Pathfinder: XQuery Compilation Techniques for Relational Database Targets

Jens Teubner

Technische Universität München, Institut für Informatik
jens.teubner@in.tum.de

Abstract: Relational database systems are highly efficient hosts to table-shaped data. It is all the more interesting to see how a careful inspection of both, the XML tree structure as well as the W3C XQuery language definition, can turn relational databases into fast and scalable XML processors.

This work shows how the deliberate choice of a relational *tree encoding* makes the XML data model—ordered, unranked trees—accessible to relational database systems. Efficient XPath-based access to these data is enabled in terms of *staircase join*, a join operator that injects full tree awareness into the relational database kernel. A *loop-lifting* compiler translates XQuery expressions into purely algebraic query plans. The representation of iteration (*i.e.*, the XQuery FLWOR construct) in terms of set-oriented algebra primitives forms the core of this compiler. Together, the techniques we describe lead to unprecedented XQuery evaluation scalability in the multi-gigabyte XML range. *Pathfinder* is an open-source implementation of a purely relational XQuery processor.

1 Introduction

The ubiquitous use of the XML file format to store, interchange, and process data raises an increasing demand to manage these data in a scalable manner. Not only since the initiative of the W3C to develop XQuery as a standard query language for XML, researchers around the globe ambitiously started to develop novel database techniques that can efficiently handle semi-structured data. The approaches pursued have been as diverse as the XML data themselves: new storage layouts can natively handle tree-structured data (*e.g.*, [FHK+02, NvdL05]), new algorithms and index structures accelerate XPath navigation primitives (*e.g.*, [BKS02, CSF+01]), and tree algebras reflect the intricate semantics of the XQuery language (*e.g.*, [JLST01]).

In this work, we want to assess how far we can get *without* the construction of such new and complex software systems. The processing model of existing relational databases—bulk operations on sets of tuples—proves versatile enough to embrace the semantics of XQuery in a standards-compliant fashion. At the same time, the maturity of existing implementations provides unprecedented scalability with interactive query response times on multi-gigabyte XML instances.

To meet these scalability goals, we contribute the purely relational XQuery processing stack shown in Figure 1 which can turn any RDBMS implementation into a processor for XQuery.

A relational *tree encoding*, derived from the XPath accelerator encoding by Grust [Gru02], provides a true isomorphism between instances of the XML data model, *ordered, unranked trees*, and relational *tables of tuples*. If B-tree indexes on such tables are chosen deliberately, interactive query response times for XML data can be observed even on commodity RDBMS implementations.

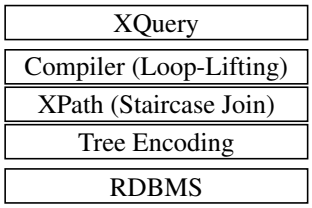


Figure 1: XQuery processing stack.

The XPath performance of such a system can further be improved if the underlying DBMS kernel is made aware of properties inherent to the used tree encoding. *Staircase join* encapsulates such knowledge in a single database operator and can accelerate tree navigation by orders of magnitude.

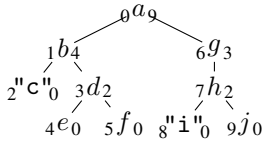
Finally, we extend the processing stack to full XQuery compliance. The *loop-lifting compilation procedure* trades XQuery's `for` iteration primitive for truly bulk-oriented operations in the relational system. By shifting the dynamic evaluation into the DBMS kernel, we make the scalability advantages of modern RDBMS implementations immediately accessible to process XQuery.

The *Pathfinder* XQuery compiler[1] is a complete implementation of the techniques we describe here. Pathfinder is part of the *MonetDB/XQuery* system, which is found among the fastest XQuery processors in existence today.

Sections 2 to 4 in the following will sketch the components of the relational XQuery processing stack. We provide performance figures obtained with MonetDB/XQuery in Section 5, before we summarize in Section 6.

