Pathfinder: XQuery—The Relational Way

Peter Boncz\textsuperscript{1} Torsten Grust\textsuperscript{2} Maurice van Keulen\textsuperscript{3}
Stefan Manegold\textsuperscript{1} Jan Rittinger\textsuperscript{1,4,}\dagger Jens Teubner\textsuperscript{2}

\textsuperscript{1}CWI Amsterdam, The Netherlands \{boncz,manegold\}@cwi.nl
\textsuperscript{2}Technische Universität München, Germany \{torsten.grust,jens.teubner\}@in.tum.de
\textsuperscript{3}University of Twente, The Netherlands m.vankeulen@utwente.nl
\textsuperscript{4}University of Konstanz, Germany rittinge@inf.uni-koenstanz.de

1 Introduction

Relational query processors are probably the best understood (as well as the best engineered) query engines available today. Although carefully tuned to process instances of the relational model (tables of tuples), these processors can also provide a foundation for the evaluation of “alien” (non-relational) query languages: if a relational encoding of the alien data model and its associated query language is given, the RDBMS may act like a special-purpose processor for the new language.

This demonstration features our XQuery compiler Pathfinder, the continuation of our earlier work on a purely relational XPath and XQuery processing stack [4, 6, 7] in which we developed relational encodings and processing strategies for the tree-shaped XML data model. The Pathfinder project is an exploration of how far we can push the idea of using mature RDBMS technology to design and build a full-fledged XQuery implementation. The demonstration will show that this line of research was and still is worth to be followed: based on the extensible relational database kernel MonetDB [2], Pathfinder provides highly efficient and scalable XQuery technology that scales beyond 10 GB XML input instances on commodity hardware.

Pathfinder requires only local extensions to the underlying DBMS’s kernel, such as the staircase join operator $\mathcal{J}$ [7, 9]. A join recognition logic in our compiler, as well as a careful consideration of order properties of relational operators [3], allow for effective optimizations that turn MonetDB into a highly efficient XQuery engine.

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 31st VLDB Conference, Trondheim, Norway, 2005

\textsuperscript{1}This work was supported by the DFG Research Training Group GK-1042 “Explorative Analysis and Visualization of large Information Spaces”.

Figure 1: XQuery development stack.

We will briefly recite these ideas in the upcoming section. Section 3 will provide experimental evidence of the viability of our approach, before we sketch the setup for this demonstration of Pathfinder.

2 Pathfinder XQuery Engine Overview

To bridge the apparent gap between the relational processing model and the XQuery data model, Pathfinder follows the step-by-step development stack of Figure 1.

Tree encoding. Relational XML processing starts off with a suitable relational encoding for XML document trees. We based our system on the XPath Accelerator [4], a numbering scheme in which each node $v$ is assigned a pre-order rank $\text{pre}(v)$ and a post-order rank $\text{post}(v)$. These numbers efficiently characterize XPath location steps as regions in a two-dimensional plane, e.g., $v'$ is a descendant of $v \iff \text{pre}(v) < \text{pre}(v') \land \text{post}(v') < \text{post}(v)$. Pathfinder employs an equivalent encoding variant that represents the location of a node as the 3-tuple $(\text{pre}(v), \text{size}(v), \text{level}(v))$ with the number of nodes in the subtree below $v$, size($v$), and $v$’s distance from the tree root, level($v$). These tuples are maintained in a table with schema $\text{pre}[\text{size}]\text{level}$.

XPath axes. This encoding scheme, turns XPath step evaluation into a relational range selection. The selection predicate describes region queries on the $(\text{pre}(v), \text{size}(v), \text{level}(v))$ space, where the region queried associates with the XPath axis [4]. These range selections are well supported by existing access and index structures.

Yet, the RDBMS gives away significant opportunities for optimization since the system is unaware of the
that helps to turn RDBMSs into functional query processors (e.g., various query optimizers). This complexity may significantly be reduced by peephole style optimization [5], though.

**Sequence encoding.** The XQuery data model is based on the sequence, an ordered (possibly heterogeneous) list of items. We implement sequence order by means of an explicit pos column, as depicted on the right. The polymorphic item column is efficiently implemented with help of MonetDB’s mposjoin operator.

**Loop lifting.** A crucial part in XQuery evaluation is the efficient implementation of its iteration primitive, the FLWOR clause. In [6], we developed a database style, bulk-oriented way of loop processing. We sketch the basic idea here.

