eXrQuy: Order Indifference in XQuery

Torsten Grust Jan Rittinger Jens Teubner
Technische Universität München, Munich, Germany
E-mail: {torsten.grust,jan.rittinger,jens.teubner}@in.tum.de

Abstract

There are more spots than immediately obvious in XQuery expressions where order is immaterial *for evaluation—this affects most notably, but not exclusively, expressions in the scope of* `unordered{}` *and the argument of* `fn:unordered()`. *Clearly, performance gains are lurking behind such expression contexts but the prevalent impact of order on the XQuery semantics reaches deep into any compliant XQuery processor, making it non-trivial to set this potential free. Here, we describe how the relational XQuery compiler* Pathfinder *uniformly exploits such order indifference in a purely algebraic fashion:* Pathfinder*-emitted plans faithfully implement the required XQuery order semantics but (locally) ignore order wherever this is admitted.*

1. Introduction

XQuery operates over two principal data structures, *ordered* unranked trees of XML nodes and *ordered* finite sequences of items (atomic values or nodes). Indeed, the XQuery Formal Semantics [7] depends on multiple notions of order—*document*, *sequence*, and *iteration order*—which interact in a variety of ways. In consequence, the proper creation and preservation of order (a) constitutes a prevalent concept in the XQuery language definition, and (b) has significant impact on the design and operation of any standards-compliant XQuery processor. As a result, order awareness is quite deeply wired into XQuery processors.

There are, however, more spots than immediately obvious in XQuery where order is either *immaterial* or *not observable* during expression evaluation. The XQuery language definition even provides explicit means—the `unordered{}` expression or, more generally, the *ordering mode* and the built-in function `fn:unordered()`—to locally flag expressions whose evaluation sensibly admits weakened order semantics. Clearly, the prime motivation for posting such flags are the evaluation performance gains lurking behind expression scopes with weak or no order requirements.

Quoting the W3C XQuery Candidate Recommendation [1], *"...a performance advantage may be realized by setting the ordering mode to* `unordered`*, thereby granting the system flexibility to return the result in the order that it finds most efficient."* It is this performance advantage that we are after in the present work. Exploiting such (explicit or implicit) order indifference in originally order-focused XQuery engines is not straightforward, though. In fact, current processors commonly disregard the latent gain and proceed as if strict ordering is required throughout (Section 6).

To demonstrate, assume that XQuery variable `$t` is bound to the XML fragment `<a><b><c/><d/></b><c/></a>`. The corresponding fragment tree is shown in Figure 1. The XPath expression[1]

$$\texttt{\$t//(c|d)}\ , \qquad (1)$$

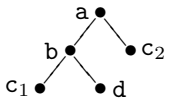


Figure 1. XML tree fragment bound to `$t`.

returns a sequence of `c` and `d` elements in document order. Typically, the query runtime will process the `child::c` and `child::d` steps separately and then *merge* the two node sequences, (c_1, c_2) and `(d)` in this example, in document order to yield (c_1, d, c_2). This evaluation strategy specifically applies to XPath implementations with efficient tag-name-based access to nodes, *e.g.*, via the element streams in *TwigStack* [5].

If we evaluate the above expression in the scope of `unordered{}`, *i.e.*, compute

```
unordered{ $t//(c|d) }
```

we are given the option to return the element nodes in *any order*. One particular order is $(c_1, c_2, \ldots, c_n, d_1, d_2, \ldots, d_m)$ in which all `c` elements precede the `d` elements. While this is only one of the overall $(n+m)!$ admissible orders, this sequence is particularly efficient to produce: the results of the `child::c` and `child::d` steps may simply be *concatenated* (note that, obviously, the two steps yield disjoint results). Simulated on the XQuery language level, the corresponding rewrite would read

```
unordered { $t//c }, unordered { $t//d }.   (2)
```

In effect, the node set union '`|`' has been traded for a low-cost sequence concatenation '`,`'.

Here, we describe how the relational XQuery compiler *Pathfinder* [4] exploits such order indifference in a purely algebraic fashion. *Pathfinder* translates XQuery expressions into relational algebra plans. Since its relational back-end database system provides an inherently unordered runtime environment, this enables *Pathfinder* to emit relational plans which judiciously compute order information only where this is indeed required. The presented technique uniformly covers the various contexts in which order does *not* affect

[1] Remember that e_1`//`e_2 is syntactic sugar for the two-step path e_1`/descendant-or-self::node()/child::`e_2.

expression evaluation: (a) XQuery's *ordering mode* (affecting FLWOR blocks, path expressions, '|', `intersect`, and `except`), (b) `fn:unordered()`, (c) the quantifiers `some` and `every` and the existential semantics of general comparisons, (d) aggregates (`fn:max()`, `fn:count()`, ...), (e) further built-in functions (`fn:empty()`, `fn:exists()`, `fn:boolean()`, ...), and (f) FLWOR blocks whose result is explicitly reordered by an `order by` clause. As of today, we are unaware of other compliant processors which exploit order indifference in XQuery to the degree described here.

We will proceed as follows. Section 2 reviews the notions of orders and their interaction in XQuery and will show that the effects of order indifference may only partially be understood on the XQuery (Core) level. The impact of order semantics on algebraic plans is discussed in Section 3, before Section 4 gives a relational and uniform account of order indifference. The experiments of Section 5 provide the evidence that the performance advantage anticipated by the W3C XQuery Working Group in [1] can indeed be realized. Research in the neighborhood of this work is reviewed (Section 6) before we wrap up in Section 7.