2 Relational Storage of XML Data

To losslessly store XML data in a relational system, we use *range encoding*, a variant of the schema-oblivious tree encoding proposed by Grust [Gru02]. We enumerate all tree nodes according to the XML document order to obtain the preorder rank $\mathsf{pre}(v)$ for each node v. Further, we maintain $\mathsf{size}(v)$ as the number of v's descendants and $\mathsf{level}(v)$, v's distance from the document root. Two properties $\mathsf{kind}(v) \in \{\mathsf{elem}, \mathsf{text}, \mathsf{comment}, \dots\}$ and $\mathsf{prop}(v)$ (holding v's tag name or textual content for text/comment nodes) account for the semantical information of each node. Figure 2 on the left illustrates this encoding for a small sample tree.



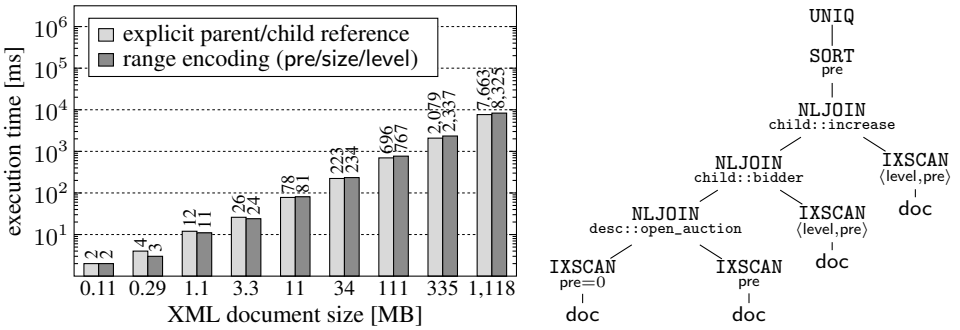
pre	size	level	kind	prop
0	9	0	elem	a
1	4	1	elem	b
2	0	2	text	c
3	2	2	elem	d
5	0	3	elem	f
⋮	⋮	⋮	⋮	⋮

Figure 2: Sample tree (with pre and size annotations) and its relational encoding.

On range-encoded data, XPath location steps translate into simple region predicates. To exemplify, the XPath `descendant` axis becomes a range condition on preorder ranks:

$$v \in c/\texttt{descendant} \Leftrightarrow \mathsf{pre}(c) < \mathsf{pre}(v) \leq \mathsf{pre}(c) + \mathsf{size}(c) \quad . \qquad \text{(DESC)}$$

[1] Pathfinder is available in open source at `http://www.pathfinder-xquery.org/`.



(a) Execution times.

(b) DB2 query plan (using partitioned B-trees).

Figure 4: Partitioned B-trees provide efficient `child` navigation performance on range-encoded XML trees without the expensive maintenance of parent/child references (IBM DB2 v8.2 on a 2×3.2 GHz Intel Xeon system with 8 GB RAM; path `//open_auction/bidder/increase`).

The evaluation of such one-dimensional range predicates is well supported by existing (*e.g.*, B-tree) index structures.

2.1 Off-the-shelf RDBMSs are Better at XPath than You Might Expect

Numbering schemes of this kind are known to provide very efficient support for axes with a recursive definition in XPath [Gru02]. At first sight, this does not hold for the important non-recursive axes `child` and `parent`, which require an additional predicate on column level to characterize their semantics, *e.g.*:

$$\begin{gathered} v \in c/\texttt{child} \Leftrightarrow \\ \mathsf{pre}(c) < \mathsf{pre}(v) \leq \mathsf{pre}(c) + \mathsf{size}(c) \ \wedge \ \mathsf{level}(v) = \mathsf{level}(c) + 1 \ . \end{gathered} \qquad \text{(CHILD)}$$

Earlier work [Gru02] had thus used explicit parent/child references to provide acceptable runtime behavior for non-recursive XPath axes. By using *partitioned B-trees* [Gra03] to index the relational XML storage, however, we can reach a similar performance without the additional storage overhead. The prepending of the level column to a B-tree on pre (to obtain a concatenated $\langle \mathsf{level}, \mathsf{pre} \rangle$ B-tree) partitions the resulting B-tree into $\mathsf{height}(t)$ regions as shown on the right in Figure 3 (where $\mathsf{height}(t)$ denotes the total height of the XML document tree).

