First publ. in: Proc. of the first Twente Data Management Workshop on XML Databases, Enschede, The Netherlands, 2004

Relational Algebra: Mother Tongue—XQuery: Fluent

Torsten Grust Jens Teubner
University of Konstanz
Department of Computer & Information Science
Box D 188, 78457 Konstanz, Germany
{grust,teubner}@inf.uni-konstanz.de

ABSTRACT

This work may be seen as a further proof of the versatility of the relational database model. Here, we add XQuery to the catalog of languages which RDBMSs are able to "speak" fluently.

Given suitable relational encodings of sequences and ordered, unranked trees—the two data structures that form the backbone of the XML and XQuery data models—we describe a compiler that translates XQuery expressions into a simple and quite standard relational algebra which we expect to be efficiently implementable on top of any relational query engine. The compilation procedure is fully compositional and emits algebraic code that strictly adheres to the XQuery language semantics: document and sequence order as well as node identity are obeyed. We exercise special care in translating arbitrarily nested XQuery `FLWOR` iteration constructs into equi-joins, an operation which RDBMSs can perform particularly fast. The resulting purely relational XQuery processor shows promising performance figures in experiments.

Keywords

XQuery, XML Query Processing, Relational Algebra

1. INTRODUCTION

Relational database back-ends have had a tremendous success over the past years. Their underlying data model, *tables of tuples*, is simple and thus efficient to implement. Typical operations, such as sequential scans, receive excellent support through read-ahead on disk-based secondary storage, or memory prefetching on modern computing hardware. If linear access is not viable, systems can rely on access structures, such as B^+-trees or hash tables. The *bulk-oriented* fashion, in which queries are described and processed, allows for effective query rewriting or parallel processing.

At the same time, the table proves to be a rather generic data structure: it is often straightforward to map other data types onto tables. Such encodings have also been proposed for *ordered, unranked trees*, the data type that forms the backbone of the XML data model. These mappings turn RDBMSs into *relational XML processors*. Furthermore, if the tree encoding is designed such that core operations on trees—XPath axis traversals—lead to efficient table operations, this can result in high-performance *relational XPath* implementations [8, 10].

In this work we extend the relational XML processing stack and propose the fully relational evaluation of XQuery [1] expressions. We give a compositional set of translation rules that compile XQuery expressions into a standard, quite primitive relational algebra. We expect any relational query engine to be able to efficiently implement the operators of this algebra. The operators were, in fact, designed to match the capabilities of modern SQL-based relational database systems (*e.g.*, the row numbering operator ϱ exactly mirrors SQL:1999 OLAP ranking functionality) [9].

By design, we only have minimalistic assumptions on the underlying tree encoding, met by several XML encoding schemes [4, 13]. Our algebra can be easily modified to operate with any such scheme.

We exercise special care in translating the XQuery `FLWOR` construct (`for` $v `in` e_1 `return` e_2). This concept of iterating the evaluation of an expression e_2 for successive bindings of a variable $v appears contrary to the set-oriented evaluation model of relational systems. In a nutshell, we thus map `for`-bound variables like $v into tables containing all bindings and translate expressions in dependence of the variable scope in which they appear. Iteration is turned into equi-joins, a table operation which RDBMS engine know how to execute most efficiently.

2. ENCODING TREES AND SEQUENCES

The dynamic evaluation phase of XQuery operates with data of two principal types: *nodes* and *atomic values* (collectively referred to as `item`-typed data). Nodes may be assembled into *ordered, unranked trees*, *i.e.*, instances of XML documents or fragments thereof. Nodes and atomic values may form *ordered, finite sequences*. We will now briefly review minimalistic relational encodings of trees as well as sequences. Both encodings exhibit just those properties necessary to support a semantically correct and efficient relational evaluation.

2.1 Trees and XPath Support

Our compilation system is designed to be adaptable to any relational tree encoding with minimalistic requirements: the

TDM'04, the first Twente Data Management Workshop on XML Databases and Information Retrieval, Enschede, The Netherlands

Axis α	Predicate $axis(c, v, \alpha)$: $v \overset{?}{\in} c/\alpha$
`descendant`	$v.pre > c.pre \wedge v.pre \leqslant c.pre + c.size$
`child`	$axis(c, v, \texttt{descendant}) \wedge v.level = c.level + 1$
`following`	$v.pre > c.pre + c.size$
`preceding`	$v.pre + v.size < c.pre$

Table 1: Predicate *axis* represents XPath axes semantics (selected axes).

encoding must support XPath step evaluation from any context node and provide a means to test for *node identity* and *document order*. These requirements are met by a number of relational XML encodings, including the numbering schemes developed in [4, 13]. We briefly sketch a suitable encoding in the sequel.

To represent node identity and document order, we assign to each node v its unique *preorder traversal rank*, $v.pre$ [8]. Extending this information by (1) $v.size$, the number of nodes in the subtree below v, and (2) $v.level$, the length of the path from the tree root to v, we can express the semantics of all 13 XPath axes—and thus support XQuery's *full axis* feature—via simple conjunctive predicates. To illustrate, for the `ancestor` axis and two nodes v and c, we have that

$$v \in c/\texttt{ancestor} \Leftrightarrow$$
$$v.pre < c.pre \ \wedge \ c.pre \leqslant v.pre + v.size \ .$$

More axes are listed in Table 1. Note that we do not require $v.size$ to be exact: as long as the XPath axis semantics are obeyed, $v.size$ may overestimate the actual number of nodes below v. Via the *pre* property we are able to ensure that the node sequence resulting from an axis step is free of duplicates and sorted in document order as required by the XPath semantics.

Support for *kind* and *name tests* is added by means of two further properties, $v.kind \in \{\texttt{"elem"}, \texttt{"text"}\}$[1] and $v.prop$. For an element node v with tag name `t`, we have $v.prop =$ `"t"`, for a text node v' with content `c`, $v'.prop =$ `"c"`.

