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 [11]. 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, and ultimately lead to unacceptable query performance.

Here, we propose a plan rewriting procedure that derives join graphs from plans generated by the XQuery compiler described in [11]. The XQuery order and duplicate semantics are preserved. The resulting plan may be equivalently expressed as a single SELECT-DISTINCT-FROM-WHERE-ORDER-BY block to be submitted for execution by an off-the-shelf RDBMS.

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 [5, 6] 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 [5], [14], [16]—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 (of size 100 MB and beyond).

We plugged join graph isolation into Pathfinder—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 [15], [19].

We start to explore this form of XQuery join graph isolation in Section II where we review the compiler’s algebraic target language, tabular document encodings, and join-based compilation rules for XPath location steps, nested for loops, and conditionals. The rewriting procedure of Section III then isolates the join graphs buried in the initial compiled plans. Cast in terms of an SQL query, IBM DB2 V9’s relational query processor is able turn these graphs into join trees which, effectively, implement a series of otherwise XQuery- and XPath-specific optimizations. A further quantitative experimental assessment demonstrates that DB2 V9’s built-in pureXML™ XQuery processor currently faces a serious challenger with its relational self if the latter is equipped with the join graph-isolating compiler (Section IV). Sections V and VI conclude this paper with reviews of related efforts and work in flux.

II. JOIN-BASED XQUERY SEMANTICS

To prepare join graph isolation, the compiler translates the XQuery fragment of Fig. I into intermediate DAG-shaped plans over the table algebra of Table I. This particularly simple 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 RAFT(b1, . . . , bn) exactly mimics SQL:1999’s RANK() OVER (ORDER BY b1, . . . , bn) AS a and is primarily used to account for XQuery’s pervasive sequence order notion. The attach operator attach c indicates that table a hosts several singleton literal tables (with columns a, b, and c). For nodes with size ⩽ 1, table doc supports value-based node access in terms of two columns that carry the node’s untyped string value [8, § 3.5.2] and, if applicable, the result of a cast to type xs:decimal3 (columns value and data, respectively). This tabular XML infoset representation may be efficiently populated (during a single parsing pass over the XML document text) and serialized again (via a table scan in pre order).

A. XML Infoset Encoding

An encoding of persistent XML infosets is provided via the 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 [17]). The following uses one such row-based format in which, for each node v, key column pre holds v’s unique document order rank to form—together with columns size (number of nodes in subtree below v) and level (length of path from v to its document root node)—an encoding of the XML tree structure (Fig. 2 and [12]). XPath kind and name tests access columns kind and name—multiple occurrences of value DOC in column kind indicate that table doc hosts several trees, distinguishable by their document URLs (in column name). For nodes with size ⩽ 1, table doc supports value-based node access in terms of two columns that carry the node’s untyped string value [8, § 3.5.2] and, if applicable, the result of a cast to type xs:decimal3 (columns value and data, respectively). This tabular XML infoset representation may be efficiently populated (during a single parsing pass over the XML document text) and serialized again (via a table scan in pre order).

B. XPath Location Steps

Further, this encoding has already been shown to admit the efficient join-based evaluation of location steps

3In the interest of space, we omit a discussion of the numerous further XML Schema built-in data types.
effective Boolean values in conditionals (via fn:boolean(·)) is explicit [8, § 4.2.1 and § 3.4.3].

D. The Compositionality Threat

To obtain an impression of typical plan features, we compile

\[
\text{doc("auction.xml")} \\
\quad /\text{descendant::open_auction[bidder]} . \quad (Q_1)
\]

After XQuery Core normalization, this query reads

\[
\text{for } x \text{ in fs:ddo(doc("auction.xml"))} \\
\quad /\text{descendant::open_auction} \\
\quad \text{return if (fn:boolean(fs:ddo(x/child::bidder)))} \\
\quad \text{then } x \text{ else } () .
\]

Fig. 4 shows the initial plan for \(Q_1\). Since the inference rules of Fig. 13 implement a fully compositional compilation scheme, we can readily identify how the subexpressions of \(Q_1\) contribute to the overall plan (to this end, observe the gray plan sections all of which yield tables with columns iter[pos][item]). XQuery is a functional expression-oriented language in which subexpressions are stacked upon each other to form complex queries. The tall plan profile with its stacked sections—reaching from a single instance of table doc (serving all node references) to the serialization point \(Q\)—directly reflects this orthogonal nesting of expressions.

Note, though, how this artifact of both, compositional language and compilation scheme, leads to plans whose shapes differ considerably from the ideal join graph + plan tail we have identified earlier. Instead, join operators occur in sections distributed all over the plan. A similar distribution can be observed for the blocking operators \(\delta\) and \(\rho\) (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 plan in stages that read and then again materialize 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 backend. (Section IV will show that join graph isolation for \(Q_1\) improves the evaluation time by a factor of 5.)