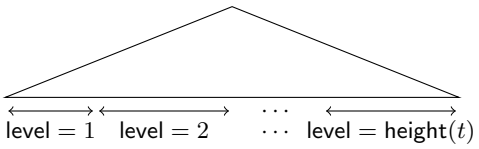
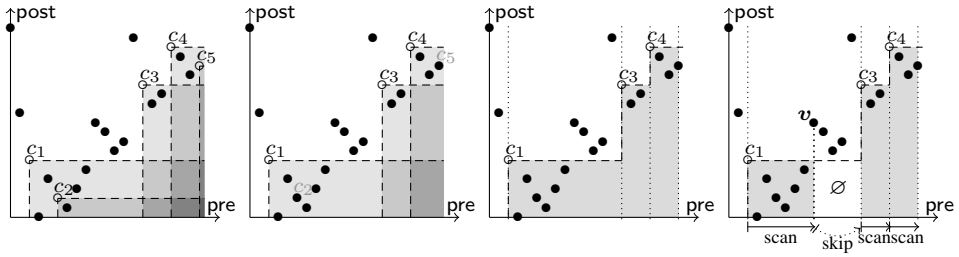


Figure 3: B-tree partitioning.

On such a partitioned B-tree, all children of a given context node appear within a single index partition and in ascending pre-order (*i.e.*, document order). As we see in Figure 4, this leads to efficient `child` evaluation on range-encoded data without the storage overhead of explicit parent/child references. Similar uses of partitioned B-trees are found to accelerate other XPath idioms as well [Teu06].



(a) Original context set. (b) Pruning. (c) Partitioning. (d) Skipping.

Figure 5: Three techniques minimize XPath processing cost in staircase join: pruning, partitioning, and skipping.

3 XPath Evaluation on Relational Back-Ends

Further improvements of an RDBMS's XPath performance can be reached if we make the system aware of the fact that underlying relational tables actually constitute the encoding of a tree. *Staircase join* encapsulates such knowledge in a single database operator that may easily be plugged into existing RDBMS implementations. Tuned for the evaluation of XPath, staircase join largely avoids to spend work on irrelevant tuples, which brings execution times down to a minimum.

Figure 5 uses the two-dimensional pre/post plane[2] and the XPath `descendant` axis to illustrate the three techniques that make staircase join an efficient means to answer XPath queries:

Pruning. Since XPath demands the result of location paths to be returned without duplicates, some context nodes may not contribute any new matches to the result set. In the pre/post plane, this surfaces as an overlap of their corresponding query regions. Staircase joins *prunes* such nodes early from the context set. This may significantly reduce the cost to eliminate duplicates from a path result.

Partitioning. After pruning the context set, the resulting query region takes the shape of a staircase in the pre/post plane. Staircase join divides this region into a distinct *partition* for each remaining context node. Each partition is scanned only once and in pre order. Regardless of the context set, the size of the document relation is now an upper bound for the number of tuples that need to be processed. The production of result tuples in document order obsoletes a subsequent sort operation as it was required in the original query plan (cf. Figure 4(b)).

Skipping. Since the pre/post plane actually constitutes the encoding of a tree, we can conclude that some regions in the plane cannot contain any nodes [GvKT03]. Staircase join *skips* over such regions, which further reduces the number of tuples to be processed from the document relation. The effect of skipping can be substantial: in earlier work [GvKT03], we found staircase join to skip over more than 90 % of all tuples.

[2]Note that the range- and pre/post-encodings are isomorph. Concepts equally apply to range-encoded data. In the pre/post plane, all descendants of a node v are to be found in the quadrant on v's bottom-right.

π	column projection, renaming	ϱ	row numbering
σ	row selection	$\dot\cup$	disjoint union (`UNION ALL`)
$\bowtie$	equi-join	$\circledcirc$	arithmetic/comparison operator $\circ$
$\times$	Cartesian product	$\scriptstyle\sqsubset\!\!\cdot$	XPath step join

Table 1: Subset of the relational algebra emitted by the loop-lifting compiler. Operator ϱ is the equivalent of SQL:1999's `ROW_NUMBER` operator.

