Pathfinder/MonetDB:
A High-Performance Relational Runtime for XQuery

Jan Rittinger*

University of Konstanz, Department of Computer & Information Science,
P.O.Box D188, 78457 Konstanz, Germany

Jan.Rittinger@uni-konstanz.de

Pathfinder/MonetDB is a collaborative effort of the University of Konstanz, the University of Twente, and the Centrum voor Wiskunde en Informatica (CWI) in Amsterdam to develop an XQuery compiler that targets an RDBMS back-end. The author of this abstract is student at the University of Konstanz and spent six months as an intern at the CWI, designing and implementing a translation of XQuery Core to (a variant of) relational algebra. His work continues in the research group at the University of Konstanz.

Pathfinder/MonetDB

Pathfinder/MonetDB can be divided into two parts: Pathfinder and MonetDB. Pathfinder is an XQuery compiler (see Figure 1) that translates XQuery expressions into a variant of relational algebra which is executable by the back-end database MonetDB. MonetDB is an extensible main-memory database system kernel which adapts database architecture concepts to the characteristics of modern hardware in order to improve the CPU and memory cache utilization [2].

The MonetDB kernel is equipped with a low-level interface language MIL (MonetDB Interpreter Language), which forms the target language for a number of different front-end query languages (*e.g.*, SQL, OQL, or XQuery). The table manipulation operations in MIL form a closed algebra on the binary table model. MIL can be extended with new primitives, data types, and associated search accelerator structures and contains a computationally complete procedural language. Pathfinder uses these features to extend MonetDB with an XQuery specific runtime module implementing a small number of additional operations (*e.g.*, XML serialization or staircase join [8]).
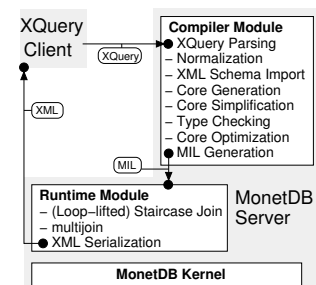

Figure 1: System architecture.

Compiling XQuery to Relational Algebra

The W3C proposes XQuery [1] as the standard query language for XML data. XQuery contains constructs to explore the *tree-structured* XML data model (plus constructors to create new XML fragments) as well as iteration primitives—notably the FLWOR block—to process *sequences* of data items. Earlier work in the context of the Pathfinder project provides the means to close the apparent gap between the set-oriented relational data model and the two principle data types which form the backbone of the XQuery data model, namely *ordered, unranked trees of nodes* and *ordered, finite sequences of items*.

We translate trees of nodes into a relational encoding—the XPath accelerator—applying the ideas presented in [5]. Item sequences, on the other hand, may be transformed using techniques originally developed for a mapping from XQuery to SQL [6, 7]. The inference rules described in these papers form a mapping from XQuery Core to an (almost) standard relational algebra. MIL perfectly supports these operators and also allows for a tight integration of extensions (*e.g.*, staircase join [8]) which are geared to support the embedded XPath sub-language. In this work, we significantly extended the compiler described in [6] to support a wide variety of XQuery constructs.

Optimizing Relational XQuery Evaluation — XQuery Join Recognition

The compilation procedure originally described in [6] opens several opportunities to improve, among them the recognition and translation of joins. Because the XQuery language lacks an explicit join oper-

---

ator, the only implicit way to perform a join is to use nested loops with an embedded filter expression. Unfortunately, the compiler described in [6] always emits algebraic Cartesian products for such nested `for`–loops and the implicit XQuery join remains undetected. Query Q_1 below shows a typical XQuery scenario which implicitly joins the two sequences (30,20,10) and (1,2,3). The original translation strategy forms the Cartesian product of the two sequence representations (yielding an intermediate result of 9 rows), despite the fact that the final query result is linear in the size of the input sequences.

```
for $u in (30, 20, 10),
    $v in (1, 2, 3)
where $u eq ($v * 10)        (Q₁)
return "hit"
```

This inefficiency may be avoided by an XQuery compilation procedure that recognizes XQuery joins. The compiler matches a given general XQuery Core pattern and, in case of a successful match, translates the XQuery expression into an relational equivalent which makes use of an algebraic join. In the current compiler, join recognition solely works at the level of XQuery Core and matches the following query pattern:

$$\text{for } \$v \text{ in } e_{in} \text{ return if } (p(e_1, e_2)) \text{ then } e_{\text{return}} \text{ else } () \;. \qquad (P_1)$$

