XQuery on SQL Hosts

Torsten Grust Sherif Sakr Jens Teubner

University of Konstanz
Department of Computer and Information Science
P.O. Box D 188, 78457 Konstanz, Germany
{grust,sakr,teubner}@inf.uni-konstanz.de

Abstract

Relational database systems may be turned into efficient XML and XPath processors if the system is provided with a suitable relational tree encoding. This paper extends this relational XML processing stack and shows that an RDBMS can also serve as a highly efficient XQuery runtime environment. Our approach is purely relational: XQuery expressions are compiled into SQL code which operates on the tree encoding. The core of the compilation procedure trades XQuery's notions of variable scopes and nested iteration (`FLWOR` blocks) for equi-joins.

The resulting relational XQuery processor closely adheres to the language semantics, *e.g.*, it obeys node identity as well as document and sequence order, and can support XQuery's *full axis* feature. The system exhibits quite promising performance figures in experiments. Somewhat unexpectedly, we will also see that the XQuery compiler can make good use of SQL's OLAP functionality.

1 Introduction

It is a virtue of the relational database model that its canonical physical representation, *tables of tuples*, is simple and thus efficient to implement. Typical operations on tables, *e.g.*, sequential scans, receive excellent support from current computing hardware in terms of prefetching CPU caches and read-ahead in disk-based secondary memory. If linear access is not viable, the regular table structure is sufficiently simple to allow for the definition of efficient *indexes*.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

At the same time, the table proves to be a generic data structure: it is often straightforward to map other data types onto tables. Among others, such encodings have been described 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. In [9, 10] we developed a tree encoding with this property: axis traversals lead to sequential table scans.

This work extends the relational XML processing stack: we devise a compilation procedure that transforms XQuery [2] expressions into SQL code. The compilation itself does not involve interaction with the database back-end. Once shipped to the DBMS, the emitted SQL code evaluates the input XQuery expression by means of a single SQL query. The result is a sequence of atomic values and node identifiers which may then be serialized by a post-processing step [11].

We assume a minimalistic encoding of both, trees and ordered sequences of atomic values and nodes. Several existing XML mapping techniques [3, 15] provide these assumptions, and our compiler can easily be modified to target any such scheme.

We exercise special care in translating the XQuery `FLWOR` construct. There is some tension between XQuery's concept of iterating the evaluation of an expression e_2 for successive bindings of a variable $\$v$ (`for` $\$v$ `in` e_1 `return` e_2) and the set- or table-oriented processing model of SQL. In a nutshell, we thus map `for`-bound variables like $\$v$ into tables containing all bindings and translate expressions in dependence of the variable scopes in which they appear. The resulting SQL code implements iteration via equi-joins, a table operation which RDBMS engines know how to execute most efficiently.

The compiler emits an SQL query with uncorrelated subqueries and does not depend on particularly advanced or "exotic" language features. It is interesting to observe, however, how the compiler can take advantage of widely available SQL/OLAP functions to speed up the evaluation of a number of XQuery constructs, *e.g.*, sequence and element construction as well as `for` expressions.

The paper proceeds as follows. Section 2 discusses relational encodings of trees and sequences, both simple by design. Support for nested variable scopes and efficient iteration affects the overall compilation process and is introduced in Section 3. Section 4 presents a compositional compilation procedure for a subset of XQuery Core in terms of inference rules. We will also see what is to be gained if OLAP ranking functionality is available. Compiler extensions and optimizations are the topic of Section 5: we will discuss bundling of XPath axis steps and how to exploit disjointness properties of tree fragments to evaluate element constructors. Section 6 reports on experiments in which IBM DB2 runs XQuery benchmarks before a review of related work summarizes (Sections 7, 8).

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 XQuery to SQL compilation.

2.1 Trees and XPath Support

We assemble the components of the relational tree encoding piece by piece. Two basic concepts of the XQuery tree data model are *node identity* and *document order* (the latter orders nodes according to the order of their opening tags in the serialized tree instance). To represent both concepts, we assign to each node v its unique *preorder traversal rank* [9] in the tree, $v.pre$. The XQuery node comparison operators `is` and `<<` then compile into comparisons of ranks.

XQuery embeds XPath as a sublanguage to navigate tree structures. Given a sequence of context nodes e, an XPath *axis step* e/α returns the sequence of nodes which are reachable from e via axis α. If we extend the tree encoding for node v 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

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

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

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

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

Support for XPath *name* and *kind tests* is added by means of two further node properties, $v.prop$ and $v.kind \in \{$`"elem"`, `"text"`$\}$.[1] 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 the unique fragment identity for each constructed fragment in the node property *frag*.

The database system maintains a table `doc` of *live nodes* (*i.e.*, nodes of persistent XML documents as well as nodes constructed at runtime) and their properties. Figure 1 depicts two XML fragments as well as their relational encoding. Note that the document order of two nodes v, v' in separate fragments is consistent with the XQuery semantics: if v precedes v' ($v \texttt{<<} v' \equiv v.pre < v'.pre$), the same is true for any pair of nodes taken from these two fragments.

Any XML encoding which provides the above properties or allows for their derivation may be plugged into the compilation procedure. One example of such an encoding is the *XPath accelerator* [9], others include [3, 15].

2.2 Sequences

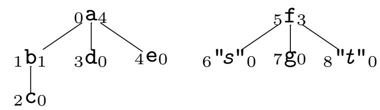
XQuery expressions evaluate to ordered, finite sequences of `item`s. Since sequences are flat and cannot be nested, a sequence may be represented by a single relation in which each tuple encodes a sequence item i. We preserve *sequence order* by means of a

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

```
<a>
  <b><c/></b>
  <d/>
  <e/>
</a>
```

```
<f>
  s<g/>t
</f>
```

(a) Two XML fragments.



(b) Fragment trees.

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

(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).

property $i.pos \geqslant 1$. In sequences, nodes are represented by their unique preorder rank (property $i.pre$) while atomic values, *i.e.*, values of type `xs:float`, `xs:string`, *etc.*, are recorded with their lexical representation $i.val$ as defined by XML Schema [1].

pos	*pre*	*val*
1	NULL	"1.0"
2	NULL	"x"
3	0	NULL
4	5	NULL

Figure 2: Relational sequence encoding.