Staircase join encapsulates full tree awareness within a single join operator. This operator easily plugs into any existing relational database kernel. We have shown staircase join's effectiveness with implementations for the MonetDB and PostgreSQL systems, for which we refer the reader to [Teu06].

4 Loop-Lifting: From XPath to XQuery

We have now seen how one of XQuery's core data structures, *ordered, unranked trees*, can suitably be mapped to relational database tables. The second principal data type in the XQuery data model, *ordered sequences* of *items*, however, seems quite contradictory to the processing model of relational systems, *unordered sets* of *tuples*. Existing systems thus often tend to escape to a programming language outside the database kernel to implement language features that are sensible to this difference.

The *loop-lifting* compilation technique, in contrast, carries these tasks into the database kernel and leverages any RDBMS implementation to full XQuery support. Our approach remains purely relational: the compiler emits plans of a standard relational algebra (see Table 1 for an excerpt) whose operators are efficiently implementable on, *e.g.*, SQL hosts. Note that this algebra operates on first normal form relations only. No XQuery-specific extensions (such as, *e.g.*, the Map operators in [RSF06]) are required to back our compiler.

4.1 A Relational Representation for XQuery Sequences

The loop-lifting compiler represents any XQuery item sequence in terms of a relational table. The table shown on the right shows the relational sequence encoding of the XQuery sequence `("a","b","c","d")`. In this table, sequence order is maintained using column pos, while the actual sequence items are stored in column item. In line with the XQuery data model, we assume that column item can host *atomic values* as well as references to XML *nodes* (*e.g.*, in terms of their preorder ranks pre(v)) in a heterogeneous fashion. See [Teu06] for ways to implement such a column.

pos	item
1	"a"
2	"b"
3	"c"
4	"d"

4.2 Turning Iteration Into Joins

The heart of the loop-lifting compiler is the standards-compliant translation of XQuery's *iteration* primitive, the `for-return` construct. This construct successively binds a variable $\$v$ to the items listed in its `in` part. The `return` body e is then evaluated for each binding

and all sub-results are assembled to form the overall expression result:

$$\texttt{for } \$v \texttt{ in } (x_1, x_2, \ldots, x_n) \texttt{ return } e \;\equiv\; (e[x_1/\$v], e[x_2/\$v], \ldots, e[x_n/\$v]) \;.$$

The semantics of this construct remains purely functional: it is sound to evaluate e for all bindings of $\$v$ in parallel. The iter|pos|item relation shown here for the variable $\$v$ reflects this situation and encodes all bindings of $\$v$ in a single relation. This *loop-lifted sequence representation* is pervasive in our approach. Each tuple $\langle i, p, x\rangle$ in it indicates that, in the ith iteration, the item at position p has the value x (note that $\$v$ is a singleton in the above expression, hence, pos $\equiv 1$).

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

We can easily derive this representation of the binding variable from the representation of the expression it is bound to: *(i)* attach a new iter column, consecutively numbered from $1, \ldots, n$ in the order given by the pos column, and then *(ii)* set the pos column to constant 1.

The *row-numbering* step *(i)* is characteristic for this approach and we assume the availability of a respective operator $\varrho_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle\|\mathsf{c}}$ to implement it. For each group identified by column c, operation $\varrho_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle\|\mathsf{c}}(R)$ extends R by a new column a that contains consecutive numbers in the order specified by $\langle \mathsf{b}_1, \ldots, \mathsf{b}_n\rangle$. Many RDBMSs readily provide an implementation for ϱ. The construct `ROW_NUMBER() OVER (PARTITION BY c ORDER BY` b_1`, ...,` b_n`)`, *e.g.*, implements $\varrho_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle\|\mathsf{c}}$ in SQL:1999 [GST04].