XQuery is not limited to query single XML documents. In general, query evaluation involves nodes from multiple documents or fragments thereof, possibly created at runtime via XQuery's element constructors. The query

```
(element a { element b { () }}, element c { () })
```

creates three element nodes in two independent fragments, for example. We thus record a fragment identifier for node v in its $v.frag$ property.

The database system keeps a table `doc` of persistently stored XML documents. Transient nodes constructed at runtime, on the other hand, are represented by means of a term Δ of the relational algebra—this term is derived during query compilation. The *disjoint union* of both relations, $\texttt{doc} \dot\cup \Delta$, comprises the set of *live nodes* at any point of query evaluation. The relational encoding of two XML fragments is depicted in Figure 1.

2.2 Sequences

XQuery expressions evaluate to ordered, finite sequences of `item`s. Since sequences are flat and cannot be nested, a sequence may be represented by a single relation in which each tuple encodes a sequence item i. We preserve *sequence order* by means of a property $i.pos \geqslant 1$. The actual value of a sequence item is recorded in $i.item$, which is one of (1) a node's *pre* value if this item is a node, or (2) the actual value if the item is an atomic value. The relational representation of the sequence `("a", "b", "c")` is shown in Figure 2. In the course of this work, we assume the *item* column to be of polymorphic type: such a column may carry node identifiers, character strings, numeric values, as well as any other atomic XQuery `item`. The empty relation encodes the empty sequence `()`. A single item i and the singleton sequence `(`i`)` are represented identically, which coincides with the XQuery semantics. Note that XQuery's positional predicates e`[`p`]`, $p \geqslant 1$, are easily evaluated if the *pos* column is populated *densely* starting at 1 as is the case in Figure 2.

pos	*item*
1	`"a"`
2	`"b"`
3	`"c"`

Fig. 2: Relational sequence encoding.

3. TURNING ITERATION INTO JOINS

The core of the XQuery language, with syntactic sugar like path expressions, quantifiers, or sequence comparison operators removed, has been designed around an *iteration* primitive, the `for-return` construct. A `for`-loop iterates the evaluation of loop body e for successive bindings of the loop variable `$`v:

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

where $e[x/\texttt{\$}v]$ denotes the consistent replacement of all free occurrences of `$`v in e by x. XQuery provides a functional style of iteration: it is semantically sound to evaluate e for all n bindings of `$`v in parallel.

3.1 Loop Lifting for Constant Subexpressions

This property of XQuery inspires our loop compilation strategy:

(1) A loop of n iterations is represented by a relation loop with a single column *iter* of n values $1, 2, \ldots, n$.
(2) If a constant subexpression c occurs inside a loop body e, the relational representation of c is *lifted* (intuitively, this accounts for the n independent evaluations of e).

For a constant atomic value c, lifting with respect to a given loop relation is computed by means of the Cartesian product

$$\text{loop} \times \begin{array}{c|c} pos & item \\ \hline 1 & c \end{array} \ .$$

Figure 3(a) exemplifies how the constant subexpression `10` is lifted with respect to the loop

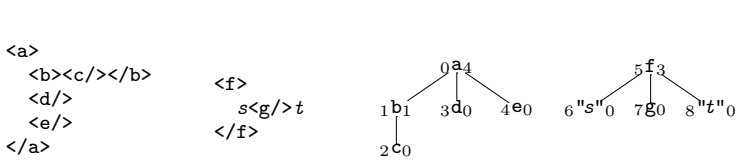
```
for $v0 in (1,2,3) return 10 .
```

If, for example, `10` is replaced by the sequence `(10,20)` in this loop, we require the lifting result to be the relation of Figure 3(b) instead.

Generally, a tuple (i, p, v) in a loop-lifted relation for subexpression e may be read as the assertion that, during the ith iteration, the item at position p in e has value v. With this in mind, suppose we rewrite the `for`-loop as

```
for $v0 in (1,2,3) return (10,$v0) .        (Q1)
```

[1] We omit the discussion of further XML node kinds for space reasons.



pre	*size*	*level*	*kind*	*prop*	*frag*
0	4	0	"elem"	"a"	0
1	1	1	"elem"	"b"	0
2	0	2	"elem"	"c"	0
3	0	1	"elem"	"d"	0
4	0	1	"elem"	"e"	0
5	3	0	"elem"	"f"	1
6	0	1	"text"	"s"	1
7	0	1	"elem"	"g"	1
8	0	1	"text"	"t"	1

(a) Two XML fragments. (b) Fragment trees. (c) Tree encoding (table doc).

Figure 1: Relational encoding of two XML fragments. Nodes in the fragment trees (b) have been annotated with their *pre* and *size* properties. Both trees are encoded as independent fragments 0 and 1 in (c).

$e ::= c$	atomic constants
$\mid$ $v	variables
$\mid$ (e,e)	sequence construction
$\mid$ $e/\alpha::n$	loc. step (axis α, node test n)
$\mid$ element t {e}	element constructor (tag t)
$\mid$ for $v in e return e	iteration
$\mid$ let $v := e return e	let binding
$\mid$ e+e	addition

Figure 4: Syntax of XQuery Core subset.

Consistent with the loop lifting scheme, the database system will represent variable v_0 as the relation shown in Figure 3(c), *e.g.*, in the second iteration ($iter = 2$), v_0 is bound to the item 2. We will shortly see how we can derive this representation of a variable from the representation of its domain (in this case the sequence (1,2,3)).

Finally, to evaluate the query Q_1, the system solely operates with the loop-lifted relations to compute the result shown in Figure 3(d).

3.2 An Algebra for XQuery

As a language with variables, XQuery demands a third piece of information (despite the relations Δ and loop) for compilation: the *environment* Γ maps all free variables in XQuery expression e to their relational representation (again, an algebraic expression).

We thus define the XQuery compiler in terms of a set of inference rules, in which a judgment of the form

$$\Gamma; \mathsf{loop}; \Delta \vdash e \Rightarrow (q, \Delta')$$

indicates that, given Γ, loop, and Δ, the XQuery expression e compiles into the algebraic expression q with a new table of transient nodes Δ'. New nodes are created by XQuery's element constructors only, otherwise $\Delta' = \Delta$.