The compilation from the iterative FLWOR clauses to bulk-oriented table manipulation is achieved through loop-lifting: a column iter added to the encoding in Figure 2 accounts for the different bindings a variable assumes when it iterates over a sequence. To illustrate, Figure 3 shows some intermediate results for the relational query plan that evaluates

\[
\text{for } \$v \text{ in } (10, 20), \text{for } \$w \text{ in } (100, 200) \text{ return } \$v + \$w.
\]

by peephole style optimization [5], though.

**Sequence encoding.** The XQuery data model is based on the sequence, an ordered (possibly heterogeneous) list of items. We implement sequence order by means of an explicit pos column, as depicted on the right. The polymorphic item column is efficiently implemented with help of MonetDB’s mposjoin operator.

**Loop lifting.** A crucial part in XQuery evaluation is the efficient implementation of its iteration primitive, the FLWOR clause. In [6], we developed a database style, bulk-oriented way of loop processing. We sketch the basic idea here.

The compilation from the iterative FLWOR clauses to bulk-oriented table manipulation is achieved through loop-lifting: a column iter added to the encoding in Figure 2 accounts for the different bindings a variable assumes when it iterates over a sequence. To illustrate, Figure 3 shows some intermediate results for the relational query plan that evaluates

\[
\text{for } \$v \text{ in } (10, 20), \text{for } \$w \text{ in } (100, 200) \text{ return } \$v + \$w.
\]

by peephole style optimization [5], though.

**Sequence encoding.** The XQuery data model is based on the sequence, an ordered (possibly heterogeneous) list of items. We implement sequence order by means of an explicit pos column, as depicted on the right. The polymorphic item column is efficiently implemented with help of MonetDB’s mposjoin operator.

**Loop lifting.** A crucial part in XQuery evaluation is the efficient implementation of its iteration primitive, the FLWOR clause. In [6], we developed a database style, bulk-oriented way of loop processing. We sketch the basic idea here.

The compilation from the iterative FLWOR clauses to bulk-oriented table manipulation is achieved through loop-lifting: a column iter added to the encoding in Figure 2 accounts for the different bindings a variable assumes when it iterates over a sequence. To illustrate, Figure 3 shows some intermediate results for the relational query plan that evaluates

\[
\text{for } \$v \text{ in } (10, 20), \text{for } \$w \text{ in } (100, 200) \text{ return } \$v + \$w.
\]

by peephole style optimization [5], though.

**Sequence encoding.** The XQuery data model is based on the sequence, an ordered (possibly heterogeneous) list of items. We implement sequence order by means of an explicit pos column, as depicted on the right. The polymorphic item column is efficiently implemented with help of MonetDB’s mposjoin operator.

**Loop lifting.** A crucial part in XQuery evaluation is the efficient implementation of its iteration primitive, the FLWOR clause. In [6], we developed a database style, bulk-oriented way of loop processing. We sketch the basic idea here.

The compilation from the iterative FLWOR clauses to bulk-oriented table manipulation is achieved through loop-lifting: a column iter added to the encoding in Figure 2 accounts for the different bindings a variable assumes when it iterates over a sequence. To illustrate, Figure 3 shows some intermediate results for the relational query plan that evaluates

\[
\text{for } \$v \text{ in } (10, 20), \text{for } \$w \text{ in } (100, 200) \text{ return } \$v + \$w.
\]

by peephole style optimization [5], though.

**Sequence encoding.** The XQuery data model is based on the sequence, an ordered (possibly heterogeneous) list of items. We implement sequence order by means of an explicit pos column, as depicted on the right. The polymorphic item column is efficiently implemented with help of MonetDB’s mposjoin operator.

**Loop lifting.** A crucial part in XQuery evaluation is the efficient implementation of its iteration primitive, the FLWOR clause. In [6], we developed a database style, bulk-oriented way of loop processing. We sketch the basic idea here.

The compilation from the iterative FLWOR clauses to bulk-oriented table manipulation is achieved through loop-lifting: a column iter added to the encoding in Figure 2 accounts for the different bindings a variable assumes when it iterates over a sequence. To illustrate, Figure 3 shows some intermediate results for the relational query plan that evaluates

\[
\text{for } \$v \text{ in } (10, 20), \text{for } \$w \text{ in } (100, 200) \text{ return } \$v + \$w.
\]

by peephole style optimization [5], though.