4.3 Independent Iterations

iter
1
2
⋮
n

Note how column iter in the loop-lifted sequence representation enumerates the iterations performed by the `for` loop. It is a principle idea of our compilation approach that each subexpression is compiled in dependence of all enclosing `for` loops. To encode the latter, we use a unary loop relation, a projection of the loop-lifted encoding of the iteration variable on column iter. The table on the left depicts the loop relation that encodes the n-fold iteration over the loop body e in the above example.

Once loop has been determined, we can use it to obtain the loop-lifted encoding of a constant subexpression by means of a Cartesian product. We say that the expression is *lifted* with respect to loop. To illustrate, the table on the right encodes the sequence `("a", "b")` in the loop

```
for $v in (10, 20) return ("a", "b") .
```

iter
1
2

loop

$\times$

pos	item
1	"a"
2	"b"

$\equiv$

iter	pos	item
1	1	"a"
1	2	"b"
2	1	"a"
2	2	"b"

To ensure compositionality, the full compilation procedure operates on loop-lifted sequence representations only. The compiler is defined in terms of a set of compilation rules, such that the algebraic expressions consumed and produced by each rule evaluate to the loop-lifted encodings of their respective XQuery equivalents, each one associated with a loop relation.

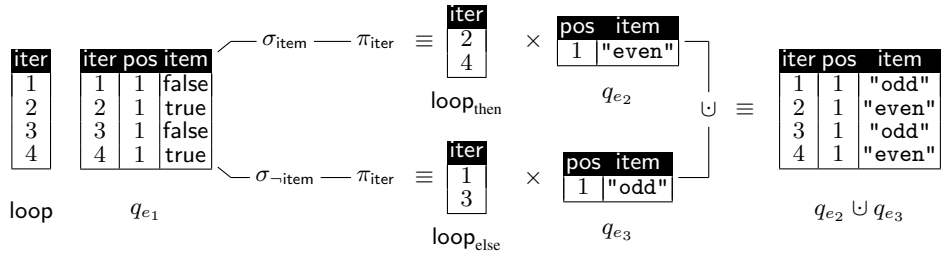


Figure 6: Evaluation trace for the loop-lifted equivalent of Query Q_1.

4.4 Compiling Arbitrary XQuery Expressions

The complete procedure to compile arbitrary XQuery expressions into their relational equivalent is beyond the scope of this paper (refer to [Teu06] for an extensive documentation). To provide an intuition of the typical plans emitted by the compiler, let us briefly review the compilation and evaluation of the XQuery expression

```
for $v in (3, 4, 5, 6) return
  if ( $v mod 2 eq 0 ) then "even" else "odd" .        (Q1)
```

The loop relation associated with the `return` body of this query is the relation shown on the left in Figure 6. This relation is used to compile the predicate subexpression e_1. We omit the details of this compilation and show its outcome as the relation q_{e_1} in Figure 6. It contains the loop-lifted representation of a single Boolean value for each of the four iterations (in the third iteration, *e.g.*, e_1 evaluates to false).

Depending on the outcome of the predicate, we need to either evaluate the `then` branch e_2 or the `else` branch e_3. Two independent selections compute the respective sets of iter values (σ_{item} selects all tuples with value true in column item, $\sigma_{\neg\text{item}}$ selects the complement) which are used to loop-lift the respective branches. Figure 6 shows the two relations loop$_{\text{then}}$ and loop$_{\text{then}}$. Cartesian products yield the loop-lifted encodings q_{e_2} and q_{e_3} of the subexpressions e_2 and e_3, respectively. The result of the `return` clause is their disjoint union $q_{e_2} \uplus q_{e_3}$ shown on the right in Figure 6.

Observe how the intermediate result q_{e_1}, the loop-lifted encoding of the predicate expression e_1, is consumed by two different sub-plans in Figure 6. This plan sharing is characteristic for query plans emitted by a loop-lifting compiler. The optimizer component of the Pathfinder XQuery compiler has thus been explicitly tuned to handle graph-shaped plans [RTG07].

