An Inflationary Fixed Point Operator in XQuery

Loredana Afanasiev • Torsten Grust ○ Maarten Marx • Jan Rittinger ○ Jens Teubner ○

• ISLA, University of Amsterdam, Amsterdam, The Netherlands
○ Technische Universität München, Munich, Germany
E-mail: |lafanasi,marx@science.uva.nl|, |grust,rittinge,teubnerj@in.tum.de|

Abstract

We introduce a controlled form of recursion in XQuery, inflationary fixed points, *familiar in the context of relational databases. This imposes restrictions on the expressible types of recursion, but we show that inflationary fixed points nevertheless are sufficiently versatile to capture a wide range of interesting use cases, including the semantics of Regular XPath and its core transitive closure construct.*

While the optimization of general user-defined recursive functions in XQuery appears elusive, we will describe how inflationary fixed points can be efficiently evaluated, provided that the recursive XQuery expressions exhibit a distributivity *property. We show how distributivity can be assessed both, syntactically and algebraically, and provide experimental evidence that XQuery processors can substantially benefit during inflationary fixed point evaluation.*

1. Introduction

The backbone of the XML data model, namely *ordered, unranked trees of nodes*, is inherently recursive and it is natural to equip the associated languages with constructs that can query such recursive structures. To get from the recursive axes in XPath, *e.g.*, `ancestor` and `descendant`, to XQuery's [7] recursive user-defined functions, language designers took a giant leap, however. User-defined functions in XQuery admit *arbitrary* types of recursion—a construct that largely evades optimization approaches beyond "procedural" improvements like tail-recursion elimination or unfolding.

This paper embarks on a journey that explores a controlled form of recursion in XQuery, the *inflationary fixed point (IFP)*, familiar in the context of relational databases [1]. While this imposes restrictions on the expressible types of recursion, IFP embraces a family of widespread use cases of recursion in XQuery, including many forms of horizontal or vertical structural recursion and the pervasive *transitive closure* problem (IFP captures *Regular XPath* [25], in particular).

```
<!ELEMENT curriculum (course)*>
<!ELEMENT course prerequisites>
<!ATTLIST course code ID #REQUIRED>
<!ELEMENT prerequisites (pre_code)*>
<!ELEMENT pre_code #PCDATA>
```

Figure 1. Curriculum data (simplified DTD).

Example 1.1 The DTD of Figure 1 (taken from [22]) describes recursive curriculum data, including courses, their lists of prerequisite courses, the prerequisites of the latter, and so on. The XQuery program of Figure 2 uses the `course` element node with code `"c1"` to seed a computation that recursively finds all prerequisite courses, direct or indirect, of course `"c1"`. For a given sequence `$x` of `course` nodes, function `fix(·)` calls out to `rec(·)` to find their prerequisites. While new nodes are encountered, `fix(·)` calls itself with the accumulated `course` node sequence. (This is not expressible in XPath 2.0.) ◁

Note that `fix(·)` implements a generic fixed point computation: only the initialization (`let $seed := ···`) and the *payload function* `rec(·)` are specific to the curriculum problem. This motivates the introduction of a syntactic form that can succinctly accommodate this pattern of computation (Section 2).

Most importantly, however, such computation in IFP form is susceptible to systematic optimization, provided that the payload (or *body*) of the recursion exhibits a specific *distributivity* property.

Unlike the general user-defined XQuery functions, this account of recursion puts the query processor into control in that it can decide whether the optimization may be safely applied. Distributivity may be assessed on a syntactical level—a non-invasive approach that can easily be realized on top of existing XQuery processors (Section 3). Further, though, if we adopt a relational view of the XQuery semantics (as in [15]), the seemingly XQuery-specific distributivity notion turns out to be elegantly and uniformly tractable on the familiar algebraic level (Section 4).

Compliance with the restriction that IFP imposes on query formulation is rewarded by significant query runtime

```
declare function rec ($cs) as node()*
{ $cs/id (./prerequisites/pre_code)
};

declare function fix ($x) as node()*
{ let $res := rec ($x)
  return if (empty ($x except $res))
         then $res
         else fix ($res union $x)
};

let $seed := doc ("curriculum.xml")
                          /course[@code="c1"]
return fix (rec ($seed))
```

Figure 2. Prerequisites for the course `"c1"` **(marks the fixed point computation).**

savings that the IFP-inherent optimization hook can offer. We document the effect for the XQuery processors *MonetDB/XQuery* [8] and *Saxon* [20] in Section 5. This is primarily due to a substantial reduction of the number of items that are fed into the recursion's payload function (the naïve implementation of Example 1.1 feeds already discovered `course` element nodes back into `rec(·)`).

In Section 6, we stop by related work on recursion on the XQuery as well as the relational side of the fence, and finally wrap-up in Section 7.

2. An Inflationary Fixed Point in XQuery

The subsequent discussion will revolve around the recursion pattern embodied by function `fix(·)` of Figure 2, known as the *inflationary fixed point (IFP)* [1]. We will introduce a new syntactic form to introduce IFP on the XQuery language level and then explore its semantics in the XQuery context, application, and optimization.

In the following, we regard an XQuery expression e_1 containing a free variable `$x` as a function of `$x`. We write $e_1(e_2)$ to denote $e_1[^{e_2}/\texttt{\$x}]$, *i.e.*, the consistent replacement of all free occurrences of `$x` in e_1 by e_2. Function $fv(e)$ returns the set of free variables of expression e. We further introduce *set-equality* ($\stackrel{s}{=}$), a relaxed notion of equality for XQuery item sequences that disregards duplicate items and order, *e.g.*, `(1,"a")` $\stackrel{s}{=}$ `("a",1,1)`.

To streamline the discussion, in the following we assume computations over sequences of type `node()*` as trees are *the* recursive data structure in the XQuery Data Model. In this case, with X_1, X_2 of type `node()*`, we have[1]

$$X_1 \stackrel{s}{=} X_2 \quad \Leftrightarrow \quad \texttt{fs:ddo}(X_1) = \texttt{fs:ddo}(X_2) \ .$$

[1] Here and in the following, `fs:ddo(·)` abbreviates the function `fs:distinct-doc-order(·)` of the XQuery Formal Semantics [9].

An extension to general sequences of type `item()*` is possible and entails the replacement of XQuery's node set operations (`union`, `except`) with appropriate variants.

