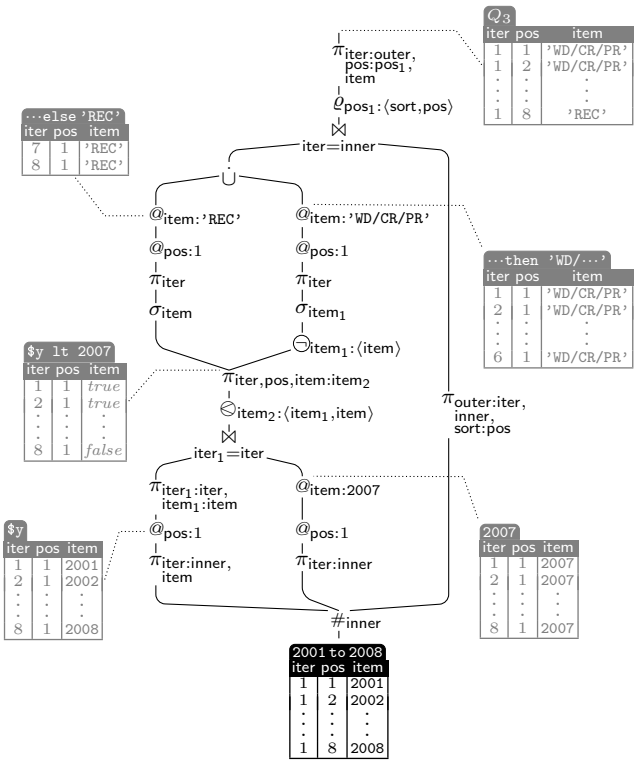


Figure 5: Loop-lifted algebraic code to evaluate Q_3 (also shows results of selected subexpressions).

unary loop tables that keep record of all iterations performed in a given scope (see Figure 3(b) for the loop tables associated with Q_3). By subtracting from table loop(s_1), the plan can compute that the `then` branch of Q_3 evaluates to `()` in iterations 7 and 8 while the `else` branch yields `()` in iterations 1 through 6. (We come back to loop tables in Section 3.)

Plan shape. Figure 5 and a peek at Figure 9 show a distinctively narrow but tall plan shape resulting from the compositional stacking of constructs that is typical for expression-oriented languages like XQUERY and its companions. Loop-lifted code exhibits plenty of sub-plan sharing opportunities (all subexpression in a scope share one loop table, for example) which naturally leads to plan DAGs rather than trees. Despite their unusual shape we have found the plans to be amenable to far-reaching analysis and simplification [15]. Nevertheless, dependent on the complexity of the input program, back-ends originally built for languages with only restricted compositionality (think SQL) may have to issue a *series of collaborating queries* to realize the semantics of the overall loop-lifted plan [12].

While this already gives a fairly complete account of loop lifting, the gory details of the translation scheme, its implications, performance, and optimization have been described elsewhere [4, 11, 15, 16].

3. MORE COLUMNS...

As a purely relational compilation technique, loop lifting inherits the versatility of the relational model. (To make this point, we built ROVER [14], an XQUERY debugger that

instruments the algebraic code to persistently save selected intermediary result tables (see Figure 5) in the back-end. ROVER issues SQL queries against these tables to offer a declarative post-mortem debugging interface that can explore huge values and deeply nested iteration scopes.)

Here, we sketch how minor extensions or restrictions of the original ternary iter|pos|item model suffice to address a number of further interesting peculiarities in XQUERY land.

Validation and type annotation (iter|pos|item|type). A side effect of successful XML document validation is *type annotation* which augments the document's nodes with their inferred XML Schema types. In a loop-lifted translation, such annotations naturally live in an extra type column. In [25], Teubner has shown how this column may be used to let the database back-end perform loop-lifted *sequence type matching*. Ultimately, this leads to an efficient set-oriented implementation of `typeswitch()` and `instance of` clauses.

Node locators (iter|pos|item|guide). Locating nodes in huge XML instances may account for a significant share of query execution time. This is where additional location information, *e.g.*, the node's representative in its document's *Data Guide* [10], pays off. *Pathfinder* uses the equivalent of a guide column—available for items of sequence type `node`—to shortcut XPATH traversal and to reduce atomization effort.