Compilation starts with the top-level expression, an empty environment $\Gamma = \emptyset$, a singleton loop relation ($\mathsf{loop} = \frac{iter}{1}$) indicating that the top-level expression is not embedded into a loop, and an empty relation Δ. All inference rules pass Γ, loop, and Δ top-down, while the resulting algebra expression is synthesized bottom-up. The compiler produces a single algebra query that operates on the tree and sequence encodings sketched in Section 2.

This paper contains inference rules to compile a subset of XQuery Core defined by the grammar in Figure 4. This subset, plus a few extensions, suffices to express the XMark benchmark query set [16], for example.[2]

[2]In fact, the subset may be extended to embrace the complete XQuery Core language. The implementation of XQuery's dynamic typing and validation features, however, requires further support from the underlying tree and sequence encoding.

$\pi_{a_1:b_1,\ldots,a_n:b_n}$	projection (and renaming)
σ_a	selection
$\dot\cup$	disjoint union
$\times$	cartesian product
$\bowtie_{a=b}$	equi-join
$\varrho_{b:\langle a_1,\ldots,a_n\rangle/p}$	row numbering
$\text{⍁}_{\alpha,n}$	XPath axis join (axis α, node test n)
ε	element construction
$\circledast_{b:\langle a_1,\ldots,a_n\rangle}$	n-ary arithmetic/comparison operator $*$
$\underline{a \downarrow b}$	literal table

Figure 5: Operators of the relational algebra (a, b column names).

The compiler's target language is a relational algebra with operators lined up in Figure 5. Most of the operators are rather standard, or even restricted, variants of the operators found in a classical relational algebra. It is sufficient for $\bowtie$, *e.g.*, to evaluate equality join predicates. The selection σ_a selects those tuples with column $a \neq 0$. Operator $\circledast_{b:\langle a_1,\ldots,a_n\rangle}$ applies the n-ary operator $*$ to columns $a_1, \ldots, a_n$ and extends the input tuples with the result column b.

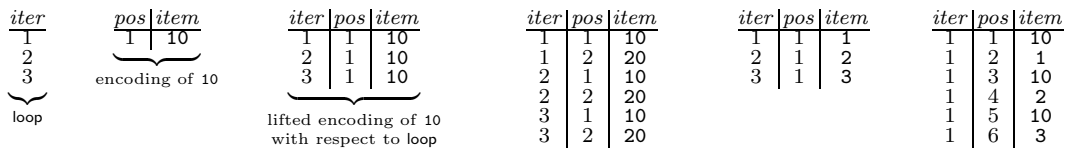
We write $sch(q)$ to denote the column schema of algebraic expression q; $+\!\!+$ concatenates column schemas. Thus, $sch(\pi_{a_1:b_1,\ldots,a_n:b_n}(q)) = \underline{a_1 \downarrow \ldots \downarrow a_n}$, $sch(\circledast_{b:\langle a_1,\ldots,a_n\rangle}(q)) = sch(q) +\!\!+ \underline{b}$ and $sch(q_1 \times q_2) = sch(q_1 \bowtie_{a=b} q_2) = sch(q_1) +\!\!+ sch(q_2)$, for example.

To encapsulate the underlying tree encoding, we extend the algebra by the operators $\text{⍁}_{\alpha,n}$ to evaluate XPath steps, and ε to construct new transient nodes.

$q \text{⍁}_{\alpha,n} \mathsf{doc}$ returns the result of the evaluation of the XPath step $\alpha::n$ originating in the context nodes returned by q. ⍁ does so for each iteration encoded in q, thus $sch(q_1 \text{⍁}_{\alpha,n} q_2) = \underline{iter \downarrow item}$. A highly efficient implementation of $\text{⍁}_{\alpha,n}$, the *staircase join*, has been presented in [10].

Given the existing set of live nodes, $\mathsf{doc} \dot\cup \Delta$, a set of tag names q_t and a sequence of nodes q_e, $\varepsilon(\mathsf{doc}\dot\cup\Delta, q_t, q_e)$ returns the new transient nodes resulting from the XQuery expression element t { e }, along with their originating iteration $iter$; $sch(\varepsilon(q_1, q_2, q_3)) = sch(\mathsf{doc}) +\!\!+ \underline{iter}$. An implementation for the sample document encoding scheme introduced in Section 2.1 is sketched in Section 5.3.

With order being an inherent concept of the XQuery data model as well as our compilation scheme, we make frequent use of the *numbering* operator ϱ. Given a sort order defined



(a) Lifting the constant `10`. (b) Loop-lifted sequence. (c) Encoding of variable $\$v_0$. (d) Result of query Q_1.

Figure 3: Loop lifting.

by columns $a_1, \ldots, a_n$, $\varrho_{b:\langle a_1,\ldots,a_n\rangle/p}(q)$ numbers consecutive tuples in q, recording the row number in the new column b. Row numbers start from 1 in each partition defined by the optional grouping column p. Many RDBMSs readily provide a ϱ operator, for example by means of the `DENSE_RANK` operator defined by SQL:1999 [15]. A database host operating on *ordered* relations may even provide such numbering for free (*cf.* the `void` columns in the MonetDB RDBMS [2]).

4. RELATIONAL FLWORS

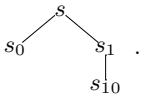
We will now generalize our loop-lifting idea and give a translation for arbitrarily nested `for`-loops.

Assume an expression with three nested `for`-loops as shown here:

$$s\left\{\begin{array}{l}\texttt{(for } \$v_0 \texttt{ in } e_0 \texttt{ return}\\ \quad s_0\,\{\ e'_0\ \ \texttt{,}\\ \quad \texttt{for } \$v_1 \texttt{ in } e_1 \texttt{ return}\\ \quad\quad s_1\left\{\begin{array}{l}\texttt{for } \$v_{10} \texttt{ in } e_{10} \texttt{ return}\\ s_{10}\,\{\ e'_{10}\end{array}\right.\\ \texttt{)}\end{array}\right.$$

The curly braces visualize the *variable scopes* in this query: variable $\$v_0$ is visible in scope s_0, variable $\$v_1$ is visible in scopes s_1 and s_{10}, while variable $\$v_{10}$ is accessible in scope s_{10} only. No variables are bound in top-level scope s. (In the context of this section, only `for` expressions are considered to open a new scope; `let` expressions are treated in Section 5.2.)