Definition 2.1 ***(Inflationary Fixed Point)*** Let e_{seed} and $e_{rec}(\texttt{\$x})$ be XQuery expressions of type `node()*`. The *inflationary fixed point (IFP) of* $e_{rec}(\texttt{\$x})$ *seeded by* e_{seed} is an XQuery expression represented by the following syntactic form:

$$\texttt{with \$x seeded by } e_{seed} \texttt{ recurse } e_{rec}(\texttt{\$x}) \ . \qquad (1)$$

The payload expression e_{rec} is called the *body*, e_{seed} is called the *seed*, and `$x` is called the *recursion variable* of the inflationary fixed point operator.

The semantics of the IFP of $e_{rec}(\texttt{\$x})$ seeded by e_{seed} is the sequence of nodes res_k, if it exists, obtained in the following manner:

$$\begin{array}{lcl} res_0 & \leftarrow & e_{rec}(e_{seed}) \\ res_{i+1} & \leftarrow & e_{rec}(res_i)\ \texttt{union}\ res_i \quad , \quad i \geqslant 0 \end{array}$$

where $k \geqslant 1$ is the minimum number for which $res_k \stackrel{s}{=} res_{k-1}$. Otherwise, the IFP of $e_{rec}(\texttt{\$x})$ seeded by e_{seed} is *undefined*. ◁

Note that if expression e_{rec} does *not* invoke node constructors (*e.g.*, `element {·} {·}` or `text {·}`), such that the query operates over a finite domain of nodes, IFP will always be defined. Otherwise, the invocation of node constructors in the recursion body might yield an infinite node domain and IFP might be undefined.

Example 2.2 In terms of the new `with···seeded by···recurse` syntactic form, we can now express the transitive closure query from Example 1.1 in a quite concise and elegant fashion:

```
with $x seeded by doc ("curriculum.xml")
                        /course[@code="c1"]   (Q1)
recurse $x/id (./prerequisites/pre_code)
```

◁

Obviously, the new form `with···seeded by···recurse` is mere syntactic sugar as it can be equivalently expressed via the recursive user-defined function template `fix(·)` (shown in in Figure 2). Since the syntactic form is a second-order construct taking an XQuery variable name and two XQuery expressions as arguments, function `fix(·)` has to be interpreted as a template in which the recursion body `rec(·)` needs to be instantiated (XQuery 1.0 does not support higher-order functions). Given this, Expression (1) is equivalent to the expression `let $x :=` e_{seed} `return fix (rec ($x))`.

Using IFP to Compute Transitive Closure. Much like in the relational context, *transitive closure* is an archetype of recursive computation over XML instances. Regular XPath [25], for example, defines the transitive closure of XPath location steps to obtain powerful primitives that express horizontal and vertical structural recursion. We can naturally extend this definition to any XQuery expression of type `node()*`.

Definition 2.3 ***(Transitive Closure)*** Let e be an expression of type `node()*`. The *transitive closure* e^+ *of* e is

$$e \texttt{ union } e/e \texttt{ union } e/e/e \texttt{ union } \cdots \quad , \tag{2}$$

if the resulting node sequence is finite. Otherwise, e^+ is *undefined*. ◁

Given simple restrictions on e, see Section 3.1, with the new IFP form e^+ is (‘`.`’ denotes the context node):

$$\texttt{with \$x seeded by . recurse \$x/}e \ .$$

IFP in SQL:1999. IFP has found its way into SQL in terms of the `WITH RECURSIVE` clause introduced by the ANSI/ISO SQL:1999 standard [21]. To exemplify, consider the table `C(course, prerequisite)` as a relational representation of the curriculum XML data (Figure 1). The prerequisites `P(course_code)` of the course with code `'c1'` then are:

```
WITH RECURSIVE P(course_code) AS
  (SELECT prerequisite                        )
   FROM   C                                   } seed
   WHERE  course = 'c1')                      )
 UNION ALL
  (SELECT C.prerequisite                      )
   FROM   P, C                                } body
   WHERE  P.course_code = C.course)           )
SELECT DISTINCT * FROM P;
```

Analogous to the XQuery variant, table `P` is seeded with the direct prerequisites of course `'c1'` before the join with table `C` in the body is iterated to also add all indirect prerequisites until `P` does not grow further.

The SQL:1999 standard dictates quite rigid syntactical restrictions for the `WITH RECURSIVE` form (the body, in particular, must be *linear*: `P` may occur only once in its `FROM` clause). We will return to this in Section 3.2 and 6.

2.1. Algorithms for IFP

The semantics of the inflationary fixed point in XQuery, *i.e.*, the specification of the node sequence res_k of Definition 2.1, can be straightforwardly turned into an iterative algorithm to compute IFP. Figure 3(a) shows the resulting

(a) Algorithm *Naïve*	(b) Algorithm *Delta*
$res \leftarrow e_{rec}(e_{seed})$;	$res \leftarrow e_{rec}(e_{seed})$; $\Delta \leftarrow res$;

do
    $res \leftarrow e_{rec}(res)$ `union` res;
while res grows ;
	do
	    $\Delta \leftarrow e_{rec}(\Delta)$ `except` res;
	    $res \leftarrow \Delta$ `union` res;
	while res grows ;

Figure 3. Algorithms to evaluate the IFP of e_{rec} given e_{seed}. Result is res.

```
declare function delta($x,$res) as node()*
{ let $delta := rec($x) except $res
  return if (empty($delta))
         then $res
         else delta($delta,$delta union $res)
};
```

Figure 4. An XQuery formulation of Delta.

procedure, commonly referred to as *Naïve* in the literature [5]. In the **do** $\cdots$ **while** loop body, the procedure calls out to the recursion's payload function $e_{rec}(\cdot)$ to determine the next portion of nodes that will augment the current intermediate result. Only if $e_{rec}(\cdot)$ cannot contribute new nodes, the procedure returns the current res.

Since res grows, this feeds the same nodes over and over again into $e_{rec}(\cdot)$. Dependent on the nature of the payload, $e_{rec}(\cdot)$'s answer might include nodes which we have seen before. Ultimately, *Naïve* risks to initiate a substantial amount of redundant computation.