The relational representation of the sequence `(1.0,"x",`v`,`v'`)` where v and v' denote the root nodes of the two XML fragments of Figure 1 is shown in Figure 2. 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.

3 Relational FLWORs: Turning Variable Scopes and 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`-expression evaluates the body e of the return clause for successive bindings of the `for`-bound variable `$`v:

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

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 $0, 1, \dots, n-1$.

(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 performed as follows:

$$\texttt{SELECT } iter, 1 \texttt{ AS } pos, \texttt{NULL AS } pre, c \texttt{ AS } val$$
$$\texttt{FROM loop} .$$

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

$$\texttt{for \$}v_0\texttt{ in (1,2,3) return 10} .$$

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 $(i, p, \texttt{NULL}, v)$ 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—an analogous interpretation applies for a tuple $(i, p, n, \texttt{NULL})$ which represents a node with preorder rank n (Section 2.1). With this in mind, suppose we rewrite the `for`-loop as

$$\texttt{for \$}v_0\texttt{ in (1,2,3) return (10,\$}v_0\texttt{)} . \qquad (Q_1)$$

Consistent with the loop lifting scheme, the database system will represent variable `$`v_0 as the relation shown in Figure 3(c). 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). The upcoming discussion of nested variable scopes and Section 4 will fill in the missing details.

3.2 Nested Scopes

In XQuery, `for`-loops nest arbitrarily and we will now generalize the loop lifting idea to support nesting.

iter
0
1
2

loop

pos	*pre*	*val*
1	NULL	"10"

encoding of 10

iter	*pos*	*pre*	*val*
0	1	NULL	"10"
1	1	NULL	"10"
2	1	NULL	"10"

lifted encoding of 10

(a) Lifting the constant `10`.

iter	*pos*	*pre*	*val*
0	1	NULL	"10"
0	2	NULL	"20"
1	1	NULL	"10"
1	2	NULL	"20"
2	1	NULL	"10"
2	2	NULL	"20"

(b) Loop-lifted sequence.

iter	*pos*	*pre*	*val*
0	1	NULL	"1"
1	1	NULL	"2"
2	1	NULL	"3"

(c) Encoding of variable $\$v_0$.

iter	*pos*	*pre*	*val*
0	1	NULL	"10"
0	2	NULL	"1"
0	3	NULL	"10"
0	4	NULL	"2"
0	5	NULL	"10"
0	6	NULL	"3"

(d) Result of query Q_1.

Figure 3: Loop lifting.

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

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

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 4.)

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

$$\begin{array}{ccc} & s & \\ s_0 & & s_1 \\ & & | \\ & & s_{10}\end{array}\ .$$

In the following, we write $s_{x \cdot y}$, $x \in \{0, 1, \dots\}^*, y \in \{0, 1, \dots\}$ 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:

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

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 $s_{x \cdot y}$ is thus given by[2]

$$\begin{array}{rl} q_{x\cdot y}(\$v_{x\cdot y}) = & \texttt{SELECT } row() \texttt{ AS } iter, 1 \texttt{ AS } pos, pre, val \\ & \quad \texttt{FROM } q_x(e_{x\cdot y}) \\ & \texttt{ORDER BY } iter, pos\ . \end{array}$$

[2]We assume the presence of a builtin function *row*() which densely numbers the tuples of an ordered table starting from 0. Section 4 discusses two possible implementations of *row*().

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

$$\begin{array}{rl} q_0(\$v_0) = & \texttt{SELECT } row() \texttt{ AS } iter, 1 \texttt{ AS } pos, pre, val \\ & \quad \texttt{FROM } q(\texttt{(1,2,3)}) \\ & \texttt{ORDER BY } iter, pos \end{array}$$

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

Constants. The compilation of an atomic constant c requires loop lifting (Section 3.1). If c occurs in scope s_x:

$$\begin{array}{l}\texttt{for } \$v_x \texttt{ in } e_x \texttt{ return}\\ \quad s_x \left\{\ \cdots c \cdots\right.\end{array}$$

we compile c into

$$\begin{array}{l}\texttt{SELECT } iter, 1 \texttt{ AS } pos, \texttt{NULL AS } pre, c \texttt{ AS } val\\ \quad \texttt{FROM } \mathsf{loop}_x\end{array}$$

in which

$$\mathsf{loop}_x = \begin{array}{l}\texttt{SELECT } iter\\ \quad\texttt{FROM } q_x(\$v_x)\ .\end{array}$$

represents the iterations of the surrounding `for`-loop. The loop relation associated with the top-level scope s is $\mathsf{loop} = \frac{iter}{0}$.

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 $s_{x \cdot x'}$, $x' \in \{0, 1, \dots\}^+$. If scope $s_{x \cdot x'}$ is viewed in isolation, such variables appear to be free.

We will derive the compiled representation of a free variable in scope $s_{x \cdot y}$ from its representation in the directly enclosing scope s_x (if the variable is also free in s_x, we repeat the process). To understand the derivation, consider the evaluation of two nested `for`-loops (note the reference to $\$v_0$ in the inner scope $s_{0 \cdot 0}$):

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

In the zeroth 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 $\$v_{0 \cdot 0}$. Then, during the next outer iteration, two further evaluations of the innermost loop body occur with $\$v_0$ bound to `2`.

iter	*pos*	·	*val*
0	1	·	"1"
1	1	·	"2"

(a) $q_0(\$v_0)$

iter	*pos*	·	*val*
0	1	·	"1"
1	1	·	"1"
2	1	·	"2"
3	1	·	"2"

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

iter	*pos*	·	*val*
0	1	·	"10"
1	1	·	"20"
2	1	·	"10"
3	1	·	"20"

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

Figure 5: Q_2: Scope-dependent representation of variables (entries in the omitted *pre* column are all NULL).

outer	*inner*
0	0
0	1
1	2
1	3

Figure 4: $\mathsf{map}_{(0,0\cdot 0)}$.

