Pathfinder: XQuery—The Relational Way

Peter Boncz[1] Torsten Grust[2] Maurice van Keulen[3]
Stefan Manegold[1] Jan Rittinger[1,4,†] Jens Teubner[2]

[1]CWI Amsterdam, The Netherlands
{boncz,manegold}@cwi.nl
[2]Technische Universität München, Germany
{torsten.grust,jens.teubner}@in.tum.de
[3]University of Twente, The Netherlands
m.vankeulen@utwente.nl
[4]University of Konstanz, Germany
rittinge@inf.uni-konstanz.de

1 Introduction

Relational query processors are probably the best understood (as well as the best engineered) query engines available today. Although carefully tuned to process instances of the relational model (tables of tuples), these processors can also provide a foundation for the evaluation of "alien" (non-relational) query languages: if a relational encoding of the alien data model and its associated query language is given, the RDBMS may *act like* a special-purpose processor for the new language.

This demonstration features our XQuery compiler *Pathfinder*, the continuation of our earlier work on a purely relational XPath and XQuery processing stack [4, 6, 7] in which we developed relational encodings and processing strategies for the tree-shaped XML data model. The Pathfinder project is an exploration of how far we can push the idea of using mature RDBMS technology to design and build a full-fledged XQuery implementation. The demonstration will show that this line of research was and still is worth to be followed: based on the extensible relational database kernel *MonetDB* [2], Pathfinder provides highly efficient and scalable XQuery technology that scales beyond 10 GB XML input instances on commodity hardware.

Pathfinder requires only local extensions to the underlying DBMS's kernel, such as the *staircase join* operator ⛦ [7, 9]. A *join recognition* logic in our compiler, as well as a careful consideration of *order properties* of relational operators [3], allow for effective optimizations that turn MonetDB into a highly efficient XQuery engine.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

[†]This work was supported by the DFG Research Training Group GK-1042 "Explorative Analysis and Visualization of large Information Spaces".

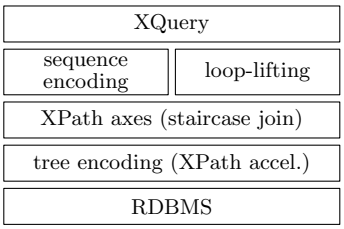

Figure 1: XQuery development stack.

We will briefly recite these ideas in the upcoming section. Section 3 will provide experimental evidence of the viability of our approach, before we sketch the setup for this demonstration of Pathfinder.

2 Pathfinder XQuery Engine Overview

To bridge the apparent gap between the relational processing model and the XQuery data model, Pathfinder follows the step-by-step development stack of Figure 1.

Tree encoding. Relational XML processing starts off with a suitable relational encoding for XML document trees. We based our system on the XPath Accelerator [4], a numbering scheme in which each node v is assigned a pre-order rank $pre(v)$ and a post-order rank $post(v)$. These numbers efficiently characterize XPath location steps as regions in a two-dimensional plane, *e.g.*, v' is a descendant of $v \Leftrightarrow pre(v) < pre(v') \wedge post(v') < post(v)$. Pathfinder employs an equivalent encoding variant that represents the location of a node as the 3-tuple $\langle pre(v), size(v), level(v)\rangle$ with the number of nodes in the subtree below v, $size(v)$, and v's distance from the tree root, $level(v)$. These tuples are maintained in a table with schema pre|size|level.

XPath axes. This encoding scheme, turns XPath step evaluation into a relational range selection. The selection predicate describes region queries on the $\langle pre(v), size(v), level(v)\rangle$ space, where the region queried associates with the XPath axis [4]. These range selections are well supported by existing access and index structures.

