Computer Science—Research and Development manuscript No.
(will be inserted by the editor)

Jens Teubner

The Relational XQuery Puzzle: A Look-Back on the Pieces Found So Far

Received: date / Accepted: date

Abstract Given the tremendous versatility of relational database implementations toward a wide range of database problems, it seems only natural to consider them as back-ends for XML data processing. Yet, the assumptions behind the language XQuery are considerably different to those in traditional RDBMSs. The underlying data model is a *tree*, data and results carry an intrinsic *order*, queries are described using explicit *iteration* and, after all, problems are everything else but regular.

Solving the *relational XQuery puzzle*, therefore, has challenged a number of research groups over the past years. The purpose of this article is to summarize and assess some of the results that have been obtained during this period to solve the puzzle. Our main focus is on the Pathfinder XQuery compiler, a full reference implementation of a purely relational XQuery processor. As we dissect its components, we relate them to other work in the field and also point to open problems and limitations in the context of relational XQuery processing.

Keywords Relational XQuery · Relational Tree Encoding · XPath · Compilation · Loop Lifting · Type Matching

1 Introduction

The W3 Consortium had not yet even set up its working group to develop what later would become XQuery [5], when some authors already suggested the use of *relational* database technology to process XML in a scalable and efficient manner [14]. The idea spurred the interest of research teams around the globe to solve the "relational XQuery puzzle."

Jens Teubner
ETH Zürich, Systems Group
Haldeneggsteig 4
8092 Zürich, Switzerland
E-mail: `jens.teubner@inf.ethz.ch`
`http://people.inf.ethz.ch/jteubner/`

A number of important pieces of this puzzle have been discovered since and, after almost a decade, it gets time to review, see what pieces have been found, and which ones are still missing. We do so by dissecting the internals of *Pathfinder* [46], a prototype of a *purely relational XQuery processor*. As we sketch its components, we relate them to ideas that complement, or could sometimes replace, the current implementation in Pathfinder.

Our goal is to construct a complete, efficient, and correct implementation of an XQuery engine. Since relational database systems are, by far, the most scalable and efficient data processors currently available, we want to implement XQuery functionality on relational foundations wherever possible. During our tour through Pathfinder, we'll see that this in fact can be done for far more language features than one would intuitively think of. The remainder of this work emphasizes those features that are particularly relevant or challenging in relational XQuery processing.

(a) Relational *encodings for trees* seeded the interest in relational XML processing. The way how XML data is represented at the relational end obviously affects the *evaluation of XPath*. These subjects are on our agenda for Sections 2 and 3 (respectively).
(b) To off-load the processing of an entire query to a relational back-end, data *and* query need to mapped into the relational world. In Section 4, we investigate ways to express the semantics of XQuery in terms of relational algebra.
(c) Substantial research efforts have been put into relational *query optimization*. In Section 5, we draw on this work to optimize XQuery on the relational level.
(d) Building an XQuery processor means to support *all* features of the language. Support for *XML Schema types* is one of them that often gets overlooked. We present an efficient implementation for *type matching* (the runtime aspect of XML Schema support) in Section 6.
(e) Some functionality in XQuery go beyond what relational systems can handle. In Section 7, we'll show

how a fair degree of *recursion* can be implemented efficiently on relational hosts.

We summarize our work in Section 8, where we also point at pieces that we think are still missing in order to complete the full puzzle.

2 XML to Tables and Back

Obviously, the performance of an RDBMS-based XQuery processor crucially depends on the representation of its principal data type, *ordered unranked trees* at the relational end. The work of Florescu and Kossmann [14] (dubbed "edge mapping") was an early attempt to establish such a representation. But although this work became a seed for the whole topic, it fell short in providing an *convincing* implementation for two key requirements in XQuery processing:

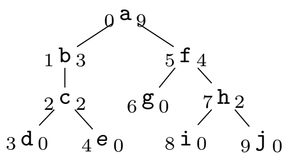
(a) The relational equivalent of two XML tree nodes v_1 and v_2 must easily be comparable to decide *node identity* (v_1 `is` v_2) and *document order* (v_1 `<<` v_2).[1]
(b) The encoding must support efficient XPath navigation along *all* twelve axes.

Inspired by the idea of constructing a high-performance XML storage solution based on relational technology, an abundance of research papers got published, all in the search for the optimal tree encoding. In retrospect, they all discovered essentially the same two approaches to relational XML storage: the ones that encode the tree structure using each nodes rank in a *pre- and postorder* tree traversal and the ones that picked up the idea of *Dewey ordering* (a popular way of organizing books in a library) and associated a vector of numbers with each node, printed typically using a dot-separated notation. A thorough treatment of the former idea has been published by Grust *et al.* [22], a popular implementation of the latter is ORDPATH [38].

2.1 pre/post-Based Tree Encodings

XPath accelerator, an encoding proposed in [19], stores the positions of each node v's occurrence in a pre- and postorder tree walk, $\mathsf{pre}(v)$ and $\mathsf{post}(v)$ (respectively), as attributes in a relational table accel. Additional columns hold a foreign key reference to the pre value of v's parent, $\mathsf{parent}(v)$, and the semantical value of v (*i.e.*, its XML node type, tag name, text value, etc.). Column pre in this encoding provides an immediate and simple implementation for the document order and node identity tests mentioned earlier.

[1] With an appropriate choice for *oid* values, this property could easily be added to [14].



(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) Encoding pre/size/level.

Fig. 1: XML document tree, annotated with $\mathsf{pre}(\cdot)$ and $\mathsf{size}(\cdot)$ information (left/right), and resulting tree encoding.

A virtue of this approach to XML storage is that it allows for a concise and machine-friendly characterization for all twelve XPath axes. For the `descendant` axis, *e.g.*, we have

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

XPath step navigation, hence, translates into a two-dimensional region query. Grust demonstrates how functionality of a commodity database implementation (R- and B-trees for that matter) is well suited to accelerate this type of queries [19].

Pre- and postorder ranks are related to each other according to the equation

$$\mathsf{pre}(v) - \mathsf{post}(v) = \mathsf{level}(v) - \mathsf{size}(v) \qquad (2)$$

for every node v in the tree ($\mathsf{level}(v)$ and $\mathsf{size}(v)$ denote v's distance from the tree root and its number of descendants, respectively).

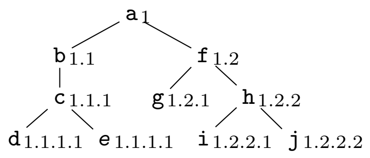
A consequence is that "new" encodings can be obtained by using subsets of the four properties to store the structural part of the XML tree.[2] One such encoding is illustrated in Figure 1, corresponding to the XML document

```
<a>
  <b><c><d/>e</c></b>
  <f>g<h><i/><j/></h></f>  .     (3)
</a>
```

This is the encoding used within the Pathfinder system. An inherent problem of pre/post-based numberings is the need to *renumber* parts of the document during updates or node construction [11]. This particular variant, however, minimizes the relabeling overhead since column size is invariant with respect to subtree copying or moving. Column level need only be shifted by a constant value in the face of either operation. For technical details regarding Pathfinder's XML storage refer to [7].

The TinyTree storage model of Saxon [30] is another example of a pre/post-based tree encoding.

[2] In fact, any *pair* of properties already suffices to encode the full tree structure, except the pair of $\mathsf{level}(v)$ and $\mathsf{size}(v)$.



(a) XML document tree.

dewey	kind	prop
1	elem	a
1.1	elem	b
1.1.1	elem	c
1.1.1.1	elem	d
1.1.1.2	text	e
1.2	elem	f
⋮	⋮	⋮

(b) Dewey encoding.

Fig. 2: Dewey labels for XML Instance (3).