III. XQUERY JOIN GRAPH ISOLATION

In a nutshell, join graph isolation pursues a strategy that moves the blocking operators \(\rho\) and \(\delta\) into plan tail positions and, at the same time, pushes join operators down into the plan. This rewriting process will isolate a plan section, the join graph, that is populated with references to the infoset encoding table doc, joins, and further pipelineable operators, like projection, selection, and column attachment (\(\pi\), \(\sigma\), \(\vartheta\)).

The ultimate goal is to form a new DAG that may 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.

A. Plan Property Inference

We account for the unusual tall shape and substantial size (of the order of 100 operators and beyond for typical benchmark-type queries) of the initial plan DAGs by a peephole-style rewriting process. For all operators @, a property inference collects relevant information about the plan vicinity of @. The applicability of a rewriting step may then be decided by inspection of the properties of a single operator (and its closer neighborhood) at a time. Tables II–IV define these properties and their inference in an operator-by-operator fashion. We rely on auxiliary function cols(·) that can determine the columns used in a predicate (e.g., cols(pre_o + size_o < pre) = {pre_o, size_o, pre} as well as the columns in the output table of a given plan fragment (e.g., cols[@iter](roles)) = {iter, pos}). Furthermore we use ⇒ to denote the reachability relation of this DAG (e.g., @ ⇒ ρ for any operator @ in the plan).

icol This property records the set of input columns strictly required to evaluate @ and its upstream plan. At the plan root ρ, the property is seeded with the set {pos, item}, the two columns required to represent and serialize the resulting XML node sequence. The icols column set is inferred top-down and accumulated whenever the DAG-walking inference enters a node more than once.

const A set with elements of the form a = c, indicating that all rows in the table output by ⊙ hold value c in column a. Seeded at the plan leaves (instances of doc or literal tables) and inferred bottom-up.

key The set of candidate keys generated by ⊙ (bottom-up).

set Boolean property set communicates whether the output rows of ⊙ will undergo duplicate elimination in the upstream plan. Inferred top-down (set is initialized to true for all operators but ρ).

B. Isolating Plan Tail and Join Graph

The isolation process is defined by three subgoals ⊙, ⊖, and ◊ (described below), attained through a sequence of goal-directed applications of the rewrite rules in Fig. 5. Note how the rules’ premises inspect the inferred plan properties, as described above. In addition to these three main goals, “house cleaning” is performed by rules defined to simplify or remove operator (Rules 1–5, 7, and 13). Otherwise, the subgoals either strictly move ρ and ⊖ towards the plan tail or push ◊ down into the plan. Progress and termination thus is guaranteed. Goal ⊙ is pursued first.

◊ Establish a single ρ operator in the plan tail. A single instance of the row ranking operator ρ suffices to correctly implement the sequence and document order requirements of the overall plan. To this end, Rule 12 trades a ρ for a projection if ρ ranks over a single column. For the compilation
Rules $\text{DDO}$ and $\text{STEP}$, which introduce row rankings of this form ($\{\text{pos}, \text{item}\}$), this effectively means that document order determines sequence order—which is indeed the case for the result of XPath location steps and $\text{fs}: \text{ddo}(\cdot)$. All other instances of $\varrho$ ($\{\text{pos}, \text{sort}, \text{pos}\}$, introduced by $\text{FOR}$) are moved towards the plan tail via Rules (14)–(17). The premises of Rules (14) and (15) are no obstacle here: for the XQuery fragment of Fig. 7 the compiler does not emit predicates over sequence positions (column $\text{pos}$). Once arrived in the plan tail, Rule (17) splices the ranking criteria of adjacent $\varrho$ operators. Rule (2) finally removes all but the topmost instance of $\varrho$.

Establish a single $\delta$ operator in the plan tail + join push-down and removal. Duplicate elimination relocation and join push-down and removal are intertwined. Fig. 6 illustrates the stages of this process (the $\oplus$ represent plan sections much like in Fig. 4). These subgoals target and ultimately delete the equi-joins introduced by the compilation Rules (16) and (17) (the latter is in focus here).