The semantics of this nested iteration may be captured by a relation $\mathsf{map}_{(0,0\cdot 0)}$ shown in Figure 4 ($\mathsf{map}_{(x,x\cdot y)}$ 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 $s_{0\cdot 0}$, 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:

$$\begin{aligned} q_{x\cdot y}(\$v_x) = \;& \texttt{SELECT}\ \mathit{inner}\ \texttt{AS}\ \mathit{iter}, \mathit{pos}, \mathit{pre}, \mathit{val} \\ & \texttt{FROM}\ \mathsf{map}_{(x,x\cdot y)}\ ,\ q_x(\$v_x) \\ & \texttt{WHERE}\ \mathit{outer} = \mathit{iter}\ \ . \end{aligned}$$

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):

$$\begin{aligned} \mathsf{map}_{(x,x\cdot y)} = \;& \texttt{SELECT}\ \mathit{iter}\ \texttt{AS}\ \mathit{outer}, \mathit{row}()\ \texttt{AS}\ \mathit{inner} \\ & \texttt{FROM}\ q_x(e_{x\cdot y}) \\ & \texttt{ORDER BY}\ \mathit{iter}, \mathit{pos}\ \ . \end{aligned}$$

Figure 5 contains a line-up of the relational variable representations involved in evaluating query Q_2. Note how the relations in Figures 5(b) and 5(c) represent the fact that, for example, in iteration 2 of the inner loop body variable $\$v_0$ is bound to 2 while $\$v_{0\cdot 0}$ is bound to 10, as desired.

The intermediate result computed by the inner loop is shown in Figure 6(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 For compilation rule in Section 4 emits the required SQL code to achieve this back-mapping. Figure 6(b) depicts the inner loop body result after it has been mapped back into scope s_0. Sequence construction (Rule Seq, Section 4) 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 6(c)).

Other expression types. The compilation procedure ensures that the correct loop relation and variable representations are available when an expression is compiled. Section 4 describes in which way (if any)

iter	*pos*	·	*val*
0	1	·	"1"
0	2	·	"10"
1	1	·	"1"
1	2	·	"20"
2	1	·	"2"
2	2	·	"10"
3	1	·	"2"
3	2	·	"20"

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

iter	*pos*	·	*val*
0	1	·	"1"
0	2	·	"10"
0	3	·	"1"
0	4	·	"20"
1	1	·	"2"
1	2	·	"10"
1	3	·	"2"
1	4	·	"20"

(b) Intermediate result in s_0.

iter	*pos*	·	*val*
0	1	·	"1"
0	2	·	"1"
0	3	·	"10"
0	4	·	"1"
0	5	·	"20"
0	6	·	"2"
0	7	·	"2"
0	8	·	"10"
0	9	·	"2"
0	10	·	"20"

(c) Final result in top-level scope.

Figure 6: Q_2: Intermediate and final results.

$e ::=$	c	atomic constants
$\mid$	`$`v	variables
$\mid$	`(`e`,`e`)`	sequence construction
$\mid$	e`/`α`::`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

Figure 7: Syntax of XQuery Core subset.

other expression types, *e.g.*, sequence construction, element constructors, or path expressions, are affected by variable scoping and iteration.

4 XQuery on SQL Hosts

The core of the XQuery to SQL compiler is defined in terms of a set of inference rules (Figure 8). In these rules, a judgment of the form

$$\Gamma; \mathsf{loop}; \mathsf{doc} \vdash e \Mapsto (q, \mathsf{doc}')$$

indicates that, given

(1) Γ (an *environment* mapping XQuery variables to their relational representation, *i.e.*, an SQL query),
(2) the current loop relation, and
(3) doc (the table of currently live nodes),

the XQuery expression e compiles into the SQL query q with a new table of live nodes doc'. New live nodes are created by XQuery's element constructors only, otherwise we have $\mathsf{doc} = \mathsf{doc}'$.

Compilation starts with the top-level expression, an empty environment[3] $\Gamma = \emptyset$, the singleton loop relation associated with the top-level scope (Section 3.2), and a table doc populated with all persistent XML document instances maintained by the RDBMS; in particular, doc may be empty. All inference rules pass Γ, loop, and doc top-down, while the emitted SQL code is synthesized bottom-up. The compiler produces a single SQL query that operates on the tree and sequence encodings of Section 2.

This paper contains inference rules to compile a subset of XQuery Core defined by the grammar in Figure 7. This subset, plus a few extensions, suffices to

[3]The initial environment Γ may already contain bindings if external variables have been defined in the input query.

iter	*pos*	·	*val*
0	1	·	"1"
1	1	·	"10"
1	2	·	"20"

(a) Encoding q_1 of e_1.

iter	*pos*	·	*val*
0	1	·	"2"
1	1	·	"30"

(b) Encoding q_2 of e_2.

iter	*pos*	·	*val*
0	1	·	"1"
0	3	·	"2"
1	1	·	"10"
1	2	·	"20"
1	3	·	"30"

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

Figure 9: Sequence construction. The dashed lines separate the represented iterations (*iter* partitions).

express the XMark benchmark query set [18], for example. We will sketch a few extensions in the sequel.[4]

Rule Const implements loop lifting for constant atomic values as introduced in Section 3.1. The variable environment Γ is updated and accessed in Rules Let and Var in a standard fashion: to compile `let` $\$v$ `:=` e_1 `return` e_2, translate e_1 in environment Γ to yield the SQL query q_1, then compile e_2 in the enriched environment $\Gamma + \{\$v \mapsto q_1\}$. A reference to $\$v$ in e_2 then yields q_1 via Rule Var.

Essentially, Rule Seq compiles the sequence construction `(`e_1`,`e_2`)` into an SQL `UNION ALL` of the relational encodings q_1 and q_2 of e_1 and e_2. Note that this evaluates the sequence construction for *all* iterations encoded in q_1, q_2 at once. Figure 9 exemplifies the operation of the compiled code. Relation q_1 encodes two sequences: `(1)` in iteration 0 and `(10,20)` in iteration 1, while q_2 encodes `(2)` in iteration 0 and `(30)` in iteration 1. The SQL code generated by Rule Seq computes the result in Figure 9(c): the sequence construction evaluates to `(1,2)` in iteration 0 and `(10,20,30)` in iteration 1, as expected.

Exploiting OLAP functionality. In Figure 9(c), note that the resulting *pos* column, in general, is not densely populated for each iteration (*i.e.*, in each *iter* partition). While this is neither a problem for the sequence encoding nor for the compilation process *per se*, we can use an alternative SQL implementation—based on the SQL/OLAP amendment defined for SQL:1999 [17]—which will generate a dense *pos* column, ascending from 1 in each *iter* partition (ordering by columns *ord*,*pos* ensures that sequence order is obeyed: items encoded in q_1 will appear before items encoded in q_2):

```
SELECT iter,
       DENSE_RANK() OVER
       (PARTITON BY iter ORDER BY ord, pos) AS pos,
       pre, val
  FROM (SELECT *, 0 AS ord FROM q1
        UNION ALL
        SELECT *, 1 AS ord FROM q2) .
```

[4]In fact, the subset may be extended to embrace the complete XQuery Core language. Support for dynamic typing and validation, however, requires extensions to the minimalistic tree and sequence encoding discussed here.