A now folklore variation of *Naïve* is the *Delta* algorithm [17] of Figure 3(b). In this variant, the payload is invoked only for those nodes that have not been encountered in earlier iterations: node sequence Δ is the difference between $e_{rec}(\cdot)$'s last answer and the current result res. In general, $e_{rec}(\cdot)$ will thus process fewer nodes.

Delta introduces a significant potential for performance improvement, especially for large node sequences and computationally expensive payloads (Section 5). Figure 4 shows the corresponding XQuery user-defined function `delta(·,·)` which, for Example 1.1 and thus Query Q1, can serve as a drop-in replacement for function `fix(·)`—line 14 then needs to be replaced by `return delta(rec($seed),())`.

Is this replacement of `fix(·)` by `delta(·,·)` *always* a valid optimization? For XQuery, the answer is *no*.

Example 2.4 Consider the following expression:

```
let $seed := (<a/>,<b><c><d/></c></b>)
return with $x seeded by $seed
       recurse if (count($x/self::a))                (Q2)
               then $x/* else ()
```

Let a, b, c, and d denote the tree fragments constructed by the seed's subexpressions `<a/>`, `<b><c><d/></c></b>`, `<c><d/></c>`, and `<d/>`, respectively. Thus, b`/*` is c and c`/*` is d.

The table below illustrates the progress of the iterations performed by algorithms *Naïve* and *Delta*. While the former computes (a,b,c,d), the latter returns (a,b,c).

Iteration	*Naïve*	*Delta*	
	res	res	Δ
0	(a,b)	(a,b)	(a,b)
1	(a,b,c)	(a,b,c)	(c)
2	(a,b,c,d)	(a,b,c)	$()$
3	(a,b,c,d)		

$\triangleleft$

What then is an effective characterization of those payloads for which *Naïve* may safely be traded for *Delta*?

3. Trading *Naïve* for *Delta*

We will now see that a simple notion of *distributivity* for XQuery expressions suffices to let an XQuery processor safely switch to a more efficient evaluation mode for `with $x seeded by` e_{seed} `recurse` e_{rec}: whenever expression e_{rec} is *distributive* (in the sense defined below), algorithm *Delta* (Figure 3(b)) preserves the desired IFP semantics. While the *distributivity* property is undecidable in general, we present two safe and effective approximations of distributivity, one formulated on the level of XQuery language syntax, and one cast in terms of an algebraic XQuery representation. The algebraic approximation will turn out to be particularly simple and uniform (Section 4).

3.1. Distributivity in XQuery

Obviously, *Delta* computes the IFP for given expressions e_{seed} and e_{rec} if the algorithm produces the same result as *Naïve* on the same inputs. In particular, the algorithms are equivalent if both yield equivalent intermediate $result$ sequences in each iteration of their **do** $\cdots$ **while** loops.

In its first loop iteration, *Naïve* yields $e_{rec}(e_{rec}(e_{seed}))$ `union` $e_{rec}(e_{seed})$ which is equivalent to *Delta*'s first intermediate result $(e_{rec}(e_{rec}(e_{seed}))$ `except` $e_{rec}(e_{seed}))$ `union` $e_{rec}(e_{seed})$. For the second and further iterations, an inductive proof can show the equivalence of all subsequent intermediate $result$ sequences, if we may assume that, for two item sequences X_1, X_2, we have

$$e_{rec}(X_1 \texttt{ union } X_2) \stackrel{s}{=} e_{rec}(X_1) \texttt{ union } e_{rec}(X_2) \ . \quad (3)$$

For lack of space, we do not reproduce the straightforward equational reasoning behind the proof here but refer to [2].

Note how (3) resembles the *distributivity property* of functions defined on sets. Such a function e is *distributive* if, for all non-empty sets X, $e(X) = \bigcup_{y \in X} e(\{y\})$. This property suggests a divide-and-conquer evaluation strategy in which e is applied to subsets (singletons) of X only. We define the corresponding *distributivity property for XQuery* as follows:

Definition 3.1 ***Distributivity property for XQuery.*** Let e be an XQuery expression in which variable `$x` may occur free. Expression e is *distributive for* `$x` if, for any item sequence $X \neq$ `()` and fresh variable `$y`,

$$\texttt{for \$y in } X \texttt{ return } e(\texttt{\$y}) \quad \stackrel{s}{=} \quad e(X) \ . \quad (4)$$

$\triangleleft$

In particular, Equality (3) is a straightforward consequence if we know that the recursion body e_{rec} is distributive for its free variable. Overall, we arrive at the following sufficient condition for the applicability of *Delta*:

Theorem 3.2 *Consider the expression* `with $x seeded by` e_{seed} `recurse` e_{rec}*. If* e_{rec} *is distributive for* `$x`*, then algorithm* Delta *computes the IFP of* e_{rec} *given* e_{seed}*.*

XPath Location Steps. XPath location steps are a prevalent example of distributive expressions in XQuery. Any expression of the form $e(\texttt{\$x}) = \texttt{\$x/}s$ is distributive for `$x` if the step subexpression s neither contains *(i)* free occurrences of `$x`, nor *(ii)* calls to `fn:position()` and `fn:last()` that refer to the context item sequence bound to `$x`, nor *(iii)* node constructors. To see this, note that the XQuery Core equivalent [9] of `$x/`s is `fs:ddo(for $fs:dot in $x return` s`)`, and then rewrite the lhs of Equation (4) into its rhs, using the definition of $\stackrel{s}{=}$.

Regular XPath. These observations about the distributivity of XPath location steps extend to Regular XPath [25] and thus also make this XPath extension susceptible to *Delta*-based evaluation. Since any Regular XPath step subexpression s is of the form prescribed by *(i)* to *(iii)* above and Regular XPath's transitive closure s^+ is equivalently expressed as `with $x seeded by . recurse $x/`s (for the simple proof see [2]), Theorem 3.2 asserts that we may indeed use algorithm *Delta* to evaluate s^+.

In contrast, expression $e(\texttt{\$x}) = \texttt{\$x[1]}$ is *not* distributive for `$x` in general. With variable `$x` bound to the sequence `(<a/>,<b/>)`, `$x[1]` evaluates to `<a/>`, while `for $y in $x return $y[1]` yields `(<a/>,<b/>)`. Effectively, this invalidates Equation (4).

