A SQL:1999 Code Generator for the Pathfinder XQuery Compiler

Torsten Grust° Jan Rittinger° Sherif Sakr• Jens Teubner°

°Technische Universität München
Munich, Germany
torsten.grust | jan.rittinger | jens.teubner@in.tum.de

•University of Konstanz
Konstanz, Germany
sakr@inf.uni-konstanz.de

ABSTRACT

The *Pathfinder* XQuery compiler has been enhanced by a new code generator that can target any SQL:1999-compliant relational database system (RDBMS). This code generator marks an important next step towards truly *relational XQuery processing*, a branch of database technology that aims to turn RDBMSs into highly efficient XML and XQuery processors without the need to invade the relational database kernel. *Pathfinder*, the retargetable front-end compiler, translates input XQuery expressions into DAG-shaped relational algebra plans. The code generator then turns these plans into sequences of either SQL:1999 statements or view definitions which jointly implement the (sometimes intricate) XQuery semantics. In a sense, this demonstration thus lets relational algebra and SQL swap their traditional roles in database query processing. The result is a code generator that (1) supports an almost complete dialect of XQuery, (2) can target any RDBMS with a SQL:1999 language interface, and (3) exhibits quite promising performance characteristics when run against high-volume XML data as well as complex XQuery expressions.

1. PURELY RELATIONAL XQUERY PROCESSING

The *Pathfinder* project develops *purely* relational XQuery processing technology under the main hypothesis that RDBMSs, while originally built to operate over table-shaped data, provide perfect infrastructure to efficiently process high-volume XML instances [2]. In the present work, we particularly emphasize "relational purity" and demonstrate a code generator for the *Pathfinder* XQuery compiler that emits strictly standard-compliant SQL:1999 statements. To faithfully implement the XQuery semantics, these statements manipulate relational encodings of XML documents (or fragments thereof) as well as ordered item sequences [7], the two principal building blocks of the XQuery data model [1]. The result is a compiler that can translate the XQuery dialect sketched in Table 1 and is able to target *any* SQL:1999-ready RDBMS. In particular, note that

(1) there is *no* need for the relational database back-end to support the SQL/XML [8] standard or to provide an `XML` column type of any kind, and that
(2) the relational database kernel remains untainted: *no* additional query processing operators (*e.g.*, structural join algorithms) are injected and *no* specific gadgets for XML storage [9] are required.

Instead, the *Pathfinder*-generated code relies on functionality already built into relational database back-ends. Among these are partitioned B-trees to accelerate XPath location step evaluation (*Pathfinder* implements XQuery's optional *full axis* feature) and SQL:1999's row numbering capabilities to support concepts like sequence order that are central to the XQuery semantics.

Most importantly, the demonstration makes the point that this purely relational approach to XQuery processing does not take second place behind specifically engineered native XQuery support in RDBMSs—quite the contrary (see Section 4). The *Pathfinder* XQuery compiler is available for download at `www.pathfinder-xquery.org`.

atomic literals	document order (e_1 `>>` e_2)
sequences (e_1 `,` e_2)	node identity (e_1 `is` e_2)
variables (`$`v)	arithmetics (`+`,`-`,`*`,`idiv`,..)
`let $`v `:=` e_1 `return` e_2	(general) comparisons (`=`, `eq`, ..)
`for $`v [`at $`p] `in` e_1 `return` e_2	Boolean connectives (`and`, `or`)
`if (`e_1`) then` e_2 `else` e_3	user-defined functions
e_1 `order by` e_2,..,e_n	`fn:doc(·)`, `fn:root(·)`, `fn:data(·)`
`unordered {`e`}`	`fn:id(·)`, `fn:idref(·)`
`element {`e_1`} {`e_2`}`	`fn:distinct-values(·)`
`attribute {`e_1`} {`e_2`}`	`op:union(·)`, `op:intersect(·)`
`text {`e`}`	`fn:count(·)`, `fn:sum(·)`, `fn:max(·)`
XPath (e_1`/`s`[`e_2`]`)	`fn:position()`, `fn:last()`
`typeswitch (`e_1`) case [$`v `as]` t `return` e_2..`default return` e_n	

Table 1: XQuery dialect supported by *Pathfinder* (s: XPath step, t: sequence type). Only a subset of the built-in functions (namespaces `fn`, `op`) shown.