Note that the compositionality and scoping rules of XQuery, in general, lead to a tree-shaped hierarchy of scopes. For the above query, we obtain



.

In the following, we write $s_{x\cdot y}$, $x \in \{0,1,\ldots\}^*, y \in \{0,1,\ldots\}$ to identify the yth child scope of scope s_x. Furthermore, let $q_x(e)$ denote the representation of expression e in scope s_x.

Bound variables. Consider a `for`-loop in its directly enclosing scope s_x:

$$s_x\left\{\begin{array}{l}\vdots\\ \texttt{for } \$v_{x\cdot y} \texttt{ in } e_{x\cdot y} \texttt{ return}\\ \quad s_{x\cdot y}\,\{\ e'_{x\cdot y}\\ \vdots\end{array}\right.$$

According to the XQuery semantics, $e_{x\cdot y}$ is evaluated in scope s_x. Variable $\$v_{x\cdot y}$ is then successively bound to each single item in the resulting sequence; these bindings are used in the evaluation of $e'_{x\cdot y}$ in scope $s_{x\cdot y}$. A suitable representation for $\$v_{x\cdot y}$ in scope $s_{x\cdot y}$ may thus be computed if we retain the values of $q_x(e_{x\cdot y})$ (to which $\$v_{x\cdot y}$ will be bound consecutively) , but assign a new *iter* property with consecutive numbers and a constant *pos* value of 1:

$$q_{x\cdot y}(\$v_{x\cdot y}) = \frac{pos}{1} \times \pi_{iter:inner,item}\big(\varrho_{inner:\langle iter,pos\rangle}(q_x(e_{x\cdot y}))\big)$$

This is exactly how we obtained the representation of variable $\$v_0$ in query Q_1 (see Figure 3(c)):

$$q_0(\$v_0) = \frac{pos}{1} \times \pi_{iter:inner,item}\big(\varrho_{inner:\langle iter,pos\rangle} q(\texttt{(1,2,3)})\big)$$

where $q(\texttt{(1,2,3)})$ simply is the relational encoding of the sequence `(1,2,3)` as introduced in Section 2.2.

Maintaining loop. The concept of loop-lifting requires the maintenance of a loop relation of independent iterations. The body of a `for`-loop in scope $s_{x\cdot y}$ needs to be evaluated once for each binding of the `for`-bound variable $\$v_{x\cdot y}$. To compile the subexpressions comprising this body, we thus define a new loop relation based on $q_{x\cdot y}(\$v_{x\cdot y})$:

$$\mathsf{loop}_{x\cdot y} = \pi_{iter}(q_{x\cdot y}(\$v_{x\cdot y}))\quad .$$

Constants. The compilation of an atomic constant c is now achieved through loop lifting as motivated in Section 3.1. The associated inference (or compilation) rule Const reads

$$\frac{}{\Gamma;\mathsf{loop};\Delta \vdash c \Rightarrow \left(\mathsf{loop} \times \frac{pos\,|\,item}{1\,|\,c}, \Delta\right)}\ . \qquad \text{(Const)}$$

Note how atomic constants do not affect the set of transient nodes Δ.

Free variables. In XQuery, an expression e may refer to variables which have been bound in an enclosing scope: a variable bound in scope s_x is also visible in any scope $s_{x\cdot x'}$, $x' \in \{0,1,\ldots\}^+$. If scope $s_{x\cdot x'}$ is viewed in isolation, such variables appear to be free.

The compiled representation $q_{x\cdot y}(\$v_x)$ of a free variable v_x in scope $s_{x\cdot y}$ depends on the value of the loop relation in $s_{x\cdot y}$, and we will now derive $q_{x\cdot y}(\$v_x)$ from the representation in the directly enclosing scope s_x. To understand the derivation, consider the evaluation of two nested `for`-loops (note the reference to $\$v_0$ in the inner scope $s_{0\cdot 0}$):

$$s\left\{\begin{array}{l}\texttt{for } \$v_0 \texttt{ in (1,2) return}\\ \quad s_0\left\{\begin{array}{l}\texttt{(} \$v_0\texttt{,}\\ \quad \texttt{for } \$v_{0\cdot 0} \texttt{ in (10,20) return}\\ \quad\quad s_{0\cdot 0}\,\{\ (\$v_0\texttt{,}\$v_{0\cdot 0})\\ \texttt{)}\end{array}\right.\end{array}\right. \qquad (Q_2)$$

iter	pos	item
1	1	"1"
2	1	"2"

(a) $q_0(\$v_0)$

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

(b) $q_{0\cdot0}(\$v_0)$

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

(c) $q_{0\cdot0}(\$v_{0\cdot0})$

Figure 6: Q_2: Scope-dependent representation of variables.

In the first outer iteration, $\$v_0$ is bound to `1`. With this binding, two evaluations of the innermost loop body occur, each with a new binding for $\$v_{0\cdot0}$. Then, during the next outer iteration, two further evaluations of the innermost loop body occur with $\$v_0$ bound to `2` (Figure 6).

outer	inner
1	1
1	2
2	3
2	4

Fig. 7: $\mathsf{map}_{(0,0\cdot0)}$.