3.2. Is Expression e_{rec} Distributive? (A Syntactic Approximation)

Whenever an XQuery processor plans the evaluation of `with $x seeded by` e_{seed} `recurse` e_{rec}, knowing the answer to "*Is* e_{rec} *distributive for* `$x`*?*" is particularly valuable: we may legitimately expect *Delta* to be a significantly more efficient IFP evaluation strategy than *Naïve* (Section 5 will indeed make this evident). While, unfortunately, there is no complete procedure to decide this question[2], still we can safely approximate the answer. Here, we will present purely syntactic, sufficient conditions for XQuery distributivity. Section 4 approaches the same challenge on an algebraic level.

Intuitively, we may *not* apply a divide-and-conquer evaluation strategy for an expression $e($`$x`$)$, if any subexpression of e inspects the sequence bound to `$x` as a whole: e is only evaluated after `$x` has been divided into individual items (see Equation 4). Obvious examples of such problematic subexpressions are `count($x)` and `$x[1]`, but also the general comparison `$x = 10` (that involves existential quantification over the sequence bound to `$x`).

Subexpressions whose value is *independent* of `$x`, on the other hand, are distributive. The only exception of this rule are XQuery's node constructors, *e.g.*, `text{·}`, which create new node identities upon each invocation. With `$x` bound to `(<a/>,<b/>)`, for example,

$$\texttt{text\{"c"\}} \overset{s}{\not=} \texttt{for \$y in \$x return text\{"c"\}} \ ,$$

since the rhs will yield a sequence of two distinct text nodes.

The inference rules of Figure 5 have been designed to implement these considerations. The rules syntactically assess the *distributivity safety* $ds_{\texttt{\$x}}(e)$ of an arbitrary LiXQuery [19] input expression e by traversing e's parse tree in a bottom-up fashion. LiXQuery is a sublanguage of XQuery that preserves Turing-completeness, removes all but the most basic types, and excludes selected, rather esoteric, language features. LiXQuery's simplification of the verbose XQuery syntax and semantics have been designed to make LiXQuery ideal for investigations of interesting language properties, yet allow findings to be transposed to full XQuery.

Rules FOR1 and FOR2 ensure that the recursion variable `$x` occurs either in the body e_2 or in the range expression e_1 of a `for`-iteration but not both. This coincides with the linearity constraint of SQL:1999. A similar remark applies to Rules STEP1 and STEP2 (in XQuery, the step operator '`/`' essentially describes an iteration over a sequence of type `node()*` [9]). Also note how Rule FUNCALL recursively infers the distributivity of the body of a called function if the recursion variable occurs free in the function argument(s).

In our context, whenever the XQuery processor is able to infer $ds_{\texttt{\$x}}(e)$ for an input expression e, then it is guaranteed that e is indeed distributive for `$x`. The proof of this implication, by induction on the syntactical structure of e, is to be found in [2].

Distributivity Hints. Still, the inference rules of Figure 5 can only check *sufficient syntactical conditions* for distributivity to hold. The processor might thus actually miss distributive expressions and will fail to infer $ds_{\texttt{\$x}}($`count($x) >= 1`$)$, for example. However, it is interesting to note that we can support the XQuery processor in its distributivity assessment, since every distributive expression is equivalent to a distributivity-safe expression:

> *If expression* $e($`$x`$)$ *is distributive for* `$x`, then it is set-equal to `for $y in $x return` $e($`$y`$)$, for which the rules of Figure 5 will successfully infer distributivity safety $ds_{\texttt{\$x}}(\cdot)$.

This is a direct consequence of Rule FOR2 (Figure 5) and Definition 3.1. Thus, at the expense of a slight query reformulation, we may provide a "syntactic distributivity hint" to the XQuery processor which effectively paves the way for IFP evaluation via algorithm *Delta*.

4. Distributivity and Relational XQuery

In this section we will, literally, follow an alternative route to decide the applicability of *Delta* for the evaluation of the IFP of an XQuery expression e_{rec}. We leave syntax aside and instead inspect *relational algebraic code* that has been compiled for e_{rec}: the equivalent algebraic representation of e_{rec} renders the check for the inherently algebraic distributivity property particularly uniform and simple.

Relational XQuery. This alternative route is inspired by the *Pathfinder* project[3] which fully implements such a purely relational approach to XQuery. *Pathfinder* compiles instances of the XQuery Data Model (XDM) and XQuery expressions into relational tables and algebraic plans over these tables, respectively, and thus follows the dashed path in Figure 6. The translation strategy built into the compiler has been carefully designed *(i)* to faithfully preserve the XQuery semantics (including compositionality, node identity, iteration and sequence order), and *(ii)* yield relational plans which exclusively rely on regular relational query engine technology (no specific operators or index structures are required, in particular) [15].

[2] If, for two arbitrary expression e_1, e_2 in which `$x` does not occur free, an XQuery processor could assess whether `if (deep-equal(`e_1`,`e_2`)) then $x else $x[1]` is distributive for `$x`, it could also decide the equivalence of e_1 and e_2 (which is impossible).

[3] `http://www.pathfinder-xquery.org/`

$$\frac{}{ds_{\texttt{\$x}}(c)}(\textsc{Const}) \quad \frac{}{ds_{\texttt{\$x}}(\texttt{\$}v)}(\textsc{Var}) \quad \frac{\texttt{\$x} \notin fv(e_1) \quad ds_{\texttt{\$x}}(e_2) \quad ds_{\texttt{\$x}}(e_3)}{ds_{\texttt{\$x}}(\texttt{if (}e_1\texttt{) then } e_2 \texttt{ else } e_3)}(\textsc{If}) \quad \frac{\oplus \in \{\texttt{,}, \texttt{|}\} \quad ds_{\texttt{\$x}}(e_1) \quad ds_{\texttt{\$x}}(e_2)}{ds_{\texttt{\$x}}(e_1 \oplus e_2)}(\textsc{Concat})$$

$$\frac{\texttt{\$x} \notin fv(e_1) \quad ds_{\texttt{\$x}}(e_2)}{ds_{\texttt{\$x}}(\texttt{for \$}v\texttt{ at \$}p\texttt{ in } e_1 \texttt{ return } e_2)}(\textsc{For1}) \quad \frac{ds_{\texttt{\$x}}(e_1) \quad \texttt{\$x} \notin fv(e_2)}{ds_{\texttt{\$x}}(\texttt{for \$}v\texttt{ in } e_1 \texttt{ return } e_2)}(\textsc{For2})$$

