XQuery on SQL Hosts

Torsten Grust  Sherif Sakr  Jens Teubner

University of Konstanz
Department of Computer and Information Science
P.O. Box D188, 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 cache 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.

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

Proceedings of the 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 $<$ 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 $e/\alpha$ returns the sequence of nodes which are reachable from $e$ via axis $\alpha$. 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

\[
v \in c/\text{ancestor} \iff v.pre < c.pre \ \text{AND} \ c.pre \leq v.pre + v.size .
\]

<table>
<thead>
<tr>
<th>Axis $\alpha$</th>
<th>Predicate $\text{axis}(c, v, \alpha)$: $v \in c/\alpha$</th>
</tr>
</thead>
<tbody>
<tr>
<td>descendant</td>
<td>$v.pre &gt; c.pre \ \text{AND} \ c.size \leq v.pre + c.size$</td>
</tr>
<tr>
<td>child</td>
<td>$\text{axis}(c, v, \text{descendant}) \ \text{AND} \ c.size = v.size + 1$</td>
</tr>
<tr>
<td>following</td>
<td>$v.size &gt; c.pre + c.size$</td>
</tr>
<tr>
<td>preceding</td>
<td>$v.pre + v.size &lt; c.pre$</td>
</tr>
</tbody>
</table>

Table 1: Predicate $\text{axis}()$ 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 $v.kind \in \{"\text{elem}","\text{text}"\}$. For an element node $v$ with tag name $t$, we have $v.prop = "t"$, for a text node $v'\prime$ 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

\[
(\text{element a}\{\text{element b}\{()\}\},\text{element c}\{()\})
\]

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 $\text{frag}$.

The database system maintains a table $\text{doc}$ 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 $v, v'\prime$ in separate fragments is consistent with the XQuery semantics: if $v$ precedes $v'\prime$ ($v < v'\prime \equiv v.pre < v'.pre$), 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 items. 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.
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\:

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

where \( e[x/\$v] \) 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, \ldots, \( n-1 \).

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 \( c \)).

For a constant atomic value \( c \), lifting with respect to a given loop relation is performed as follows:

\[
\text{SELECT iter, 1 as pos, NULL as pre, c as val FROM loop .}
\]

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

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

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 \((i,p,\text{NULL},v)\) in a loop-lifted relation for subexpression \( e \) may be read as the assertion that, during the \( i \)th iteration, the item at position \( p \) in \( e \) has value \( v \)—an analogous interpretation applies for a tuple \((i,p,n,\text{NULL})\) which represents a node with preorder rank \( n \) (Section 2.1). With this in mind, suppose we rewrite the for-loop as

\[
\text{for } \$v_0 \text{ in } (1,2,3) \text{ return } (10,\$v_0) . \quad (Q_1)
\]

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.
XQuery, in general, lead to a tree-shaped hierarchy of scopes. For the above query, we obtain

\[
q_0(\$v_0) = \text{SELECT row() AS iter, 1 AS pos, pre, val FROM } q((1,2,3)) \text{ ORDER BY iter, pos}
\]

where \(q((1,2,3))\) 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 \(s_x\):

\[
\text{for } \$v_x \text{ in } e_x \text{ return } s_x \{ \cdots c \cdots\}
\]

we compile \(c\) into

\[
\text{SELECT iter, 1 AS pos, NULL AS pre, c AS val FROM loop}_x
\]

in which

\[
\text{loop}_x = \text{SELECT iter FROM } q_x(\$v_x)
\]

represents the iterations of the surrounding for-loop. The loop relation associated with the top-level scope \(s\) is loop = \(\text{iter} \in \{0\}^\ast\).