**Sequence encoding.** The XQuery data model is based on the sequence, an ordered (possibly heterogeneous) list of items. We implement sequence order by means of an explicit pos column, as depicted on the right. The polymorphic item column is efficiently implemented with help of MonetDB’s mposjoin operator.

**Loop lifting.** A crucial part in XQuery evaluation is the efficient implementation of its iteration primitive, the FLWOR clause. In [6], we developed a database style, bulk-oriented way of loop processing. We sketch the basic idea here.

The compilation from the iterative FLWOR clauses to bulk-oriented table manipulation is achieved through loop-lifting: a column iter added to the encoding in Figure 2 accounts for the different bindings a variable assumes when it iterates over a sequence. To illustrate, Figure 3 shows some intermediate results for the relational query plan that evaluates
MonetDB. We designed Pathfinder as an XQuery compiler: XQuery expressions are translated into relational query plans. Their execution on a database back-end evaluates the input expression by means of a single algebraic query. A simple post-processor then serializes the relational result to form a response in terms of the XQuery data model.

The compiler currently targets MonetDB, an extensible relational database system, available in open source [2]. Its main-memory optimized implementation and features like virtual object identifiers (that, e.g., make the row-numbering operator \( g \) a no-cost operator) make MonetDB particularly suitable for our objective: scalable XQuery processing. The use of alternative back-ends (e.g., SQL) is current work in progress.

3 Scalability and Performance

Our implementation is meant to assess the viability of using relational back-ends for XQuery. RDBMSs are known to scale well with increasing data volumes—an inevitable feature if XQuery systems are to support large amounts of XML data.

We used the XML generator XMLgen from the XMark benchmark set [10] to create XML instances of sizes from 11 MB to 11 GB (50,000 to 500 million nodes, respectively). Documents were loaded into our MonetDB back-end to run the 20 XMark queries.

3.1 Storage Overhead

Pathfinder stores the structural part of XML documents in the pre\[size\]level encoding, as sketched in Section 2. A prop column stores surrogates of XML node properties. Actual property values (tag names, text node content, etc.) are maintained in separate property BATs and kept unique therein. These node properties are identified by their surrogates, where nodes with identical properties share the same surrogate.

This surrogate sharing not only avoids expensive string comparisons at query execution time, but also reduces space consumption on secondary storage. For the XMark benchmark set, disk space requirements range between 147\% (11 MB instance) and 125\% (110 MB instance) of the original XML document.\(^1\)

\(^1\)For larger instances, XMark documents tend to degrade: more and more duplicate text nodes bring down disk space usage below 80\% of the original XML instance.

<table>
<thead>
<tr>
<th>11 MB</th>
<th>110 MB</th>
<th>1.1 GB</th>
<th>11 GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>X-Hive</td>
<td>X-Hive</td>
<td>X-Hive</td>
</tr>
<tr>
<td></td>
<td>PF</td>
<td>PF</td>
<td>PF</td>
</tr>
<tr>
<td>1</td>
<td>0.37</td>
<td>0.05</td>
<td>1.29</td>
</tr>
<tr>
<td>2</td>
<td>0.45</td>
<td>0.07</td>
<td>1.75</td>
</tr>
<tr>
<td>3</td>
<td>0.65</td>
<td>0.28</td>
<td>5.66</td>
</tr>
<tr>
<td>4</td>
<td>0.10</td>
<td>0.08</td>
<td>1.00</td>
</tr>
<tr>
<td>5</td>
<td>0.13</td>
<td>0.05</td>
<td>2.01</td>
</tr>
<tr>
<td>6</td>
<td>1.07</td>
<td>0.02</td>
<td>10.17</td>
</tr>
<tr>
<td>7</td>
<td>1.57</td>
<td>0.03</td>
<td>24.84</td>
</tr>
<tr>
<td>8</td>
<td>0.85</td>
<td>0.14</td>
<td>3.51</td>
</tr>
<tr>
<td>9</td>
<td>32.25</td>
<td>0.20</td>
<td>12200.66</td>
</tr>
<tr>
<td>10</td>
<td>5.28</td>
<td>0.80</td>
<td>442.37</td>
</tr>
<tr>
<td>11</td>
<td>99.91</td>
<td>1.18</td>
<td>19927.29</td>
</tr>
<tr>
<td>12</td>
<td>23.39</td>
<td>0.11</td>
<td>5100.19</td>
</tr>
<tr>
<td>13</td>
<td>0.10</td>
<td>0.07</td>
<td>1.03</td>
</tr>
<tr>
<td>14</td>
<td>0.72</td>
<td>0.26</td>
<td>11.16</td>
</tr>
<tr>
<td>15</td>
<td>0.03</td>
<td>0.09</td>
<td>0.49</td>
</tr>
<tr>
<td>16</td>
<td>0.03</td>
<td>0.11</td>
<td>0.52</td>
</tr>
<tr>
<td>17</td>
<td>0.09</td>
<td>0.07</td>
<td>0.85</td>
</tr>
<tr>
<td>18</td>
<td>0.08</td>
<td>0.04</td>
<td>0.64</td>
</tr>
<tr>
<td>19</td>
<td>0.67</td>
<td>0.11</td>
<td>12.15</td>
</tr>
<tr>
<td>20</td>
<td>0.11</td>
<td>0.24</td>
<td>1.40</td>
</tr>
</tbody>
</table>