XQUERY Full-Text (iter|pos|item|score). In XQUERY Full-Text selections [3, § 3], each retrieved item is associated with a match *score* that is accessible during iteration (in terms of `for $x score $s in` ⋯). Such extensions to the original XQUERY Data Model (DM) may be straightforwardly understood in terms of column extensions (here: a score column from which bindings for `$s` may be drawn just like column item provides the domain for iteration variable `$x`).

Tuples in the XQUERY DM (iter|pos|item$_1$|⋯|item$_n$). Through column extension the loop-lifted processing model is prepared to support *tuple constructors* in the XQUERY DM. While explicit tuple syntax may never make it into XQUERY, its support in the language processor may benefit compilation and optimization. One common opportunity is to trade pairs of transient node construction and subsequent XPATH-based deconstruction (often used to describe the result of grouping in XQUERY) for real tuple processing in the back-end.

3.1 ...LESS COLUMNS

`fn:unordered()` and `unordered{}` (iter|item). Sequence or iteration order may be selectively ignored in the scope of `fn:unordered()` or `unordered{}`. In a loop-lifted translation, this corresponds to the absence (or arbitrary population) of column pos. In [15], we show how this simple algebraic characterization of order indifference can have sweeping effects on query plans and execution time.

Top-level expressions (pos|item). Loop lifting introduces no overhead if there is no enclosing iteration. On the XQUERY top-level, for example, column iter is statically identified to carry the constant 1 (see Figure 3(b)) and thus projected away by the *Pathfinder* compiler. Loop tables are obsolete in such contexts.

Existential semantics (iter). XQUERY's central notion of *effective Boolean value* [7, § 3.4.3] relies on *existential semantics* (refer to the occurrences of built-in function `exists(·)` in Figure 2). In such contexts, sequence position or actual item values are irrelevant: knowledge about the iterations encoded in column iter already suffices to decide the outcome of an application of `exists(·)`.

```
for $o in doc('auction.xml')//open_auctions/open_auction
return
 forseq $w in $o//bidder tumbling window                    ①
  start prevItem $p when xs:decimal($p/increase) gt 0.5
  end   curItem  $c when xs:decimal($c/increase) gt 0.5
where count($w) ge 4                                        ②
return
 <micro> { $p, $w } </micro>
                                                            ③
```

Figure 6: Identify phases of "micro bidding" for all open auctions (Query Q_4). Subexpression annotations in ◯ refer to the subplans in Figure 9.

4. WHEN ITERATION TUMBLES

At the time of writing (May 2008), all odds are that—with the advent of XQUERY 1.1—`for` will lose its monopoly of being the language's only iteration primitive.[2] The new `forseq $w in` ⋯ construct [5] binds its iteration variable `$w` to a non-empty *sequence* (or *window*) of items before the evaluation of the loop body is performed—`for` creates single-item bindings instead. As we will see shortly, these windowed bindings—including the new `prevItem`, `nextItem`, ... options—naturally map onto the loop lifting scheme. The new window border clauses `start` ⋯ `when` and `end` ⋯ `when`, however, violate the principle of iteration independence (Section 1). With loop lifting, this violation surfaces in terms of (sub)plan complexity but we conjecture that any truly set-oriented XQUERY 1.1 processor will face similar intricacies.

Query Q_4 in Figure 6 uses `forseq` to identify phases of "micro bid `increases`" (here: no more than 50 ¢) typically issued by `bidder`s right before an active auction closes. Once such a window of bids has been identified, `$w` is bound to the *sequence* of associated `bidder` elements—in Q_4's `where` clause we can therefore constrain the duration `count($w)` of the overall bid phase and build `micro` result elements that include all relevant bids (we chose to include the two higher bids that frame the micro bid phase).