There are some preconditions which have to be met to guarantee that the above syntactic pattern actually describes a join (in a nutshell, the conditions below guarantee that the join inputs are indeed *independent* of each other). To be more concrete, the pattern P_1 above qualifies for an XQuery join, only if

(i) variable $\$v$ appears free in e_2 only[1],

(ii) variables occurring free in e_2 and e_{in} are bound in any enclosing scope, except for the scope that *directly* encloses P_1, and

(iii) predicate p is supported by the theta-join implementation of the relational back-end (*i.e.*, typically, p will be `eq`, `=`, `lt`, `<`, ...).

```
for $u in (30, 20, 10)
return $u
              and             (Q₂)
for $v in (1, 2, 3)
return $v * 10
```

Since all restrictions hold for Query Q_1, this query can be split into two independent sub-queries (Q_2) which compute the join inputs: The final overall translation does not involve Cartesian products at all.

The gory details and a formal description of the join compilation can be captured by means of an inference rule similar to the ones gives in [6]. This work is currently under submission.

Performance Results

To show that our relational approach may indeed yield a high-performance XQuery processor, we conducted experiments[2] in which we focused on XMark [9], as the most frequently used benchmark for evaluating XQuery efficiency and scalability. We performed measurements at scaling factors 0.1, 1 and 10 (which yield documents of respectively 10 MB, 100 MB, and 1 GB), using Pathfinder/MonetDB as well as the latest versions of Galax (0.4.0) [4] and X-Hive (6.0) [10].

XQuery Join Recognition. In our first runs, Pathfinder/MonetDB was not able to evaluate the XMark join queries Q8–Q12 on the 100 MB and 1 GB sizes, due to excessive running time and resource consumption.

The reason was the generation of huge intermediate Cartesian products. Figure 2 contrasts the results for the 10 MB document with the performance we obtained with join recognition enabled in our MIL generation. It is obvious that the execution of XQuery statements with join predicates simply *requires* join recognition when the query is run on significant XML document sizes.
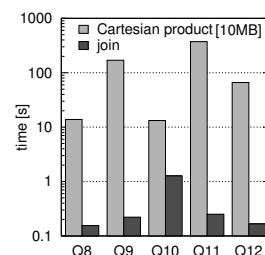
Figure 2: Benefits of XQuery join recognition.

Scalability. In Figure 3 all numbers are normalized to the elapsed time on the 100 MB document. The figure clearly shows that Pathfinder/MonetDB scales linearly with document size. The only outliers are query Q11 and Q12. The culprit in this queries is a theta-join predicate ($>$) which generates an intermediate result almost the size of the Cartesian product of its inputs. Note that this concerns the query

---

[1]The roles of e_1, e_2 may be arbitrarily swapped.

[2]The platform was a 1.6 GHz AMD Opteron 242 (1 MB L2 cache) processor with 8 GB RAM and a RAID-5 disk subsystem (3ware 7810, configured with eight 250 GB IDE disks of 7200 RPM). The operating system was Linux 2.6.9, using a 64-bit address space.

result, whose computation cannot be avoided (though the end result becomes small, due to subsequent aggregation) and thus *any* XQuery processor will face this problem.

Figure 3: Scaling with respect to document size.

System Comparison. For our own system, we generated MIL query plans[3], which range from 318 lines (XMark Query Q6) to 11,509 lines (Q10) with an average of about 2,000 lines. The next step, the document loading[4], is followed by the query evaluation and the serialization[5]. Our two reference systems are Galax, which is the most popular "native" XQuery engine available in open-source, and X-Hive, which is one of the faster native XML database systems (shown in [3]). The performance results of X-Hive and Pathfinder/MonetDB contain only query evaluation times, while Galax still[6] includes serialization times.

The table on the right side shows our full experimental results (elapsed time in seconds). Galax failed to process the queries once the XMark documents were of size 100 MB or larger. Compared to our system Galax is marginally faster on queries Q2, Q5, Q13 (obvious, as we use it as base reference), Q16, and Q19. X-Hive also finishes the execution of non-join queries in reasonable time. For the join queries (Q8–Q12) both Galax and X-Hive are dominated by the Cartesian products. X-Hive avoids quadratic complexity in Q8 due to the value indices we created. However if the queries join *intermediate* query results, indices cannot be used and performance degrades strongly. With the help of join recognition, Pathfinder/MonetDB clearly outperforms the other two systems on these queries.