Node test n	Predicate $test(v, n)$
`*`	$v.kind$ = `"elem"`
t (tag name)	$v.kind$ = `"elem"` `AND` $v.prop$ = `"`t`"`
`text()`	$v.kind$ = `"text"`
`node()`	`TRUE`

Table 2: Predicate *test*() represents XPath node tests.

This variant (1) executes substantially faster in our experimental setup (Section 6), (2) avoids early `INTEGER` overflow in the *pos* column, and (3) works correctly in case relation q_1 is empty (the original SQL code in Rule Seq requires a slight adaption to ensure this).

Rule Step compiles an XPath location step α`::`n. The SQL code yields a node sequence that obeys the XPath semantics: while the `DISTINCT` clause removes duplicate nodes, we use the nodes' preorder rank—which reflects document order—to order the sequence ($d.pre$ `AS` pos). Property *frag* is tested to avoid that step evaluation escapes the document fragment of the current context node e'.

Rule Step uses region queries as described in [9] to evaluate XPath axis steps. For some common location steps we listed predicate *axis*() in Table 1 to evaluate axis α, predicate *test*() encodes the associated node (name or kind) test (Table 2).

Rule For essentially implements the compilation procedure for `for`-loops as introduced in Section 3.2. Note how the rule makes use of the `map` relation to map all variables $\$v_i$ in the environment into the scope opened by the `for` expression. The SQL code emitted by Rule For implements the back-mapping step explained in Section 3.2.

In this context, the OLAP function `DENSE_RANK()` may serve as an efficient implementation of the hypothetical *row*() function introduced in Section 3.2. If `DENSE_RANK()` (or equivalent functionality, *e.g.*, `ROW_NUMBER`) is not provided by the SQL dialect of the target RDBMS, we can rephrase the definition of `map` as follows:

```
SELECT iter AS outer,
       iter * m.pos + e1.pos AS inner
  FROM q1 AS e1, (SELECT MAX(pos) AS pos FROM q1) AS m
```

(q_v may be rewritten accordingly). To illustrate, given the *iter* and *pos* columns of relation q_1 as shown in Figure 10(a), this SQL query computes the `map` relation of Figure 10(b)—which performs inferior to the OLAP variant but is good enough to ensure correct compilation.

Rule Elem emits SQL code for the evaluation of an XQuery element constructor `element` t `{` e `}` in which subexpression e is required to evaluate to a sequence of nodes `(`v_1`,`v_2`,...,`v_n`)`: (1) a new element node

$$\frac{}{\Gamma;\mathsf{loop};\mathsf{doc} \vdash c \Mapsto \left(\begin{array}{l}\texttt{SELECT } l.\mathit{iter}, 1 \texttt{ AS } \mathit{pos},\\ \qquad \texttt{NULL AS } \mathit{pre}, c \texttt{ AS } \mathit{val}\\ \quad \texttt{FROM } \mathsf{loop} \texttt{ AS } l\end{array}, \mathsf{doc}\right)}\ (\textsc{Const}) \qquad \frac{}{\{\dots,\$v\mapsto q_v,\dots\};\mathsf{loop};\mathsf{doc}\vdash \$v \Mapsto (q_v,\mathsf{doc})}\ (\textsc{Var})$$

$$\frac{\Gamma;\mathsf{loop};\mathsf{doc}\vdash e_1 \Mapsto (q_1,\mathsf{doc}') \qquad \Gamma+\{\$v\mapsto q_1\};\mathsf{loop};\mathsf{doc}'\vdash e_2 \Mapsto (q_2,\mathsf{doc}'')}{\Gamma;\mathsf{loop};\mathsf{doc}\vdash \texttt{let } \$v \texttt{ := } e_1 \texttt{ return } e_2 \Mapsto (q_2,\mathsf{doc}'')}\ (\textsc{Let})$$

$$\frac{\Gamma;\mathsf{loop};\mathsf{doc}\vdash e_1 \Mapsto (q_1,\mathsf{doc}') \qquad \Gamma;\mathsf{loop};\mathsf{doc}'\vdash e_2 \Mapsto (q_2,\mathsf{doc}'')}{\Gamma;\mathsf{loop};\mathsf{doc}\vdash \texttt{(}e_1\texttt{,}e_2\texttt{)} \Mapsto \left(q_1 \texttt{ UNION ALL } \begin{array}{l}\texttt{SELECT } \mathit{iter}, e_2.\mathit{pos}+m.\mathit{pos} \texttt{ AS } \mathit{pos}, \mathit{pre}, \mathit{val}\\ \quad \texttt{FROM } q_2 \texttt{ AS } e_2,\\ \qquad (\texttt{SELECT MAX(}\mathit{pos}\texttt{) AS } \mathit{pos} \texttt{ FROM } q_1) \texttt{ AS } m\end{array}, \mathsf{doc}''\right)}\ (\textsc{Seq})$$

$$\frac{\Gamma;\mathsf{loop};\mathsf{doc}\vdash e \Mapsto (q_e,\mathsf{doc}')}{\Gamma;\mathsf{loop};\mathsf{doc}\vdash e\texttt{/}\alpha\texttt{::}n \Mapsto \left(\begin{array}{l}\texttt{SELECT DISTINCT } e.\mathit{iter}, d.\mathit{pre} \texttt{ AS } \mathit{pos}, d.\mathit{pre}, \texttt{NULL AS } \mathit{val}\\ \quad \texttt{FROM } q_e \texttt{ AS } e, \mathsf{doc}' \texttt{ AS } e', \mathsf{doc}' \texttt{ AS } d\\ \quad \texttt{WHERE } e'.\mathit{pre} = e.\mathit{pre}\\ \qquad \texttt{AND } e'.\mathit{frag} = d.\mathit{frag}\\ \qquad \texttt{AND } \mathit{axis}(e',d,\alpha) \texttt{ AND } \mathit{test}(d,n)\end{array}, \mathsf{doc}'\right)}\ (\textsc{Step})$$