Sequence bindings. True to the theme of our discussion, let us zoom in on the essentials of iteration and consider the following simple `forseq` instance (Query Q_5):

```
forseq $w in (a,b,c,d,e) tumbling window
  start position $s, prevItem $p when $s = (1,2,4)
  end   position $e, nextItem $n when $e = (3,5)
return $w
```

The clauses `start`/`end position` ⋯ `when` define positional window borders that ultimately lead (see below) to the two sequence bindings `(a,b,c)` and `(d,e)` for `$w`. In a loop-lifted compilation scheme, this translates into the table representing iteration variable `$w` shown here: the `forseq` loop will perform two

`$w`

iter	pos	item
1	1	a
1	2	b
1	3	c
2	1	d
2	2	e

[2] The XQUERY 1.1 Requirements Working Draft lists *windowing* with status SHOULD [8, § 2.3.8] and more than half of the first XQUERY 1.1 Use Cases Working Draft [18] is devoted to the treatment of `forseq`.

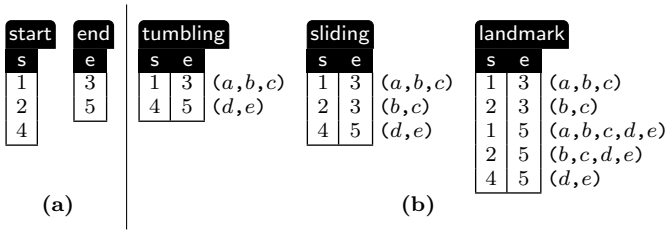


Figure 7: Windows defined over the 5-item sequence (a,b,c,d,e) if the `start` $\cdots$ `when` and `end` $\cdots$ `when` clauses evaluate to *true* for the sequence positions given by tables start and end in (a), respectively (Query Q_5).

```
WITH
landmark(s,e) AS
  (SELECT s,e
   FROM   start, end
   WHERE  s < e),

sliding(s,e) AS
  (SELECT s,(SELECT MIN(l2.e)
             FROM   landmark l2
             WHERE  l2.s = l1.s)
   FROM   landmark l1
   GROUP BY s), ...
```

```
...
tumbling(s,e) AS
  (SELECT s,e
   FROM   sliding s1
   WHERE  NOT EXISTS (
     SELECT 1
     FROM   sliding s2
     WHERE  s2.s < s1.s
     AND    s1.s < s2.e))
```

Figure 8: SQL:1999 simulation of the `forseq` window border automata (tables names refer to Figure 7).

iterations (iter $\in \{1, 2\}$), with `$w` bound to 3-item and 2-item sequences, respectively (note the composite $\langle$iter, pos$\rangle$ key).

The evaluation of the `position`, `prevItem`, `nextItem`, and `curItem` binding clauses is iterated for each item of the input sequence: their associated variables are bound to single items such that the clauses mimic the behavior of the iteration variable (and the optional `at`-bound sequence position variable) of the well-known `for` construct. Loop-lifted encodings of `$s` and `$p` are shown on the side. The representation of `$p`, for example, is obtained through a single equi-join of the representation of the input sequence with itself (effectively shifting column pos by 1, see the subplan labeled `prevItem` in Figure 9). Also, the table contains no row with iter $= 1$, reflecting that a `prevItem` variable is bound to `()` in the first iteration of a `forseq` loop [5].

`$s`

iter	pos	item
1	1	1
2	1	2
3	1	3
4	1	4
5	1	5

`$p`

iter	pos	item
2	1	*a*
3	1	*b*
4	1	*c*
5	1	*d*

Up to here, `forseq` still is a good citizen in terms of the iteration independence notion.

Stateful window borders. Whenever the `start` $\cdots$ `when` clause evaluates to *true* in a given iteration, *zero, one, or more* windows open:

(1) In `tumbling` mode, no new window is created if a window opened in an earlier iteration is still open (tumbling windows do not overlap), otherwise one window opens.
(2) In `sliding` mode, exactly one new window opens.
(3) In `landmark` mode, the number of subsequent *true* evaluations of the `end` $\cdots$ `when` clause determines the number of windows opened.

Whenever the `end` $\cdots$ `when` clause evaluates to *true* (or in the last iteration), for each earlier iteration, one window—if present—is closed.

This is an inherently *stateful* and *order-dependent* semantics of window borders (*e.g.*, in the `tumbling` mode Query Q_5 above, no window is opened in iteration 2 because the *earlier* iteration 1 has switched the processor to *state* "window open"). Indeed, [5] calls upon state automata to define the semantics of `start`/`end` $\cdots$ `when`.