2. SQL:1999 CODE GENERATION

Intermediate language: relational algebra. In *Pathfinder*, a rather classical variant of relational algebra assumes the role of the compiler's intermediate language. To translate the expression-oriented, almost functional XQuery language with its explicit variable binding and iteration constructs (`let`, `for`), *Pathfinder* relies on a compilation technique known as *loop-lifting* [7]. In a nutshell, loop-lifting "unrolls" XQuery FLWOR blocks to expose inter-iteration

data parallelism which the compiler then maps onto the bulk-processing primitives of relational algebra. This choice of intermediate language has its particular strengths. On the one hand, the semantics of relational algebra are well-defined and independent of any particular database back-end. *Pathfinder* back-ends for the *MonetDB* column-oriented RDBMS and the *Idefix* XML-aware file system layer are described in [2] and [4], respectively. On the other hand, the algebraic primitives model the internals of RDBMS query engines sufficiently exact such that the generation of efficient code remains feasible. To further facilitate the latter, *Pathfinder*'s relational algebra adheres to restrictions that are inspired by the actual processing capabilities of SQL-centric database kernels: column projection (π) does not eliminate duplicate rows, for example.

The resulting loop-lifted algebraic plans typically exhibit a wealth of sharing opportunities and thus are maintained as plan DAGs. *Pathfinder* makes aggressive use of relational properties—*e.g.*, keys, functional and multi-valued dependencies, inclusion of active attribute domains—to reduce the size of the plan DAGs (cutting down the typical operator count of 30–300 by more than 50 %) and to realize other plan enhancements. A number of non-trivial XQuery-specific optimizations are compactly described using relational algebra. In algebraic plans derived by loop-lifting, a robust, non-syntactical detection of value-based joins in XQuery may be based on a simple analysis of multi-valued dependencies [5] and an elegant implementation of `unordered{}` and `fn:unordered()` can be formulated in terms of projection pushdown [6].

Target language: SQL:1999. Due to the compositionality of the XQuery language in which all constructs nest orthogonally as long as typing rules are obeyed, plan shapes significantly diverge from the well-known π–σ–$\bowtie$ pattern generated by SQL compilers. This observation led to a code generation approach where the algebraic plans are chopped to let the RDBMS back-end evaluate the overall plan in separate chunks.

To identify suitable plan chunks, the code generator walks the plan DAG to find *basic blocks*—straight-line operator sequences with no sideways entries—much like compilers for programming languages. Inside these basic blocks, *Pathfinder* applies template instantiation to collapse groups of adjacent operators: a template describes a group of algebraic operators that can be equivalently implemented by a single SQL statement.

Pathfinder issues SQL:1999 code in one of three modes:

(1) each instantiated template is compiled into a SQL:1999 statement that, upon execution, populates a temporary result table exempt from logging or other transactional overhead (strategy ■ in Figure 1),
(2) instantiations are compiled into SQL:1999 view definitions plus a single query that drives evaluation (strategy ▩), or
(3) instantiations yield SQL:1999 view definitions as in (2) except when the code generator decides that a group of view definitions leads to a nested SQL query so complex that intermediate materialization as in (1) is beneficial (strategy □).

The statements are assembled into a SQL script whose execution evaluates the input XQuery expression. Note that both, shared temporary tables as well as view definitions,

SQL Construct	Used by/for (Remarks)
`ROW_NUMBER() OVER (PARTITION BY a ORDER BY b)`	implementation of XQuery order semantics
`UNION ALL`	item sequence construction
`[RIGHT OUTER] JOIN`	(predominantly equi-joins)
aggregates (`SUM()`, `MAX()`, ...)	maintain node encoding
range conditions	XPath location step evaluation
type casting (`CAST a AS` τ)	polymorphic item sequences
(non-logged) temporary tables	full materialization strategy
nested queries in `FROM` clauses	basic block forming

Table 2: Excerpt of SQL constructs used by the code generator (a,b: column names, τ: SQL data type).

can reflect the substantial degree of sharing in the DAG-shaped algebraic plans emitted by the compiler front-end.