2.2 Dewey-Based Encodings

Dewey-based encodings are the second major avenue the researchers followed to represent XML document trees in a tabular format. Each node is assigned a *vector* of integer values (typically separated by dots in print). For each node v, this vector consists of (a) the vector of the parent of v, extended by (b) the position of v among its siblings (according to document order). The resulting encoding is illustrated in Figure 2 for our earlier XML example.

Microsoft SQL Server uses Dewey-based ORDPATH [38] labels to encode the XML structure in its "primary XML index." Other implementations that use variants of the Dewey idea are IBM's DB2 9 pureXML or XTC's SPLIDs [27].

The strength of the Dewey approach is its update-friendliness. This is particularly the case of the ORDPATH implementation that allows for arbitrary tree modifications without the need to re-label major parts of the document. The "trick" here is to construct node labels only from odd numbers during an initial document load and reserve even numbers in-between as placeholders for future updates. Such updates are then accommodated by "careting in", without affecting the remainder of the tree [38].

The price of the update-friendliness is the dependence on *variable-sized node labels* (up to 20 bytes in SQL Server) and an increased CPU overhead to compare two node labels for their relationship in the XML document tree. Also note that this type of XML storage requires explicit support from the underlying RDBMS back-end, namely a "Dewey id" data type.[3]

Interestingly, both classes of tree encodings have very similar behavior during typical XPath step processing. Preorder ranks and Dewey ids both obey the same XML document order. Scans along indices over either representation, therefore, usually exhibit the same access patterns on secondary storage.

[3] Interestingly, ORDPATH has become a first-class (*i.e.*, user-accessible) data type in the 2008 release of SQL Server.

3 Stepping Through XML Forests

Any tree encoding would be meaningless if we used the relational database as a mere storage container for XML trees. In this section, we are going to explore efficient ways to perform *XPath step evaluation* over encoded tree data. In the spirit of this work, we first consider RDBMS functionality that is available in off-the-shelf systems. In Sections 3.2 and 3.3, we then look at potential additions to the DBMS kernel that could speed-up the processing of XPath.

3.1 XPath Evaluation Off-The-Shelf

The XPath axis characterization over pre/post-based tree data in Section 2.1 lends itself to the use of index structures with efficient support for *range queries*. At closer look, the region to scan typically is a *one-dimensional* interval only, as we can see if we characterize `descendant` based on v's pre and size values:

$$\begin{array}{c} v' \in v/\texttt{descendant} \\ \Leftrightarrow \\ \mathsf{pre}(v) < \mathsf{pre}(v') \leq \mathsf{pre}(v) + \mathsf{size}(v) \end{array} \tag{4}$$

or using the Dewey label of v, $v_1.v_2.\cdots.v_n$:

$$\begin{array}{c} v' \in v/\texttt{descendant} \\ \Leftrightarrow \\ v_1.v_2.\cdots.v_n < \mathsf{dewey}(v') \leq v_1.v_2.\cdots.(v_n+1) \end{array} . \tag{5}$$

Needless to say that such interval queries are well-supported by conventional B-tree indices.

Oftentimes, the `descendant` interval needs to be *filtered* to answer the actual user query. Examples are node tests that ask for a certain *node type* (*e.g.*, `descendant::text()`) or such that constrain element *tag names* (*e.g.*, `descendant::open_auction`. The output of an XQuery `child` step, in fact, is a filtered `descendant` result, too:

$$\begin{array}{c} v' \in v/\texttt{child} \\ \Leftrightarrow \\ v' \in v/\texttt{descendant} \ \wedge\ \mathsf{level}(v') = \mathsf{level}(v) + 1 \end{array} . \tag{6}$$

These filter criteria all have two important properties in common:

(a) they have a very low selectivity (there are elements of only ~ 70 different names in XMark [42] data, for instance, at levels smaller than 12),
(b) they are equality predicates.

As such, the entire step can be answered using a single scan along a *concatenated* $\langle f, \mathsf{pre}\rangle$ B-tree,[4] where f is the column that contains the respective filter criterion. Such a scan will *not* encounter any false hits, but directly yield the step result (in document order). To answer a

[4] In the interest of readability, we assume pre/size-encoded data. Most of our observations hold for Dewey-encoded data, too.

step along the `child` axis, *e.g.*, it is sufficient to scan a ⟨level, pre⟩ B-tree over Pathfinder's XML representation.

The performance advantage of this flexibility in XML indexing is significant: in [25], we showed how a relational XPath processor can out-perform a native XML processor by orders of magnitude. Note that this advantage comes at only little overhead. The low-selectivity prefixes lead to B-tree *partitioning* [18], which makes the indices particularly susceptible to *prefix compression* [4].

Readers interested in further techniques to accelerate XPath performance on commodity RDBMS implementations are referred to [25] for an in-depth treatment.

3.2 Tree Awareness with Staircase Join

The indexes sketched in the previous paragraphs strive to tell the relational query engine as many details as possible about data distributions in the encoded tree data and guide the system in navigating the data by standard relational means. Further performance improvements can be achieved by extending the system with tailor-made tree navigation algorithms, hence, "inoculating" it with tree awareness.

An example of such an algorithm is *staircase join* [21], which encapsulates tree awareness inside a single join algorithm. At the cost of only a local change to the RDBMS kernel, staircase join provides all knowledge about the tree origin of the stored data that is required for high-performance XPath processing. In addition, staircase join guarantees a duplicate-free evaluation result in XML document order, therefore lifts the requirement to explicitly *sort* any expression result only to comply with XPath.

Staircase join draws its advantage from three principal techniques:

(a) *Pruning* redundant context nodes before processing reduces overhead as well as the number of duplicate result nodes encountered.
(b) *Partitioning* the document relation and scanning partitions strictly sequentially guarantees a duplicate-free, document-ordered result and yields cache-optimal access patterns to secondary storage.
(c) *Skipping* avoids the scanning of significant amounts of data data by ignoring parts which can early be detected to not contain any result candidates. The decision to skip is based on tree knowledge inside the algorithm.

In [34], we used the open-source system PostgreSQL to demonstrate how staircase join could be incorporated into *any* relational database back-end. The most significant effect is that the time necessary to evaluate an XPath step now only depends on the size of the step's *result*. Contrast to the off-the-shelf system, the modified system was unaffected by the size of the queried *document*.

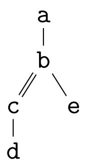
Pathfinder's distribution version MonetDB/XQuery ships with a MonetDB extension module that contains an implementation of staircase join. In addition, a *loop-lifted* variant is capable of evaluating a step over multiple context sets in parallel [6].

3.3 Holistic XPath Evaluation

Our discussion so far only considered evaluation strategies that break path expressions into individual steps for evaluation one after another. Bruno *et al.* [9] proposed to evaluate XPath expressions in a more *holistic* fashion that looks at an entire path at once. The two algorithms *PathStack* and *TwigStack* assume a tree encoding that makes `child` and `descendant` relationships between nodes easily decidable. Any of the encodings we discussed earlier would do.

To evaluate a straight k-step path, PathStack reads k individual tuple *streams* T_i, each providing a list of nodes in document order. Stream T_i is typically pre-filtered to yield only those nodes that satisfy, *e.g.*, the name test for step i in the user query. (Such a stream could be a concatenated B-tree as described in Section 3.1.) While reading the k streams, TwigStack maintains a combination of k stacks that hold (partial) query results in a very compact way. As soon as a match is found, PathStack emits a k-tuple of nodes (corresponding to a "binding" for each of the k steps in the query).

TwigStack extends this idea to the evaluation of *twig-shaped* paths. The figure on the right, *e.g.*, illustrates the twig that corresponds to the path



```
/descendant::a/child::b
  [descendant::c/child::d] .
  /child::e
```

Given such a twig pattern, TwigStack runs a variant of PathStack for every root-to-leaf path in the twig, then merges their result tuples to compute the overall result. Rather than running a set of PathStack operators independently, TwigStack synchronizes the processing of all root-to-leaf paths, thereby minimizing the production of partial results that cannot qualify for the overall operator output (this is very much like what the classical merge join algorithm does).