$$\frac{\texttt{\$x} \notin fv(e_1) \quad ds_{\texttt{\$x}}(e_2)}{ds_{\texttt{\$x}}(\texttt{let \$}v\texttt{ := } e_1 \texttt{ return } e_2)}(\textsc{Let1}) \quad \frac{ds_{\texttt{\$x}}(e_1) \quad \texttt{\$x} \notin fv(e_2) \quad ds_{\texttt{\$}v}(e_2)}{ds_{\texttt{\$x}}(\texttt{let \$}v\texttt{ := } e_1 \texttt{ return } e_2)}(\textsc{Let2})$$

$$\frac{\texttt{\$x} \notin fv(e_1) \quad ds_{\texttt{\$x}}(c_i)_{i=1\dots n+1}}{ds_{\texttt{\$x}}\left(\begin{array}{l}\texttt{typeswitch (}e_1\texttt{)} \\ \quad \texttt{case } \tau_1 \texttt{ return } c_1 \\ \qquad \vdots \\ \quad \texttt{case } \tau_n \texttt{ return } c_n \\ \quad \texttt{default return } c_{n+1}\end{array}\right)}(\textsc{TypeSw}) \quad \frac{\texttt{\$x} \notin fv(e_1) \quad ds_{\texttt{\$x}}(e_2)}{ds_{\texttt{\$x}}(e_1\texttt{/}e_2)}(\textsc{Step1}) \quad \frac{ds_{\texttt{\$x}}(e_1) \quad \texttt{\$x} \notin fv(e_2)}{ds_{\texttt{\$x}}(e_1\texttt{/}e_2)}(\textsc{Step2})$$

$$\frac{\texttt{declare function } f\texttt{(\$}v_1\texttt{,}\dots\texttt{,\$}v_n\texttt{)\{}e_0\texttt{\}} \quad (\texttt{\$x} \in fv(e_i) \Rightarrow ds_{\texttt{\$x}}(e_i) \wedge ds_{\texttt{\$}v_i}(e_0))_{i=1\dots n}}{ds_{\texttt{\$x}}(f\texttt{(}e_1\texttt{,}\dots\texttt{,}e_n\texttt{)})}(\textsc{FunCall})$$

Figure 5. Distributivity-safety $ds_{\texttt{\$x}}(\cdot)$: A syntactic approximation of the distributivity property for LiXQuery expressions.

$$\begin{tikzcd}
\text{XDM} \arrow[r, "\text{XQuery}"] \arrow[d, dashed] & \text{XDM} \\
\text{Tables} \arrow[r, dashed, "\text{Relational Algebra}"'] & \text{Tables} \arrow[u, dashed]
\end{tikzcd}$$

Figure 6. Relational XQuery (dashed path) faithfully implements the XQuery semantics.

The compiler emits a dialect of relational algebra that mimics the capabilities of modern SQL query engines (Table 1). Note that the non-textbook operators, like ε or $⏋$, merely are macros representing "micro plans" composed of standard relational operators: expanding $⏋_{\alpha::n}$ reveals $\bowtie_p$, where p is a conjunctive range predicate that realizes the semantics of an XPath location step along axis α with node test n, for example. The row numbering operator ϱ directly compares with SQL:1999's `ROW_NUMBER`. The plans operate over relational encodings of XQuery item sequence held in flat (1NF) tables with schema iter|pos|item. In these tables, columns iter and pos are used to properly reflect `for`-iteration and sequence order, respectively. Column item carries encodings of XQuery items, *i.e.*, atomic values or nodes.

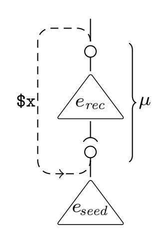
Further details of Relational XQuery do not affect our present discussion of distributivity or IFP evaluation and may be found in [15]. In the following, let $\triangle_e$ denote the algebraic plan that has been compiled for XQuery expression e.

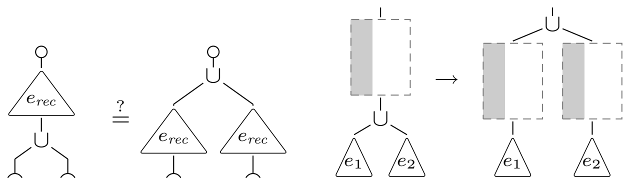
Operator	Semantics	Push?
$\pi_{\mathsf{a}_1:\mathsf{b}_1,\dots,\mathsf{a}_n:\mathsf{b}_n}$	project onto col.s a_i, rename b_i into a_i	$\odot$
σ_{b}	select rows with column $\mathsf{b} = \mathit{true}$	$\odot$
$\bowtie_p$	join with predicate p	$\oplus$
$\times$	Cartesian product	$\oplus$
δ	duplicate elimination (`DISTINCT`)	$-$
$\cup$	union	$\oplus$
$\setminus$	disjoint difference (`EXCEPT ALL`)	$-$
$\mathtt{count}_{\mathsf{a}:/\mathsf{b}}$	aggregates (group by b, result in a)	$-$
$⊚_{\mathsf{a}:\langle \mathsf{b}_1,\dots,\mathsf{b}_n\rangle}$	n-ary arithmetic/comparison operator $\circ$	$\odot$
$\#_{\mathsf{a}}$	unique row tagging (tag in a)	$\odot$
$\varrho_{\mathsf{a}:\langle \mathsf{b}_1,\dots,\mathsf{b}_n\rangle/p}$	ordered row numbering (by $\mathsf{b}_1,\dots,\mathsf{b}_n$)	$-$
$⏋_{\alpha::n}$	XPath step join (axis α, node test n)	$\odot$
$\varepsilon, \tau, \dots$	node constructors	$-$
μ, μ^{Δ}	fixpoint operators	$\odot$

Table 1. Relational algebra dialect emitted by the Pathfinder compiler.

