Pathfinder: XQuery Off the Relational Shelf

Torsten Grust Jan Rittinger
Universität Tübingen, Germany
torsten.grust@uni-tuebingen.de
jan.rittinger@uni-tuebingen.de

Jens Teubner
ETH Zürich, Switzerland
jens.teubner@inf.ethz.ch

Abstract

The Pathfinder project makes inventive use of relational database technology—originally developed to process data of strictly tabular shape—to construct efficient database-supported XML and XQuery processors. Pathfinder targets database engines that implement a set-oriented mode of query execution: many off-the-shelf traditional database systems make for suitable XQuery runtime environments, but a number of off-beat storage back-ends fit that bill as well. While Pathfinder has been developed with a close eye on the XQuery semantics, some of the techniques that we will review here will be generally useful to evaluate XQuery-style iterative languages on database back-ends.

1 The Rectangularization of XQuery: Purely Relational XML Processing

If you zoom back in time to dig for the semantic roots of XQuery [5], you will find that the language's core construct, the `for–let–where–order by–return` (FLWOR) block is one particular incarnation of a very general idea: the *comprehension* [26]. Many language-related concepts may be uniformly understood in comprehension form, but comprehensions provide a particularly concise and elegant way to express iteration over collections of objects—in the case of XQuery: finite, ordered sequences of XML nodes and atomic values (or *items*) [1]. Any program or query expressed in comprehension form is subject to a number of useful equivalence-preserving rewriting rules (the *monad laws*) and so is XQuery's FLWOR block. Once you look closely, a wide range of seemingly XQuery-specific optimizations realized by compilers and interpreters today, *e.g.*, `for` loop fusion or unnesting, in fact put the monad laws to work.

The family of programming and query languages whose semantic core may be cast in comprehension form is large. Among its members, specifically, is SQL, *the* relational database language. This observation sparked a whole line of work that we will review in the following pages:

> Exploit the common semantic ground of XQuery and SQL and try to turn relational database systems (*i.e.*, processors for strictly tabular, or rectangular, data) into efficient and scalable XQuery processors.

XQuery processors of this type should be able to benefit from the 30+ years of research and engineering experience that shaped relational database technology. This is, in fact, what we repeatedly observed in the course of

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

the *Pathfinder* project (initiated in late 2001), an effort to construct a *Purely Relational XQuery Processor* [25]. As of today, *Pathfinder* can compile XQuery expressions into code (different variants of table algebras or SQL) ready for consumption by relational-style database back-ends. The back-ends evaluate this code against tabular encodings of XML instances and item sequences and thus *act like* XQuery processors. We have found the resulting systems to exhibit runtime performance characteristics that often surpass specifically-built "native" XQuery engines. On top of that, standing on the shoulders of relational giants provides stability, scalability, instant and wide availability as well as the seamless coexistence of XML instances and tabular data.

Input to this purely relational approach to XQuery processing are relational encodings of the XML data model, *i.e.*, ordered unranked trees of nodes of several kinds. We thus start our tour of the *Pathfinder* technology in 2 with a brief review of suitable tabular encodings of XML and then see how XPath location steps may be efficiently evaluated in terms of (self-)joins over the resulting tables. We turn to XQuery's dynamic semantics in 3 and sketch *loop lifting*, a compilation strategy that derives efficient set-oriented execution plans from nested FLWOR blocks. 4 shows how a purely relational account of the XQuery semantics can provide insights and optimization hooks that would be hard to find and formulate on the XQuery language level. Different kinds of database systems have already been turned into *Pathfinder* back-ends. 5 discusses selected systems and how they fare in their new role as XQuery runtime environments. Finally, as we said, the comprehension construct can explain aspects of a large family of languages: 6 sheds some light on how other recent programming and query language proposals with an iterative core could be "rectangularized"—and thus put on top of relational back-ends.

2 How Many Rows Does Your Tree Have? (Tabular XML Encoding)

The performance of any RDBMS-backed solution depends critically on how its data is represented in the relational format, tables of tuples. A purely relational XQuery processor is no different in this respect and the choice of a good relational *tree encoding* is an important factor to the functioning of a relational XQuery setup. Two principal features must be provided by the XML-to-tables translation, both dictated by the semantics of XQuery:

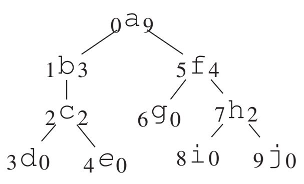
(i) It has to maintain XML *document order* and XQuery's concept of *node identity*. More explicitly, we expect the existence of a *surrogate* γ_v for each node v such that $\gamma_{v_1} = \gamma_{v_2}$ iff v_1 `is` v_2 and $\gamma_{v_1} < \gamma_{v_2}$ iff v_1 `<<` v_2.