Holistic (twig-oriented) path evaluation algorithms can play their trump in the evaluation of longer, pattern-type queries, where a single operator of moderate complexity may excel over many simple single-step joins. The *crux* of twig-style path processing is that the twig model is considerably off the syntax and semantics of the XPath language. Michiels *et al.* [28; 36] describe the approach taken in Galax [15] to detect opportunities for twig processing.

It is worth noting that step-by-step evaluation is mostly a mental model for the evaluation strategies we described earlier. In a pipelined execution engine, the k joins that evaluate a k-step path all run in parallel, which effects in an actual data access pattern that very much resembles what happens in PathStack.

Twig join algorithms are in real-world use, *e.g.*, in DB2's XML subsystem pureXML [29].

4 From XPath to XQuery

Equipped with efficient evaluation mechanisms for the XQuery sub-language XPath, the next step to take is the construction of a full XQuery processor based on purely relational foundations. Unfortunately, these foundations assume a data model that is significantly different from the one we need to support XQuery. The data model of XQuery, *ordered sequences of items*, faces *unordered tables of tuples* on the relational end, explicit *iteration* faces *set-oriented processing.*

Two approaches are conceivable to minimize this gap:

(a) force the *physical* processing order in the relational engine to be aligned with the order semantics in XQuery or
(b) lift order and iteration to the *logical* level by making both concepts explicit using column values.

We first take a look into Microsoft SQL Server, which follows route (a) to implement the XQuery semantics.

4.1 XQuery Compilation in SQL Server

The relational treatment of arbitrary XQuery expressions mostly hinges on an appropriate translation of the FLWOR construct, the XQuery way to inspect and generate the particular order of an XQuery item sequence. To implement the semantics of FLWOR expressions, SQL Server relies on its existing `APPLY` operator, available, *e.g.*, in terms of the `CROSS APPLY` syntax at the surface level in Microsoft's SQL dialect Transact-SQL.

The semantics of `APPLY` is to read its left-hand input relation R and run a parameterized execution of the right hand-side expression $S(\$x)$ for each tuple in R. All execution results are then collected to obtain the output of the overall expression $R\,\texttt{APPLY}_{\$x}\,S$ [16]:

$$R\,\texttt{APPLY}_{\$x}\,S = \bigcup_{r \in R} \left(\{r\} \times S\,[\{r\}/\$x]\right) \quad . \tag{7}$$

Provided that the system preserves the *physical order* of both arguments, `APPLY` is a direct implementation of XQuery's `for-return` iteration primitive. The following compilation rule illustrates how SQL Server uses `APPLY` to generate the algebraic equivalence of an XQuery `for` clause [39] (read $\mapsto$ as "compiles to"):

$$\frac{e_1 \mapsto q_1 \qquad e_2 \mapsto q_2}{\texttt{for}\ \$x\ \texttt{in}\ e_1\ \texttt{return}\ e_2 \mapsto q_1\ \texttt{APPLY}_{\$x}\ q_2} \quad . \tag{8}$$

Before actual execution, SQL Server will massage the resulting `APPLY` expression using its existing rewrite mechanisms (which were originally targeted to optimize SQL sub-queries and aggregates [16]).

The use of `APPLY` to implement XQuery's iteration primitive nicely exploits existing machinery in the query engine of SQL Server. The simplicity of this approach, however, is also its Achilles' heel. The dependence on a given execution order may prevent interesting opportunities for order-related optimizations, which in Section 5.1.3 we'll find to be very attractive in the context of XQuery.

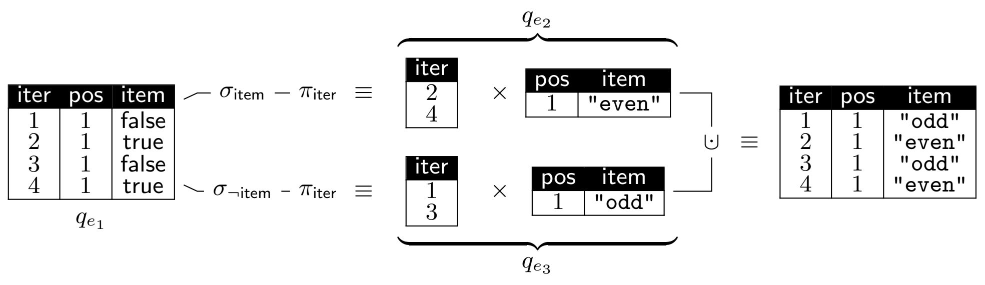
$\pi_{\ldots,\mathsf{b:a},\ldots}$	column projection, renaming (a into b)
σ_{a}	selection (select rows with $\mathsf{a} = \texttt{true}$)
$\bowtie_{\mathsf{a=b}}$, $\times$	equi-join, Cartesian product
$\dot{\cup}$, $\setminus$	disjoint union (append), difference
δ	duplicate row elimination
$\varrho_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle\Vert\mathsf{c}}$	row numbering (grouped by c)
$\circledast_{\mathsf{a}:(\mathsf{b}_1,\mathsf{b}_2)}$	arithmetic/comparison operator $*$
$\lrcorner_{\mathsf{a}:ax::nt(\mathsf{b})}$	XPath step operator ($\mathsf{a} = \mathsf{b}/ax::nt$)
$⚛_{\mathsf{a:b}}$	XQuery atomization ($\mathsf{a} = \texttt{fn:data(b)}$)
$\texttt{doc}_{\mathsf{a:b}}$	XML document access ($\mathsf{a} = \texttt{fn:doc(b)}$)
ε, τ	element/text node construction
$agg_{\mathsf{a}\Vert\mathsf{b}}$	aggregation, grouped by b

Table 1: Relational algebra used by the Pathfinder XQuery compiler ($agg \in \{\texttt{count}, \texttt{sum}, \texttt{max}, \ldots\}$).

4.2 Loop Lifting: Order Made Explicit in Logical Plans

As discussed in [16], operator `APPLY` does not add expressive power to standard functionality available in any RDBMS (textbook-style relational algebra plus grouping operators as required for SQL). Therefore, it is possible to compile XQuery FLWOR expressions into standard SQL code. This route is taken by the Pathfinder XQuery compiler which translates arbitrary XQuery expressions into a textbook-style relational algebra, enriched only with few operators to exploit available functionality on particular RDBMS back-ends [20; 23]. This algebra can then be externalized for consumption by numerous back-ends, including MonetDB [8], kdb+ [43], and SQL:1999-compatible systems.