4.5 Optimizing and Evaluating Loop-Lifted XQuery Plans

The loop-lifting compilation procedure turns arbitrary XQuery expressions into a query plan composed of a rather standard set of algebraic operators (see Table 1). Besides the scalability advantages that result from this approach, the use of relational algebra as an

equivalent representation for XQuery expressions can help to solve a number of problems that proved hard on the level of the XQuery language:

Indifference of Order. Different notions of *order* are wired deeply into the XQuery language (document order, sequence order, and iteration order). In loop-lifted query plans, this surfaces as the maintenance of iter and pos information throughout the plan.

There are many situations in XQuery, however, where order does *not* matter to the outcome of a query, *e.g.*, in the inputs of existential predicates or the context sets of XPath location steps. In the relational plans, this usually means that an iter or pos column generated for some XQuery subexpression is never inspected by any upstream plan operator. The Pathfinder compiler uses a specific variant of *projection pushdown* to counter this effect, such that order information is only generated if indeed prescribed by the semantics of the query [GRT07].

Robust Join Detection. Since, in XQuery, there is no explicit join construct, the syntactical variations to express *value-based joins* are quite diverse. Based on the inference and inspection of functional dependencies, the Pathfinder compiler recognizes join situations in loop-lifted XQuery evaluation plans. This recognition is independent of syntactical variations and will detect, *e.g.*, the value-based join in `let $d := fn:doc(···) for $a in $d//a return $d//b[@c = $a/@d]` [RTG07].

Dependable Cardinality Estimates. The availability of dependable estimates for (intermediate) result sizes can significantly improve query optimization and execution, *e.g.*, to efficiently allocate resources in the physical plan. Unfortunately, the determination of such estimates is hard on the basis of the XQuery language. Existing techniques cover only rather limited subsets of the language.

In contrast, cardinality inference for relational query plans is a well-investigated field in database research. Loop-lifting makes this work immediately accessible to the estimation of result sizes for arbitrary XQuery expressions. Depending on the workload, this approach can be a suitable means to infer cardinality estimates for XQuery [Teu06].

5 Experimental Assessment

The prime motivation to re-use relational database technology for XML query processing was the expected scalability that we can inherit from mature RDBMS implementations. Pathfinder is a full implementation of the loop-lifting compilation procedure. Together with a staircase join extension to the MonetDB database kernel, it constitutes the open-source XQuery implementation *MonetDB/XQuery* [BGvK+06].

We used the XMark benchmark [SWK+02] and MonetDB/XQuery version 4.10.2 to verify whether the system indeed meets our scalability goals. At the top of Figure 7, we listed the query execution times (in milli-seconds) required to process the 20 XMark queries on a 111 MB XML instance (the system used for testing was equipped with 2×3.2 GHz Intel Xeon processors and 8 GB of main memory).

We further measured execution times on XMark instances of different sizes. Normalized to the elapsed times observed for the 111 MB instance, the resulting figures are illustrated in

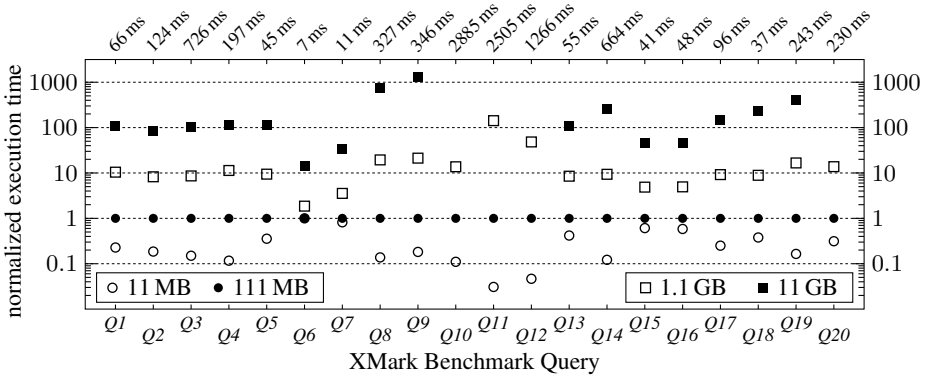


Figure 7: MonetDB/XQuery scalability with respect to document size. Figures on top: execution times on a 111 MB XMark instance. Execution times in the graph are normalized to these figures.