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

We will derive the compiled representation of a free variable in scope \(s_{x,y}\) 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 \(s_0\)):

\[
q_{x,y}(\$v_{x,y}) = \text{SELECT row() AS iter, 1 AS pos, pre, val FROM } q_x(e_{x,y}) \text{ ORDER BY iter, pos .}
\]

We assume the presence of a built-in function \(\text{row}()\) which densely numbers the tuples of an ordered table starting from 0. Section 4 discusses two possible implementations of \(\text{row}()\).
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

\[ \Gamma; \text{loop} \vdash e \Rightarrow (q, \text{doc'}) \]

indicates that, given

1. The initial environment \( \Gamma \) may already contain bindings if external variables have been defined in the input query.
2. The environment \( \Gamma \) maps each XQuery variable to an SQL relation.
3. The loop relation \( \text{loop} \) is defined as the SQL relation \( \text{loop} \).
4. The XQuery expression \( e \) compiles into the SQL query \( q \) with a new table of live nodes \( \text{doc'} \).
5. New live nodes are created by XQuery’s element constructors only, otherwise we have \( \text{doc} = \text{doc'} \).
6. The XQuery expression \( e \) compiles into the SQL query \( q \) with a new table of live nodes \( \text{doc'} \).
7. New live nodes are created by XQuery’s element constructors only, otherwise we have \( \text{doc} = \text{doc'} \).
8. The XQuery expression \( e \) compiles into the SQL query \( q \) with a new table of live nodes \( \text{doc'} \).
9. New live nodes are created by XQuery’s element constructors only, otherwise we have \( \text{doc} = \text{doc'} \).
10. The initial environment \( \Gamma \) may already contain bindings if external variables have been defined in the input query.

![Figure 7: Syntax of XQuery Core subset.](image)

Other expression types. The compilation procedure ensures that the correct \( \text{loop} \) relation and variable representations are available when an expression is compiled. Section 4 describes in which way (if any)
we can use an alternative SQL implementation based on an experimental setup (Section 6), (2) avoids early INTEGER overflow in the pos column, and (3) works correctly in case relation q1 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 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:

\[
\text{SELECT iter AS outer, iter * m.pos + e1.pos AS inner FROM q1 AS e1, (SELECT MAX(pos) AS pos FROM q1) AS m}
\]

\[
(q_e, \text{may be rewritten accordingly). To illustrate, given the iter and pos columns of relation q1 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 $(e_1, e_2, \ldots, e_n)$: (1) a new element node

---

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

<table>
<thead>
<tr>
<th>Node test n</th>
<th>Predicate test(v, n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>v.kind = &quot;elem&quot;</td>
</tr>
<tr>
<td>t (tag name)</td>
<td>v.kind = &quot;elem&quot; AND v.prop = &quot;t&quot;</td>
</tr>
<tr>
<td>text()</td>
<td>v.kind = &quot;text&quot;</td>
</tr>
<tr>
<td>node()</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

---

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 q1 will appear before items encoded in q2):

\[
\text{SELECT iter,}
\]

\[
\text{DENSE_RANK() OVER (PARTITION BY iter ORDER BY ord, pos) AS pos,}
\]

\[
\text{pre, val}
\]

\[
\text{FROM (SELECT *, 0 AS ord FROM q1}
\]

\[
\text{UNION ALL}
\]

\[
\text{SELECT *, 1 AS ord FROM q2)}
\]
Figure 8: XQuery to SQL8 compilation procedure.
Consider the query

\[
\text{let } v := e//b \text{ return element } r \{ v \}
\]

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 \( \text{doc} \) table:

1. the new root node \( r \) is appended and assigned the next available preorder rank \( \text{MAX}(\text{pre} + \text{size}) + 1 \),
2. the nodes in the affected subtrees are appended to \( \text{doc} \) (with \( \text{pre} \geq \text{MAX}(\text{pre} + \text{size}) + 2 \)) with their \( \text{size}, \text{kind}, \) and \( \text{prop} \) properties unchanged, and their \( \text{level} \) 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 \( \text{pre} \) column and an overestimation of the \( \text{size} \) 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 \textit{effective boolean value} [2] of a sequence in different iterations via the absence or presence of \( \text{iter} \) values in its encoding (Figures 12(a) and 12(b)). This enables the compilation of XQuery’s \textit{conditional expression} \( \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \) as shown in Figure 12(c).