Table 1 illustrates the set of operators emitted by the Pathfinder XQuery compiler. In this table, operators ε, τ, $\texttt{doc}_{\mathsf{a:b}}$, $⚛_{\mathsf{a:b}}$, and $\lrcorner_{\mathsf{a}:ax::nt(\mathsf{b})}$ are syntactic shorthands for micro-plans composed of remaining operators (*e.g.*, step navigation is a join with the pre/size document table). Operator $\varrho_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle\Vert\mathsf{c}}$ is the Pathfinder-internal representation of the SQL:1999 construct `ROW_NUMBER () OVER (PARTITION BY c ORDER BY` $\mathsf{b}_1, \ldots, \mathsf{b}_n$`) AS a`.


q_{e_2}
$\sigma_{\text{item}} - \pi_{\text{iter}}$ ≡
$\sigma_{\neg\text{item}} - \pi_{\text{iter}}$ ≡
×
×
⊍ ≡
q_{e_1}
q_{e_3}

iter	pos	item
1	1	false
2	1	true
3	1	false
4	1	true

iter
2
4

pos	item
1	"even"

iter
1
3

pos	item
1	"odd"

iter	pos	item
1	1	"odd"
2	1	"even"
3	1	"odd"
4	1	"even"

Fig. 3: Execution trace for loop-lifted execution of Query Q_1. Using textbook-style algebra operators, the loop-lifted sequence representation is maintained for all subexpressions (illustrated for subexpressions e_1 through e_3).

iter	pos	item
1	1	$v_{1,1}$
⋮	⋮	⋮
1	s_1	v_{1,s_1}
⋮	⋮	⋮
n	1	$v_{n,1}$
⋮	⋮	⋮
n	s_n	v_{n,s_n}

To maintain compliance with the XQuery semantics over an unordered data model, Pathfinder makes sequence and iteration order explicit in its data representation. The evaluation result of *any* XQuery subexpression e is represented as shown on the right. In a *single* table, this *loop-lifted* sequence encoding holds the value of e for *all* iterations e occurs in. A tuple $\langle i, p, v_{i,p} \rangle$ in this encoding indicates that, in the i-th iteration, the value of e has item $v_{i,p}$ at position p.

Pathfinder's compilation procedure makes sure that iter- and pos-columns are properly maintained during query execution and that *each* subexpression result is obtained in its loop-lifted representation. The relational evaluation trace shown in Figure 3 illustrates this for the query

$$\texttt{for \$}x\texttt{ in (3,4,5,6) return}$$
$$\texttt{if } \underbrace{\texttt{(\$}x\texttt{ mod 2 eq 0)}}_{e_1} \texttt{ then } \underbrace{\texttt{"even"}}_{e_2} \texttt{ else } \underbrace{\texttt{"odd"}}_{e_3} \;. \qquad (Q_1)$$

We refer to [20; 23] for a detailed description of Pathfinder's compilation procedure and the resulting query plans.

Observe that the resulting plans have a strictly *set-oriented* semantics, a property inherited from the definition of their individual operators. Thus, the system is free to evaluate queries like Q_1 in any order it sees fit—or even using a parallel mode of execution.

5 Relational XQuery Optimization

We've just seen how arbitrary compositions of XQuery expressions can be turned into purely relational evaluation plans, using either syntactic sugar available in SQL Server (the APPLY operator), or the loop lifting technique that maps XQuery directly into relational algebra. Neither technique, however, can hide the full compositionality of XQuery and generated plans take a shape that is very different to the π-σ-$\bowtie$ pattern preferred by typical RDBMS optimizers, as shown in Figure 4 for the plan obtained by a loop-lifting compiler for Query Q8 from the XMark benchmark set. This section illustrates how Pathfinder deals with plans of such shape and how the application of techniques from the relational domain can lead to new insights into XQuery problems.

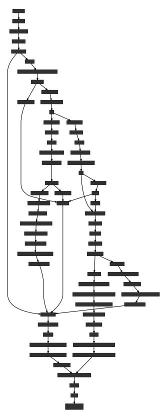


Fig. 4: Typical plan shape.

5.1 Rewriting and Join Graph Isolation

The strength of relational database systems certainly is their ability to process *joins* in a highly efficient manner. The best *order* in which joins should be applied is determined by sophisticated join optimization algorithms. To perform their work, however, these algorithms need to have a clear view on all involved join operators. Unfortunately, in the case of relational XQuery evaluation plans, this view is obstructed by the stacked plan shape we saw just a moment ago. Pathfinder's optimizer, therefore, tries to *isolate* join graphs, then hand them over to a traditional join enumeration algorithm [26].

5.1.1 Peephole-Style Optimization

Toward this end, Pathfinder employs its *peephole-style* plan optimizer, which both, (a) is flexible to cover a wide range of optimizations and plan analyses and (b) guarantees scalability with plan sizes, since operators are looked at only one at a time.[5]

Optimization is performed in two distinct phases:

1. An *inference phase* traverses the entire plan tree once. For each operator □, the plan analyzer collects any relevant information about the vicinity of □ and stores this information in an annotation to □.
2. This gives the subsequent *rewrite phase* enough information to decide on plan rewrites by looking at the annotations to □ only.

[5] Loop-lifted compilation typically leads to plan sizes of a hundred and more operators prior to optimization.

$$\frac{\mathsf{a} \in \mathit{key}\,(q)}{\mathit{key}\,(\pi_{\ldots,\mathsf{b:a},\ldots}(q)) \supseteq \{\mathsf{b}\}}(\textsc{Key-1}) \qquad \frac{\mathsf{b} \in \mathit{key}\,(q_2)}{\mathit{key}\,(q_1 \underset{\mathsf{a=b}}{\bowtie} q_2) \supseteq \mathit{key}\,(q_1)}(\textsc{Key-2}) \qquad \frac{}{\mathit{key}\,\left(\varrho_{\mathsf{a}:\langle \mathsf{b_1},\ldots,\mathsf{b_n}\rangle}(q)\right) \supseteq \{\mathsf{a}\}}(\textsc{Key-3})$$

$$\frac{}{\mathit{const}\,(\pi_{\mathsf{a}:v}(q)) \supseteq \{\mathsf{a}^{=v}\}}(\textsc{Const-1}) \qquad \frac{\mathsf{a}^{=v} \in \mathit{const}\,(q)}{\mathit{const}\,(\pi_{\ldots,\mathsf{b:a},\ldots}(q)) \supseteq \{\mathsf{b}^{=v}\}}(\textsc{Const-2}) \qquad \frac{}{\mathit{const}\,(\sigma_{\mathsf{a}}(q)) \supseteq \left\{\mathsf{a}^{=\mathsf{true}}\right\}}(\textsc{Const-3})$$

Fig. 5: Peephole-style inference rules to obtain operator annotations *key* (·) (key columns) and *const* (·) (columns holding a constant value).

$$\frac{\square \notin \{\delta\} \qquad \{\mathsf{a},\mathsf{b}\} \subseteq \mathit{cols}(q_1) \cup \mathit{cols}(q_2)}{\square(q_1) \bowtie_{\mathsf{a=b}} q_2 \longrightarrow \square\,(q_1 \bowtie_{\mathsf{a=b}} q_2)} \qquad \frac{\mathit{key}\,(q) \neq \emptyset}{\delta(q) \longrightarrow q}$$

$$\frac{\square \in \{\sigma_p, \delta\} \qquad p \text{ independent of } \mathsf{a}}{\square\left(\varrho_{\mathsf{a}:\langle \mathsf{b}_1,\ldots\mathsf{b}_n\rangle}(q)\right) \longrightarrow \varrho_{\mathsf{a}:\langle \mathsf{b}_1,\ldots\mathsf{b}_n\rangle}\,(\square(q))} \qquad \frac{}{\varrho_{\mathsf{a}:\langle\rangle\|\mathsf{c}}(q) \longrightarrow q \times \begin{array}{|c|}\hline \mathsf{a} \\ \hline 1 \\ \hline \end{array}}$$

$$\frac{\mathsf{b}_i^{=v} \in \mathit{const}\,(q)}{\varrho_{\mathsf{a}:\langle\ldots,\mathsf{b}_{i-1},\mathsf{b}_i,\mathsf{b}_{i+1}\ldots\rangle\|\mathsf{c}}(q) \longrightarrow \varrho_{\mathsf{a}:\langle\ldots,\mathsf{b}_{i-1},\mathsf{b}_{i+1}\ldots\rangle\|\mathsf{c}}(q)}$$

Fig. 6: Simplified rewrite rules to implement join graph isolation in Pathfinder [26]. Rules move obstructing operators toward the plan root to clear the view for join planning algorithms.