A join of this type preserves the keys established by $\#_{\text{inner}}$ and thus emits unique rows. The introduction of a new $\delta$ instance at the top of the plan fragment thus does not alter the plan semantics (Rule (9), see Figures 3(a) and (b)). This renders the original instance of $\delta$ obsolete as duplicate elimination now occurs upstream (Rule (6), Fig. 3(c)). The following stages push the join towards the plan base, leaving a trail of plan sections that formerly occurred in the join input branches (Rule (11)). Figures 3(c) and (d). The condition $\varrho_2 \not\supseteq \varrho_1$ in the premise of Rule (11) prevents its further application in the situation of Fig. 4(d), otherwise, the rewrite would introduce a cycle in the plan. Instead, Rule (2) detects that the join has degenerated into a key join over identical inputs and thus may be removed (Fig. 3(c)). Finally, this renders the remaining instance of $\#_{\text{inner}}$ obsolete (column inner not referenced, Rule (11)).

C. XQuery in the Guise of SQL SFW–Blocks

Fig. 7 depicts the isolation result for Query (2) (original plan shown in Fig. 4). The new plan features a bundle of operators in which—besides instances of $\pi$, $\sigma$—the only remaining joins originate from applications of compilation Rule $\text{STEP}$ implementing the semantics of XPath location steps. The joins consume rows from the XML infoset encoding table doc which now is the only shared plan node in the DAG. As desired, we can also identify the plan tail (in the case of Q2) no extra row ranking is required since the document order ranks of the elements resulting from the $\text{descendant}:: \text{open auction}$ step—in column pre produced by the topmost $\pi$ operator—already determine the overall order of the result.

Quite unlike the initial plans emitted by the compositional compiler, XQuery join graph isolation derives plans that are truly indistinguishable from the algebraic plans produced by a
regular SQL translator. We thus let an off-the-shelf relational database back-end autonomously take over from here. It is now reasonable to expect the system to excel at the evaluation of the considered XQuery fragment as it will face a familiar workload. (This is exactly what we observe in Section [V].)

Most importantly, the 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 (1) we thus ship the SQL query (Fig. 8) for execution by the database back-end.

Plan tail. The interaction of for loop iteration and sequence order of the final result becomes apparent in the plan tail of the following query (traversing XMark data [19] to return the names of those auction categories in which expensive items were sold at prices beyond $500):

```
let $a := doc("auction.xml")
for $ca in $a//closed_auction[price > 500],
   $i in $a//item,
   $c in $a//category
where $ca/itmeref/@item = $i/@id
and $i/incategory/@category = $c/@id
return $c/name
```

This example features let bindings and general value comparisons of atomized nodes, two extensions of the language fragment of Fig. 1 that have been shown to readily fit into the loop-lifting compilation approach [11]. XQuery Core normalization, compilation and subsequent isolation yields the SQL join graph query in Fig. 9 which describes a 12-fold self-join over table doc. Note how the ORDERBY and DISTINCT clauses—which represent the plan tail—reflect the XQuery sequence order and duplicate semantics:

Order The nesting of the three for loops in (Q2) principally determines the order of the resulting node sequence: in (Q2)
SELECT DISTINCT d2.*
FROM doc AS d1, doc AS d2, doc AS d3
WHERE d1.kind = DOC
AND d1.name = 'auction.xml'
AND d2.kind = ELEM
AND d2.name = 'open_auction'
AND d2 BETWEEN d1.pre+1 AND d1.pre+d1.size
AND d3.kind = ELEM
AND d3.name = 'bidder'
AND d3 BETWEEN d2.pre+1 AND d2.pre+d2.size
AND d2.level+1 = d3.level
ORDER BY d2.pre

Fig. 8. SQL encoding of (Q1)’s join graph.

SELECT DISTINCT d12.*,
d2.pre AS item1, d4.pre AS item2,
d5.pre AS item3
FROM doc AS d1, ..., doc AS d12
WHERE ...
ORDER BY d2.pre, d4.pre, d5.pre, d12.pre

Fig. 9. SQL encoding of (Q2) (focus on plan tail: order, duplicate removal).

row variables d2, d4, d5 range over closed_auction.
item, category element nodes, respectively. The
document order of the name elements (bound to d12) is least
relevant in this example and orders the nodes within each
iteration of the innermost for loop.

Duplicates The XPath location step semantics requires du-
plicate node removal (row variable d12 appears in the
DISTINCT clause). Duplicates are retained, however, across
for loop iterations (keys d2.pre, d4.pre, and d5.pre
appear in the DISTINCT clause).

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 for the join-graph
isolating compiler. In this context, DB2 V9 appears to be
especially interesting, because the system
(1) has the ability to autonomously adapt the design of its
physical layer, indexes in particular, in response to a given
workload (this will help to assess whether an RDBMS
can indeed cope with XQuery specifics, like the pervasive
sequence and document order notions), and
(2) features the built-in XQuery processor pureXML™ which
implements a “native” XML document storage and specific
primitives for XPath evaluation, but nevertheless relies on
the very same database kernel infrastructure. This will pro-
vide an insightful point of reference for the performance
assessment of Section IV-B

**Autonomous index design.** The workload produced by the
join-graph-isolating compiler is completely regular: as long as
the input expressions adhere to the XQuery dialect of Fig. [1] all
emitted queries will, for example, evaluate predicates against
ranges with endpoints pre, pre + size and always use column
pre as ordering criterion which finds perfect support in a
clustered B-tree on the primary key column pre. (Here, column
size is exclusively used as a summand in pre + size—we thus
replaced column size with a computed column that contains the
sum.)