$$\frac{\begin{array}{l}\{\dots,\$v_i\mapsto q_{v_i},\dots\};\mathsf{loop};\mathsf{doc}\vdash e_1 \Mapsto (q_1,\mathsf{doc}') \qquad \mathsf{loop}' \equiv (\texttt{SELECT } \mathit{iter} \texttt{ FROM } q_v)\\ q_v \equiv \left(\begin{array}{l}\quad\texttt{SELECT } \mathit{row}() \texttt{ AS } \mathit{iter}, 1 \texttt{ AS } \mathit{pos}, \mathit{pre}, \mathit{val}\\ \qquad \texttt{FROM } q_1\\ \texttt{ORDER BY } \mathit{iter}, \mathit{pos}\end{array}\right) \qquad \mathsf{map} \equiv \left(\begin{array}{l}\quad\texttt{SELECT } \mathit{iter} \texttt{ AS } \mathit{outer}, \mathit{row}() \texttt{ AS } \mathit{inner}\\ \qquad \texttt{FROM } q_1\\ \texttt{ORDER BY } \mathit{iter}, \mathit{pos}\end{array}\right)\\ \left\{\dots,\$v_i\mapsto \begin{array}{l}\texttt{SELECT } \mathit{inner} \texttt{ AS } \mathit{iter}, \mathit{pos}, \mathit{pre}, \mathit{val}\\ \quad \texttt{FROM } \mathsf{map}, q_{v_i}\\ \texttt{WHERE } \mathit{outer} = \mathit{iter}\end{array}, \dots\right\} + \{\$v\mapsto q_v\};\mathsf{loop}';\mathsf{doc}'\vdash e_2 \Mapsto (q_2,\mathsf{doc}'')\end{array}}{\{\dots,\$v_i\mapsto q_{v_i},\dots\};\mathsf{loop};\mathsf{doc}\vdash \texttt{for } \$v \texttt{ in } e_1 \texttt{ return } e_2 \Mapsto \left(\begin{array}{l}\texttt{SELECT } \mathit{outer} \texttt{ AS } \mathit{iter},\\ \qquad e_2.\mathit{iter} * m.\mathit{pos} + e_2.\mathit{pos} \texttt{ AS } \mathit{pos},\\ \qquad e_2.\mathit{pre}, e_2.\mathit{val}\\ \quad \texttt{FROM } \mathsf{map}, q_2 \texttt{ AS } e_2,\\ \qquad (\texttt{SELECT MAX(}\mathit{pos}\texttt{) AS } \mathit{pos}\\ \qquad\quad \texttt{FROM } q_2) \texttt{ AS } m\\ \quad \texttt{WHERE } \mathit{inner} = e_2.\mathit{iter}\end{array}, \mathsf{doc}''\right)}\ (\textsc{For})$$

$$\frac{\begin{array}{l}\Gamma;\mathsf{loop};\mathsf{doc}\vdash e \Mapsto (q_e,\mathsf{doc}')\\ \mathit{subtree\text{-}copies} \equiv \left(\begin{array}{l}\texttt{SELECT } d.\mathit{pre} - e'.\mathit{pre} + m_d.\mathit{pre} + 2 + (e.\mathit{iter} * m_e.\mathit{pos} + e.\mathit{pos}) * m_d.\mathit{size} \texttt{ AS } \mathit{pre}, d.\mathit{size},\\ \qquad d.\mathit{level} - e'.\mathit{level} + 1 \texttt{ AS } \mathit{level}, d.\mathit{kind}, d.\mathit{prop}, m_d.\mathit{frag} + 1 + e.\mathit{iter} \texttt{ AS } \mathit{frag}\\ \quad \texttt{FROM } q_e \texttt{ AS } e, \mathsf{doc}' \texttt{ AS } e', \mathsf{doc}' \texttt{ AS } d,\\ \qquad (\texttt{SELECT MAX(}\mathit{pos}\texttt{) AS } \mathit{pos} \texttt{ FROM } q_e) \texttt{ AS } m_e,\\ \qquad (\texttt{SELECT MAX(}\mathit{pre}+\mathit{size}\texttt{) AS } \mathit{pre}, \texttt{MAX(}\mathit{size}\texttt{)} + 1 \texttt{ AS } \mathit{size}, \texttt{MAX(}\mathit{frag}\texttt{) AS } \mathit{frag}\\ \qquad\quad \texttt{FROM } \mathsf{doc}') \texttt{ AS } m_d\\ \quad \texttt{WHERE } e'.\mathit{pre} = e.\mathit{pre} \texttt{ AND } e'.\mathit{frag} = d.\mathit{frag} \texttt{ AND } \mathit{axis}(e', d, \texttt{descendant-or-self})\end{array}\right)\\ \mathit{new\text{-}roots} \equiv \left(\begin{array}{l}\texttt{SELECT } l.\mathit{iter}, l.\mathit{iter} * m_e.\mathit{pos} * m_d.\mathit{size} + m_d.\mathit{pre} + 1 \texttt{ AS } \mathit{pre},\\ \qquad m_e.\mathit{pos} * m_d.\mathit{size} \texttt{ AS } \mathit{size}, 0 \texttt{ AS } \mathit{level}, \texttt{"elem" AS } \mathit{kind},\\ \qquad t \texttt{ AS } \mathit{prop}, m_d.\mathit{frag} + 1 + l.\mathit{iter} \texttt{ AS } \mathit{frag}\\ \quad \texttt{FROM } \mathsf{loop} \texttt{ AS } l, (\texttt{SELECT MAX(}\mathit{pos}\texttt{) AS } \mathit{pos} \texttt{ FROM } q_e) \texttt{ AS } m_e,\\ \qquad (\texttt{SELECT MAX(}\mathit{pre}+\mathit{size}\texttt{) AS } \mathit{pre}, \texttt{MAX(}\mathit{size}\texttt{)} + 1 \texttt{ AS } \mathit{size}, \texttt{MAX(}\mathit{frag}\texttt{) AS } \mathit{frag}\\ \qquad\quad \texttt{FROM } \mathsf{doc}') \texttt{ AS } m_d\end{array}\right)\end{array}}{\Gamma;\mathsf{loop};\mathsf{doc}\vdash \texttt{element } t \texttt{ \{}e\texttt{\}} \Mapsto \left(\begin{array}{l}\texttt{SELECT } \mathit{iter}, 1 \texttt{ AS } \mathit{pos},\\ \qquad \mathit{pre}, \texttt{NULL AS } \mathit{val}\\ \quad \texttt{FROM } \mathit{new\text{-}roots}\end{array}, \begin{array}{l}\mathsf{doc}'\\ \texttt{UNION ALL}\\ \mathit{subtree\text{-}copies}\\ \texttt{UNION ALL}\\ \texttt{SELECT } \mathit{pre}, \mathit{size}, \mathit{level}, \mathit{kind}, \mathit{prop}, \mathit{frag}\\ \quad \texttt{FROM } \mathit{new\text{-}roots}\end{array}\right)}\ (\textsc{Elem})$$