The semantics of this nested iteration may be captured by a relation $\mathsf{map}_{(0,0\cdot0)}$ shown in Figure 7 ($\mathsf{map}_{(x,x\cdot y)}$ will be used to map representations between scopes s_x and $s_{x\cdot y}$). A tuple (o, i) in this relation indicates that, during the ith iteration of the inner loop body in scope $s_{0\cdot0}$, the outer loop body in scope s_0 is in its oth iteration. This is the connection we need to derive the representation of a free variable $\$v_x$ in scope $s_{x\cdot y}$ via the following equi-join:

$$q_{x\cdot y}(\$v_x) = \pi_{iter:inner,pos,item}\left(q_x(\$v_x) \bowtie_{iter=outer} \mathsf{map}_{(x,x\cdot y)}\right)$$

If we insert the binding $\$v \mapsto q_{x\cdot y}(\$v)$ into the variable environment Γ, a reference to variable $\$v$ simply compiles to a lookup in Γ:

$$\frac{}{\{\ldots, \$v \mapsto q_v, \ldots\}\,;\mathsf{loop};\Delta \vdash \$v \Rrightarrow (q_v, \Delta)}\ . \qquad (\textsc{Var})$$

Note that relation $\mathsf{map}_{(x,x\cdot y)}$ is easily derived from the representation of the domain $e_{x\cdot y}$ of variable $\$v_{x\cdot y}$ (much like the representation of $\$v_{x\cdot y}$ itself):

$$\mathsf{map}_{(x,x\cdot y)} = \pi_{outer:iter,inner}\left(\varrho_{inner:\langle iter,pos\rangle}(q_x(e_{x\cdot y}))\right)\ .$$

Figure 6 contains a line-up of the relational variable representations involved in evaluating query Q_2. Note how the relations in Figures 6(b) and 6(c) represent the fact that, for example, in iteration 3 of the inner loop body variable $\$v_0$ is bound to `2` while $\$v_{0\cdot0}$ is bound to `10`, as desired.

Mapping back. The intermediate result computed by the inner loop of Q_2 is shown in Figure 9(a). To use this result in scope s_0 (as is required due to the sequence construction in line 2 of Q_2), we need to map its representation back into s_0. This back-mapping from scope $s_{x\cdot y}$ into the parent scope s_x may, again, be achieved via an equi-join with $\mathsf{map}_{(x,x\cdot y)}$. The required join forms the compilation result of compilation rule For (Figure 8). The rule also ensures that the correct loop relation and variable expressions are available when an expression is compiled.

Figure 9(b) depicts the inner loop body result after it has been mapped back into scope s_0. Sequence construction (Rule Seq, Section 5.1) and a second back-mapping step (from scope s_0 into the top-level scope s via $\mathsf{map}_{(\,,0)}$) produces the final result of Q_2 (Figure 9(c)).

5. OTHER EXPRESSION TYPES

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

(a) Intermediate result in $s_{0\cdot0}$.

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

(b) Intermediate result in s_0.

iter	pos	item
1	1	"1"
1	2	"1"
1	3	"10"
1	4	"1"
1	5	"20"
1	6	"2"
1	7	"2"
1	8	"10"
1	9	"2"
1	10	"20"

(c) Final result in top-level scope.

Figure 9: Q_2: Intermediate and final results.

iter	pos	item
1	1	"1"
2	1	"10"
2	2	"20"

(a) Encoding q_1 of e_1.

iter	pos	item
1	1	"2"
2	1	"30"

(b) Encoding q_2 of e_2.

iter	pos	item
1	1	"1"
1	2	"2"
2	1	"10"
2	2	"20"
2	3	"30"

(c) Encoded result of `(`e_1`,`e_2`)`.

Figure 10: Sequence construction. The dashed lines separate the represented iterations (*iter* partitions).

5.1 Sequence Construction

Essentially, Rule Seq (Figure 8) compiles the sequence construction `(`e_1`,`e_2`)` into a disjoint union of the relational encodings q_1 and q_2 of e_1 and e_2. Correct ordering is ensured by temporarily adding a column *ord* to q_1 and q_2 and a subsequent renumbering of the result via ϱ. Note that this evaluates the sequence construction for *all* iterations encoded in q_1, q_2 at once. Figure 10 exemplifies the operation of the compiled algebraic expression. Relation q_1 encodes two sequences: `(1)` in iteration 1 and `(10,20)` in iteration 2, while q_2 encodes `(2)` in iteration 1 and `(30)` in iteration 2. The algebraic expression generated by Rule Seq computes the result in Figure 10(c): the sequence construction evaluates to `(1,2)` in iteration 1 and `(10,20,30)` in iteration 2, as expected.

5.2 Variable Binding/Usage

Variables are handled in a standard fashion: to compile `let` $\$v$ `:=` e_1 `return` e_2, translate e_1 in environment Γ to yield the expression q_1, then compile e_2 in the enriched environment $\Gamma + \{\$v \mapsto q_1\}$:

$$\frac{\begin{array}{l}\Gamma;\mathsf{loop};\Delta \vdash e_1 \Rrightarrow (q_1, \Delta_1)\\ \Gamma + \{\$v \mapsto q_1\}\,;\mathsf{loop};\Delta_1 \vdash e_2 \Rrightarrow (q_2, \Delta_2)\end{array}}{\Gamma;\mathsf{loop};\Delta \vdash \texttt{let}\ \$v\ \texttt{:=}\ e_1\ \texttt{return}\ e_2 \Rrightarrow (q_2, \Delta_2)}\ . \qquad (\textsc{Let})$$

A reference to $\$v$ in e_2 then yields q_1 via Rule Var.

5.3 Element Construction

The relation Δ of transient nodes is populated by the XQuery element construction operator `element` t `{` e `}`, in which subexpression e is required to evaluate to a sequence of nodes `(`v_1`,`v_2`,...,`v_k`)`. To comply with XQuery semantics, the k subtrees rooted at the nodes v_i are copied to the relation Δ of transient nodes. A new node r with tag name t is then added to Δ and made the common root of the subtree copies; r is then returned as the overall query result.

