

XQuery Join Graph Isolation

(Celebrating 30+ Years of XQuery Processing Technology)

Torsten Grust

Manuel Mayr

Jan Rittinger

*Eberhard-Karls-Universität Tübingen
Tübingen, Germany*

torsten.grust | manuel.mayr | jan.rittinger@uni-tuebingen.de

Abstract—A purely relational account of the true XQuery semantics can turn any relational database system into an XQuery processor. Compiling nested expressions of the fully compositional XQuery language, however, yields odd algebraic plan shapes featuring scattered distributions of join operators that currently overwhelm commercial SQL query optimizers.

This work rewrites such plans before submission to the relational database back-end. Once cast into the shape of join graphs, we have found off-the-shelf relational query optimizers—the B-tree indexing subsystem and join tree planner, in particular—to cope and even be autonomously capable of “reinventing” advanced processing strategies that have originally been devised specifically for the XQuery domain, *e.g.*, XPath step reordering, axis reversal, and path stitching. Performance assessments provide evidence that relational query engines are among the most versatile and efficient XQuery processors readily available today.

I. INTRODUCTION

SQL query optimizers strive to produce query plans whose primary components are *join graphs*—bundles of relations interconnected by join predicates—while a secondary, peripheral *plan tail* performs further filtering, grouping, and sorting. Plans of this particular type are subject to effective optimization strategies that, taking into account the available indexes and applicable join methods, derive equivalent join trees, ideally with a left-deep profile to enable pipelining. For more than 30 years now, relational query processing infrastructure has been tuned to excel at the evaluation of plans of this shape.

SQL’s rather rigid syntactical block structure facilitates its compilation into join graphs. The compilation of *truly compositional* expression-oriented languages like XQuery, however, may yield plans of unfamiliar shape [7]. The arbitrary nesting of *for* loops (iteration over ordered item sequences), in particular, leads to plans in which join and sort operators as well as duplicate elimination occur throughout. Such plans overwhelm current commercial SQL query optimizers: the numerous occurrences of sort operators block join operator movement, effectively separate the plan into fragments stacked upon each other, and ultimately lead to unacceptable query performance.

Here, we derive join graphs from plans generated by the XQuery compiler described in [7]. The XQuery order and duplicate semantics are preserved. The resulting plan may be

```
Expr → for $VarName in Expr return Expr
      | $VarName
      | if (BoolExpr) then Expr else ()
      | doc(StringLiteral)
      | Expr / ForwardAxis NodeTest
      | Expr / ReverseAxis NodeTest
BoolExpr → Expr
          | Expr GeneralComp Literal
GeneralComp → = | != | < | <= | > | >= [60]
ForwardAxis → descendant:: | following:: | ... [73]
ReverseAxis → parent:: | ancestor:: | ... [76]
NodeTest → KindTest | NameTest [78]
Literal → NumericLiteral | StringLiteral [85]
VarName → QName [88]
StringLiteral → "... " [144]
```

Fig. 1. Relevant XQuery subset (source language). Annotations in [·] refer to the grammar rules in [2, Appendix A].

equivalently expressed as a single SELECT-DISTINCT-FROM-WHERE-ORDERBY block to be submitted for execution by an off-the-shelf RDBMS. The database system then evaluates this query over a schema-oblivious tabular encoding of XML documents to compute the encoding of the resulting XML node sequence (which may then be serialized to yield the expected XML text).

In this work we restrict ourselves to the XQuery Core fragment, defined by the grammar in Fig. 1, that admits the orthogonal nesting of *for* loops over XML node sequences (of type *node()**), supports the 12 axes of XQuery’s *full axis* feature, arbitrary XPath name and kind tests, as well as general comparisons in conditional expressions whose *else* clause yields the empty sequence (). As such, the fragment is considerably more expressive than the widely considered *twig* queries [3], [4] and can be characterized as XQuery’s data-bound “workhorse”: XQuery uses this fragment to collect, filter, and join nodes from participating XML documents.