Figure 8: XQuery to SQL compilation procedure.

iter	*pos*	·	·
0	1	·	·
0	2	·	·
1	1	·	·
1	2	·	·
1	3	·	·

(a) Encoding q_1.

outer	*inner*
0	1
0	2
1	4
1	5
1	6

(b) Resulting map.

Figure 10: Computing map without OLAP extensions.

r with tag name t is appended to the table doc of live nodes, (2) the n subtrees rooted at the nodes v_i are extracted (the code effectively evaluates the location step `$`v_i`/descendant-or-self::node()`) and then appended to doc, and (3) r is made the common new root of the subtree copies.

Consider the query

`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).[5] 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. The latter also constitutes the result of the overall expression.

Figure 11(c) illustrates how the new fragment is appended to the doc table:

(1) the new root node r is appended and assigned the next available preorder rank MAX($pre + size$) + 1,
(2) the nodes in the affected subtrees are appended to doc (with $pre \geqslant$ MAX($pre + size$) + 2) with their *size*, *kind*, and *prop* properties unchanged, and their *level* property updated.

To simplify the generated SQL code, we overestimate the size of the copied subtrees to be the size of the largest subtree. In general, this leads to gaps in the *pre* column and an overestimation of the *size* property of the new root node: in Figure 11(c), root r is recorded with size 4 while the actual number of nodes below r is 3. Again, this does not affect correctness (see Section 2.1) and can be fully remedied if OLAP functionality is available.

5 Extensions and Optimizations

The compilation procedure could be extended to embrace a significantly larger subset of XQuery Core than presented here.

Consistent with our sequence encoding and with the XQuery semantics we can, for example, define the *effective boolean value* [2] of a sequence in different iterations via the absence or presence of *iter* values in its encoding (Figures 12(a) and 12(b)). This enables the compilation of XQuery's *conditional expression* `if` e_1 `then` e_2 `else` e_3 as shown in Figure 12(c).

With the conditional available, the language subset may be further extended by (1) *predicates* e`[`e`]`, (2) the existential and universal *quantifiers* (`some`, `every`), (3) the *general comparison operators* for sequences, and (4) the XQuery `where` e clause which is an optional part of the syntactical `FLWOR` construct.

5.1 Exploiting the Disjointness of Fragments During Element Construction

Recall that the evaluation of an element constructor places both, the newly created element node and the subtree copies in a new separate fragment in table doc. The new current table of live nodes, computed in Rule ELEM via SQL's `UNION ALL` operator, may thus be written as the *disjoint union*

$$\mathsf{doc} \mathbin{\dot\cup} \Delta$$

where doc is the table of persistent XML nodes and Δ denotes the transient nodes in the new fragment.

Now consider the evaluation of a second element constructor `element` t `{` e `}` with $e \subseteq \mathsf{doc} \mathbin{\dot\cup} \Delta$. Rule ELEM performs the XPath location step

e`/descendant-or-self::node()`

to extract the subtrees which need to be copied into the new fragment. Since the evaluation of an XPath location step never escapes the fragment of its context node, the following would be an equivalent way to compute the nodes in the subtrees:

$$(e \cap \mathsf{doc})\texttt{/descendant-or-self::node()}$$
$$\dot\cup$$
$$(e \cap \Delta)\texttt{/descendant-or-self::node()}$$

Although more complex at first sight, this variant performs the bulk of the work[6] on the persistent doc table and thus can fully benefit from the presence of indexes (Section 6). The former variant, on the other hand, has to evaluate the `descendant-or-self` axis step on the derived table doc `UNION ALL` Δ which lacks index support.

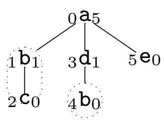
After evaluation of the second element constructor, the new table of live nodes is

$$(\mathsf{doc} \mathbin{\dot\cup} \Delta) \mathbin{\dot\cup} \Delta' = \mathsf{doc} \mathbin{\dot\cup} (\Delta \mathbin{\dot\cup} \Delta')$$

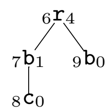
such that this optimization remains applicable after an arbitrary number of element constructor evaluations. More importantly, note that XPath step evaluation in general can benefit from this disjointness of fragments.

[5]Here, for ease of presentation, we assume that e encodes a node sequence in a single iteration. The SQL code in Rule ELEM handles the general case.

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



(a) Original tree.



(b) New tree fragment.

pre	*size*	*level*	*kind*	*prop*	*frag*
0	5	0	"elem"	"a"	0
1	1	1	"elem"	"b"	0
2	0	2	"elem"	"c"	0
3	1	1	"elem"	"d"	0
4	0	2	"elem"	"b"	0
5	0	1	"elem"	"e"	0

pre	*size*	*level*	*kind*	*prop*	*frag*
0	5	0	"elem"	"a"	0
1	1	1	"elem"	"b"	0
2	0	2	"elem"	"c"	0
3	1	1	"elem"	"d"	0
4	0	2	"elem"	"b"	0
5	0	1	"elem"	"e"	0
6	4	0	"elem"	"r"	1
7	1	1	"elem"	"b"	1
8	0	2	"elem"	"c"	1
9	0	1	"elem"	"b"	1
10					

(c) Table `doc` before (left) and after element construction. The *size* of node `r` has been overestimated.