Due to this high predictability, we expected the DB2 au-
tomated design advisor, db2advis [1], to be able to suggest
a reasonable, tailored set of vanilla B-tree indexes to support
the typical XQuery join graph workload.

To provide the RDBMS with complete information about
the expected incoming queries, we instructed the compiler to
make the semantics of the serialization point $\varphi$ explicit—this
adds one extra descendant-or-self::node() step to any
Query $Q$, originating in its result node sequence:

$$\text{for } x \text{ in } Q \text{ return } x/\text{descendant-or-self::node()}.\$$

This produces all XML nodes required to fully serialize the
result (surfacing as the additional topmost self-join in the join
plans of Figures [10] and [11]).

For (Q2) as an representative of the prototypical expected
query workload, the DB2 design advisor suggests the B-tree
index set of Table [VI] configured to exploit an unlimited time
and index space budget, db2advis proposes indexes that add
up to a total size of 300 MB for a 110 MB instance of the
XMark auction.xml document. Due to the regularity of the
emitted SQL code, the utility of the proposed indexes will be
high for any workload that exhibits a significant fraction of
XQuery join graphs.

**Partitioned B-tree index support for XQuery.** The majority of the
index keys proposed in Table [VI] are prefixed with low cardinality
column(s), e.g., n, nk, or nlk. An XMark XML instance features 77
distinct element tag and attribute names, regardless of the
document size. Similar observations apply to the 7 XML node kinds (column kind) and the typical
XML document height ($0 \leq \ell \leq 14$ for the XMark
instances). 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 [5], [6].

| Index key columns | Index deployment |
|-------------------|------------------|
| $\text{nlk}$      | XPath node test and axis step, access document node (doc($\cdot$)) |
| $\text{nlkps}$     | Atomization, value comparison with subsequent/preceding XPath step |
| $\text{nlkp}$      | Serialization support (supplies XML infoset in document order) |
| $\text{nlkp}$      | $p:pre, s:pre + size, l:level, k:kind, n:name, v:value, d:data$ |
Partitioned B-trees enjoy a number of desirable properties [10]. In a name-partitioned B-tree with key prefix compression, for example, each partition will contain its element name exactly once (zero redundancy tag name storage).

The design advisor further suggests an index with key vnk whose value column prefix supports atomization and the general value comparisons between (attribute) nodes featured in \( n_{k} \). 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 pureXML\textsuperscript{TM} (Section 4.6).

The proposal of the unique clustered index with key \( n_{k} \text{vlk}s \) is owed to the descendant-or-self::node() step, introduced to enable XML serialization. A forward scan of this index provides access to the full XML infoset for any subtree encoded in table doc, in document order. Since db2advis has observed that such scans will be the principal (here: only) use of this index, it proposes \( n_{k} \text{vlk}s \) as the only key column—all other columns (name through \( \text{pre} + \text{size} \)) are specified in DB2’s INCLUDE() index creation clause [1] and thus merely occupy space on the index leaf pages.

A. XPath Continuations

How exactly does DB2 V9’s query optimizer deploy the indexes proposed by its design advisor companion? An answer to this question can be found through an analysis of the plan trees generated by the optimizer. We have, in fact, observed a few not immediately obvious “tricks” that have found their way into the execution plans. Most of these observations are closely related to query evaluation techniques that have originally been described as XPath-specific [5], [14], [16], outside the relational domain. Since we have transferred all responsibility for the XQuery runtime aspects to the RDBMS, we think this is quite interesting.

The optimized DB2 execution plan found for Query 2 of Section II-D is shown in Fig. 10. We are reproducing these execution plans in a form closely resembling the output of DB2’s visual explain facility. Nodes in these plans represent operators of DB2’s variant of physical algebra—all operators relevant for the present discussion are introduced in Table VII.