Isolating the join graph implied by the input XQuery expression lets the relational database query optimizer face a problem known inside out despite the source language not being SQL: in essence, the join graph isolation process emits a bundle of self-joins over the tabular XML document encoding connected by conjunctive equality and range predicates. Most interestingly, we have found relational query optimizers to be autonomously capable of translating these join graphs into join trees that, effectively, (1) perform cost-based shuffling of

the evaluation order of XPath location steps and predicates, (2) exploit XPath axis reversal (e.g., trade ancestor for descendant), and (3) break up and stitch complex path expressions. In recent years, all of these have been described as specific evaluation and optimization techniques in the XPath and XQuery domain [3], [10], [12]—here, instead, they are the *automatic result* of join tree planning solely based on the availability of vanilla B-tree indexes and associated statistics. The resulting plans fully exploit the relational database kernel infrastructure, effectively turning the RDBMS into an XQuery processor that can perfectly cope with large XML instances.

We plugged join graph isolation into *Pathfinder*¹—a full-fledged compiler for the complete XQuery language specification targeting conventional relational database back-ends—and observed significant query execution time improvements for popular XQuery benchmarks, e.g., XMark or the query section of TPoX [11], [14].

II. JOIN-BASED XQUERY SEMANTICS

To prepare join graph isolation, the compiler translates the XQuery fragment of Fig. 1 into intermediate DAG-shaped plans consisting of table references, projection (π), selection (σ), join (\bowtie), cross product (\times), duplicate elimination (δ), and rank operators (ϱ).² This particularly simple table algebra dialect has been designed to match the capabilities of SQL query engines: operators consume tables (not relations) and duplicate row elimination is explicit (in terms of δ). The row rank operator $\varrho_{a:(b_1, \dots, b_n)}$ exactly mimics SQL:1999’s `RANK() OVER (ORDER BY b1, . . . , bn) AS a` and accounts for XQuery’s pervasive sequence order notion. By explicitly encoding order information in table columns using ϱ , the maintenance of order becomes tractable in the relational optimization process.

A. XML Infoset Encoding

An encoding of persistent XML infosets is provided via a designated table `doc`. In principle, any schema-oblivious node-based encoding of XML nodes that admits the evaluation of XPath node tests and axis steps fits the bill (e.g., *ORDPATH* [13]). Here, table `doc` has a `pre|size|level|kind|name|value` schema where each row represents a node with its unique document order rank, its subtree size, distance from the root, node kind, tag name, and its untyped string value [5, § 3.5.2], respectively.

B. XPath Location Steps

This encoding has already been shown to admit the efficient join-based evaluation of location steps $\alpha::n$ along all 12 XPath axes α [8]. While the structural node relationship expressed by α maps into a conjunctive range join predicate over columns `pre`, `size`, `level`, the step’s kind and/or name test n yields equality predicates over kind and name. With their ability to perform range scans, regular B-tree indexes, built over table `doc`, perfectly support this style of location step evaluation [8].

¹<http://www.pathfinder-xquery.org/>

²For more details please refer to the extended version of this paper [6].

C. A Loop-Lifting XQuery Compiler

From [7] we adopt a view of the dynamic XQuery semantics, *loop lifting*, that revolves around the `for` loop as the core language construct. Any subexpression is considered to be iteratively evaluated inside its innermost enclosing `for` loop. The compilation process, specified in terms of inference rules (taken from [7]) is discussed in the extended version of this paper [6]. They form a *compositional* algebraic compilation scheme for the XQuery dialect of Fig. 1.

D. The Compositionality Threat

Note, though, how this artifact of both, compositional language and compilation scheme, leads to *stacked plans*—plans whose shapes differ considerably from the ideal *join graph + plan tail* we have identified earlier (see e.g., Fig. 2). Instead, join operators occur in sections distributed all over the plan. A similar distribution can be observed for the blocking operators δ and ϱ (duplicate elimination and row ranking). This is quite unlike the algebraic plans produced by SQL `SELECT-FROM-WHERE` block compilation.

The omnipresence of blocking operators obstructs join operator movement and planning and leads industrial-strength optimizers, e.g., IBM DB2 UDB V9, to execute the stacked plan in stages that materialize and then again read temporary tables. In the following we will therefore follow a different route and instead *reshape* the plan into a join graph that becomes subject to efficient one-shot execution by the SQL database back-end.