(ii) Efficient mechanisms must exist that implement core operations on XML data. In particular, given a node surrogate γ_v, there must be a way to compute all surrogates for the node sequence $v/ax{::}nt$, where ax and nt are axis and node test of an XPath location step, respectively.

A variety of encodings has been published which provide both features, including ORDPATH [21], dynamic intervals [3], or XPath accelerator [14]. *Pathfinder* uses a variant of the latter, which we illustrate in a moment. As a drop-in replacement, the others could be plugged into *Pathfinder* seamlessly.

Pathfinder's relational XML storage, represents documents as a five-column table as shown in Figure 1 on the right for the XML instance

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



(a) XML document tree.

pre	size	level	kind	prop
0	9	0	elem	a
1	3	1	elem	b
2	2	2	elem	c
3	0	3	elem	d
4	0	3	text	e
5	4	1	elem	f
⋮	⋮	⋮	⋮	⋮

(b) Relational tree encoding.

Figure 1: XML document tree, annotated with pre(·) and size(·) information (left/right), and resulting tree encoding.

For each node v, the table holds v's rank in a preorder tree traversal, $\mathsf{pre}(v)$, the number of descendants below v, $\mathsf{size}(v)$, its distance to the tree root, $\mathsf{level}(v)$, and the two columns kind and prop to represent XML information set characteristics of v, *i.e.*, its XML node type (one of elem, text, attr, $\ldots$) its tag name or typed value (depending on $\mathsf{kind}(v)$, refer to [14] for details). It is easy to see that $\mathsf{pre}(v)$ provides a suitable implementation for γ_v.

Evaluating XPath. Based on $\mathsf{pre}(\cdot)$, $\mathsf{size}(\cdot)$, and $\mathsf{level}(\cdot)$ properties, all twelve XPath axes can be characterized in a concise and machine-friendly manner. For axis `descendant`, *e.g.*, we have

$$\begin{array}{c} v' \in v/\texttt{descendant} \\ \Leftrightarrow \\ \mathsf{pre}(v) < \mathsf{pre}(v') \leq \mathsf{pre}(v) + \mathsf{size}(v) \end{array} . \qquad (1)$$

Range expressions of this kind lend themselves to the use of *B-tree indexes* for efficient XPath evaluation. And, in fact, with appropriate index support, a relational XPath evaluation setup can outperform industry-strength "native" XML processors by significant margins [11].

The relational system plays its trump by organizing and indexing the relational tree encoding in the way that fits any given XPath query workload best. In [11], we found that *partitioned B-trees* form a particularly interesting class of indexes for XPath processing. The physical layout of a partitioned $\langle \mathsf{level}, \mathsf{pre} \rangle$ index, for instance, ideally matches the access pattern of an XPath `child` step. In addition, since a k-step XPath expression compiles into a k-way self-join at the relational end, the system's optimizer can gear the *order* of these joins to its liking [7]. This way, an *off-the-shelf* RDBMS solves formerly challenging problems in a purely mechanical way. This includes rewriting into forward-only plans [20] or top-down *vs.* bottom-up XPath evaluation [17].

Tree Awareness with Staircase Join. With a suitable tree encoding and the right selection of indexes, we enabled the relational system to act as an efficient tree processor, even though the system remained entirely unaware of the tree-structure that the encoded data originated from. Additional performance gains can be achieved by *injecting* such awareness into the RDBMS kernel.

Staircase join [13] is such an injection that can significantly improve relational XPath performance with only a local change to the RDBMS kernel. While evaluating an XPath location step, staircase join provides tree awareness by

- *pruning* nodes from the context set whose result nodes are already produced by other context nodes,
- *partitioning* the document space to (a) guarantee a duplicate-free result, sorted in document order and (b) achieve a strictly sequential, hence cache-efficient, access pattern to the underlying storage, and
- *skipping* parts of the document table which are early discovered (based on knowledge about the tree-origin) to not contain any result candidates.