A set-oriented language processor (an RDBMS, for example) can faithfully simulate the transitions of these automata. Dependent on the window mode, however, the system has to jump through one or more hoops to get there. Cast in terms of table operations, a plan has to derive table tumbling (sliding, landmark) from tables start and end to determine the window borders (see Figure 7). In `landmark` mode, a simple θ-join does the job; in the, presumably simpler, modes `sliding` and `tumbling`, a simulation of the stateful semantics involves grouping, aggregation, and anti-semijoin. Figure 8 sketches the resulting SQL queries. Returning to the micro bid phase Query Q_4 and its plan excerpt (Figure 9), the subplan marked `tumbling` further indicates how the algebraic window border logic accounts for a significant runtime effort. In fact, `forseq` stands closer to grouping constructs, like *positional grouping* proposed by Kay in [17], rather than actual iteration.

Optimization requires landmark decisions. The original `forseq` proposal suggests optimizations that trade `landmark` for `sliding` and `sliding` for `tumbling` windows—a stateful, streaming implementation will benefit if less state and smaller item backlogs are to be kept at query runtime [5, § 6.3]. In light of the foregoing, however, it appears that *just the opposite* window preference applies to database-supported processors in the style we have discussed them here: although, in general, `landmark` mode yields the most windows (and thus the most `forseq` loop body evaluations), its window border semantics requires the least runtime investment.

As inhabitants of the database camp, we hope that this lurking cost of `forseq` receives attention while XQuery 1.1 is shaped in the months and years and to come.

5. REFERENCES

[1] ActiveRecord in Ruby on Rails. `http://ar.rubyonrails.org/`.

[2] Ambition (Ruby). `http://ambition.rubyforge.org/`.

[3] S. Amer-Yahia, C. Botev, S. Buxton, P. Case, J. Doerre, M. Holstege, J. Melton, M. Rys, and J. Shanmugasundaram. XQuery 1.0 and XPath 2.0 Full-Text (Working Draft). W3 Consortium, May 2007. `http://www.w3.org/TR/xpath-full-text-10/`.

[4] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. SIGMOD*, Chicago, USA, June 2006.

[5] I. Botan, D. Kossmann, P.M. Fischer, T. Kraska, D. Florescu, and R. Tamosevicius. Extending XQuery With Window Functions. In *Proc. VLDB*, Vienna, Austria, September 2007.

[6] E. Cooper, S. Lindley, P. Wadler, and J Yallop. Links: Web Programming Without Tiers. In *Proc. FMCO*, Amsterdam, Netherlands, 2006.