4.1. Is Expression e_{rec} Distributive? (An Algebraic Account)

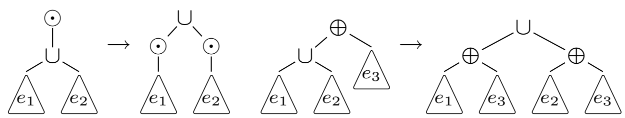
An occurrence of the new `with $x seeded by` e_{seed} `recurse` e_{rec} form in a source XQuery expression will be compiled into a plan fragment as shown here on the right. Operator μ, the algebraic representation of algorithm *Naïve* (Figure 3(a)), iterates the evaluation of the algebraic plan for e_{rec} and feeds its output $♀$ back to its input $\perp$ until the IFP is reached. If we can guarantee that the plan for e_{rec} is distributive, we may safely trade μ for its *Delta*-based variant μ^{Δ}





(a) Is e_{rec} distributive? (b) Taking a *big step*: Pushing $\cup$ through a plan template

Figure 7. Algebraic distributivity assessment.



(a) Unary operators $\odot$ (b) Binary operators $\oplus$

Figure 8. Pushing $\cup$ through unary ($\odot$) and binary ($\oplus$) operators.

which, in general, will feed significantly less items back in each iteration (see Figure 3(b) and Section 5).

In this algebraic setting, if the recursion body e_{rec} is distributive, its relational plan will satisfy the equality shown in Figure 7(a). This equality is the algebraic expression of a divide-and-conquer evaluation strategy for e_{rec} (Section 3.1): evaluating e_{rec} over a composite input (lhs, ⅄∪⅄) yields the same result as the union of the evaluation of e_{rec} over a partitioned input (rhs). Effectively, the union operator $\cup$ has been completely pushed up through all branches of the DAG-shaped algebraic plan for e_{rec}. Zooming in from the plan to the operator level, Figure 8 depicts how $\cup$ is pushed up through unary ($\odot$) and binary ($\oplus$) operators. Column 'Push?' of Table 1 indicates whether $\cup$ may indeed be validly pushed through a given operator. Note that this push through is prohibited by exactly those operators that require to consume their *complete* input to produce the result. This affects, *e.g.*, aggregates, difference, and the row numbering operator. As before, the occurrence of node constructors renders e_{rec} non-distributive.

Because our primary goal is distributivity assessment (as opposed to query *evaluation*—but see Section 5), we may actually employ simplified variants of e_{rec} in this context. In particular, since the definition of distributivity disregards duplicates and order (Definition 3.1), the compiler may choose to remove code from e_{rec} that is used to eliminate duplicate nodes after XPath location steps as well as omit those parts of the plan that realize the proper XQuery order semantics [14].

Further, the plans generated by the XQuery compiler typically contain numerous instantiations of *plan templates*, closed plan fragments with single entry and exit points (enclosed by in Figure 9). These templates embody algebraic implementations of basic XQuery constructs, *e.g.*, the semantics of `for`-iteration or XPath location steps. Assessing the distributivity of such plan templates is a one time effort. Once this has been done, whenever a distributive template is encountered, the $\cup$ push up process may disregard the template's contents and instead perform a single *big step* across the template (see Figure 7(b)).

For the XQuery processor, this suggests the following simple procedure as a replacement for $ds_{\texttt{\$x}}(\cdot)$ (Section 3.2) to assess the distributivity of e_{rec}:

> Start with the algebraic plan for e_{rec} with its input ⅄ replaced by ⅄∪⅄;
> **while** not all $\cup$ have reached ⚲ **do**
>   Perform a *big step* or push $\cup$ up through its parent operator, if possible. Otherwise **return** $false$;
> **return** $true$;

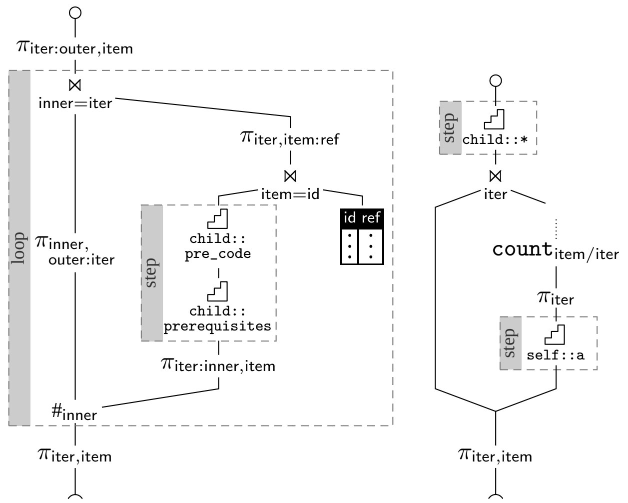
Figure 9 depicts the algebraic representations of the recursion bodies of the Queries Q1 and Q2 (Section 2). For Query Q1, to push $\cup$ through from ⅄ to ⚲, the distributivity check will succeed after it has performed two steps across the two peripheral projections plus one intermediate *big step* across the `for`-iteration that implements the semantics of the `$x/id(·)` lookup. For Query Q2, $\cup$ will be pushed through $\pi_{\mathsf{iter,item}}$ and then upwards the two branches of the DAG-shaped plan. In the right branch, the aggregate $\mathtt{count}_{\mathsf{item/iter}}$ blocks the process (Table 1) which indicates that the processor may *not* use algorithm *Delta* (or the μ^{Δ} variant of the fixed point operator) to evaluate Query Q2.

Algebraic vs. Syntactic Approximation. Compared to the syntactic approximation $ds(\cdot)$, this algebraic account of distributivity draws its conciseness from the fact that the rather involved XQuery semantics and substantial number of built-in functions nevertheless map to a small number of algebraic primitives (given suitable relational encodings of the XDM). Further, for these primitives, the algebraic distributivity property is readily decided.

To make this point, consider this slight yet equivalent variation of Query Q1 in which variable `$x` now occurs free in the argument of function `id(·)`:

```
with $x seeded by doc("curriculum.xml")
                    /course[@code="c1"]
recurse id($x/prerequisites/pre_code)  .
```

If we unfold the implementation of the XQuery built-in function `id(·)` (effectively, this expansion is performed when Rule FUNCALL recursively invokes $ds_{\texttt{\$x}}(\cdot)$ to assess



(a) e_{rec} of Query Q1 (b) e_{rec} of Query Q2

Figure 9. Relational representations of the recursion bodies e_{rec} of Queries Q1 and Q2.

the distributivity of the function body of `id(·)`), we obtain

```
with $x seeded by doc("curriculum.xml")
                    /course[@code="c1"]
recurse
  for $c in doc("curriculum.xml")/course
  where $c/@code = $x/prerequisite/pre_code
  return $c  .
```

The syntactic approximation will flag the recursion body as non-distributive because of the general comparison (=) in the `where` clause (Section 3.2). While the algebraic approach would be unaffected by the variation, the rule set of Figure 5 would need a specific rule for `id(·)` to be able to infer its actual distributivity.