Both phases are driven by extensible sets of rules. Figure 5, *e.g.*, exemplifies some of the inference rules that infer annotation *key* (□) for operators π, $\bowtie$, and ϱ (*key* (□) lists key columns in the output of □) and annotation *const* (□) for π and σ (*const* (□) $\supseteq \{\mathsf{c}^{=v}\}$ indicates that column c in □ is constant and has value v). Both annotations are inferred bottom-up (but others may also be derived top-down). In the following, we sometimes assume the presence of an annotation *cols*(□) that holds schema information about □'s output.

5.1.2 Join Graph Isolation

The annotated information is the basis for subsequent rewrite rules. Figure 6 illustrates some of the plan modifications that lead to an isolation of join graphs. Their joint goal is to "pull" obstructing plan operators (such as blocking ϱ or δ operators) toward the plan root, leaving join graphs behind near the plan leaves. Rules 1 and 3, for instance, push down equi-joins and pull out row numbering operators, respectively. Rule 2 avoids duplicate elimination if the input contains a key column, while Rules 4 and 5 simplify or eliminate instances of row numbering. Refer to [26] for a more extensive documentation of Pathfinder's plan rewrite rule set.

With join graphs isolated, a traditional join optimizer (*e.g.*, in an SQL:1999 back-end to Pathfinder) is now free to move around joins and evaluate them in any order it sees fit. Joins in relational XQuery evaluation plans may come from different sources:

(a) *XPath location steps* translate into joins in the relational plan,
(b) *iteration* in XQuery (the `for` clause) is compiled into a join much like the dependent join in SQL Server (see Section 4.1), and
(c) *value-based joins* in the input query end up as (scattered) σ-$\times$ pairs in loop-lifted plans. XQuery lacks an explicit join construct, but value-based joins are common and can be expressed in different syntactical ways.

The effect of join ordering on sources (a) and (c) is particularly interesting. Reordering joins that evaluate XPath location steps effectively alters the direction of path navigation. A traditional join optimizer, therefore, solves XPath optimization problems that challenged researchers in the past, such as rewriting into forward-only paths [37] or deciding top-down vs. bottom-up path evaluation [35].

Detecting instances of source (c) is a known hard problem in XQuery processing. The feature richness of XQuery allows value-based joins to be expressed in a large variety of ways, making them hard to detect based on syntax analyses. Pathfinder's join detection is based on the rewrite principles we mentioned before. In [7], we found it to be the only XQuery processor capable of detecting all join scenarios in the XMark benchmark set [42]. Moreover, with join graph isolation, Pathfinder can optimize queries *across* all join sources (a)–(c).

5.1.3 Omnipresence and Lack of Order in XQuery

Another rewrite strategy implemented in the optimizer of Pathfinder is the *pushdown of projections.* A peephole-style data flow analysis discovers table columns produced but never consumed by any upstream plan operator. Such columns will be discarded early in the plan DAG or, most importantly, their generation be *avoided* if possible.

Remember that, in loop-lifted XQuery plans, table columns may contain other information than only user data. Columns iter and pos are Pathfinder's device to encode *order* in otherwise set-oriented evaluation plans. Though technically this releases the back-end from any prescribed tuple order, the maintenance of both columns may ultimately still impose a specific row order. This constraint gets lifted, once columns iter or pos can successfully be *projected out* from the plan. The system is then free to do its task in *any physical order.*

It turns out that a column projection of this kind is applicable more often than one might think. Existential semantics, aggregation, explicit requests for sorted output (`order by` clause), or explicit user-requested order relaxation (`fn:unordered(·)`, `unordered{·}`) are

$$\frac{}{|q_1 \uplus q_2| = |q_1| + |q_2|}(\textsc{Card-1}) \qquad \frac{}{|q_1 \times q_2| = |q_1| \cdot |q_2|}(\textsc{Card-2})$$

$$\frac{}{|\sigma_{\mathsf{a}}(q)| = |q| \cdot {}^{1}\!/_{10}}(\textsc{Card-3}) \qquad \frac{\mathsf{a}^{\text{▙}\,H} \in \mathit{hist}\,(q)}{|\sigma_{\mathsf{a}}(q)| = |q| \cdot H[\mathsf{true}]}(\textsc{Card-4})$$

Fig. 7: System R-style cardinality estimation rules [45].

all situations where projection pushdown instantly leads to unordered processing. On realistic workloads, this may lead to orders of magnitude in performance improvement [24].

5.2 Dependable Cardinality Forecasts for XQuery

The inference of a good *physical* plan (*e.g.*, the choice of the proper join order for the join graphs in Section 5.1.2) is only possible if the system can make accurate predictions on the *cost* of potential plan alternatives. These predictions, in turn, usually depend on *cardinality estimates* for (intermediate) XQuery expression results.

It is fairly well understood how such estimates can be computed for basic XPath expressions. Data guides [17], *e.g.*, succinctly summarize an XML document tree by reducing multiple nodes with an identical root-to-leaf path to a single instance in the guide. Annotated with such statistical information, such a summary provides a high estimation accuracy for common (and order-insensitive) paths with only small space overhead. Follow-up work has improved on the ideas of data guides by reducing the memory footprint of collected statistics [1] or providing support for order-sensitive axes [32] or branching paths [41].

The syntactic diversity of XQuery, combined with a blurred distinction between schema and data, makes it hard to lift such results to the level of all XQuery. This knot can be cut by using relational algebra as a framework that links available XPath estimation work with traditional techniques, such as data histograms or the seminal System R estimator. The concise semantics of algebraic operators thereby serves as a common ground for meaningful reasoning over cardinalities.

The query analyzer in [45] is an implementation of this idea that was shown to provide high-quality estimates for a wide range of XQuery workloads. In line with Pathfinder's peephole-style plan assessment, it consumes and produces plan annotations to ultimately yield annotation $|\square|$, the projected cardinality for operator $\square$.

Figure 7 illustrates how basic estimation rules from System R fit into this inference mechanism. The cardinality of disjoint union $\uplus$ or Cartesian product $\times$ operators can be determined by adding or multiplying both input cardinalities (Rules Card-1 and Card-2). A histogram can be used to judge the selectivity of a selection predicate if available (Rule Card-4). If not, Rule Card-3 applies the System R 10 % heuristic to obtain the output cardinality.

5.2.1 Data Flow Analysis and Value Domains

The stacked shape of loop-lifted XQuery evaluation plans is reflected in the rule set of [45], too. An annotation *dom* ($\square$) speculates over the size of the runtime *value domain* of each column c (*i.e.*, the number of distinct values in c) and reasons over known inclusion relationships between domains. This implements *data flow analysis* just to the amount necessary for cardinality estimation.

Domain sizes and table cardinalities often interact. If operator ϱ is used, *e.g.*, to attach a new key column to the output of q, the new column is going to range exactly over values $1, \ldots, |q|$ ($\|\alpha\|$ indicates the size of domain α):

$$\frac{}{\mathit{dom}\left(\varrho_{\mathsf{a}:\langle \mathsf{b}_1,\ldots,\mathsf{b}_n\rangle}(q)\right) \supseteq \mathit{dom}\,(q) \cup \left\{\mathsf{a}^{\alpha} \wedge \|\alpha\| \overset{!}{=} |q|\right\}}$$

Likewise, the number of groups in an aggregate function is determined by the domain size of the grouping criterion c:

$$\frac{\mathsf{b}^{\beta} \in \mathit{dom}\,(q)}{\left|\mathit{agg}_{\mathsf{a}\|\mathsf{b}}(q)\right| = \|\beta\|}(\textsc{Card-5}) \ .$$

See [45] for details on peephole-style data flow tracking.

5.2.2 Interfacing with XPath Estimation

Relational XQuery cardinality estimation does not imply any particular technique to estimate XPath subexpressions. Rather, a generic interface allows plugging in any of the published techniques for XPath estimation.