Injecting staircase join into a main memory-oriented database system [13] or a traditional disk-based system [16] took only little changes to the systems' code. The change in runtime performance, however, was significant: we observed speed-ups of several orders of magnitude on both systems.

Types are Data. Other than traditional database query languages, XQuery blurs the distinction between data and its type. XML Schema types, *e.g.*, can be used as node tests in XPath location steps, just like tag names or node kinds. Likewise, the *runtime type* of arbitrary XQuery items can dynamically be inspected using the `instance of` and `typeswitch` constructs just like the item's value. A relational encoding for XQuery type annotations types, therefore, is called for.

The type system of XQuery, incidentally, has a structure that we already know how to deal with efficiently. All XML Schema types relate to each other in a *tree shape*. Pairs of pre and size values ("type ranks") are a

```
for $_ in (0) return
  for $x in (5,6,7) return
    if ($x mod 2 ne 0)
    then ("odd",xs:string($x))
    else "even"
```

(a) Sample XQuery FLWOR block.

iter	pos	item
1	1	5
1	2	6
1	3	7

(b) `(5,6,7)`

iter	pos	item
1	1	"odd"
1	2	"5"
2	1	"even"
3	1	"odd"
3	2	"7"

(c) $(\underbrace{\texttt{"odd","5"}},\underbrace{\texttt{"even"}},\underbrace{\texttt{"odd","7"}})$

Figure 2: An XQuery FLWOR block with the loop-lifted representations of its binding sequence and result. The gray outer pseudo loop establishes a single-iteration context for the top-level item sequence `(5,6,7)`.

logical way to account for that, with pre(·) as a concise implementation for type annotations. But the virtue of using type ranks is not only their support for tree navigation. As we showed in [23], type ranks enable interesting, database-style evaluation strategies for queries on types. *Type aggregation*, *e.g.*, can accelerate the processing of `instance of` or `typeswitch` clauses with sequence-valued input. Type-constrained XPath expressions can profit from relational indexes over columns that encode type information, or even from combined type/data indexes.

3 Drawing Independent Work from Lack of Side Effects

Pathfinder's main XQuery compilation strategy, dubbed *loop lifting*, revolves around the XQuery FLWOR block as the main language construct: *any* subexpression e is considered to be iteratively evaluated in the scope of its innermost enclosing `for` loop (if e is a top-level expression, we install a pseudo single-iteration loop `for $_ in (0) return` e such that variable `$_` does not occur free in e). In line with the comprehension notion, the FLWOR block

$$\texttt{for \$x in (}v_1\texttt{,}v_2\texttt{,}\ldots\texttt{,}v_n\texttt{) return } e$$

describes the n-fold *side effect-free* evaluation of e under unmodifiable bindings of `$x` to items v_i. The result will be $(e[v_1/\$x],\ e[v_2/\$x],\ \ldots,\ e[v_n/\$x])$ (note that the resulting sequence will be flat according to the XQuery data model). Since the individual evaluations of loop body e cannot interfere, the system may perform the evaluation in any order or even in parallel [4, § 4.8.2]. This leads to a significant load of *independent work*, the principal source of set-orientation and potential parallelism in *Pathfinder*-generated query plans.

The "Great Invariant." To implement this idea on a relational back-end, *Pathfinder* compiles an XQuery subexpression e into an algebraic plan fragment that, at runtime, will yield a ternary table encoding e*'s result for all its* n *iterated evaluations* [6, 12]. These tables uniformly adhere to the schema iter|pos|item in which a row $\langle i, p, v\rangle$ indicates that, in the ith iteration, the evaluation of e returned a sequence in which item v occurs at position p—in a sense, we obtain a fully unrolled representation of the result of e's enclosing `for` loop.

Consider the sample XQuery FLWOR block in Figure 2a. The top-level binding sequence `(5,6,7)` is evaluated once only while the inner `for` loop body undergoes three individual evaluations and thus contributes three subsequences (marked by $\sqcup$ in Figure 2c) to the final result (to illustrate: row $\langle 3, 1, \texttt{"odd"}\rangle$ indicates that the third iteration contributes a sequence with item `"odd"` at position 1).