$$\frac{\{\ldots, \$v_i \mapsto q_{v_i}, \ldots\}; \mathsf{loop}; \Delta \vdash e_1 \Rightarrow (q_1, \Delta_1) \qquad q_v \equiv \frac{pos}{1} \times \pi_{iter:inner,item}\left(\varrho_{inner:\langle iter,pos\rangle} q_1\right)}{\mathsf{loop}_v \equiv \pi_{iter} q_v \qquad \mathsf{map} \equiv \pi_{outer:iter,inner}\left(\varrho_{inner:\langle iter,pos\rangle} q_1\right)}$$
$$\frac{\{\ldots, \$v_i \mapsto \pi_{iter:inner,pos,item}\left(q_{v_i} \bowtie_{iter=outer} \mathsf{map}\right), \ldots\} + \{\$v \mapsto q_v\}; \mathsf{loop}_v; \Delta_1 \vdash e_2 \Rightarrow (q_2, \Delta_2)}{\{\ldots, \$v_i \mapsto q_{v_i}, \ldots\}; \mathsf{loop}; \Delta \vdash \texttt{for } \$v \texttt{ in } e_1 \texttt{ return } e_2 \Rightarrow \left(\pi_{iter:outer,pos:pos_1,item}\left(\varrho_{pos_1:\langle iter,pos\rangle/outer}\left(q_2 \bowtie_{iter=inner} \mathsf{map}\right)\right), \Delta_2\right)} \quad \text{(For)}$$

$$\frac{\Gamma; \mathsf{loop}; \Delta \vdash e_1 \Rightarrow (q_1, \Delta_1) \qquad \Gamma; \mathsf{loop}; \Delta_1 \vdash e_2 \Rightarrow (q_2, \Delta_2)}{\Gamma; \mathsf{loop}; \Delta \vdash (e_1, e_2) \Rightarrow \left(\pi_{iter,pos:pos_1,item}\left(\varrho_{pos_1:\langle ord,pos\rangle/iter}\left(\left(\frac{ord}{1} \times q_1\right) \dot\cup \left(\frac{ord}{2} \times q_2\right)\right)\right), \Delta_2\right)} \quad \text{(Seq)}$$

$$\frac{\Gamma; \mathsf{loop}; \Delta \vdash e \Rightarrow (q_e, \Delta_1)}{\Gamma; \mathsf{loop}; \Delta \vdash e/\alpha::n \Rightarrow \left(\varrho_{pos:\langle item\rangle/iter}\left(\left(\pi_{iter,item} q_e\right) \text{◿}_{\alpha,n} \left(\mathsf{doc} \dot\cup \Delta_1\right)\right), \Delta_1\right)} \quad \text{(Step)}$$

$$\frac{\Gamma; \mathsf{loop}; \Delta \vdash e_1 \Rightarrow (q_1, \Delta_1) \qquad \Gamma; \mathsf{loop}; \Delta_1 \vdash e_2 \Rightarrow (q_2, \Delta_2) \qquad n \equiv \varepsilon(\mathsf{doc} \dot\cup \Delta_2, q_1, q_2)}{\Gamma; \mathsf{loop}; \Delta \vdash \texttt{element } e_1 \texttt{ \{ } e_2 \texttt{ \}} \Rightarrow \left(\pi_{iter,item:pre}\left(roots(n)\right) \times \frac{pos}{1}, \Delta_2 \dot\cup \pi_{sch(\mathsf{doc})} n\right)} \quad \text{(Elem)}$$

$$\frac{\Gamma; \mathsf{loop}; \Delta \vdash e_1 \Rightarrow (q_1, \Delta_1) \qquad \Gamma; \mathsf{loop}; \Delta_1 \vdash e_2 \Rightarrow (q_2, \Delta_2)}{\Gamma; \mathsf{loop}; \Delta \vdash e_1 \texttt{ + } e_2 \Rightarrow \left(\pi_{iter,pos,item:res}\left(\oplus_{res:\langle item,item'\rangle}(q_1 \bowtie_{iter=iter'} (\pi_{iter':iter,item':item} q_2))\right), \Delta_2\right)} \quad \text{(Plus)}$$

Figure 8: Compilation rules For, Seq, Step, Elem, and Plus.

Element construction naturally depends on the XML document representation. These specifics are encapsulated in the operator ε. Given the three arguments $\mathsf{doc} \dot\cup \Delta$ (the set of live nodes), q_t (the set of tag names), and q_e (the content of the new element), ε returns a relation with newly constructed nodes, along with their originating iteration *iter*. Note how Rule Elem (Figure 8) adds the resulting nodes to Δ to reflect the construction of the new transient nodes.

Figure 11 exemplifies the usage of the ε operator if the XML encoding scheme of Section 2.1 is used to evaluate the XQuery expression

```
let $v := e//b return element r { $v }
```

in which we assume that e evaluates to the singleton sequence containing the root element node `a` of the tree depicted in Figure 11(a). After XPath step evaluation, $\$v$ will be bound to the sequence containing the two element nodes with tag `b` (preorder ranks $1, 4$). Figure 11(b) shows the newly constructed tree fragment: the copies of the subtrees rooted at the two `b` nodes now share the newly constructed root node `r`.

Figure 11(c) illustrates how ε constructs the new tree fragment:

(1) the new root node `r` is assigned the next available preorder rank (6 in our case),
(2) the nodes in the affected subtrees are appended with their *size*, *kind*, and *prop* properties unchanged, and their *level* property updated.
(3) Each entry in the resulting relation n is labeled with the originating iteration *iter* from q_t.

The result n of ε contains two pieces of information: The projection on the schema of the document representation $sch(\mathsf{doc})$ represents the set of new transient nodes to be appended to Δ, while the root nodes in n constitute the result of the overall expression. Rule Elem determines the latter via the auxiliary function $roots(n)$ which may be implemented as

$$roots(n) = \pi_{sch(\mathsf{doc}) +\!\!+ \underline{iter}}\left(\sigma_{res}(\ominus_{res:\langle level,zero\rangle}(n \times \frac{zero}{0}))\right)$$

for a *pre/size/level* encoding scheme.

5.4 XPath Evaluation

Our work is complementary to techniques for efficient XPath evaluation. We encapsulate document encoding and the access to XML tree nodes in the algebra operator ◿. Given an *unordered set* of context nodes c (represented as as a relation $\underline{iter}|\underline{item}$) and the live nodes , c ◿$_{\alpha,n}$ $(\mathsf{doc} \dot\cup \Delta)$ returns all nodes reachable from c via XPath step $\alpha::n$, where duplicate elimination is performed for each *iter* value in separation. The compiled algebraic expression obeys the XPath semantics: the resulting nodes are assembled into a sequence whose order is given by the nodes' preorder ranks (which reflect document order) using the ϱ operator.