To this end, Pathfinder tracks the application of ⏍ operators (XPath step navigation) and constructs navigation traces very much like the *projection paths* in the Galax XQuery processor [33]. Each node-valued column c in the loop-lifted plan is annotated with the navigation path that has been followed to obtain the nodes in c:

$$\frac{\mathsf{b}^{\Rightarrow p} \in \mathit{path}\,(q)}{\mathit{path}\left(\text{⏍}_{\mathsf{a}:\mathit{ax}::\mathit{nt}(\mathsf{b})}(q)\right) \supseteq \mathsf{a}^{\Rightarrow p/\mathit{ax}::\mathit{nt}} \cup \mathit{path}\,(q)} \ .$$

A set of related rules ensures that the information is properly propagated through the operator graph.

Tracked path information is then used to invoke the XPath estimation subsystem. Assuming a mechanism to predict the *fanout* of a location step $\mathit{ax}::\mathit{nt}$ that originates in a node set reachable via the path p,

$$\mathsf{Pr}_{\mathit{ax}::\mathit{nt}}\,(p) := \frac{\texttt{fn:count}\,(p/\mathit{ax}::\mathit{nt})}{\texttt{fn:count}\,(p)} \ ,$$

the cardinality of the step operator ⏍ can be estimated according to

$$\frac{\mathsf{b}^{\Rightarrow p} \in \mathit{path}\,(q)}{\left|\text{⏍}_{\mathsf{a}:\mathit{ax}::\mathit{nt}(\mathsf{b})}(q)\right| = |q| \cdot \mathsf{Pr}_{\mathit{ax}::\mathit{nt}}\,(p)}(\textsc{Card-6}) \ .$$

Further examples are detailed in [45].

The attractiveness of using relational query plans to estimate XQuery cardinalities is its *robustness* with respect to the syntactic diversity of XQuery and potential mis-estimations for intermediate expressions. An extensive experimental assessment in [45] demonstrated that a relational XQuery estimator can cope with a wide range of query workloads.

6 Scalable XQuery Type Matching

Unlike most existing programming or query languages, XQuery blurs the distinction between data and its type. Type names can be used, *e.g.*, as the node test in XPath location steps. Likewise, the type of any XQuery item can be inspected at runtime by means of the `typeswitch` or `instance of` constructs. In this section, we look into relational support for such functionality.

The approach taken in [44] is inspired by the XQuery Data Model specification [13]. In XQuery, every item x is defined to be a *pair*, consisting of a *value* v and its *type annotation* t (a reference to a named XML Schema type):

$$x = v \; \texttt{of type} \; t \;\; .$$

Provided a suitable representation for named types, this definition can directly be used to enrich a relational sequence encoding with dynamic type information: each instance of an item column becomes a pair of value and type. The modified loop-lifted sequence encoding shown on the right, *e.g.*, assumes the availability of *surrogates* τ_t to represent the type annotation t.

iter	pos	value	type
⋮	⋮	⋮	⋮
i	p	v	τ_t
⋮	⋮	⋮	⋮

6.1 Sequence Type Matching

The common ground for all XQuery operations on runtime type information is the *type matching* process defined in the XQuery Formal Semantics [12]. In a nutshell, a singleton XQuery item $x = v$ `of type` t' *matches* a named type t if the type annotation of x, t', references a named type definition that has been *derived* from t (by extension or restriction). A sequence $x =$ `(`x_1`,`$\ldots$`,`x_l`)` matches a sequence type $t\Box$ if all singletons in x match t and the length l of x is compatible with the occurrence indicator $\Box$.

Implementing a singleton type match, therefore, implies a lookup in the derives from hierarchy. One insight in [44] is that derives from describes a proper tree structure. We already saw how trees can efficiently handled by relational means: preorder ranks in a pre/size-encoded type hierarchy or Dewey ids for type relationships would both make for appropriate implementations of τ_t.[6] Figure 8 illustrates a pre/size encoding, termed *type ranks* in [44], for a subset of the predefined XDM type hierarchy, enriched with user-defined simple (`Price` and `Currency`) and complex types (`AuctionItem` and `CarAuctionItem`).

```
⟨0, xs:anyType, 12⟩
 ├─ ⟨1, xs:untyped, 0⟩
 ├─ ⟨2, xs:anySimpleType, 8⟩
 │   └─ ⟨3, xs:anyAtomicType, 7⟩
 │       ├─ ⟨4, xs:boolean, 0⟩
 │       ├─ ⟨5, xs:decimal, 2⟩
 │       │   ├─ ⟨6, xs:integer, 0⟩
 │       │   └─ ⟨7, Price, 0⟩
 │       ├─ ⟨8, xs:string, 1⟩
 │       │   └─ ⟨9, Currency, 0⟩
 │       └─ ⟨10, xs:untypedAtomic, 0⟩
 └─ ⟨11, AuctionItem, 1⟩
     └─ ⟨12, CarAuctionItem, 0⟩
```

Fig. 8: Sample type hierarchy, annotated with pre(·) (left) and size(·) (right) numbers.

Assuming a pre/size-encoded type hierarchy, predicate matches can then by characterized as

$$\frac{\begin{array}{c} x = v \; \texttt{of type} \; t' \\ \mathsf{pre}(t) \leq \mathsf{pre}(t') \leq \mathsf{pre}(t) + \mathsf{size}(t) \end{array}}{x \; \mathsf{matches} \; t} \; . \qquad (9)$$

Compare this to the pre/size-based characterization of the XPath `descendant` axis in Section 3.1 and note that, due to syntactical constraints in XQuery, type t (*i.e.*, the two interval ends) are always known at query compilation time. Only t' is runtime-dependent in this judgment.

The use of type ranks avoids the runtime recursion required by existing XQuery processors to resolve the derives from property (*e.g.*, [30; 15]).

6.2 Type Aggregation

Lifted to sequence-valued operands, Judgment 9 reads

$$\frac{\begin{array}{c} \forall(x_i = v_i \; \texttt{of type} \; t_i) : \\ \mathsf{pre}(t) \leq \mathsf{pre}(t_i) \leq \mathsf{pre}(t) + \mathsf{size}(t) \\ \text{length of } x \text{ compatible with } \Box \end{array}}{x \; \mathsf{matches} \; t\Box} \; . \qquad (10)$$

Informally, we test the type annotation t_i for each item x_i in x against the interval defined by the pre and size values of t.

There is a different (and more database-friendly) way to obtain the same result. Instead of separately testing each $\mathsf{pre}(t_i)$ against the given interval boundaries,

[6] Note that we are encoding trees of *types* here, not XML document trees.

$$\frac{e \mapsto q_e \qquad t \text{ is a named atomic type}}{e\ \texttt{instance of}\ t\square \mapsto \left(\begin{array}{l} \texttt{SELECT iter, 1 AS pos,} \\ \qquad \texttt{CASE WHEN (MIN(type) >= } \mathsf{pre}(t) \texttt{ AND MAX(type) <= } \mathsf{pre}(t)+\mathsf{size}(t) \\ \qquad\qquad\qquad \texttt{AND COUNT(*) IS COMPATIBLE WITH } \square \texttt{)} \\ \qquad\quad \texttt{THEN 'true' ELSE 'false' END AS value,} \\ \qquad \texttt{pre(xs:boolean) AS type} \\ \quad \texttt{FROM } q_e \\ \texttt{GROUP BY iter} \end{array}\right)} \ (\textsc{InstanceOf})$$

Fig. 9: Translation of `instance of` for atomic types in a loop-lifting compiler (for details refer to [44]).

we could as well determine the *minimum* and *maximum* type ranks in x first, then do the interval test only once (*type aggregation*):

$$\frac{\begin{array}{c} \min\limits_{(x_i=v_i \texttt{of type}\, t_i)\in x}(\mathsf{pre}(t_i)) \geq \mathsf{pre}(t) \\ \max\limits_{(x_i=v_i \texttt{of type}\, t_i)\in x}(\mathsf{pre}(t_i)) \leq \mathsf{pre}(t)+\mathsf{size}(t) \\ \text{length of } x \text{ compatible with } \square \end{array}}{x \ \mathsf{matches}\ t\square} \ . \tag{11}$$

Needless to say that this rewrite enables the use of advanced algorithms for aggregation on modern RDBMS back-ends.