Table 2 lists some of the SQL:1999 constructs used by the code generator. The resulting SQL queries are reasonably "good-natured", *e.g.*, all `UNION` operations are over disjoint tables, nested queries in `FROM` clauses are uncorrelated, and most of the occurring `JOIN` operators are equi-joins that implement the behavior of nested `for`-iteration scopes [7]. Further θ-joins are only introduced by XQuery join detection (see above).

In the XQuery data model, *ordered* finite sequences of items are pervasive. The code generator employs the SQL:1999 `ROW_NUMBER()` OLAP ranking primitive to create and manipulate tabular representations of such ordered sequences. In *Pathfinder*, row ranks are interpreted as sequence positions (column pos) [1] as depicted here in the relational encoding of the item sequence ($i_1, i_2, \ldots, i_n$). It is critical that `ROW_NUMBER()`-based orderings are encoded in the data itself and thus may be communicated from statement to statement (which would be impossible with SQL's `ORDER BY` construct).

pos	item
2	i_2
n	i_n
$\vdots$	$\vdots$
1	i_1

Most perceivable implementations of `ROW_NUMBER()` introduce blocking sort operations in the final physical query execution plans. *Pathfinder*'s optimizer thus invests considerable effort to avoid row numbering wherever this is possible, for example in the scope of `unordered{}`, the existential semantics of general comparisons (`=`, `<`, ...), aggregate functions, *etc.* [6].

3. RDBMSs AS XQUERY RUNTIME ENVIRONMENTS

Pathfinder can operate over any node-based (one XML node $\widehat{=}$ one row) relational tree encoding that preserves node identity and document order. One such encoding yields a table with schema pre|size|level which, for each node v, records v's rank in document order, the size of the subtree below v and the length of the path from the root to v, respectively. An RDBMS can efficiently query and maintain this encoding: (1) document order is present in the data itself (column pre) which helps to save sorting effort, (2) location steps along the 12 XPath axes map into SQL range predicates against the encoding table, and (3) the encoding of transient nodes (generated by XQuery's node constructors) may be derived using SQL aggregate functions and arithmetics [7].

Standard B-tree index structures suffice to accelerate the operations over this schema-oblivious XML encoding. We

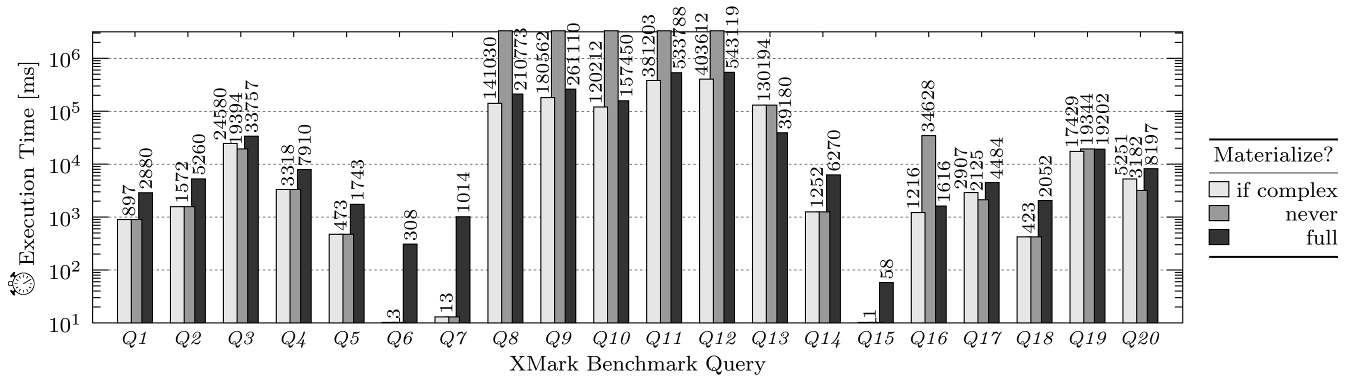

Figure 1: Elapsed times for the 20 XMark queries ran against a 115 MB XML instance (≈ 5,000,000 nodes) hosted by DB2 9. The SQL code generator has been configured to apply different materialization strategies.