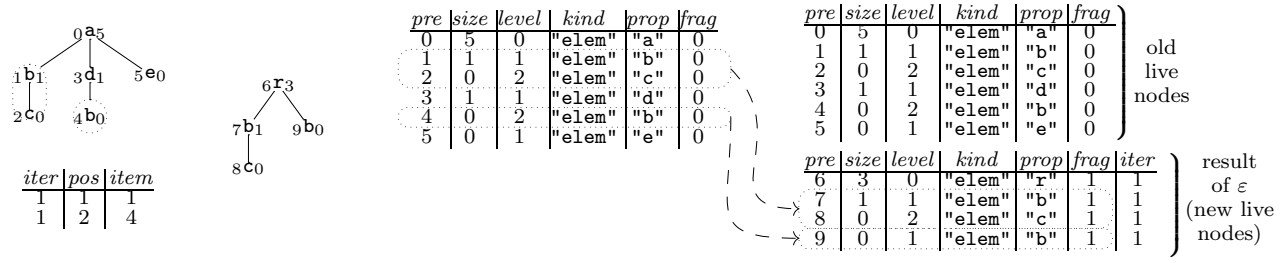
Disjointness of fragments. To evaluate ◿$_{\alpha,n}$, the full set of live nodes has to be queried, a *disjoint* union of persistently stored nodes (doc), and transient nodes constructed at runtime (Δ). Our compilation rules take care to keep these two parts separate during compilation which opens the door for interesting optimizations.

Since the evaluation of an XPath step never escapes the fragment of its context node, the step may safely be evaluated on doc and Δ in separation:

$$c \text{◿}_{\alpha,n} \left(\mathsf{doc} \dot\cup \Delta\right) = (c \text{◿}_{\alpha,n} \mathsf{doc}) \dot\cup (c \text{◿}_{\alpha,n} \Delta)$$

Although more complex at first sight, the latter variant performs the bulk of the work[3] on the persistent doc table and thus can fully benefit from the presence of indexes. The former variant, on the other hand, has to evaluate the axis step on the derived table $\mathsf{doc} \dot\cup \Delta$ which lacks index support.

[3]Typically, $|\Delta| \ll |\mathsf{doc}|$.



(a) Content expression. (b) New tree fragment. (c) Live nodes before (left) and after element construction. The ε operator returns the relation n containing new life nodes, labeled with their originating *iter* value.

Figure 11: Element construction and the resulting extension for table doc.

Bundling XPath steps. Even if a query addresses nodes in only moderately complex XML documents, XPath path expressions are usually comprised of *multiple*, say $k > 1$, location steps (let c denote a sequence of context nodes):

$$c/\alpha_1::n_1/\alpha_2::n_2/\cdots/\alpha_k::n_k \ . \qquad (Q_3)$$

Operator / associates to the left such that the above is seen by the compiler as

$$\big(\cdots((c/\alpha_1::n_1)/\alpha_2::n_2)/\cdots\big)/\alpha_k::n_k$$

which also suggests the evaluation mode of such a multi-step path. Proceeding from left to right, the ith location step computes the context node sequence (in document order and with duplicates removed) for step $i+1$. For each of the k steps, the system

(1) joins the current context node sequence with doc to retrieve the necessary context node properties (only the preorder rank property *pre* is available in the sequence encoding),
(2) performs the doc self-join to evaluate the XPath axis and node test, and finally
(3) removes duplicate nodes generated in step (2).

Especially the latter proves to be quite expensive [11].

With the definition of operator ⍁—the output of ⍁ may serve as the input to a subsequent step—we can do better: if we extend Rule Step to translate multi-step paths *as a whole*, queries of the general form Q_3 can be compiled into a k-way self-join on $\mathsf{doc} \,\dot\cup\, \Delta$ (let q_c denote the compilation result for expression c):

$$\Big(\Big((\pi_{iter,item}(q_c)) \between_{\alpha_1,n_1} (\mathsf{doc} \,\dot\cup\, \Delta)\Big) \between_{\alpha_2,n_2} (\mathsf{doc} \,\dot\cup\, \Delta)\Big) \cdots$$

This, in turn, enables the RDBMS to choose and optimize join order, or—if suitable support is available (*e.g.*, [3])—compute the entire XPath step as a whole. If the XML encoding supports the efficient exploitation of fragment disjointness, whole XPath expressions may be evaluated on doc and Δ in separation, before merging the overall result. Furthermore, sorting and duplicate removal is now required only once. If the RDBMS kernel includes a tree-aware join operator, *e.g.*, *staircase join* [10], duplicate removal may even become obsolete.

5.5 Arithmetic Expressions

Our set-oriented execution model requires a means to evaluate operations on *atomic* values, such as arithmetics, in a bulk fashion. Given the relational representations q_1 and q_2 of two XQuery values e_1 and e_2 in multiple iterations, the expression e_1 + e_2 can be compiled as follows (Rule Plus):

(1) join q_1 and q_2 over their iterations *iter*,
(2) for each tuple, compute the sum of both *item* values, and
(3) project to form the final result:

$$\pi_{iter,pos,item:res}\big(\oplus_{res:\langle item,item'\rangle}(q_1 \between_{iter=iter'} (\pi_{iter':iter,item':item}(q_2)))\big)$$

This evaluation strategy is in line with XQuery semantics which demands the result of an arithmetic expression to be the empty sequence if either operand is the empty sequence (*i.e.*, one or more *iter* values are completely missing in q_1 or q_2).

XMark Query	execution time [s] 1.1 MB	110 MB	1.1 GB
XMark 1	0.003	0.003	0.002
XMark 2	0.036	3.277	136.286
XMark 6	0.007	0.175	1.794
XMark 7	0.009	0.523	5.261

Table 2: Execution times for the XMark benchmark set run on documents of various sizes.