Yet, the RDBMS gives away significant opportunities for optimization since the system is unaware of the

π	column projection, renaming
σ	row selection
$\dot\cup$,$\setminus$	disjoint union, difference
δ	duplicate elimination
$\bowtie$, $\times$	equi-join, Cartesian product
ϱ	row-numbering
$\triangle$	staircase join
ε, τ	element/text node construction
$\circledast$	arithmetic/comparison operator $*$

Table 1: The relational algebra used by Pathfinder.

atomic literals	document order (e_1 `<<` e_2)
sequences (e_1 `,` e_2)	node identity (e_1 `is` e_2)
variables (`$`v)	arithmetics (`+`, `-`, ...)
`let $`v `:=` e_1 `return` e_2	comparisons (`eq`, `lt`, ...)
`for $`v `in` e_1 `return` e_2	Boolean operators (`and`, `or`, ...)
`if` e_1 `then` e_2 `else` e_3	`fn:doc(`e`)`
`typeswitch` clauses	`fn:root(`e`)`
`element {` e_1 `} {` e_2 `}`	`fn:data(`e`)`
`text {` e `}`	`fs:distinct-doc-order(`e`)`
e_1 `order by` e_2`,...,`e_n	`fn:count(`e`)`, `fn:sum(`e`)`
XPath ($e/\alpha::\nu$)	`fn:empty(`e`)`
user defined functions	`fn:position()`, `fn:last()`

Table 2: XQuery dialect supported by Pathfinder. Expressions may be composed arbitrarily; α denotes an XPath axis, ν a node test.

isomorphism between the tree-shaped XML data and its relational encoding. Injecting such *tree-awareness*, however, is possible in terms of a special join operator, *staircase join* $\triangle$, that helps to turn RDBMSs into highly efficient XPath processors [7].

Relational XQuery evaluation. Pathfinder compiles XQuery expressions into a purely *relational* query plan. This target algebra solely uses standard operators, available in off-the-shelf RDBMSs (Table 1). Note that the extra operators $\triangle$, ε, and τ are just short-hands for efficient implementations of equivalent algebraic expressions. A row-numbering operator ϱ is provided by many existing RDBMSs, *e.g.*, in terms of MonetDB's `mark` operator, or the `DENSE_RANK()` function in SQL:1999. The algebra still suffices to express the XQuery dialect listed in Table 2, a subset that, *e.g.*, suffices to express all XMark queries [10]. Figure 5 in Section 3 depicts the query plan of a typical XQuery `FLWOR` clause.

Our query plans use a very explicit, "assembly-style" algebra. The exploitation of specific restrictions which hold for this algebra (*e.g.*, all joins are equi-joins, π does not imply duplicate elimination, all unions are disjoint) enables additional, quite far-reaching optimizations. All in all, this makes this particular variant of relational algebra very efficiently implementable on any relational DBMS. Query plans can become quite large (XMark query Q8 [10], *e.g.*, prior to optimization, compiles to a plan DAG of 120 operators). This complexity may significantly be reduced by peep-hole style optimization [5], though.

iter	pos	item
1	1	10
1	2	20

(a) `(10,20)` in s_0.

iter	pos	item
1	1	10
2	1	20

(b) `$v` in scope s_1.

iter	pos	item
1	1	10
2	1	10
3	1	20
4	1	20

(c) `$v` in scope s_2.

iter	pos	item
1	1	100
2	1	200
3	1	100
4	1	200

(d) `$w` in scope s_2.

iter	pos	item
1	1	110
2	1	210
3	1	120
4	1	220

(e) `$v + $w` in s_2.

inner	outer
1	1
2	1
3	2
4	2

(f) $\mathsf{map}_{(s_1,s_2)}$.

iter	pos	item
1	1	110
1	2	210
1	3	120
1	4	220

(g) Result (scope s_0).

Figure 3: Intermediate results in evaluation of `for $v in (10,20), $w in (100,200) return $v + $w`.

Sequence encoding. The XQuery data model is based on the *sequence*, an ordered (possibly heterogeneous) list of *items*. We implement sequence order by means of an explicit `pos` column, as depicted on the right. The polymorphic `item` column is efficiently implemented with help of MonetDB's `mposjoin` operator.

pos	item
1	5
2	"x"
3	<a/>
4	"x"

Fig. 2: Sequence encoding of `(5, "x",<a/>,"x")`.