Table 3: Query evaluation times (in seconds) for different XMark instance sizes (scale factors 0.1 to 100).

3.2 Experimental Setup

We tested Pathfinder’s performance on a 1.6 GHz AMD Opteron system, equipped with 8 GB RAM. For comparison, we installed X-Hive/DB on the same system [8]. X-Hive appears to be one of the faster XML database systems and copes with large XML instances quite well. We tuned the X-Hive database with value indices on the buyer/@person and profile/@income paths. These indices significantly reduced execution times of some computationally intensive XMark join queries (e.g., Q8).

3.3 Pathfinder Performance

The execution times, shown in Table 3, confirm the strengths of our purely relational approach to XQuery evaluation. Pathfinder handles simple path queries (Q1–Q5, Q13–Q20) 2 to 20 times faster than X-Hive. If paths include recursive axes, we benefit from our staircase join implementation that outperforms X-Hive by two orders of magnitude (Q6 and Q7).

It is not surprising that the join queries (Q8–Q12) benefit most from our relational back-end. While X-Hive almost competes with Pathfinder for the simple join query Q8, its performance strongly degrades if intermediate query results are to be joined. Pathfinder compiles these queries into join plans [3] and takes advantage of efficient join implementations in our back-end. It is thus able to execute queries in reasonable time which we were not able to run on our X-Hive installation (Q9–Q12).

3.4 Scalability

To assess the scalability of our approach, we printed XMark execution times normalized to the elapsed times for the 110 MB XML instance. As shown in Figure 4, Pathfinder scales linearly for most of the queries. Even the join queries (Q8–Q10) are of almost linear complexity.
Figure 4: Pathfinder scalability: Execution times for the XMark benchmark set, normalized to timings for the 110 MB instance.

The only two exceptions are queries Q11/Q12. Their quadratic scaling stems from a theta-join (predicate $\sigma$) that produces intermediate results of about 120 k, 12 M, and 1.2 G tuples for the 11 MB, 110 MB, and 1.1 GB instances, respectively. Note that this concerns the query result, whose computation cannot be avoided. Thus, any XQuery implementation will face this complexity.

4 Demonstration Setup

Our system operates in a front-end/back-end fashion. The front-end, the Pathfinder compiler, parses incoming XQuery expressions and translates them into a relational algebra expression tree, represented in terms of a MIL\(^2\) program. The code is shipped to a MonetDB server that then executes the entire query and serializes the encoding of the result.

The demonstration features both of these components, with various interesting hooks to trace the execution of XQuery expressions on our relational backend. Our system implements the W3C October 2004 working drafts [1], including schema import, static typing, and full axis features.

The compiler provides facilities to look “under the hood” of relational XQuery compilation. An output of type-annotated XQuery Core expression equivalents demonstrates optimization techniques applied to the incoming XQuery code before compiling it into relational algebra. A graphical output of relational query plans at different compilation stages (much like the plan in Figure 5) illustrates the loop-lifting concept. Relational plans may be traced to reveal the result computed for any subexpression.

Our demonstration system will be pre-loaded with XMark [10] instances of different sizes up to 1.1 GB. The query texts for the 20 benchmark queries will be ready to run, but users may as well state their own ad hoc queries.

The Pathfinder compiler is available in open source as part of the MonetDB/XQuery implementation via http://pathfinder-xquery.org/.

\(^2\)MIL stands for MonetDB Interpreter Language

Figure 5: Relational query plan that evaluates the query for $\sigma v in (10,20) return v + 100$.

References