Figure 7. Over a large range of document sizes, we see a linear scaling with the document size, the only real outliers being Queries *Q11* and *Q12*. Both queries follow a quadratic scaling that stems from an intermediate result with quadratic complexity. For more in-depth experimental studies on the techniques we described, we refer to the experiments performed in [Teu06].

6 Summary

Our work demonstrates once more the versatility of the relational data model. We have shown how relational database systems can serve as efficient hosts to process XQuery. To suitably store the underlying XML data, we described *range encoding*, a variant of the XPath accelerator encoding developed in earlier work. A novel join operator, *staircase join*, provides an efficient implementation for XPath navigation steps over encoded tree data.

Our key contribution that allows the execution of arbitrary XQuery expressions on relational back-ends, however, is the *loop-lifting* compilation technique. Using a suitable encoding for XQuery's basic data type, sequences of items, the loop-lifting technique turns the `for` iteration primitive into a bulk-oriented execution strategy on the relational system.

To demonstrate the effectiveness of the resulting *relational XQuery processing stack*, we used the software developed in the context of the Pathfinder and MonetDB/XQuery companion projects. We showed how MonetDB/XQuery reaches linear scaling and interactive query response times beyond the gigabyte XML size limit.

References

[BGvK+06] 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 2006 ACM SIGMOD Int'l Conference on Management of Data*, Chicago, IL, USA, June 2006.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. of the 2002 ACM SIGMOD Int'l Conference on Management of Data*, pages 310–321, Madison, WI, USA, 2002.

[CSF+01] Brian Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. A Fast Index for Semistructured Data. In *Proc. of the 27th Int'l Conference on Very Large Databases (VLDB)*, pages 341–350, Rome, Italy, September 2001.

[FHK+02] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, and Robert Schiele. Anatomy of a native XML base management system. *The VLDB Journal*, 11(4):292–314, December 2002.

[Gra03] Goetz Graefe. Sorting and Indexing with Partitioned B-Trees. In *Proc. of the 1st Int'l Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, January 2003.

[GRT07] Torsten Grust, Jan Rittinger, and Jens Teubner. eXrQuy: Order Indifference in XQuery. In *Proc. of the 23th Int'l Conference on Data Engineering (ICDE)*, Beijing, China, April 2007.

[Gru02] Torsten Grust. Accelerating XPath Location Steps. In *Proc. of the 2002 ACM SIGMOD Int'l Conference on Management of Data*, pages 109–120, Madison, WI, USA, June 2002.

[GST04] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*, pages 252–263, Toronto, Canada, September 2004.

[GvKT03] 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.

[JLST01] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. TAX: A Tree Algebra for XML. In *Database Programming Languages (DBPL), 8th Int'l Workshop*, pages 149–164, Frascati, Italy, September 2001.

[NvdL05] Matthias Nicola and Bert van der Linden. Native XML Support in DB2 Universal Database. In *Proc. of the 31st Int'l Conference on Very Large Databases (VLDB)*, pages 1164–1174, Trondheim, Norway, September 2005.

[RSF06] Christopher Re, Jérôme Siméon, and Mary F. Fernández. A Complete and Efficient Algebraic Compiler for XQuery. In *Proc. of the 22nd Int'l Conference on Data Engineering (ICDE)*, Atlanta, GA, USA, April 2006.

[RTG07] Jan Rittinger, Jens Teubner, and Torsten Grust. Pathfinder: A Relational Query Optimizer Explores XQuery Terrain. In *Proc. of the 2007 BTW Conference (Datenbanksysteme für Business, Technologie und Web)*, Aachen, Germany, March 2007.

[SWK+02] Albrecht R. Schmidt, Florian Waas, Martin L. Kersten, Michael J. 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)*, pages 974–985, Hong Kong, China, August 2002.

[Teu06] Jens Teubner. *Pathfinder: XQuery Compilation Techniques for Relational Database Targets*. PhD thesis, Technische Universität München, October 2006. Verlag Dr. Hut, München, ISBN 3-89963-440-3.