Conclusion

This present work builds on both, an XPath-aware relational encoding of XML trees and a relational XQuery compiler, to turn a relational database back-end into an XQuery processor. The outcome is a prototype implementation backed by the extensible MonetDB RDBMS which exhibits the efficiency and scalability provided by a relational database. XML input documents of 1 GB size and beyond can be queried in interactive time.

| | 10 MB | | | 100 MB | | 1 GB | |
Q	Galax	X-Hive	PF/M	X-Hive	PF/M	X-Hive	PF/M
1	0.13	0.37	0.04	1.29	0.37	9.9	4.1
2	0.08	0.45	0.09	1.75	0.72	33.0	6.5
3	0.16	0.65	0.29	5.66	2.63	25.0	22.2
4	0.36	0.10	0.10	0.99	1.00	18.1	10.6
5	0.02	0.13	0.06	1.17	0.43	20.7	4.2
6	1.27	1.07	0.06	10.17	0.54	178.0	5.4
7	2.87	1.57	0.12	24.84	1.09	278.4	10.4
8	127.28	0.85	0.16	3.51	1.38	49.1	18.2
9	142.74	32.25	0.22	2280.66	1.81	DNF	22.2
10	18.16	5.28	1.27	442.37	13.68	DNF	150.6
11	218.23	98.91	0.25	9927.29	6.29	DNF	683.5
12	63.66	23.39	0.17	5100.19	2.98	DNF	347.7
13	—	0.09	0.07	1.03	0.53	12.9	5.2
14	2.13	0.72	0.17	11.17	1.69	110.2	16.9
15	0.08	0.03	0.09	0.49	0.80	10.6	7.8
16	0.06	0.03	0.10	0.51	0.87	10.9	8.2
17	0.14	0.09	0.10	0.85	0.80	11.8	7.6
18	0.03	0.08	0.05	0.64	0.43	14.8	4.2
19	1.32	0.67	0.21	12.15	1.70	254.5	16.9
20	0.52	0.11	0.21	1.40	1.79	24.6	19.0

References

[1] S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. World Wide Web Consortium, Oct. 2004. http://www.w3.org/TR/xquery/.

[2] P. Boncz and M. Kersten. MIL Primitives For Querying a Fragmented World. *The VLDB Journal*, 8(2):101–119, Mar. 1999.

[3] D. DeHaan, D. Toman, M. Consens, and M. Özsu. A Comprehensive XQuery to SQL Translation Using Dynamic Interval Encoding. In *Proc. SIGMOD Conf.*, pages 623–634, San Diego, CA, USA, June 2003.

[4] M. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax Experience. In *Proc. VLDB Conf.*, pages 1077–1080, Berlin, Germany, Sept. 2003.

[5] T. Grust. Accelerating XPath Location Steps. In *Proc. SIGMOD Conf.*, pages 109–120, Madison, WI, USA, June 2002.

[6] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. VLDB Conf.*, pages 252–263, Toronto, Canada, Sept. 2004.

[7] T. Grust and J. Teubner. Relational Algebra: Mother Tongue—XQuery: Fluent. In *Twente Data Management Workshop on XML Databases and Information Retrieval (TDM)*, pages 7–14, Enschede, The Netherlands, June 2004.

[8] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. VLDB Conf.*, pages 524–535, Berlin, Germany, Sept. 2003.

[9] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB Conf.*, pages 974–985, Hong Kong, China, Aug. 2002.

[10] X-Hive/DB. http://www.x-hive.com/.

---

[3]The times for running the XQuery compiler varies between 60 and 100 ms for all XMark queries. The compiler timings are excluded from our performance results.

[4]The XML loader of Pathfinder/MonetDB imports the 10 MB XMark document in 1.15 seconds.

[5]Serialization times are excluded from the performance numbers. For the 10 MB instance they are all below 50 ms (except Q10: 690 ms).

[6]Galax is a file-oriented system that parses the XML file on each query. To (over-)compensate for XML parsing time, we subtracted 8.25 seconds (which is the time of the fastest query Q13) fro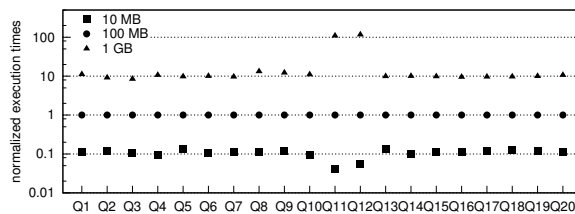m all Galax performance figures.