As detailed in [44], all remaining type-related tasks can be turned into similar aggregation problems, including the check for the occurrence indicator □ and support for XML Schema substitution groups.

6.3 Loop-Lifted Type Matching

The aggregation semantics is easily expressible in the context of a loop-lifted XQuery compiler. Figure 9 shows the (simplified) translation rule for the XQuery `instance of` operator. Observe how this rule emits a four-column loop-lifted output representation to ensure compositionality with arbitrary XQuery expressions. The output of an `instance of` expression is a Boolean value. Column type, therefore, is populated with the preorder rank of the `xs:boolean` type, a constant determined at query compilation time.

In experiments on top of an SQL database back-end we found that, besides providing a slim and efficient implementation for derives from, type ranks integrate particularly well with existing strategies for relational query processing. A relational database will immediately take advantage of, *e.g.*, a type column that happens to be physically sorted. Further, for XPath node tests on type *and* tag name, the system may now, depending on estimated costs, freely choose among indexes on types, tag names, or even on a combination of both.

7 Hitting the Limits

It is well-known that XQuery is a Turing-complete language, which relational algebra is not [31]. No encoding or compilation strategy can get around this limit in expressive power.

```
$res \leftarrow e_{body}(e_{seed})$
do
  $res \leftarrow e_{body}(res)$ union $res$
while $res$ grows
return $res$
```

(a) Algorithm *Naïve*.

```
$res \leftarrow e_{body}(e_{seed})$
$\Delta \leftarrow res$
do
  $\Delta \leftarrow e_{body}(\Delta)$ except $res$
  $res \leftarrow e_{body}(res)$ union $res$
while $res$ grows
return $res$
```

(b) Algorithm *Delta*.

Fig. 10: Potential implementations for the `with`$\cdots$ construct. Algorithm *Naïve* is correct for all e_{body}, but less efficient than *Delta*.

The source of Turing-completeness in XQuery is primarily the allowance of arbitrary recursion in user-defined functions.[7] At the same time, recursion is a highly desirable feature in a query language that operates over an inherently recursive data structures such as XML trees. And while the general problem is proven hard, limited types of recursion may still be tractable on relational back-ends, yet be useful in actual applications.

An extension recently built into Pathfinder [3] aims to explore one particular flavor of recursion, the support for *transitive closure*. Its evaluation has been studied extensively in the context of deductive databases, and any decent SQL processor readily ships with support for transitive closure.

To relieve the query compiler from detecting transitive closure operations in function declarations in the input query, Pathfinder requires users to make their intentions explicit using the

$$\texttt{with } \$v \texttt{ seeded by } e_{seed} \texttt{ recurse } e_{body}(\$v)$$

construct in a Pathfinder-specific extension to XQuery. (Based on an initial binding of $\$v$, expression e_{body} is recursively evaluated until a fixed point is reached.)

While the semantics of the `with`$\cdots$ construct can directly be translated into a back-end implementation, the resulting Algorithm *Naïve* (Figure 10a) may often not be the most efficient strategy to compute the fixed point. Its counterpart *Delta* is the folklore variant in deductive

[7] It is not surprising that several commercial XQuery implementations built around relational technology do not support user-defined functions.

databases and avoids the repeated re-computation of the results obtained in early loop iteration.

In XQuery, unfortunately, Algorithm *Delta* turns out to be *not* correct for all instances of e_{body} and, therefore, cannot serve as a general-purpose implementation for `with` $\cdots$. But if those instances that satisfy a *distributivity property* can be detected reliably, an optimizer can trade *Naïve* for *Delta*. Based on a relational representation for e_{body}, the distributivity property can be proven by well-defined algebraic plan rewrites. In a nutshell, Pathfinder tries to push a *union operator* $\uplus$ through the plan that implements e_{body}. If this can be done successfully, Algorithm *Naïve* is replaced by *Delta* [2].

The application of *Delta* can lead to significant performance advantages for fixed-point computations (or may enable their use on relational back-ends at all). Pathfinder's current implementation still depends on user hints in terms of the `with` $\cdots$ clause. This is certainly one situation where XQuery compilers could benefit from existing work on compilers for general-purpose programming languages.

The work in [40] notes that a common query pattern in XQuery is *structural recursion* along the XML document tree. The authors propose an analysis based on static type information to *unfold* finite instances of recursive invocations. The technique could be a way to extend the class of queries that can be handled in a purely relational XQuery setup. It does not bridge the expressivity gap per se, however,

8 Conclusions

This work gave a (biased) review of existing techniques to turn relational database back-ends into processors for XQuery. We have based our tour on the paths chosen in Pathfinder, an open-source implementation of a purely relational XQuery compiler.

Our review demonstrates that the relational approach to XQuery has come a long way since its first steps roughly a decade ago. Pathfinder is able to support a large subset of the XQuery specification at unprecedented speeds and with scalability far into the gigabyte range. Similar approaches have already found their way into commodity database software, such as the XQuery implementation in Microsoft SQL Server.

Yet, to solve the full puzzle, not all of the necessary pieces have been found so far. We feel that the most interesting question in relational XQuery processing is the role of pattern-based query evaluation in the XQuery picture (see Section 3.3). Some systems have come up with very efficient algorithms to answer queries formulated in a pattern style (and we have sketched one prominent representative). Unfortunately, due to significant semantical differences between XPath and pattern notations, these algorithms still remain largely inaccessible for compliant XQuery evaluation. The work in [28; 36] does a significant step toward closing that gap. The full piece, however, seems still missing to fit into the puzzle.

Section 7 hints at another aspect of XQuery processing that has not yet been fully explored. In many senses, XQuery is at the verge to a general-purpose programming language, as one can also tell from recent developments in the W3C Working Group [10]. Pathfinder and other XQuery processors already picked up a number of techniques originally designed for programming language compilers. But we feel that the general application of such techniques to query processing problems is still not sufficiently understood. Research on recent scripting additions to XQuery may bring up interesting synergies contributed from both research fields, databases and programming languages/compiler construction (and recursion is only one aspect).

References

1. Aboulnaga A, Alameldeen AR, Naughton JF (2001) Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In: Proc. of the 27th Int'l Conference on Very Large Databases (VLDB), Rome, Italy, pp 591–600
2. Afanasiev L, Grust T, Marx M, Rittinger J, Teubner J (2007) An Inflationary Fixed Point in XQuery. URL `http://arxiv.org/abs/0711.3375`
3. Afanasiev L, Grust T, Marx M, Rittinger J, Teubner J (2008) An Inflationary Fixed Point in XQuery. In: Proc. of the 24th Int'l Conference on Data Engineering (ICDE), Cancún, Mexico
4. Bayer R, Unterauer K (1977) Prefix B-Trees. ACM Transactions on Database Systems (TODS) 2(1):11–26
5. Boag S, Chamberlin D, Fernández MF, Florescu D, Robie J, Siméon J (2007) XQuery 1.0: An XML Query Language. World Wide Web Consortium Recommendation, URL `http://www.w3.org/TR/xquery/`
6. Boncz P, Grust T, van Keulen M, Manegold S, Rittinger J, Teubner J (2005) Loop-Lifted Staircase Join: From XPath to XQuery. Tech. Rep. INS-E0510, CWI, Amsterdam
7. Boncz P, Grust T, van Keulen M, Manegold S, Rittinger J, Teubner J (2006) 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
8. Boncz PA (2002) Monet: A Next-Generation DBMS Kernel for Query-Intensive Applications. PhD thesis, Universiteit van Amsterdam
9. Bruno N, Koudas N, Srivastava D (2002) Holistic Twig Joins: Optimal XML Pattern Matching. In: Proc. of the 2002 ACM SIGMOD Int'l Conference on Management of Data, Madison, WI, USA, pp 310–321
10. Chamberlin D, Engovato D, Florescu D, Ghelli G, Melton J, Siméon J (2008) XQuery Scripting Extension 1.0. W3C Working Draft, URL `http://www.w3.org/TR/xquery-xs-10/`
11. Cohen E, Kaplan H, Milo T (2002) Labeling Dynamic XML Trees. In: Proc. of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), Madison, WI, USA, pp 271–281
12. Draper D, Fankhauser P, Fernández M, Malhotra A, Rose K, Rys M, Siméon J, Wadler P (2007) XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Recommendation, URL `http://www.w3.org/TR/xquery-semantics/`