III. XQUERY JOIN GRAPH ISOLATION

In a nutshell, join graph isolation pursues a strategy that moves the blocking operators (ϱ and δ) into plan tail positions and, at the same time, pushes join operators down into the plan [6]. This rewriting process is enhanced by attaching properties such as constantness, keyness, (absence of) duplicates, and utilization to the columns of intermediate result tables. For the plan in Fig. 2, e.g., column `iter` is marked constant in all operators and column `pos` generated in the lower ϱ operator is marked unused. Based on these properties, peep-hole style rewrite rules isolate a plan section, the *join graph* (see Fig. 3), that is populated with references to the infoset encoding table `doc`, joins, and further pipelineable operators, like projection and selection (π , σ).

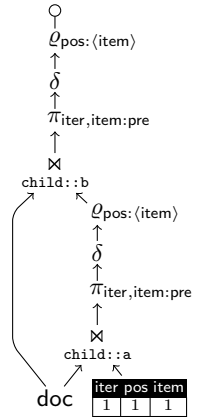


Fig. 2. Initial algebra plan for `/a/b`.

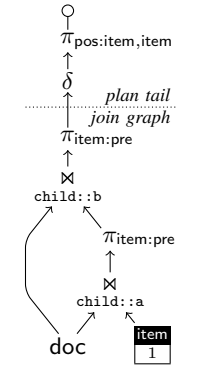


Fig. 3. Optimized algebra plan for `/a/b`.

A. XQuery in the Guise of SQL SFW-Blocks

The result of join graph isolation is a new DAG that can readily be translated into a *single* SELECT-DISTINCT-FROM-WHERE-ORDER BY block in which

- (1) the FROM clause lists the required doc instances,
- (2) the WHERE clause specifies a conjunctive self-join predicate over doc, reflecting the semantics of XPath location steps and predicates, and
- (3) the SELECT-DISTINCT and ORDER BY clauses represent the plan tail.

Join graphs provide a *complete* description of the input query’s true XQuery semantics but *do not prescribe* a particular order of XPath location step or predicate evaluation. It is our intention to let the RDBMS decide on an evaluation strategy, based on its very own cost model, the availability of join algorithms, and supporting index structures. As a consequence, it suffices to communicate the join graph in form of a standard SQL SELECT-DISTINCT-FROM-WHERE-ORDER BY-block—*i.e.*, in a declarative fashion barring any XQuery-specific annotations or similar clues. For query /a/b, we thus ship the following SQL query for execution by the database back-end:

```
SELECT DISTINCT d3.pre as item
FROM doc AS d1, doc AS d2, doc AS d3
WHERE d1.pre = 1 AND d1.level = 0 AND d1.kind = 0
AND d1.pre < d2.pre AND d2.pre < d1.pre + d1.size
AND d1.level + 1 = d2.level
AND d2.name = "a" AND d2.kind = 1
AND d2.pre < d3.pre AND d3.pre < d2.pre + d2.size
AND d2.level + 1 = d3.level
AND d3.name = "b" AND d3.kind = 1
ORDER BY d3.pre .
```

Note how the DISTINCT and ORDER BY clauses—realizing the plan tail of Fig. 3—reflect the XQuery sequence order and duplicate semantics.

IV. IN LABORATORY WITH IBM DB2 V9 (EXPERIMENTS)

The SQL language subset used to describe the XQuery join graphs—flat self-join chains, simple ordering criteria, and no grouping or aggregation—is sufficiently simple to let any SQL-capable RDBMS assume the role of a back-end for XQuery evaluation. We do *not* rely on SQL/XML functionality, in particular. In what follows, we will observe how IBM DB2 UDB V9 acts as a runtime environment for the join-graph-isolating compiler.

Partitioned B-tree index support for XQuery. Based on the query set in Table I, DB2’s index advisor db2adv1s [1] *autonomously* proposed a set of B-tree indexes with keys (p:pre, s:pre + size, l:level, k:kind, n:name, v:value): nkspl, nlkps, nksp, lksp, nlkp, vnlkp.

The majority of the proposed index keys are prefixed with *low cardinality* column(s), *e.g.*, nk, nlk, or lk. A B-tree that is primarily organized by such a low cardinality column will, in consequence, *partition* the XML infoset encoding into few disjoint node sets. Note how, in a sense, a name-prefixed index key leads to a B-tree-based implementation of the *element tag*

streams, the principal data access path used in the so-called *twig join* algorithms [3], [4].

The design advisor further suggests an index with key vnlkp whose value column prefix supports atomization; all other columns back subsequent XPath steps. A B-tree of this type bears some close resemblance with the XPath-specific indexes (CREATE INDEX ... GENERATE KEY USING XMLPATTERN ... AS SQL VARCHAR(n)) employed by DB2 V9’s pureXML™ (Section IV-B).