Path stitching and branching. Consider the B-tree index with key \( n_{k} \text{spl} \). Due to its \( n_{k} \) prefix, this index primarily provides support for XPath name and kind tests. In the execution plan for \( Q_{1} \), the index is used to access the requested document node (name = ‘auction.xml’ \& kind = DOC). Additionally, however, the index delivers the infoset properties \( sp \) and thus provides all necessary information to step along the XPath descendant axis—namely the interval \([\text{pre}, \text{pre} + \text{size}]\), see Fig. 3—from those nodes that have been found during index lookup. In the following, we will denote the result of such index lookups as

\[
\text{doc("auction.xml")/ descendant}_{1}
\]

or, generally \( n_{k}^{0}[\alpha] \) (read: perform the specified node test \( n \), then prepare a subsequent step along axis \( \alpha \)). In the execution plan of Fig. 10, the bottom index nested-loop join continues this “half-cooked” step: a lookup in the index \( n_{k} \text{spl} \) retrieves columns \( n_{k} \) to (1) perform the due name and kind test (name = ‘open_auction’ \& kind = ELEM) and (2) complete the structural descendant axis traversal (check \( p \) for containment in the \([\text{pre}, \text{pre} + \text{size}]\) interval obtained in the first half of the step). In the annotated plans, we write \( \text{circ}_{1}::\text{open_auction} \) (read: resume axis step and perform specified name and kind test). Stitched at the matching continuation points (here: those with subscript 1), we obtain the complete XPath location step again:

\[
\text{doc("auction.xml")/ descendant}_{1}\text{::open_auction}.
\]

The lookup in index \( n_{k} \text{spl} \) further provides the necessary infoset properties to prepare the now current continuations \( /\text{ descendant-or-self}_{2} \) (columns \( sp \)) as well as \( /\text{child}_{3} \) (columns \( pl \)). Such continuations with multiple resumption points are the equivalent of the branching nodes discussed in the context of holistic twig joins [5].

Given the tailored B-tree index set in Table VII, the DB2 query optimizer consistently manages to select the index access path that provides just the required XML infoset properties. Resuming the child continuation at \( \text{circ}_{3}::\text{bidder} \) requires columns \( nk \) to perform the name and kind test plus columns \( pl \) to complete the evaluation of the range predicate
that implements the child step (again, see Fig. 5 columns pre, size, level are provided by the /child, continuation). The optimizer thus selected the index with key nklp. Finally, as anticipated, the plan scans index nkspl to traverse all nodes in the subtrees below the query’s XML result nodes (resuming from /descendant-or-self).

**XPath step reordering and axis reversal.** Two further phenomena in the execution plans can be explained in terms of the XPath continuation notion. For \( \mathcal{Q}_2 \) the order of the XPath location steps specified in the input query did coincide with the join order in the execution plan (access document node of XML instance auction.xml, then perform a descendant::open_auction step, finally evaluate child::bidder). The B-tree index entries provide sufficient context information, however, to allow for arbitrary path processing orders—right-to-left strategies are conceivable as are strategies that start in the middle of a step sequence and then work their way towards the path’s endpoints. The latter can be witnessed in the execution plan of Query \( \mathcal{Q}_2 \) (Fig. 11).

The plan’s very first index scan over nkspl evaluates the name and kind test for elements with tag closed_auction before the continuation for resumption point \( \bowtie_1 \) has provided any context node information. The index key columns spl are used to prepare the continuation /child, which is then immediately resumed by the node test for element price. Index nkdlp is deployed to implement this node test as well as node atomization and subsequent value comparison (the index key contains column data of table doc). At this point, the plan has evaluated the path fragment

\[
\bowtie_1::closed_auction/child,::price[data(.)>500]
\]

which is still context-less. The due context is only provided by the subsequent NLJOIN-IXSCAN pair which verifies that the closed_auction elements found so far indeed are descendants of auction.xml’s document node. Observe that in this specific evaluation order of the location steps, the closed_auction nodes now assume the context node role: the plan effectively determines the closed_auction elements that have the document node of auction.xml in their ancestor axis. The reversal of axes—in this case, trading descendant for ancestor—is based on the dualities pre \( \leftrightarrow \) pre, size \( \leftrightarrow \) size, in the predicates axis (ancestor) and axis (descendant), defined in Fig. 3. This observation applies to all other pairs of dual axes [16] (e.g., parent/child, following/preceding) and, due to the attribute encoding used in table doc, also to the attribute/owner relationship between element nodes and the attributes they own.

The query optimizer decides on the reordering of paths and the associated reversal of XPath axes based on its “classical” selectivity notion and the availability of eligible access paths: for a B-tree with name-prefixed keys, the RDBMS’s data distribution statistics capture tag name distribution while value-prefixed keys lead to statistics about the distribution of the (untyped) element and attribute values.

In the case of \( \mathcal{Q}_2 \) this enabled the optimizer to decide that the access path nkdlp, directly leading to price nodes (key prefix nk) with a typed decimal value of greater than 500 (key column d), is highly selective (only 9,750 of the 4.7 million nodes in the 110 MB XML instance are price elements and only a fraction of these has a typed value in the required range).