have indeed found XPath location step evaluation to account for only a minor fraction of the overall query processing time [6]. Access patterns which are specific to the plans produced by the SQL:1999 code generator, *e.g.*, node access by column level or by element tag name, are readily supported by secondary *partitioned* B-tree indexes [3]. A low-selectivity key prefix partitions the encoding and quickly guides B-tree lookups to find only relevant nodes. The code generator can optionally take advantage of DataGuide-style path summaries which may also be realized in terms of partitioned B-trees: all nodes represented by a common node in the DataGuide reside in the same partition. Effectively, each such B-tree partition materializes the result of an absolute XPath location path.

To benefit from these indexes even in the presence of nodes that a query may construct at runtime—these transient nodes do not appear in any index—the system maintains the relational encodings of persistently stored XML documents and transient XML fragments in *separate* tables. Since the evaluation of an XPath location step never escapes the tree fragment of its context node, we can fully benefit from indexing whenever steps are evaluated against persistent nodes (which is the common case) [7].

Once query evaluation is complete, a database client can perform a single sequential scan over the result table to serialize the final XML output.

4. DEMONSTRATION SETUP

Figure 1 depicts the elapsed query execution times for the 20 queries of the XMark benchmark [10] when the *Pathfinder*-generated SQL:1999 scripts were executed on top of IBM DB2 9[1]. The reported times include SQL parsing and plan generation—the scripts contain 10–100 statements per benchmark query—but omit serialization. On the very same database host and workload, we have found *Pathfinder* to use about 1/5 to 1/3 of the time required by pureXML®, the database kernel-resident XQuery engine operating on native XML storage that has been introduced in DB2 9 [9]. We firmly believe that the purely relational approach to XQuery deserves to be pursued further.

Note that the longer execution times for *Q8* through *Q12* are due to XQuery joins that inevitably produce substantial intermediate XML results (*e.g.*, more than 31,500,000 nodes for *Q11*). For these queries, pureXML® could not complete evaluation within a time frame of a few hours.

Side-by-side on DB2 9: Purely Relational XQuery and pureXML®. The live demonstration will feature a side-by-side setup of *Pathfinder* and the pureXML® XQuery processor on DB2 9. The database will be populated with persistent XML documents of varying schema and size—once relationally encoded, once imported into the native XML storage. Users may evaluate ad-hoc XQuery expressions or run canned queries on both systems. Hooks will be installed in *Pathfinder* and DB2 9 to allow the inspection of the result of various compilation stages (relational algebra plan DAGs, SQL:1999 scripts, physical execution plans).

[1] The database host was Linux-based and equipped with two 3.2 GHz Intel Xeon® CPUs, 8 GB of primary and 280 GB SCSI disk-based memory.

5. REFERENCES

[1] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. W3 Consortium, June 2006.

[2] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. SIGMOD*, Chicago, USA, 2006.

[3] G. Graefe. Sorting and Indexing with Partitioned B-trees. In *Proc. CIDR*, Asilomar, CA, USA, 2003.

[4] C. Grün, A. Holupirek, M. Kramis, M. H. Scholl, and M. Waldvogel. Pushing XPath Accelerator to its Limits. In *Proc. EXPDB*, Chicago, USA, 2006.

[5] T. Grust. Purely Relational FLWORs. In *Proc. XIME-P Workshop*, Maryland, USA, 2005.

[6] T. Grust, J. Rittinger, and J. Teubner. eXrQuy: Order Indifference in XQuery. In *Proc. ICDE*, Istanbul, Turkey, 2007.

[7] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. VLDB*, Toronto, Canada, 2004.

[8] International Standards Organization (ISO). Information Technology—Database Language SQL, 2003.

[9] M. Nicola and B. van der Linden. Native XML Support in DB2 Universal Database. In *Proc. VLDB*, Trondheim, Norway, 2005.

[10] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB*, Hong Kong, China, 2002.