A. XPath Continuations

How exactly does DB2 V9’s query optimizer deploy the indexes proposed by its design advisor companion? Plan tree analysis provides an answer. We observed query evaluation techniques that have originally been described as XPath-specific [3], [10], [12], outside the relational domain. Since we have shifted all responsibility for the XQuery runtime aspects to the RDBMS, we think this is quite interesting.

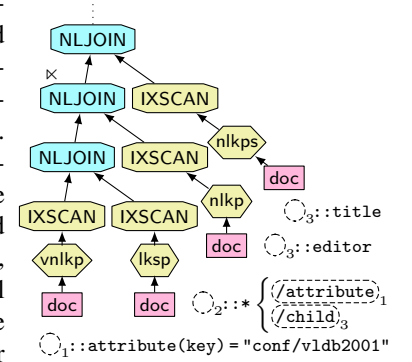


Fig. 4. Partial DB2 V9 execution plan for Q_3 with continuation annotations.

A snippet of the optimized left-deep DB2 execution plan for Query Q_3 (see Table I) is shown in Fig. 4. Annotations (such as $C_3::title$) furthermore document the role of the planned index scans. DB2 starts the evaluation of Query Q_3 with the most selective test using a vnlkp index: find attribute nodes with name key having string value "conf/v1db2001". At this point, the evaluated path fragment is still context-nodeless. The due context C_1 is only provided by the subsequent NLJOIN-IXSCAN pair which retrieves the owner elements of the key attributes. Note that key assumes the context node role. In the next NLJOIN-IXSCAN pair (where the *early-out* flag is set) columns pre and size from the child::* step provide the continuation for the editor child step. The same context information is used a second time to fill “hole” C_3 for the title nodes.

In addition to Query Q_3 we analyzed many more query plans in DB2 and made the following observations:

- The “classical” selectivity notion directly leads to the planned access to *element tag stream* indexes and value-based indexes.
- DB2’s optimizer takes into account all pairs of dual axes [12] and treats *e.g.*, axis parent for child.
- The B-tree index entries provide sufficient context information to allow for arbitrary path processing orders. In the terminology of [10], we have observed the optimizer to generate the whole variety of *Scan* (strict left-to-right location path evaluation), *Lindex* (right-to-left evaluation), and *Bindex* plans (hybrid evaluation, originating in a context node set established via tag name selection).

TABLE I

SAMPLE QUERY SET TAKEN FROM [9] (RIGHTMOST COLUMN SHOWS THE QUERY IDENTIFIER USED IN [9]). WE REPLACED THE NON-STANDARD `return-tuple (Q4)` BY AN SQL/XML `XMLTABLE` CONSTRUCT.

| | Query | Data [9] |
|----------------|--|----------|
| Q ₁ | <code>/site/people/person[@id = "person0"] /name/text()</code> | XMark 9a |
| Q ₂ | <code>//closed_auction/price/text()</code> | XMark 9c |
| Q ₃ | <code>/dblp/*[@key = "conf/vldb2001" and editor and title]/title</code> | DBLP 8c |
| Q ₄ | <code>for \$thesis in /dblp/phdthesis [year < "1994" and author and title] return-tuple \$thesis/title, \$thesis/author, \$thesis/year</code> | DBLP 8g |

TABLE II

OBSERVED RESULT SIZES AND WALL CLOCK EXECUTION TIMES (AVERAGE OF 10 RUNS).

| Query | # nodes | DB2 + <i>Pathfinder</i> | | DB2 pureXML TM | |
|----------------|---------|-------------------------|-----------------------|---------------------------|----------------------|
| | | stacked ⌚ (sec) | join graph ⌚ (sec) | whole ⌚ (sec) | segmented ⌚ (sec) |
| Q ₁ | 1 | 60.582 | 0.017 | 0.891 | 0.001 |
| Q ₂ | 9,750 | 32.246 | 0.309 | 6.455 | 7.438 |
| Q ₃ | 1 | 442.745 | 0.391 | 48.066 | 0.001 |
| Q ₄ | 59 | 0.026 | 0.004 | 1.292 | 0.017 |

- We spotted continuations with multiple resumption points—an equivalent of the branching nodes discussed in the context of *holistic twig joins* [3].