This "great invariant" drives the design of the whole compiler and enables a truly compositional style of translation from XQuery to relational algebra—prepared to cope with `for` loop nesting hierarchies of arbitrary depth. Loop-lifted algebraic plans diverge from the classical σ-π-$\bowtie$ pattern emitted by SQL compilers: instead, the plans exhibit a narrow "stacked" shape [7] reflecting the orthogonal expression nesting that is typical for a functional language like XQuery. Figure 3 sketches the plan shape for XMark Query Q8 [22] (each box denotes an algebraic operator, see 5). The resulting plans

(*i*) are truly set-oriented, *e.g.*, the algebraic plan for `$x mod 2` evaluates the subexpression for *all* bindings of `$x`, in some order the database back-end sees fit,
(*ii*) offer a range of effective optimization hooks (see 4), and
(*iii*) are sufficiently versatile to embrace a family of further iterative languages (6).

4 Relational Insights Into XQuery Affairs

The previous two sections invested considerable effort to "rectangularize" XQuery, press it into some shape that is digestible by a relational back-end. This section explores how we can benefit from a relational formulation of XQuery problems thanks to advanced and well-understood optimization techniques.

A particular example of such well-understood optimization techniques is the early dismissal of irrelevant information from the processing pipeline, also known as *selection* and *projection pushdown*. The latter idea, the disposal of columns not inspected by any upstream operator, has interesting consequences when applied to a loop-lifting compiler.

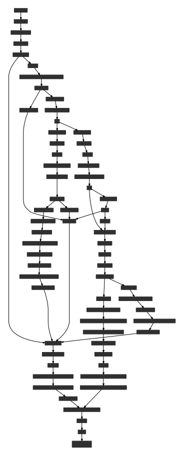


Figure 3: Plan shape model.

And Order is Data, Too. With *Pathfinder*, XQuery's various notions of order are encoded *in the data* (*i.e.*, the surrogates γ_v reflect document order, the columns `iter` and `pos` reflect iteration and sequence order): generated algebraic plans do *not* rely on some prescribed physical row order. Yet, the computation of encoded order information ultimately may still enforce such row order—and therefore incur a significant cost. With order made explicit on the data level, however, we now have a handle to *control* the dependence on ordered processing. By *projecting out* order-encoding columns, operators that were previously needed to ensure physical order will automatically be eliminated by *Pathfinder*'s optimizer.

In [10], we demonstrated how this effects in execution plans that can exploit opportunities to process (sub-) queries in an *unordered* fashion (*e.g.*, in the scope of XQuery's `unordered{}`), an opportunity that proved hard to discover by traditional query analyses on the level of XQuery. For XMark benchmark queries, *e.g.*, this led to a many-fold speed-up even when the dependence on order is not apparent in the source query.

Dependable Cardinality Forecasts for XQuery. Finding the most efficient execution plan for a given query often depends on the availability of accurate *result size estimates*. Though fairly well understood in the context of XPath, the problem of computing such estimates proved notoriously hard to solve for the complete XQuery language. The problem gets tangible once we look at it in the "rectangular" world. Relational equivalents for XQuery expressions provide the necessary fabric to connect existing work on XPath estimation with traditional relational techniques, such as the ones known from System R or different flavors of data statistics (*e.g.*, histograms) [24].

The outcome is a cardinality estimator for *arbitrary* XQuery (sub-)expressions whose accuracy we demonstrated for a wide range of different XQuery workloads [24]. And, most importantly, the estimator shows a high *robustness* with respect to intermediate estimation errors. Rather than piling up such errors during the estimation process, we found it often to recover gracefully and still come up with a meaningful estimate for the overall expression.

5 Off-the-Shelf and Off-Beat XQuery Runtime Environments

Loop-lifting turns the input XQuery expression into an algebraic plan solely operating at the table level. Plans are, generally, DAG-shaped (Figure 3) owing to a common sub-plan analysis stage installed in *Pathfinder*'s compilation pipeline. No traces of variable binding and reference, XPath traversal and node construction, explicit `for` iteration, conditionals (`if` or `typeswitch`), and similar source language features are left. This renders a wide range of set-oriented execution environments suitable runtime environments for XQuery, most of which have not originally been designed to act as XML processors.