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

iter
0
1
2

(a) Current `loop`.

iter	*pos*	·	·
0	1	·	·
0	2	·	·
2	1	·	·

(b) Encoded sequence.

```
 SELECT *
   FROM q2 AS e
  WHERE EXISTS (SELECT e.iter FROM q1 AS c WHERE c.iter = e.iter)
UNION ALL
 SELECT *
   FROM q3 AS e
  WHERE NOT EXISTS (SELECT e.iter FROM q1 AS c WHERE c.iter = e.iter)
```

(c) Evaluation of a conditional expression (q_i encodes expression e_i).

Figure 12: The effective boolean value of the encoded sequence (b) in the current loop is *true* in iterations 0 and 2, and *false* (*i.e.*, the empty sequence) in iteration 1. SQL code generated for `if` e_1 `then` e_2 `else` e_3 in (c).

5.2 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 e denote a sequence of context nodes):

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

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

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

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 $k = 2$, the normalized XQuery Core [5] equivalent reads (slightly simplified):

```
distinct-doc-order(
  for $v1 in e return
    distinct-doc-order(
      for $v2 in $v1/α1::n1 return
        $v2/α2::n2))
```

Note that Rule Step already improves on this naive evaluation scheme: while the above iterates the step evaluation for each context node, the compiler emits SQL code that applies a location step to a *whole context node sequence*. In a sense, Rule Step implements the above iteration implicitly via the self-join of table `doc`.

Nevertheless, the compilation of a k-step path, and thus the k-fold application of Rule Step, leads to an SQL query that is nested to depth k. The nesting is not a problem *per se* for the RDBMS—in the terminology of Kim [13], Rule Step generates uncorrelated *type N* nested queries. However, at each nesting level, *i.e.*, k times, 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 [12].

Since we target a relational database backend, we can do better: the tree encoding of Section 2.1 allows us to evaluate a multi-step path *as a whole* [9, 10]. In a modified compiler, Rule Step is replaced by a new Rule Steps which is applicable to queries of the general form Q_3. For a k-step path, the new rule emits a flat k-way self-join of table `doc` (plus a single join with the initial context node sequence e). This, in turn, enables the RDBMS to choose and optimize join order. In our experiments (Section 6) we observed that the system decided to evaluate certain paths in a “backward” fashion. Furthermore, 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.

6 Experiments: DB2 Runs XQuery

An RDBMS can be an efficient host to XQuery. To support this claim and in order to assess the viability and performance of our approach, we ran a number of queries from the XMark benchmark series [18] on the IBM DB2 UDB V8.1 database system. The database was hosted on a dual 2.2 GHz Pentium 4 Xeon system with 2 GB RAM, running a version 2.4 Linux kernel. The experiment was the only client connected to the database. No other processes were active besides a small number of system daemons.

We used the XML generator XMLgen from the XMark project to create XML document instances with sizes ranging from 110 KB to 1.1 GB (5,000 to 50 million nodes). An instance of the `doc` table was created for each document size and then populated with the encoded XMark XML documents as described in Section 2. The database resided on a single SCSI disk, with the buffer pool size set to 200,000 pages.

To make the point that an RDBMS can indeed be an efficient host to XQuery, we presented the result of the compilation process to the system's workload analysis tools. The DB2 *index advisor* `db2advis` was used to recommend a set of indexes to optimally support our workload. The recommendations included indexes on the *pre* column of the `doc` table to support queries on the XML tree structure, and indexes on the *prop* column to support node tests.

We created the recommended indexes and issued DB2's `reorg` command to optimize the physical data placement on secondary storage. No other "wizardry" was applied. Experiments were run with a "warm" database buffer cache, each query was run multiple times with timings averaged.

6.1 Impact of OLAP Availability and XPath Step Bundling

Sections 4 and 5 described the use of SQL OLAP functions as well as the bundling of successive XPath steps as two promising optimization hooks. To verify the effectiveness of these techniques, we repeatedly executed query XMark 1 on a 110 KB XML document with and without these optimizations applied. The effects are substantial: execution times are reduced by orders of magnitude (Table 3).

It turns out, that our choice of sequence encoding and representation of iteration, *i.e.*, a *single* relation encodes the sequence value for *all* iterations of a `for`-loop, is a perfect match for the SQL/OLAP ranking and partitioning functionality. The compiler can repeatedly make use of the idiom

`DENSE_RANK() (PARTITION BY` *iter* `ORDER BY` *iter*, *pos*`)`

Optimization	exec. time [s]	# tbl. acc.
no optimization	5995	196
use of OLAP functions	0.14	43
bundled XPath steps	0.02	24
OLAP and bundled XPath	0.002	13

Table 3: Effectiveness of optimizations. Execution times and number of accesses to the `doc` relation for XMark 1 run on a 110 KB document with different optimizations applied.

to compute dense sequence positions (property *pos*) inside an iteration, *i.e.*, inside an *iter* partition. Likewise, the compiler may emit

`DENSE_RANK() (ORDER BY` *iter*, *pos*`)`

to densely populate *iter* columns, *e.g.*, during the computation of `map` (Section 3.2).

Furthermore, most XMark queries feature multi-step path expressions—typical path lengths are 3 or 4 steps—such that these queries are also subject to the XPath step bundling optimization (Section 5.2). Taken together, both optimizations reduced the number of accesses to the persistent `doc` relation by a factor of 15.

6.2 Disjointness of Fragments

Remember that the construction of new element nodes essentially leads to a `UNION ALL` operation that extends the persistent `doc` relation by a disjoint transient set of nodes.

XMark 13 features two successive element constructors[7] and thus is a typical candidate for the disjoint fragments optimization of Section 5.1:

```
for $i in fn:doc("auction.xml")/site
                        /regions/australia/item
return
 element item { (element name { $i/name/text() },
                 $i/description) }
```

Document `"auction.xml"` resided in the persistent `doc` table and thus received full index support.

To evaluate the query, the system eventually created the `name` element nodes and subtree copies and extended the `doc` table accordingly. Note that the situation in XMark 13 perfectly matches the scenario of Section 5.1: when the `item` element nodes are created, their child nodes are taken from both, the persistent document (`$i/description`) and the transient live nodes (the `name` element nodes).

With the optimization applied, access without index support was only required for the relatively few transient `name` nodes. Without this optimization, *all*

[7] In the original XMark 13 query, the inner constructor creates an attribute node. Our discussion is not affected by this adaptation.