Loop lifting. A crucial part in XQuery evaluation is the efficient implementation of its iteration primitive, the `FLWOR` clause. In [6], we developed a database style, bulk-oriented way of loop processing. We sketch the basic idea here.

The compilation from the iterative `FLWOR` clauses to bulk-oriented table manipulation is achieved through *loop-lifting*: a column `iter` added to the encoding in Figure 2 accounts for the different bindings a variable assumes when it iterates over a sequence. To illustrate, Figure 3 shows some intermediate results for the relational query plan that evaluates

$$s_0 \begin{cases} \texttt{for \$v in (10,20)} \\ s_1 \begin{cases} \texttt{for \$w in (100,200)} \\ s_2 \{ \texttt{return \$v + \$w} \end{cases} \end{cases} .$$

The loop-lifting concept makes any relational sequence encoding depend on its associated iteration scope s_i. The initial expression `(10,20)` is evaluated in the top-level scope s_0, with a constant `iter` value of 1 (Figure 3(a)). The `for` loop introduces new `iter` values for each tuple to represent `$v` in s_1 (Figure 3(b)): for example, in the second iteration in scope s_1, variable `$v` will be bound to the single item 20. The `return` expression in scope s_2 is to be evaluated four times, represented by four different `iter` values for expressions `$v`, `$w`, and `$v + $w` (Figures 3(c)–(e)).

The semantics of nested iteration implies that during the first two iterations of scope s_2, scope s_1 is still in its first iteration, while for iterations 3 and 4 of scope s_2, s_1 is in its second iteration. We capture this

semantics by a relation map (Figure 3(f)) that relates iter values between scopes s_1 and s_2.

Back-mapping the encoding in Figure 3(e) to the top-level scope finally forms the overall expression result in Figure 3(g). Observe that our bulk-oriented way of evaluation is quite different from other XQuery engines, which in a sense only do nested loop, *i.e.*, recursive, processing. For details on the loop-lifting concept, refer to [6].

MonetDB. We designed Pathfinder as an XQuery *compiler*: XQuery expressions are translated into relational query plans. Their execution on a database back-end evaluates the input expression by means of a single algebraic query. A simple post-processor then serializes the relational result to form a response in terms of the XQuery data model.

The compiler currently targets MonetDB, an extensible relational database system, available in open source [2]. Its main-memory optimized implementation and features like virtual object identifiers (that, *e.g.*, make the row-numbering operator ϱ a no-cost operator) make MonetDB particularly suitable for our objective: scalable XQuery processing. The use of alternative back-ends (*e.g.*, SQL) is current work in progress.

3 Scalability and Performance

Our implementation is meant to assess the viability of using relational back-ends for XQuery. RDBMSs are known to scale well with increasing data volumes—an inevitable feature if XQuery systems are to support large amounts of XML data.

We used the XML generator XMLgen from the XMark benchmark set [10] to create XML instances of sizes from 11 MB to 11 GB (50,000 to 500 million nodes, respectively). Documents were loaded into our MonetDB back-end to run the 20 XMark queries.

3.1 Storage Overhead

Pathfinder stores the structural part of XML documents in the pre|size|level encoding, as sketched in Section 2. A prop column stores surrogates of XML node properties. Actual property values (tag names, text node content, etc.) are maintained in separate property BATs and kept unique therein. These node properties are identified by their surrogates, where nodes with identical properties share the same surrogate.

This surrogate sharing not only avoids expensive string comparisons at query execution time, but also reduces space consumption on secondary storage. For the XMark benchmark set, disk space requirements range between 147 % (11 MB instance) and 125 % (110 MB instance) of the original XML document.[1]

[1]For larger instances, XMark documents tend to degrade: more and more duplicate text nodes bring down disk space usage below 80 % of the original XML instance.