An analogous observation about the distribution of untyped string values in the value column—the key prefix in the vnlp B-tree—has led the optimizer to evaluate the general attribute value comparison

\[
\bowtie_5::id/data(.) =
\text{doc("auction.xml")}/.../attribute::item/data(.)
\]

before the “hole” \( \bowtie_5 \) has been filled. The elements owning the @id attributes are resolved subsequently, effectively reversing the attribute axis. (This constellation repeats for the second attribute value comparison in \( \mathcal{Q}_2 \) resumption point \( \bowtie_6 \).)

**B. Pure SQL vs. pureXML™**

With DB2 Version 9, IBM released the built-in pureXML™ XQuery processor. This opens up a chance for a particularly insightful quantitative assessment of the potential of the purely
TABLE VIII
SAMPLE QUERY SET TAKEN FROM [13] (RIGHTMOST COLUMN SHOWS THE QUERY IDENTIFIER USED IN [13]). WE REPLACED THE NON-STANDARD return-tuple (Q2) BY AN SQL/XML XMLTABLE CONSTRUCT.

| Query                                                                 | Data [13] |
|----------------------------------------------------------------------|-----------|
| Q3 /site/people/person[@id = "person0"] /name/text()               | XMark 9a  |
| Q4 //closed_auction/price/text()                                     | XMark 9c  |
| Q5 /dblp/*[@key = "conf/vldb2001"] and editor and title/title      | DBLP 8c   |
| Q6 for $thesis in /dblp/thesis/year < "1994" and author and title   | DBLP 8g   |

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

| Query                                    | DB2 + Pathfinder # nodes (sec) | Pathfinder join graph (sec) | DB2 pureXML™ whole (sec) | pureXML™ segmented (sec) |
|------------------------------------------|--------------------------------|----------------------------|--------------------------|--------------------------|
| Q1                                       | 1,625,157                      | 63.011                     | 11.788                   | 10.073                   |
| Q2                                       | 318                            | DNF                        | 0.544                    | DNF                      |
| Q3                                       | 1                              | 60.582                     | 0.017                    | 0.891                    |
| Q4                                       | 9,750                          | 32.246                     | 0.309                    | 6.455                    | 7.438                   |
| Q5                                       | 1                              | 442.745                    | 0.391                    | 48.066                   |
| Q6                                       | 59                             | 0.026                      | 0.004                    | 1.292                    |

We then translated the query set with Pathfinder, an XQuery compiler that includes a faithful implementation of loop lifting and join graph isolation described in Sections II-C and III. Pathfinder was configured to emit the SQL code derived from both, the original stacked plan and the isolated join graph. The resulting SQL queries ran against a database populated with tabular XML infost structures of the XMark and DBLP instances, using a B-tree index setup as described in Table VII. Both, pureXML™ and Pathfinder used the same DB2 UDB V9.1 instance hosted on a dual 3.2 GHz Intel Xeon™ computer with 8 GB of primary and SCSI-based secondary disk memory, running a Linux 2.6 kernel.

The impact of join graph isolation. Table IX summarizes the average wall clock execution times we observed. For Query Q4, for example, isolating the join graph (Fig. 4) yields a five-fold reduction of execution time. Compositional compilation leads to tall stacked plans that exhibit a significant number of intermediate ϱ and δ operators. Pathfinder translates such plans into a SQL common table expression (WITH . . . ) that features an equally large number of DISTINCT and RANK().

Although a B-tree with key vnklp resembles an XMLPATTERN index of type VARCHAR(n)—both index (the string values of) atomized XML nodes—the B-tree is deployed quite differently: a lookup yields document order ranks (column pre)
The more than 20-fold advantage of *Pathfinder* engine from visiting a significant part of the XML instance. Query being the root of a subtree containing 500 nodes on average. Query sequence contains 3,249 node sequences and 3,249 open auction elements, each being the root of a subtree containing 500 nodes on average. Query *Q* primarily relies on path traversal performance as no value-based index can save the query engine from visiting a significant part of the XML instance. The more than 20-fold advantage of *Pathfinder* suggests that B-tree-supported location step evaluation will remain a true challenger for the XSCAN-based implementation inside pureXML™. Queries *Q* and *Q*-5 to some extent) yield singleton (short) node sequences and constitute the best case for the segmented pureXML™ setup: here, XMLPATTERN index lookups return a single or few RID(s), directly leading the system to small XML segment(s)—the remaining traversal effort for XSCAN then is marginal. For the whole document setup, however, an index lookup could only point to the single monolithic XML instance: XSCAN thus does all the heavy work (the wildcard * in *Q* forces the engine to scan the entire 400 MB DBLP instance). Despite the extensive index options available to support pureXML™ is not able to finish evaluation within 20 hours: the system appears to miss the opportunity to perform value-based selections and joins early (recall the discussion of Fig. 1 (1) and ultimately is overwhelmed by the Cartesian product of all closed auction, item, and category elements. The indexes largely remain unused (the predicate price[. ] > 500) is, in fact, evaluated second to last in the execution plan generated by pureXML™).