5. Practical Impact of Distributivity and *Delta*

Recasting a recursive XQuery query as an inflationary fixed point computation imposes restrictions. Such recasting, however, also puts the query processor into control since the applicability of a promising optimization, trading *Naïve* for *Delta*, becomes effectively decidable. This section provides the evidence that significant gains can indeed be realized, much like in the relational domain.

To quantify the impact, we implemented the two fixed point operator variants μ and μ^{Δ} (Section 4.1) in *MonetDB/XQuery 0.18* [8], an efficient and scalable XQuery processor that consequently implements the Relational XQuery approach (Section 4). Its algebraic compiler front-end *Pathfinder* has been enhanced *(i)* to process the syntactic form `with···seeded by···recurse`,

```
declare variable $doc := doc("auction.xml");

declare function bidder ($in as node()*) as node()*
{ for $id in $in/@id
  let $b := $doc//open_auction[seller/@person = $id]
              /bidder/personref
  return $doc//people/person[@id = $b/@person]
};

for $p in $doc//people/person
return <person>
         { $p/@id }
         { data ((with $x seeded by $p
                  recurse bidder ($x))/@id) }
       </person>
```

Figure 10. XMark bidder network query.

and *(ii)* to implement the algebraic distributivity check. All queries in this section were recognized as being distributive by *Pathfinder*. To demonstrate that any XQuery processor can benefit from optimized IFP evaluation in the presence of distributivity, we also performed the transition from *Naïve* to *Delta* on the XQuery source level and let *Saxon-SA 8.9* [20] process the resulting user-defined recursive queries (cf. Figures 2 and 4). All experiments were conducted on a Linux-based host (64 bit), with two 3.2 GHz Intel Xeon® CPUs, 8 GB of primary and 280 GB SCSI disk-based secondary memory.

Table 2 summarizes our observations for four query types, chosen to inspect the systems' behavior for growing input XML instance sizes and varying result sizes at each recursion level (the maximum recursion depth ranged from 5 to 33).

XMark Bidder Network. To assess scalability, we computed a bidder network—recursively connecting the sellers and bidders of auctions (Figure 10)—over XMark [24] XML data of increasing size (from scale factor 0.01, small, to 0.33, huge). If *Delta* is used to compute the IFP of this network, *MonetDB/XQuery* (2.2 to 3.3 times faster) as well as *Saxon* (1.2 to 2.7 times faster) benefit significantly. Most importantly, note that the number of nodes in the network grows quadratically with the input document size. Algorithm *Delta* feeds significantly less nodes back in each recursion level which positively impacts the complexity of the value-based join inside recursion payload `bidder(·)`: for the huge network, *Delta* exactly feeds those 10 million nodes into `bidder(·)` that make up the result—*Naïve* repeatedly revisits intermediate results and processes 9 times as many nodes.

Romeo and Juliet Dialogs. Far less nodes are processed by a recursive expression that queries XML markup of

Query	*MonetDB/XQuery* Naïve	*MonetDB/XQuery* Delta	*Saxon-SA 8.9* Naïve	*Saxon-SA 8.9* Delta	Total # of Nodes Fed Back Naïve	Total # of Nodes Fed Back Delta	Recursion Depth
Bidder network (small)	362 ms	165 ms	2,307 ms	1,872 ms	40,254	9,319	10
Bidder network (medium)	5,010 ms	1,995 ms	15,027 ms	7,284 ms	683,225	122,532	16
Bidder network (large)	40,785 ms	13,805 ms	123,316 ms	52,436 ms	5,694,390	961,356	15
Bidder network (huge)	9 m 46 s	176,890 ms	32 m 40 s	12 m 04 s	87,528,919	9,799,342	24
Romeo and Juliet	6,795 ms	1,260 ms	1,150 ms	818 ms	37,841	5,638	33
Curriculum (medium)	183 ms	135 ms	1,308 ms	1,040 ms	12,301	3,044	18
Curriculum (large)	1,466 ms	646 ms	3,485 ms	2,176 ms	127,992	19,780	35
Hospital (medium)	734 ms	497 ms	1,301 ms	1,290 ms	99,381	50,000	5

Table 2. Naïve vs. Delta: Comparison of query evaluation times and total number of nodes fed back.

Shakespeare's Romeo and Juliet[4] to determine the maximum length of any uninterrupted dialog. Seeded with `SPEECH` element nodes, each level of the recursion expands the currently considered dialog sequences by a single `SPEECH` node given that the associated `SPEAKER`s are found to alternate (horizontal structural recursion along the `following-sibling` axis—we do not reproduce the query here for space reasons.) Although the recursion is shallow (depth 6 on average), Table 2 shows how both, *MonetDB/XQuery* and *Saxon*, completed evaluation up to 5 times faster because the query had been specified in a distributive fashion.

Transitive Closures. Two more queries, taken directly from related work [22, 11], compute transitive closure problems (we generated the data instances with the help of ToXgene [6]). The first query implements a consistency check over the curriculum data (cf. Figure 1) and finds courses that are among their own prerequisites (Rule 5 in the Curriculum Case Study in Appendix B of [22]). Much like for the bidder network query, the larger the query input (medium instance: 800 courses, large: 4,000 courses), the better *MonetDB/XQuery* and *Saxon* exploited *Delta*.

The last query in the experiment explores 50,000 hospital patient records to investigate a hereditary disease [11]. In this case, the recursion follows the hierarchical structure of the XML input (from patient to parents), recursing into subtrees of a maximum depth of 5. Again, *Delta* makes a notable difference even for this computationally rather "light" query.

We believe that this renders this particular controlled form of XQuery recursion and its associated distributivity notion attractive, even for processors that do not implement a dedicated fixed point operator (like *Saxon*).

[4] `http://www.ibiblio.org/xml/examples/shakespeare/`

6. More Related Work

Bringing adequate support for recursion to XQuery is a core research matter on various levels of the language. While the efficient evaluation of the recursive XPath axes (*e.g.*, `descendant` or `ancestor`) is well understood by now [3, 16], the optimization of recursive user-defined functions has been found to be tractable only in the presence of restrictions: [23, 13] propose exhaustive inlining of functions but require that functions are *structurally* recursive (use axes `child` and `descendant` to navigate into subtrees only) over *acyclic* schemata to guarantee that inlining terminates. Note that, beyond inlining, this type of recursion does not come packaged with an effective optimization hook comparable to what the inflationary fixed point offers.