	11 MB		110 MB		1.1 GB		11 GB
Q	X-Hive	PF	X-Hive	PF	X-Hive	PF	PF
1	0.37	0.05	1.29	0.17	9.9	1.2	13
2	0.45	0.07	1.75	0.30	33.0	2.4	25
3	0.65	0.28	5.66	1.51	25.1	12.5	126
4	0.10	0.08	1.00	0.45	18.1	3.8	36
5	0.13	0.05	0.90	0.16	20.7	1.2	11
6	1.07	0.02	10.17	0.05	178.1	0.3	3
7	1.57	0.03	24.84	0.07	278.4	0.4	4
8	0.85	0.14	3.51	0.75	49.1	10.4	208
9	32.25	0.20	12280.66	0.87	DNF	12.9	289
10	5.28	0.80	442.37	5.31	DNF	55.0	1882
11	98.91	0.18	19927.29	3.48	DNF	960.9	DNF
12	23.39	0.14	5100.19	1.66	DNF	431.3	DNF
13	0.10	0.07	1.03	0.22	12.9	1.3	13
14	0.72	0.26	11.16	2.20	110.2	21.3	6463
15	0.03	0.09	0.49	0.28	10.6	1.7	16
16	0.03	0.11	0.52	0.26	10.9	1.8	18
17	0.09	0.07	0.85	0.30	11.8	2.8	26
18	0.08	0.04	0.64	0.13	14.8	0.9	9
19	0.67	0.11	12.15	0.55	254.5	5.3	88
20	0.11	0.24	1.40	0.62	24.6	4.9	50

Table 3: Query evaluation times (in seconds) for different XMark instance sizes (scale factors 0.1 to 100).

3.2 Experimental Setup

We tested Pathfinder's performance on a 1.6 GHz AMD Opteron system, equipped with 8 GB RAM. For comparison, we installed X-Hive/DB on the same system [8]. X-Hive appears to be one of the faster XML database systems and copes with large XML instances quite well. We tuned the X-Hive database with value indices on the `buyer/@person` and `profile/@income` paths. These indices significantly reduced execution times of some computationally intensive XMark join queries (*e.g.*, Q8).

3.3 Pathfinder Performance

The execution times, shown in Table 3, confirm the strengths of our purely relational approach to XQuery evaluation. Pathfinder handles simple path queries (Q1–Q5, Q13–Q20) 2 to 20 times faster than X-Hive. If paths include recursive axes, we benefit from our staircase join implementation that outperforms X-Hive by two orders of magnitude (Q6 and Q7).

It is not surprising that the join queries (Q8–Q12) benefit most from our relational back-end. While X-Hive almost competes with Pathfinder for the simple join query Q8, its performance strongly degrades if *intermediate* query results are to be joined. Pathfinder compiles these queries into join plans [3] and takes advantage of efficient join implementations in our back-end. It is thus able to execute queries in reasonable time which we were not able to run on our X-Hive installation (Q9–Q12).

3.4 Scalability

To assess the scalability of our approach, we printed XMark execution times normalized to the elapsed times for the 110 MB XML instance. As shown in Figure 4, Pathfinder scales linearly for most of the queries. Even the join queries (Q8–Q10) are of almost linear complexity.

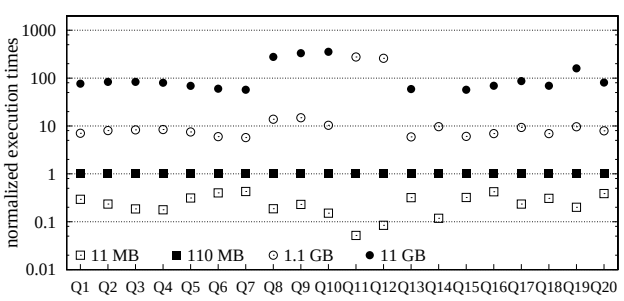

Figure 4: Pathfinder scalability: Execution times for the XMark benchmark set, normalized to timings for the 110 MB instance.