Running XQuery on Off-the-Shelf SQL:1999 RDBMSs. The data-embedded order representation (4) makes off-the-shelf RDBMSs perfectly valid compilation targets and execution platforms for *Pathfinder*-generated code. A SQL:1999 code generator is, in fact, among the most advanced code generators available for *Pathfinder* today [8]. The compiler emits a no-frills table algebra dialect in which (groups of) operators have straightforward SQL equivalents. Its row numbering operator ϱ, for example, has its direct correspondence in the SQL:1999 clause `ROW_NUMBER() OVER(PARTITION BY ··· ORDER BY ···)`. *Pathfinder*'s SQL code generator implements a greedy template instantiation strategy—much like programming language compilers—that identifies plan sections whose semantics may be expressed in terms of a single SQL `SELECT-FROM-WHERE` block. The resulting SQL fragments are reasonably "good-natured", *e.g.*, all `UNION` operations are over disjoint tables, nested queries in `FROM` clauses are uncorrelated, and most occurring `JOIN` operations are equi-joins that implement the behavior of nested `for` iteration scopes.

The code generator introduces plan section boundaries,

(*i*) voluntarily, to share runtime evaluation effort if a (sub-)plan's output is input to more than one upstream branch in the plan DAG, or

(*ii*) by necessity, whenever the plan's stacked shape and the occurrence of a row numbering or duplicate removal operation does not allow to further grow the current SQL block.

The generated sequence of SQL code pieces are assembled into a *common table expression* (`WITH ···`) to jointly realize the semantics of the input XQuery expression on an off-the-shelf SQL back-end.

Lab experiments have shown how this approach turns SQL RDBMSs, hosting rectangularized XML instances as described in 2, into capable XQuery processors that do not stumble if document sizes get large [8, 11]. Quite the contrary: for queries against XMark instances beyond 100 MB size, we have seen IBM DB2 V9—running on loop-lifted SQL code—outperform its own built-in native XQuery processor *pureXML*$^{\text{TM}}$ [7].

Off-Beat XQuery Targets. If the underlying database back-end *does* operate over deterministically ordered tables, embedding order in the data appears wasteful: most perceivable implementations of ϱ lead to blocking sort operations in the final physical query execution plans. *Pathfinder*'s code generator for *MonetDB*, CWI Amsterdam's extensible database kernel tuned for in-memory operation [15], exploits explicit control over physical row order [2]. The narrow iter|pos|item tables that are pervasive in loop-lifted plans (3) prove to be a good match for the strictly column-oriented data and query model realized by *MonetDB*. The openness of the *MonetDB* kernel permits the injection of an implementation of staircase join that can particularly benefit from *MonetDB*'s ability to address rows, *i.e.*, encoded XML nodes, by document order rank [13]. *Pathfinder* plus *MonetDB* is distributed as *MonetDB/XQuery* [19]—a purely relational implementation of an XQuery compiler and runtime environment that can process Gigabyte-range XML instances in interactive time [2].

6 Compiling More Iterative Languages

Turning More Semantics into Data? The past few years with *Pathfinder* have taught us that RDBMSs can be turned into efficient processors for "alien" (*i.e.*, non-relational) languages if relevant pieces of the language's semantics are cast into data. To understand XQuery, in particular, we introduced relational representations of XML node identity and document order, XPath axes semantics, type annotations, sequence order, and nested `for` iteration.

```
concat [ ifodd xthen [”odd”, show x]else[”even”]|x < −[5, 6, 7] ]        (Haskell)
[5, 6, 7].collect {|x|x % 2! = 0? [”odd”, x.to_s] : ”even”}.flatten   (Ruby)
```

Figure 4: Haskell and Ruby equivalents of the XQuery FLWOR block of Figure 2a.

This recipe should be applicable to more languages, especially if their core iteration construct may be understood in terms of comprehensions and thus loop lifting. Comprehensions are indeed to be found, under varying coats of syntactic sugar, in a large family of languages. Among these are the programming languages Haskell and Ruby (Figure 4) [9], or Microsoft's LINQ [18]. A rectangularization of the relevant aspects of the language's semantics—*i.e.*, data types like ordered lists and dictionaries, or constructs like conditionals, variable assignment, and reference—plus loop lifting enables a *relational database engine to seamlessly participate in program evaluation*. Programmers continue to use their language's very own syntax, idioms, and functions—the system is in charge to decide *where* the computation described by a given program fragment will take place: on the heap or inside the relational database back-end. Programs that touch and move huge amounts of data, think *Computational Science*, will benefit the most from this support off the relational shelf.

References

[1] Scott Boag, Don Chamberlin, Mary Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. W3C Recommendation, January 2007. `http://www.w3.org/TR/xquery/`.

[2] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. of the 25th ACM SIGMOD Int'l Conference on Management of Data*, Chicago, USA, June 2006.