6. EXPERIMENTS: DB2 RUNS XQUERY

An RDBMS can be an efficient host to XQuery. In [9], we implemented the set of algebraic operators of Figure 5 in SQL. This resulted in a purely relational SQL-based XQuery processor. We then compiled and ran a number of queries from the XMark benchmark set [16] to support our claim. We recorded timings on a dual 2.2 GHz Pentium 4 Xeon host, running the IBM DB2 UDB V8.1 database system. Execution times for XML document sizes from 1.1 MB to 1.1 GB are listed in Table 2, for detailed experiments we refer to [9].

The results confirm that our approach can indeed turn relational databases into efficient XQuery processors, scaling well up to and probably beyond document sizes of 1 GB. The database takes advantage of efficient indexing techniques, best visible in the millisecond range timings for XMark 1 that essentially measures XPath performance. We have observed similar figures in earlier work [8, 10].

7. RELATED RESEARCH AND SYSTEMS

As of today, we are not aware of any other published work which succeeded in hosting XQuery *efficiently* on a relational DBMS. A recent survey paper suggests the same [12]. The compilation procedure described here (1) is compositional, (2) does *not* depend on the presence of XML Schema or DTD knowledge (the compiler is *schema-oblivious* unlike [14,17]), and, (3) is *purely relational*. There is no need to invade or extend the database kernel to make the approach perform well (although we may benefit from such extensions [10]).

The work described in [5] comes closest to what we have developed here. Based on a dynamic interval encoding for XML instances, the authors present a compositional translation from a subset of XQuery Core into a set of SQL view definitions. The translation scheme falls short, however, of preserving fundamental semantic properties of XQuery: the omission of a back-mapping step in the translation of `for`-expressions prevents arbitrary expression nesting and, lacking an explicit encoding of sequence positions, the encoding cannot distinguish between sequence and document order.

We feel that the most important drawback, however, is the complexity and execution cost of the SQL view definitions generated in [5]. The compilation of path expressions, for example, leads to nested *correlated* queries—the RDBMS falls back to nested-loops plans, which renders the relational backend a poor XQuery runtime environment.

8. CONCLUSIONS AND CURRENT WORK

The XQuery compiler described in this paper targets relational database backends and thus extends the relational XML processing stack, which was already known to be capable of providing XML mass storage as well as efficient XPath support. The compilation procedure is largely based on a specific encoding of sequences (the principal data structure in the XQuery data model apart from trees) which allows for the set-oriented evaluation of nested `for`-loops (the principal query building block in XQuery). The compiler relies on the presence of the numbering operator ϱ, which can be efficiently implemented using widely available OLAP functionality in the SQL:1999 standard.

Our XQuery compiler offers a variety of interesting hooks for extension and optimization, many of which we were not able to present here. Current work in flux is related to a considerable generalization of the *disjoint fragments* observation of Section 5.4. Since the early days of the development of XQuery Core, it has been observed that certain language constructs, in particular `FLWOR` expressions, enjoy homomorphic properties—in [6] this was shown by reducing `FLWOR` expressions to list (or sequence) comprehensions. This may open the door for compiler optimizations [7] that minimize those parts of a query which need to operate on transient live nodes.

9. REFERENCES

[1] S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language. World Wide Web Consortium, Nov. 2003.

[2] P. A. Boncz and M. L. Kersten. MIL Primitives for Querying a Fragmented World. *The VLDB Journal*, 8(2), 1999.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. Madison, Wisconsin, USA, June 2002.

[4] S. Chien, Z. Vagena, D. Zhang, V. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proc. of the 28th Int'l Conference on Very Large Databases (VLDB)*, Hong Kong, China, Aug. 2002.

[5] D. DeHaan, D. Toman, M. Consens, and M. Öszu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *Proc. of the 22nd Int'l ACM SIGMOD Conference on Management of Data*, San Diego, California, USA, June 2003.

[6] M. Fernández, J. Simeon, and P. Wadler. A Semi-monad for Semi-structured Data. In *Proc. of the 8th Int'l Conference on Database Theory (ICDT)*, London, UK, Jan. 2001.

[7] D. Gluche, T. Grust, C. Mainberger, and M. Scholl. Incremental Updates for Materialized OQL Views. In *Proc. of the 5th Int'l Conference on Deductive and Object-Oriented Databases (DOOD)*, Montreux, Switzerland, Dec. 1997.

[8] T. Grust. Accelerating XPath Location Steps. In *Proc. of the 21st Int'l ACM SIGMOD Conference on Management of Data*, Madison, Wisconsin, USA, June 2002.

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

[10] 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 (VLDB)*, Berlin, Germany, Sept. 2003.

[11] J. Hidders and P. Michiels. Avoiding Unnecessary Ordering Operations in XPath. In *Proc. of the 9th Int'l Workshop on Database Programming Languages (DBPL)*, Potsdam, Germany, Sept. 2003.

[12] R. Krishnamurthy, R. Kaushik, and J. Naughton. XML-to-SQL Query Translation Literature: The State of the Art and Open Problems. In *Proc. of the 1st Int'l XML Database Symposium (XSym)*, Berlin, Germany, Sept. 2003.

[13] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *Proc. of the 27th Int'l Conference on Very Large Databases (VLDB)*, Rome, Italy, Sept. 2001.

[14] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries over Heterogeneous Data Sources. In *Proc. of the 27th Int'l Conference on Very Large Databases (VLDB)*, Rome, Italy, Sept. 2001.

[15] J. Melton. *Advanced SQL:1999: Understanding Object-Relational and Other Advanced Features*. Morgan Kaufmann Publishers, Amsterdam, 2003.

[16] A. Schmidt, F. Waas, M. Kersten, M. 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 (VLDB)*, Hong Kong, China, Aug. 2002.

[17] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. Querying XML Views of Relational Data. In *Proc. of the 27th Int'l Conference on Very Large Databases (VLDB)*, Rome, Italy, Sept. 2001.