[7] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and

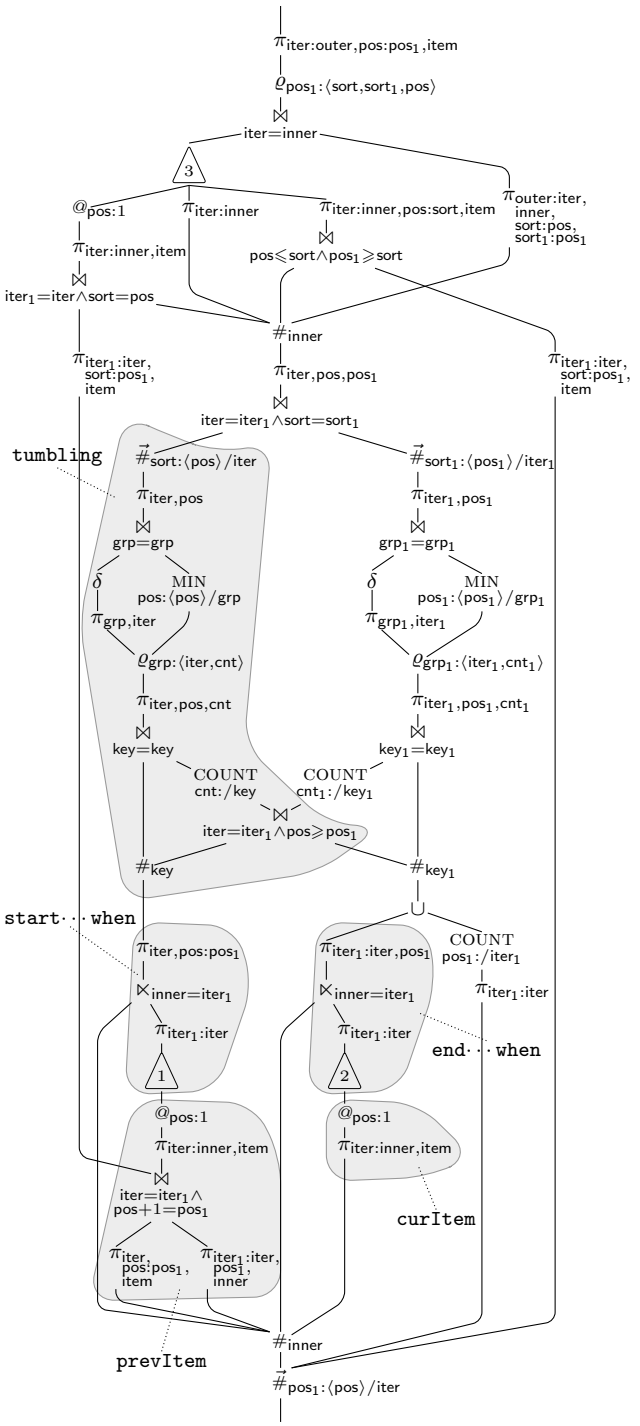


Figure 9: An algebraic account of the XQuery 1.1 `forseq` semantics (plan excerpt for Q_4).

P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3 Consortium, January 2007. `http://www.w3.org/TR/xquery-semantics/`.

[8] D. Engovatov. XQuery 1.1 Requirements (Working Draft). W3 Consortium, March 2007. `http://www.w3.org/TR/xquery-11-requirements/`.

[9] P. Fankhauser, M. Fernández, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 Formal Semantics (Working Draft). W3 Consortium, June 2001. `http://www.w3.org/TR/2001/WD-query-semantics-20010607/`.

[10] R. Goldman and J. Widom. Data Guides: Enabling Query Formulation and Optimization. In *Proc. VLDB*, Athens, Greece, August 1997.

[11] T. Grust. Purely Relational FLWORs. In *Proc. XIME-P*, Baltimore, MD, USA, June 2005.

[12] T. Grust, M. Mayr, J. Rittinger, S. Sakr, and J. Teubner. A SQL:1999 Code Generator for the Pathfinder XQuery Compiler. In *Proc. SIGMOD*, Beijing, China, June 2007.

[13] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. A Ferry Across the Great Database and Programming Languages Divide, 2008. Submitted for publication.

[14] T. Grust, J. Rittinger, and J. Teubner. Data-Intensive XQuery Debugging with Instant Replay. In *Proc. XIME-P*, Beijing, China, June 2007.

[15] T. Grust, J. Rittinger, and J. Teubner. eXrQuy: Order Indifference in XQuery. In *Proc. ICDE*, Istantbul, Turkey, April 2007.

[16] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. VLDB*, Toronto, Canada, 2004.

[17] M. Kay. Positional Grouping in XQuery. In *Proc. XIME-P*, Chicago, USA, June 2006.

[18] T. Kraska. XQuery 1.1 Use Cases (Working Draft). W3 Consortium, March 2008. `http://www.w3.org/TR/xquery-11-use-cases/`.

[19] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling Objects, Relations, and XML in the .NET Framework. In *Proc. SIGMOD*, Chicago, USA, 2006.

[20] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. SIGMOD*, Paris, France, 2004.

[21] S. Peyton Jones. The Haskell 98 Language. *Journal of Functional Programming*, 13(1), 2003.

[22] The Python Programming Language. `http://www.python.org/`.

[23] The Ruby Programming Language. `http://www.ruby-lang.org/`.

[24] A. Schmidt, F. Waas, M. Kersten, M.J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB*, Hong Kong, China, 2002.

[25] J. Teubner. Scalable XQuery Type Matching. In *Proc. EDBT*, Nantes, France, March 2008.

[26] Philip Wadler. Comprehending Monads. In *Proc. ACM Conference on LISP and Functional Programming*, Nice, France, June 1990.