The only two exceptions are queries Q11/Q12. Their quadratic scaling stems from a theta-join (predicate >) that produces intermediate results of about 120 k, 12 M, and 1.2 G tuples for the 11 MB, 110 MB, and 1.1 GB instances, respectively. Note that this concerns the query *result*, whose computation cannot be avoided. Thus, *any* XQuery implementation will face this complexity.

4 Demonstration Setup

Our system operates in a front-end/back-end fashion. The front-end, the Pathfinder compiler, parses incoming XQuery expressions and translates them into a relational algebra expression tree, represented in terms of a MIL[2] program. The code is shipped to a MonetDB server that then executes the entire query and serializes the encoding of the result.

The demonstration features both of these components, with various interesting hooks to trace the execution of XQuery expressions on our relational backend. Our system implements the W3C October 2004 working drafts [1], including schema import, static typing, and full axis features.

The compiler provides facilities to look "under the hood" of relational XQuery compilation. An output of type-annotated XQuery Core expression equivalents demonstrates optimization techniques applied to the incoming XQuery code before compiling it into relational algebra. A graphical output of relational query plans at different compilation stages (much like the plan in Figure 5) illustrates the loop-lifting concept. Relational plans may be traced to reveal the result computed for any subexpression.

Our demonstration system will be pre-loaded with XMark [10] instances of different sizes up to 1.1 GB. The query texts for the 20 benchmark queries will be ready to run, but users may as well state their own ad hoc queries.

The Pathfinder compiler is available in open source as part of the MonetDB/XQuery implementation via `http://pathfinder-xquery.org/`.

[2]MIL stands for MonetDB Interpreter Language

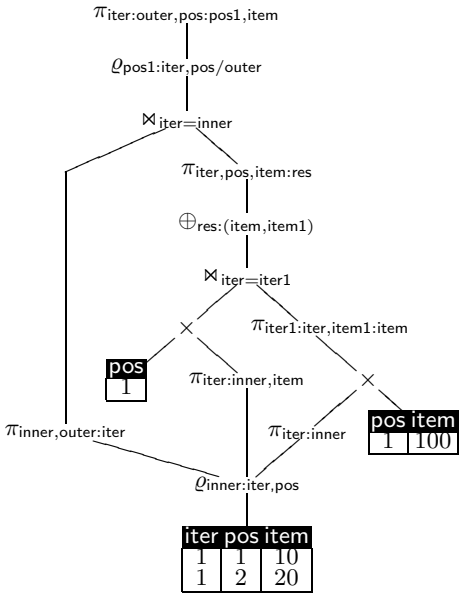

Figure 5: Relational query plan that evaluates the query `for $v in (10,20) return $v + 100`.

References

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

[2] P. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, University of Amsterdam, The Netherlands, May 2002.

[3] P. Boncz, T. Grust, S. Manegold, J. Rittinger, and J. Teubner. Pathfinder: Relational XQuery Over Multi-Gigabyte XML Inputs In Interactive Time. Technical Report INS-E0503, CWI, Amsterdam, The Netherlands, March 2005.

[4] T. Grust. Accelerating XPath Location Steps. In *Proc. of the 21st Int'l ACM SIGMOD Conf. on Management of Data*, pages 109–120, Madison, Wisconsin, USA, June 2002.

[5] T. Grust. Purely Relational FLWORs. In *Proc. of the ACM SIGMOD/PODS 2nd Int'l Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, Baltimore, MD, USA, June 2005.

[6] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. of the 30th Int'l Conference on Very Large Data Bases*, Toronto, Canada, August 2004.

[7] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its Axis Steps. In *Proc. of the 29th Int'l Conference on Very Large Databases*, Berlin, Germany, September 2003.

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

[9] S. Mayer, T. Grust, M. van Keulen, and J. Teubner. An Injection of Tree Awareness: Adding Staircase Join to PostgreSQL. In *Proc. of the 30th Int'l Conference on Very Large Data Bases*, pages 1305–1308, Toronto, Canada, August 2004.

[10] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. of the 28th Int'l Conference on Very Large Databases*, pages 974–985, Hong Kong, China, August 2002.