With the conditional available, the language subset may be further extended by (1) \textit{predicates} \( e [c] \), (2) the existential and universal \textit{quantifiers} \( \text{some, every} \), (3) the \textit{general comparison operators} for sequences, and (4) the XQuery \textit{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 \( \text{doc} \). The new current table of live nodes, computed in Rule \text{ELEM} via SQL’s \text{UNION ALL} operator, may thus be written as the \textit{disjoint union}

\[
\text{doc} \cup \Delta
\]

where \( \text{doc} \) is the table of persistent XML nodes and \( \Delta \) denotes the transient nodes in the new fragment.

Now consider the evaluation of a second element constructor \textit{element} \( t \{ e \} \) with \( e \subseteq \text{doc} \cup \Delta \). Rule \text{ELEM} performs the XPath location step

\[
e/\text{descendant-or-self::node()}
\]

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:

\[
(e \cap \text{doc})/\text{descendant-or-self::node()}
\cup
(e \cap \Delta)/\text{descendant-or-self::node()}
\]

Although more complex at first sight, this variant performs the bulk of the work\(^6\) on the persistent \( \text{doc} \) table and thus can fully benefit from the presence of indexes (Section 6). The former variant, on the other hand, has to evaluate the \textit{descendant-or-self} axis step on the derived table \( \text{doc} \cup \Delta \) which lacks index support.

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

\[
(\text{doc} \cup \Delta) \cup \Delta' = \text{doc} \cup (\Delta \cup \Delta')
\]

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 \text{ELEM} handles the general case.

\(^6\)Typically, \( |\Delta| \ll |\text{doc}| \).
usually comprised of complex XML documents, XPath path expressions are

\[
thankyou{5.2} \text{ Bundling XPath Steps}
\]

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):

\[
e / \alpha_1 : n_1 / \alpha_2 : n_2 / \cdots / \alpha_k : n_k. \quad (Q_3)
\]

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

\[
( \cdots ((e / \alpha_1 : n_1) / \alpha_2 : n_2) / \cdots ) / \alpha_k : n_k
\]

which also suggests the evaluation mode of such a multi-step path. Proceeding from left to right, the \( i \)th 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):

\[
\text{distinct-doc-order}()
\begin{align*}
& \text{for } \$v_1 \text{ in } e \text{ return } \\
& \text{distinct-doc-order}()
\begin{align*}
& \text{for } \$v_2 \text{ in } \$v_1 / \alpha_1 : n_1 \text{ return } \\
& \$v_2 / \alpha_2 : n_2)
\end{align*}
\end{align*}
\]

Note that Rule \text{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 \textit{whole context node sequence}. In a sense, Rule \text{STEP} implements the above iteration implicitly via the self-join of table \textit{doc}.

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

(1) joins the current context node sequence with \textit{doc} to retrieve the necessary context node properties (only the preorder rank property \textit{pre} is available in the sequence encoding),

(2) performs the \textit{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 \textit{a whole} [9, 10]. In a modified compiler, Rule \text{STEP} is replaced by a new Rule \text{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 \textit{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

\[
\text{DENSE\_RANK() (PARTITION BY iter ORDER BY iter, pos)}
\]

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

\[
\text{DENSE\_RANK() (ORDER BY iter, pos)}
\]

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 and thus is a typical candidate for the disjoint fragments optimization of Section 5.1:

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

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

<table>
<thead>
<tr>
<th>Optimization</th>
<th>1.1 MB</th>
<th>11 MB</th>
<th>55 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>no optimization</td>
<td>1.1</td>
<td>48.7</td>
<td>1088</td>
</tr>
<tr>
<td>fragment disjointness</td>
<td>0.31</td>
<td>2.9</td>
<td>14.7</td>
</tr>
</tbody>
</table>

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

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

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 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 back-mapping 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