The sub-second execution times observed for *Pathfinder* indicate that the effort to compile into particularly simply-shaped self-join chains pays off. The DB2 V9 built-in monitor facility provides further evidence in this respect: the queries enjoy a buffer cache hit ratio of more than 90% since merely table doc and indexes fight for page slots.

V. MORE RELATED WORK

One key ingredient in the join graph isolation process are the rewrites that move order maintenance and duplicate elimination into tail positions. Their importance is underlined by similar optimizations proposed by other research groups [9], [18], [21]: Fernández et al. remove order constraints and duplicates based on the XQuery Core representation [9]—an effect achieved by Rules 2 and 3 of Figure 5. The principal data structure of XQuery Core—item sequences—however prohibits merging of multiple orders as in Rule 17.

In [21], a algebra working on ordered tables is the subject of order optimization. An order context framework provides minimal order semantics by removing—much like Rule 2—superfluous Sortby operators. In addition, order is merged in join operators and pushed through the plan in an Orderby Pull up much like in Section III. In the presence of order-destroying operators such as δ they however fail to propagate the order information to the plan tail (compare to Rule 14).

An extension of the tree algebra (TLC-C) in the research project Timber introduces order on a global level [18] and generates tree algebra plans that—if tree patterns are mapped to self-join chains—might be transformed into SQL queries similar to join graph isolation.

In Section IV-A we have seen how a selectivity-based reordering of XPath location steps can also lead to a reversal of axes. In effect, the optimizer mimics a family of rewrites that has been developed in [16]. These rewrites were originally designed to trade reverse XPath axes for their forward duals, which can significantly enlarge the class of expressions tractable by streaming XPath evaluators. Here, instead, we have found the optimizer to exploit the duality in both directions—in fact, a descendant axis step has been traded for an ancestor step in the execution plan for *Q* (Fig. 11). The evaluation of rooted /descendant::n steps—pervasively introduced in [16] to establish a context node set of all elements with tag n in a document—is readily supported by the n-prefixed B-tree indexes. Since the XQuery compiler implements the full axis feature, it can actually realize a significant fraction of the rewrites in [16].

Although we exclusively rely on the vanilla B-tree indexes that are provided by any RDBMS kernel, cost-based join tree planning and join reordering leads to a remarkable plan versatility. In the terminology of [14], 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; cf. the initial closed auction node test in Fig. 11).

The path branching and stitching capability (Section IV-A) makes the present XQuery compilation technique a distant relative of the larger family of holistic twig join algorithms [5], [6], [7]. We share the language dialect of Fig. 11—coined generalized tree pattern queries in [5], [7]—but add to this the full axis feature. Quite differently, though, we (1) let the RDBMS shoulder 100% of the evaluation-time and parts of the compile-time effort invested by these algorithms (e.g., the join tree planner implements the findOrder(·) procedure of [7] for free), and (2) use built-in B-tree indexes over table-shaped data where TwigStack [5] and TwigStack [6] rely on special-purpose runtime data structures, e.g., chains or hierarchies of linked stacks and modified B-trees, which call for significant invasive extensions to off-the-shelf database kernels.
Finally, this work may be read as one possible response to a list of open issues identified in the context of cost-based XQuery processing with DB2 V9 pureXML™ (here, we directly refer to the specific issues raised in [3, page 316 ff.]):

**XML index exploitation** The infoset encoding (Fig. 2) is truly node-based: location paths may originate in any individual node and the document order rank (column pre) is sufficient to “point into” a document.

**Deferred XPath evaluation** The B-tree index keys (Table VI) are self-contained: XPath processing may resume without any consultation of a document’s infoset encoding.

**Cost estimation** Since the generated executions plans exclusively feature the well-known operators (Table VI), established procedures for statistics collection and cost estimation remain applicable [20] (but see Section VI).

**Order optimization** Join graph isolation leads to a sound—also in the presence of nested for loops—yet compact representation of the XQuery order semantics in the plan tail, which ultimately finds its way into a single simple SQL ORDER BY clause.

VI. Work in Flux

This work rests on the maturity and versatility of database technology for strictly table-shaped data, resulting from 30+ years of experience. We (1) discussed relational encodings of the true XQuery semantics that are accessible for today’s SQL query optimizers, but (2) also saw that some care is needed to unlock the potential of a set-oriented query processor.