[3] David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Özsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *Proc. of the 22nd ACM SIGMOD Int'l Conference on Management of Data*, pages 623–634, San Diego, CA, USA, June 2003.

[4] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kris Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3 Consortium, January 2007. `http://www.w3.org/TR/xquery-semantics/`.

[5] Peter Fankhauser, Mary Fernández, Ashok Malhotra, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 Formal Semantics. W3C Working Draft, June 2001. `http://www.w3.org/TR/2001/WD-query-semantics-20010607/`.

[6] Torsten Grust. Purely Relational FLWORs. In *Proc. of the 2nd ACM SIGMOD Int'l Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, Baltimore, MD, USA, June 2005.

[7] Torsten Grust, Manuel Mayr, and Jan Rittinger. XQuery Join Graph Isolation. In *Proc. of the 25th Int'l Conference on Data Engineering (ICDE)*, Shanghai, China, March 2009.

[8] Torsten Grust, Manuel Mayr, Jan Rittinger, Sherif Sakr, and Jens Teubner. A SQL:1999 Code Generator for the Pathfinder XQuery Compiler. In *Proc. of the 26th ACM SIGMOD Int'l Conference on Management of Data*, Beijing, China, June 2007.

[9] Torsten Grust and Jan Rittinger. Jump Through Hoops to Grok the Loops. In *Proc. of the 5th Int'l ACM SIGMOD Workshop on XQuery Implementation, Experience, and Perspectives (XIME-P)*, Vancouver, Canada, June 2008.

[10] Torsten Grust, Jan Rittinger, and Jens Teubner. eXrQuy: Order Indifference in XQuery. In *Proc. of the 23rd Int'l Conference on Data Engineering (ICDE)*, Istanbul, Turkey, April 2007.

[11] Torsten Grust, Jan Rittinger, and Jens Teubner. Why Off-The-Shelf RDBMSs are Better at XPath Than You Might Expect. In *Proc. of the 26th ACM SIGMOD Int'l Conference on Management of Data*, Beijing, China, June 2007.

[12] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*, Toronto, Canada, September 2004.

[13] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. of the 29th Int'l Conference on Very Large Databases (VLDB)*, pages 524–535, Berlin, Germany, September 2003.

[14] Torsten Grust, Maurice van Keulen, and Jens Teubner. Accelerating XPath Evaluation in Any RDBMS. *ACM Transactions on Database Systems (TODS)*, 29(1):91–131, March 2004.

[15] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access. *The VLDB Journal*, 9(3):231–246, December 2000.

[16] Sabine Mayer, Torsten Grust, Maurice van Keulen, and Jens Teubner. An Injection with Tree Awareness: Adding Staircase Join to PostgreSQL. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*, pages 1305–1308, Toronto, Canada, September 2004.

[17] Jason McHugh and Jennifer Widom. Query Optimization for XML. In *Proc. of the 25th Int'l Conference on Very Large Databases (VLDB)*, pages 315–326, Edinburgh, Scotland, UK, September 1999.

[18] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Objects, Relations, and XML in the .NET Framework. In *Proc. of the 25th ACM SIGMOD Int'l Conference on Management of Data*, Chicago, USA, June 2006.

[19] MonetDB/XQuery. `http://www.monetdb-xquery.org/`.

[20] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking Forward. In *XML-Based Data Management and Multimedia Engineering, EDBT 2002 Workshops, Revised Papers*, pages 109–127, Prague, Czech Republic, March 2002.

[21] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. of the 23rd ACM SIGMOD Int'l Conference on Management of Data*, pages 903–908, Paris, France, June 2004.

[22] Albrecht Schmidt, Florian Waas, Martin Kersten, Mike Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *Proc. of the 28th Int'l Conference on Very Large Databases (VLDB)*, Hong Kong, China, 2002.

[23] Jens Teubner. Scalable XQuery Type Matching. In *Proc. 11th Int'l Conference on Extending Database Technology (EDBT)*, Nantes, France, April 2008.

[24] Jens Teubner, Torsten Grust, Sebastian Maneth, and Sherif Sakr. Dependable Cardinality Forecasts for XQuery. In *Proc. of the 34th Int'l Conference on Very Large Databases (VLDB)*, August 2008.

[25] The Pathfinder XQuery Compiler. `http://www.pathfinder-xquery.org/`.

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