Optimization	execution time [s] 1.1 MB	11 MB	55 MB
no optimization	1.1	48.7	1088
fragment disjointness	0.31	2.9	14.7

Table 4: XMark 13 on various XML document sizes with and without exploitation of fragment disjointness.

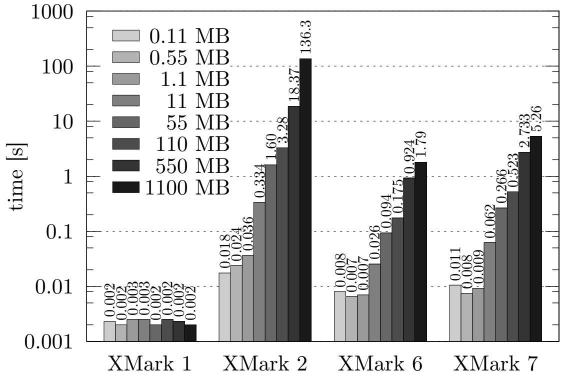


Figure 13: XMark queries run on documents of various sizes.

child nodes of the newly created `item` elements resided in a derived table with no persistent index support at all. We ran both variants on our test database and observed the execution times documented in Table 4. The experiment clearly indicates the potential of this optimization technique.

6.3 XMark on DB2

Finally, to evaluate our compilation procedure on a range of document sizes, we chose a set of queries from the XMark benchmark. The set comprises the XQuery constructs which have been discussed in the foregoing, namely `FLWOR` and XPath expressions (all queries), and element construction (XMark 2). XMark 6 and 7 further contain XQuery aggregate functions (`fn:count`) and can benefit from the efficient implementation of their SQL counterparts in the relational system.

All queries were compiled with optimizations applied. The results are depicted in Figure 13 and confirm the scalability of our approach with respect to the document size. Execution times are reasonable even for the 1 GB XMark document instance. The milli-second range timings for XMark 1 stem from the fact that this query essentially measures XPath performance. We have observed similar figures in earlier work [9, 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

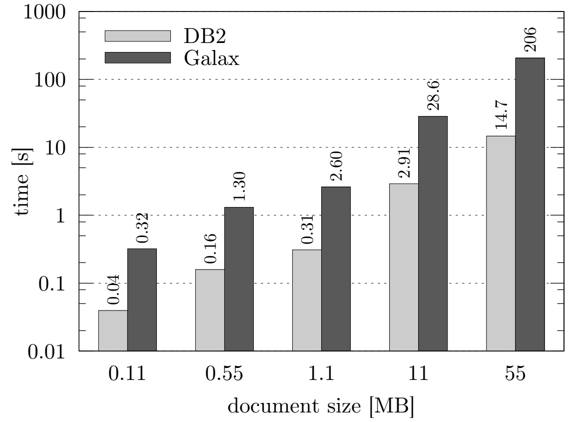


Figure 14: XMark 13: DB2 and Galax 0.3.5 compared.

an SQL-based RDBMS. A recent survey paper suggests the same [14]. 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 [16, 19]), and, (3) is *purely relational* in the sense that the compiler translates XQuery into standard SQL:1999 plus OLAP extensions: 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]).

Evidence for the latter is also provided by experiments in which we compared the relational XQuery host and the XQuery processor Galax [6]. Galax operates on an in-memory representation of XML documents and implements the XQuery Formal Semantics specification quite literally, *i.e.*, nested `for`-loops are evaluated in a nested-loops fashion, XPath path expressions are evaluated step-by-step, *etc.* We thus expected the strengths of relational technology to come in useful especially with increasing document sizes—this is exactly what the measured execution times for XMark 13 indicate (Figure 14).

The work described in [4] comes closest to what we have developed here. Based on a dynamic interval encoding for XML instances, the paper presents 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 backmapping step in the translation of `for`-expressions prevents arbitrary expression nesting and, lacking an explicit treatment 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 [4]. 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. To achieve acceptable performance, the authors indeed proposed modifications to the relational engine specifically geared to support the dynamic interval encoding (SQL-based timings were never published).

8 Conclusions and Work in Flux

The XQuery compiler described in this paper targets SQL-based 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). Operations on this encoding receive excellent support from widely available OLAP extensions to the SQL:1999 standard.

Our XQuery to SQL 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 5.1. 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 [7] this was shown by reducing `FLWOR` expressions to list (or sequence) comprehensions. This may open the door for compiler optimizations [8] that minimize those parts of a query which need to operate on transient live nodes.

References

[1] P.V. Biron and A. Malhotra. XML Schema Part 2: Datatypes. World Wide Web Consortium, May 2001. `http://www.w3.org/TR/xmlschema-2/`.

[2] S. Boag, D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language. World Wide Web Consortium, November 2003. W3C Working Draft `http://www.w3.org/TR/xquery/`.

[3] S. Chien, Z. Vagena, D. Zhang, V.J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. In *Proc. of the 28th Int'l Conference on Very Large Databases (VLDB)*, pages 263–274, Hong Kong, China, August 2002.

[4] D. DeHaan, D. Toman, M.P. Consens, and M.T. 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*, pages 623–634, San Diego, California, USA, June 2003.

[5] D. Draper, P. Fankhauser, M.F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Simeon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. World Wide Web Consortium, February 2004. W3C Working Draft `http://www.w3.org/TR/xquery-semantics/`.

[6] M.F. Fernández, J. Simeon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax Experience. In *Proc. of the 29th Int'l Conference on Very Large Data Bases (VLDB)*, pages 1077–1080, Berlin, Germany, September 2003.

[7] M.F. 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)*, pages 263–300, London, UK, January 2001.

[8] D. Gluche, T. Grust, C. Mainberger, and M.H. Scholl. Incremental Updates for Materialized OQL Views. In *Proc. of the 5th Int'l Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 52–66, Montreux, Switzerland, December 1997.

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

[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, September 2003.

[11] T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath Evaluation in Any RDBMS. *ACM Transactions on Database Systems*, 29(1):91–131, March 2004.

[12] 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, September 2003.

[13] W. Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems*, 7(3):443–469, September 1982.

[14] 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)*, pages 1–18, Berlin, Germany, September 2003.

[15] 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)*, pages 361–370, Rome, Italy, September 2001.

[16] 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, September 2001.

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

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

[19] J. Shanmugasundaram, J. Kiernan, E.J. 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)*, pages 261–270, Rome, Italy, September 2001.