13. Fernández MF, Malhotra A, Marsh J, Nagy M, Walsh N (2007) XQuery 1.0 and XPath 2.0 Data Model (XDM). W3C Recommendation, URL `http://www.w3.org/TR/xpath-datamodel/`
14. Florescu D, Kossmann D (1999) Storing and Querying XML Data Using an RDBMS. IEEE Data Engineering Bulletin 22(3):27–34
15. Galax (2008) Galax: An Implementation of XQuery. URL `http://www.galaxquery.org/`
16. Galindo-Legaria CA, Joshi MM (2001) Orthogonal Optimization of Subqueries and Aggregation. In: Proc. of the 2001 ACM SIGMOD Int'l Conference on Management of Data, Santa Barbara, CA, USA, pp 571–581
17. Goldman R, Widom J (1997) DataGuides: Enabling Query Formulation and Optimization. In: Proc. of the 23rd Int'l Conference on Very Large Databases (VLDB), Athens, Greece, pp 436–445
18. Graefe G (2003) Sorting and Indexing with Partitioned B-Trees. In: Proc. of the 1st Int'l Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA
19. Grust T (2002) Accelerating XPath Location Steps. In: Proc. of the 2002 ACM SIGMOD Int'l Conference on Management of Data, Madison, WI, USA, pp 109–120
20. Grust T, Teubner J (2004) Relational Algebra: Mother Tongue—XQuery: Fluent. In: Proc. of the 1st Twente Data Management Workshop (TDM), Enschede, The Netherlands, pp 7–14
21. Grust T, van Keulen M, Teubner J (2003) Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In: Proc. of the 29th Int'l Conference on Very Large Databases (VLDB), Berlin, Germany, pp 524–535
22. Grust T, van Keulen M, Teubner J (2004) Accelerating XPath evaluation in any RDBMS. ACM Transactions on Database Systems (TODS) 29(1):91–131
23. Grust T, Sakr S, Teubner J (2004) XQuery on SQL Hosts. In: Proc. of the 30th Int'l Conference on Very Large Databases (VLDB), Toronto, Canada, pp 252–263
24. Grust T, Rittinger J, Teubner J (2007) eXrQuy: Order Indifference in XQuery. In: Proc. of the 23rd Int'l Conference on Data Engineering (ICDE), Istanbul, Turkey
25. Grust T, Rittinger J, Teubner J (2007) Why Off-The-Shelf RDBMSs are Better at XPath Than You Might Expect. In: Proc. of the 2007 ACM SIGMOD Int'l Conference on Management of Data, Beijing, China
26. Grust T, Mayr M, Rittinger J (2009) XQuery Join Graph Isolation. In: Proc. of the 25th International Conference on Data Engineering (ICDE), Shanghai, China
27. Härder T, Haustein M, Mathis C, Wagner M (2007) Node Labeling Schemes for Dynamic XML Documents Reconsidered. Data and Knowledge Engineering 6(1):126–149
28. Hidders J, Michiels P, Siméon J, Vercammen R (2007) How to Recognise Different Kinds of Tree Patterns From Quite a Long Way Away. In: Proc. of the 2007 Workshop on Programming Language Technologies for XML (PLAN-X), Nice, France
29. Josifovski V, Fontoura M, Barta A (2005) Querying XML Streams. The VLDB Journal 14(2):197–210
30. Kay M (2008) The Saxon XSLT and XQuery Processor. URL `http://saxon.sf.net/`
31. Kepser S (2004) A Simple Proof for the Turing-Completeness of XSLT and XQuery. In: Proc. of the Extreme Markup Languages 2004, Montréal, Quebec, Canada
32. Li H, Lee ML, Hsu W, Cong G (2006) An Estimation System for XPath Expressions. In: Proc. of the 22nd Int'l Conference on Data Engineering (ICDE), Atlanta, GA, USA
33. Marian A, Siméon J (2003) Projecting XML Documents. In: Proc. of the 29th Int'l Conference on Very Large Databases (VLDB), Berlin, Germany
34. Mayer S, Grust T, van Keulen M, Teubner J (2004) An Injection with Tree Awareness: Adding Staircase Join to PostgreSQL. In: Proc. of the 30th Int'l Conference on Very Large Databases (VLDB), Toronto, Canada, pp 1305–1308
35. McHugh J, Widom J (1999) Query Optimization for XML. In: Proc. of the 25th Int'l Conference on Very Large Databases (VLDB), Edinburgh, Scotland, UK, pp 315–326
36. Michiels P, Mihaila GA, Siméon J (2007) Put a Tree Pattern in Your Algebra. In: Proc. of the 23rd Int'l Conference on Data Engineering (ICDE), pp 246–255
37. Olteanu D, Meuss H, Furche T, Bry F (2002) XPath: Looking Forward. In: XML-Based Data Management and Multimedia Engineering, EDBT 2002 Workshops, Revised Papers, Prague, Czech Republic, pp 109–127
38. O'Neil PE, O'Neil EJ, Pal S, Cseri I, Schaller G, Westbury N (2004) ORDPATHs: Insert-Friendly XML Node Labels. In: Proc. of the 2004 ACM SIGMOD Int'l Conference on Management of Data, Paris, France, pp 903–908
39. Pal S, Cseri I, Seeliger O, Rys M, Schaller G, Yu W, Tomic D, Baras A, Berg B, Churin D, Kogan E (2005) XQuery Implementation in a Relational Database System. In: Proc. of the 31st Int'l Conference on Very Large Databases (VLDB), Trondheim, Norway, pp 1175–1186
40. Park CW, Min JK, Chung CW (2002) Structural Function Inlining Technique for Structurally Recursive XML Queries. In: Proc. of the 28th Int'l Conference on Very Large Databases (VLDB), Hong Kong, China
41. Polyzotis N, Garofalakis MN (2006) XSKETCH Synopses for XML Data Graphs. ACM Transactions on Database Systems (TODS) 31(3):1014–1063
42. Schmidt AR, Waas F, Kersten ML, Carey MJ, Manolescu I, Busse R (2002) XMark: A Benchmark for XML Data Management. In: Proc. of the 28th Int'l Conference on Very Large Databases (VLDB), Hong Kong, China, pp 974–985
43. kx Systems (2008) The kdb+ Database. URL `http://www.kx.com/`
44. Teubner J (2008) Scalable XQuery Type Matching. In: Proc. 11th Int'l Conference on Extending Database Technology (EDBT), Nantes, France
45. Teubner J, Grust T, Maneth S, Sakr S (2008) Dependable Cardinality Forecasts for XQuery. In: Proc. of the 34th Int'l Conference on Very Large Databases (VLDB)
46. The Pathfinder XQuery Compiler (2001–2008) URL `http://www.pathfinder-xquery.org/`