In the context of the open-source Pathfinder project, we continue to pursue the idea of a purely relational XQuery processor. On the workbench lie DB2’s statistical views—in our case, pre-formulated descendant and child location steps for which the system records statistical properties but not the result itself—which promise to give insight into the structural node distribution of an encoded XML document. This may further improve join tree planning.

The scope of this work reaches beyond XQuery. Tall stacked plan shapes with scattered distributions of group operators (Fig. 1) also are an artifact of the compilation of complex SQL/OLAP queries (in which functions of the RANK() family are pervasive). The observations of Section V suggest that the rewriting procedure of Fig. 5 can benefit commercial query optimizers also in this domain.

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

**REFERENCES**

[1] DB2 for Linux, UNIX and Windows Manuals, 2007. [http://www.ibm.com/software/data/db2/udb/]

[2] A. Balmin, K. Beyer, F. Özcan, and M. Nicola. On the Path to Efficient XML Queries. In Proc. VLDB, 2006.

[3] A. Balmin, T. Eliaz, J. Hornbrook, L. Lim, G. Lohman, D. Simon, M. Wang, and C. Zhang. Cost-based Optimization in DB2 XML. IBM Systems Journal, 45(2), 2006.

[4] S. Boag, D. Chamberlin, and M. Fernández. XQuery 1.0: An XML Query Language. W3 Consortium, 2007. [http://www.w3.org/TR/xquery/]

[5] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In Proc. SIGMOD, 2002.

[6] 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.

[7] Z. Chen, H. Jagadish, L. Lakshmanan, and S. Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In Proc. VLDB, 2003.

[8] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Walder. XQuery 1.0 and XPath 2.0 Formal Semantics. W3 Consortium, 2007. [http://www.w3.org/TR/xquery-semantics/]

[9] M. Fernández, J. Hidders, P. Michiels, J. Siméon, and R. Verammenn. Optimizing Sorting and Duplicate Elimination in XQuery Path Expressions. In Proc. DEXA, 2005.

[10] G. Graefe. Sorting and Indexing with Partitioned B-Trees. In Proc. CIDR, 2003.

[11] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In Proc. VLDB, 2004.

[12] T. Grust, J. Teubner, and M. van Keulen. Accelerating XPath Evaluation in Any RDBMS. ACM TODS, 29(1), 2004.

[13] V. Josifovski, M. Fontoura, and A. Barta. Querying XML Streams. VLDB Journal, 14(2), 2004.

[14] J. McHugh and J. Widom. Query Optimization for XML. VLDB Journal, 1999.

[15] M. Nicola, I. Kogan, and B. Schiefer. An XML Transaction Processing Benchmark. In Proc. SIGMOD, 2007.

[16] D. Olteanu, H. Meuss, T. Furchè, and F. Bry. XPath: Looking Forward. In Proc. XMLDM (EDBT Workshop), 2002.

[17] P. E. O’Neill, E. J. O’Neill, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In Proc. SIGMOD, 2004.

[18] S. Paparizos and H. V. Jagadish. Pattern tree algebras: sets or sequences? In Proc. VLDB, 2005.

[19] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In Proc. VLDB, 2002.

[20] J. Teubner, T. Grust, S. Maneth, and S. Sakr. Dependable Cardinality Forecasts for XQuery. In Proc. VLDB, 2008.

[21] S. Wang, E. A. Rundensteiner, and M. Mani. Optimization of Nested XQuery Expressions with Orderby Clauses. DKE Journal.
The inference rule set of Fig. 13 (adopted from [11]) implements a loop-lifting XQuery compiler for the XQuery subset in Fig. 1 taking into account the XML encoding sketched in Section 11. The rule set defines a judgment

\[ \Gamma; \text{loop} \vdash e \Rightarrow q, \]

indicating that the XQuery expression \( e \) compiles into the algebraic plan \( q \), given

1. \( \Gamma \), an environment that maps XQuery variables to their algebraic plan equivalent, and
2. loop, a table with a single column \( \text{iter} \) that invariantly contains \( n \) arbitrary but distinct values if \( e \) is evaluated in \( n \) loop iterations.

An evaluation of the judgment \( \Omega; \text{iter} \vdash e_0 \Rightarrow q_0 \) invokes the compiler for the top-level expression \( e_0 \) (the singleton loop relation represents the single iteration of a pseudo loop wrapped around \( e_0 \)). The inference rules pass \( \Gamma \) top-down and synthesize the plan \( q_0 \) in a bottom-up fashion. A serialization operator at the plan root completes the plan to read \( \varphi(q_0) \).