B. Pure SQL vs. pureXMLTM

With DB2 Version 9, IBM released the built-in pureXMLTM XQuery processor. This opens up a chance for a particularly insightful quantitative assessment of the potential of the approach to XQuery processing discussed here: not only can a comparison with pureXMLTM be performed on the same machine, but even in the context of a single query processing infrastructure.

PureXMLTM's path steps are based on the TurboXPath algorithm [9]. We thus chose sample queries that directly stem from [9]—these queries are displayed in Table I. This query set exhibits runtime characteristics representative for the much larger query set we investigated in the course of this work.

Queries Q₁ and Q₂ ran against an XMark instance of 110 MB, Q₃ and Q₄ queried an XML representation of the DBLP publication database (400 MB). We ran all experiments on the same DB2 UDB V9.1 instance hosted on a dual 3.2 GHz Intel XeonTM computer with 8 GB of primary and SCSI-based secondary disk memory, running a Linux 2.6 kernel.

For pureXMLTM we stored the XML documents in columns of type XML and generated an extensive set of XMLPATTERN indexes. As an index scan in pureXMLTM yields RIDs—the row IDs containing matching XML documents—a refinement query is necessary to retrieve the actual result nodes. By cutting

the XMark instance into 23,000 segments of 1–6 KB and the DBLP instance into about 1,000,000 distinct publications (each 1 KB), we helped pureXMLTM utilize its indexes.

We then translated the query set with *Pathfinder*, an XQuery compiler that includes a faithful implementation of join graph isolation as described in Section III. *Pathfinder* was configured to emit the SQL code derived from both, the original stacked plan and the isolated join graph.

The impact of join graph isolation. Table II summarizes the average wall clock execution times we observed. The results lead to five interesting observations:

- 1) Join graph isolation beats the stacked approach in most cases by more than two orders of magnitude.
- 2) On large (monolithic) documents, join graph isolation surpasses pureXMLTM (especially if pureXMLTM is forced to scan the whole document, *e.g.*, due to the wildcard step in Q₃).
- 3) The raw path performance of the B-tree supported path step evaluation (see Q₂) seems hard to surpass for pureXMLTM.
- 4) Q₁ and Q₃ are best case scenarios for the segmented pureXMLTM setup—index lookups with a single result node and marginal post-processing.
- 5) The sub-second execution times observed for *Pathfinder* indicate that the effort to compile into particularly simply-shaped self-join chains pays off.

Acknowledgments. This research is supported by the German Research Council (DFG) under grant GR 2036/2-1.

REFERENCES

- [1] DB29 for Linux, UNIX and Windows Manuals, 2007. <http://www.ibm.com/software/data/db2/udb/>.
- [2] S. Boag, D. Chamberlin, and M. Fernández. XQuery 1.0: An XML Query Language. W3 Consortium, 2007. <http://www.w3.org/TR/xquery/>.
- [3] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. SIGMOD*, 2002.
- [4] S. Chen, H. Li, J. Tatemura, W. Hsiung, D. Agrawal, and K. Selçuk Candan. Twig²Stack: Bottom-Up Processing of Generalized-Tree-Pattern Queries over XML Documents. In *Proc. VLDB*, 2006.
- [5] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3 Consortium, 2007. <http://www.w3.org/TR/xquery-semantics/>.
- [6] T. Grust, M. Mayr, and J. Rittinger. XQuery Join Graph Isolation (extended version), 2008. <http://arxiv.org/abs/0810.4809>.
- [7] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. VLDB*, 2004.
- [8] T. Grust, J. Teubner, and M. van Keulen. Accelerating XPath Evaluation in Any RDBMS. *ACM TODS*, 29(1), 2004.
- [9] V. Josifovski, M. Fontoura, and A. Barta. Querying XML Streams. *VLDB Journal*, 14(2), 2004.
- [10] J. McHugh and J. Widom. Query Optimization for XML. *VLDB Journal*, 1999.
- [11] M. Nicola, I. Kogan, and B. Schiefer. An XML Transaction Processing Benchmark. In *Proc. SIGMOD*, 2007.
- [12] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Proc. XMLDM (EDBT Workshop)*, 2002.
- [13] P. E. O’Neil, E. J. O’Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. SIGMOD*, 2004.
- [14] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB*, 2002.