The distinguished use case for inflationary fixed point computation is transitive closure. This is also reflected by the advent of XPath dialects like Regular XPath [25] and the inclusion of a dedicated `dyn:closure(·)` construct in the EXSLT function library [10]. We have seen applications in Section 5 [22, 11] and recent work on data integration and XML views adds to this [12].

In the domain of relational query languages, *Naïve* is the most widely described algorithmic account of the inflationary fixed point operator [5]. Its optimized *Delta* variant, in focus since the 1980's, has been coined *delta iteration* [17], *semi-naïve* [5], or *wavefront* [18] strategy in earlier work.

Since our work rests on the adaption of these original ideas to the XQuery Data Model and language, the large "relational body" of work in this area should be directly transferable, even more so in the Relational XQuery context. In particular, optimization techniques like *Magic Set* rewriting [4] should apply (this has not been investigated in the present paper).

The adoption of inflationary fixed point semantics by Datalog and SQL:1999 with its `WITH RECURSIVE` clause (Section 2) led to investigations of the applicability of *Delta* for these recursive relational query languages. For stratified Datalog programs [1], *Delta* is applicable in *all* cases: positive Datalog maps onto the distributive operators of relational algebra (π, σ, $\bowtie$, $\cup$, $\cap$) while stratification yields partial applications of the difference operator $x \setminus R$ in which R is fixed ($f(x) = x \setminus R$ is distributive).

SQL:1999, on the other hand, imposes rigid *syntactical* restrictions [21] on the iterative fullselect (recursion body) inside `WITH RECURSIVE` that make *Delta* applicable: grouping, ordering, usage of column functions (aggregates), and nested subqueries are ruled out, as are repeated references to the virtual table computed by the recursion. Replacing this coarse syntactic check by an algebraic distributivity assessment (Section 4) would render a larger class of queries admissible for efficient fixed point computation.

7. Wrap-Up

This paper may be read in two ways:

(i) As a proposal to add an inflationary fixed point construct, along the lines of `with` $\cdots$ `seeded by` $\cdots$ `recurse`, to XQuery (this has actually been discussed by the W3C XQuery working group in the very early XQuery days of 2001[5] but then dismissed because the group aimed for a first-order language design at that time).

(ii) As a guideline for query authors as well XQuery processor designers to check for and then exploit distributivity during the evaluation of recursive queries.

We have seen how such distributivity checks can be used to safely unlock the optimization potential, namely algorithm *Delta*, that comes tightly coupled with the inflationary fixed point semantics. *MonetDB/XQuery* implements this distributivity check on the algebraic level and significantly benefits whenever the *Delta*-based operator μ^{Δ} may be used for fixpoint computation. Even if the approach is realized on the coarser syntactic level *on top of* an existing XQuery processor, feeding back less nodes in each recursion level yields substantial performance improvements.

Remember that the distributivity notion suggests a divide-and-conquer evaluation strategy (Section 3.1) in which parts of a computation may be performed independently (before a merge step forms the final result). Beyond recursion, this may lead to improved XQuery compilation strategies for back-ends that can exploit such independence, *e.g*, set-oriented relational query processors (cf. loop-lifting [15]) as well as parallel or distributed execution platforms.

Acknowledgments. Loredana Afanasiev is supported by the Netherlands Organization for Scientific Research (NWO) under project number 612.000.207. Jan Rittinger is supported by the German Research Foundation (DFG) under grant GR 2036/2-1. We thank Massimo Franceschet for input in an early stage of this work.

[5] `http://www.w3.org/TR/2001/WD-query-semantics-20010607/` (Issue 0008).

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.
[2] L. Afanasiev. Distributivity for XQuery Expressions. Technical report, University of Amsterdam, 2007. Available at `http://staff.science.uva.nl/~lafanasi/Publications/2007/dist`
[3] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. ICDE*, 2002.
[4] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. PODS*, 1985.
[5] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proc. SIGMOD*, 1986.
[6] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: A template-based Data Generator for XML. In *Proc. SIGMOD*, 2002.
[7] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. W3C Recommendation, 2007.
[8] 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*, 2006.
[9] D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Recommendation, 2007.
[10] EXSLT: A Community Initiative to Provide Extensions to XSLT. `http://www.exslt.org/`.
[11] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. SMOQE: A System for Providing Secure Access to XML. In *Proc. VLDB*, 2006.
[12] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Rewriting Regular XPath Queries on XML Views. In *Proc. ICDE*, 2007.
[13] M. Grinev and D. Lizorkin. XQuery Function Inlining for Optimizing XQuery Queries. In *Proc. ADBIS*, 2004.
[14] T. Grust, J. Rittinger, and J. Teubner. eXrQuy: Order Indifference in XQuery. In *Proc. ICDE*, 2007.
[15] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. VLDB*, 2004.
[16] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. VLDB*, 2003.
[17] U. Güntzer, W. Kieling, and R. Bayer. On the Evaluation of Recursion in (Deductive) Database Systems by Efficient Differential Fixpoint Iteration. In *Proc. ICDE*, 1987.
[18] J. Han, G. Z. Qadah, and C. Chaou. The Processing and Evaluation of Transitive Closure Queries. In *Proc. EDBT*, 1988.
[19] J. Hidders, P. Michiels, J. Paredaens, and R. Vercammen. LiXQuery: A Foundation for XQuery Research. *SIGMOD Record*, 3(4), 2005.
[20] M. Kay. The Saxon XSLT and XQuery Processor. `http://saxon.sf.net/`.
[21] J. Melton and A. R. Simon. *SQL: 1999 - Understanding Relational Language Components*. Morgan Kaufmann, 2002.

[22] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2), 2002.
[23] C.-W. Park, J.-K. Min, and C.-W. Chung. Structural Function Inlining Technique for Structurally Recursive XML Queries. In *Proc. VLDB*, 2002.
[24] 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*, 2002.
[25] B. ten Cate. Expressivity of XPath with Transitive Closure. In *Proceedings of PODS*, pages 328–337, 2006.