2. Order Semantics in XQuery: Prevalent But Detachable

Whenever expression evaluation is performed under the `ordered` *ordering mode*[2], the semantics of XQuery mirrors the inherent order of its underlying XML data model in various ways. By design, order in the base data, *i.e.*, *document order* (short: *doc*), is maintained whenever such data is accessed: XPath expressions yield node sequences in document order. Likewise, sequence order (*seq*) is observable by an expression when a `for` iteration draws variable bindings from a sequence (*iteration order*, or *iter* for short).

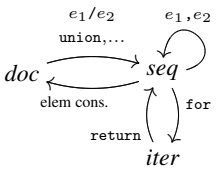


Figure 2. Interaction of orders in XQuery (*ordering mode* `ordered`).

This interaction of order notions in XQuery is captured by Figure 2. Arrow *doc* $\rightarrow$ *seq* for example, indicates that document order determines sequence order during the evaluation of location steps (e_1/e_2) or node set operations (`union` or `|`, `except`, and `intersect`). Since the different types of interactions (as well as their absence) lie at the heart of this work, let us review the effects in more detail. As before, we assume variable `$t` to be bound to the XML fragment of Figure 1.

① **Document order determines sequence order** (*doc* $\rightarrow$ *seq*). We have already observed the effect of this interaction when we discussed the expression `$t//(c|d)` in Section 1: even if document-ordered access paths are available, an XQuery processor may face the requirement to sort node sequences to establish document order.

② **Sequence order establishes document order** (*seq* $\rightarrow$ *doc*). Whenever a query constructs an element, its content sequence determines document order in the newly constructed XML fragment. Consider

```
let $b := $t//b, $d := $t//d,
    $e := <e>{ $d, $b }</e>                (3)
return ($b << $d, $e/b << $e/d)
```

which evaluates to `(true,false)` if applied to the tree fragment of Figure 1: in the new fragment rooted in `e`, element `d` precedes element `b` in document order.

③ **Sequence order determines iteration order** (*seq* $\rightarrow$ *iter*). XQuery adopts a functional style of iteration in which the individual evaluations of the `return` clause in a FLWOR block cannot observe each others' effect. In a `for` iteration like

```
for $x at $p in ("a","b","c")
return <e pos="{ $p }">{ $x }</e>          (4)
```

the evaluations of the `return` clause for the three bindings of `$x` and `$p` may thus occur in arbitrary order or, in principle, in parallel. The individual results of each iteration, however, are then assembled in the sequence order determined by the binding sequence `("a","b","c")` to form the overall result `(<e pos="1">a</e>,<e pos="2">b</e>,<e pos="3">c</e>)`.

④ **Iteration order determines sequence order** (*iter* $\rightarrow$ *seq*). Lastly, the internal order of the individual sequences contributing to the result of a `for` iteration is preserved, *i.e.*,

```
for $x in (1,2) return ($x, $x * 10)       (5)
```

returns `(1,10,2,20)`. (Sequences in XQuery are flat, the two contributing sequences have been marked by ⌊⌋.)

2.1. Partially Detached Order: `unordered{}` and `fn:unordered()`

Depending on the query kind, an XQuery processor may devote a significant share of query evaluation time to properly realize these various interactions of order. (Section 5 contains an exemplary breakdown of where time goes during XQuery expression evaluation.) In response to this observation, XQuery has been equipped with explicit, global as well as local, control over the order semantics in selected parts of an expression:

***Ordering mode* and `unordered{}`.** Any XQuery expression is evaluated under a given *ordering mode* $\in$ {`ordered`, `unordered`}. Local control over the *ordering mode* is exercised by means of the expressions `unordered{`e`}` and `ordered{`e`}` which determine the *ordering mode* for expression e and its sub-expressions. In the query prolog, the declaration `declare ordering` globally sets the *ordering mode*.

[2] Note that *ordering mode* `ordered` merely is a "perceived default": an implementation may choose to operate in default mode `unordered` (Section 2.1) instead [1].

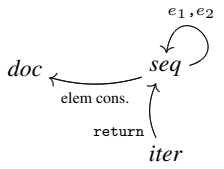


Figure 3. *Ordering mode* `unordered`**.**

Effectively, an *ordering mode* of `unordered` corresponds to the weakened order interaction shown in Figure 3: path expressions may return out-of-document-order node sequences and the bindings of `for`-bound variables may be generated in non-deterministic order. Further order interactions are *not* affected.

To illustrate, removal of $doc \longrightarrow seq$ led to the equivalence of the XQuery Expressions (1) and (2) (see Section 1). Further, Expression (4) may now be evaluated to the sequence `(<e pos="2">b</e>,<e pos="1">a</e>,<e pos="3">c</e>)` or any other permutation of the `e` elements. (Observe, however, that variable `$p`, and thus attribute `pos`, still consistently reflects the position in the binding sequence.) Finally, Expression (5) is admitted to evaluate to `(2,20,1,10)` but *not* `(1,20,2,10)`: order interaction *iter* $\longrightarrow$ *seq* remains intact in Figure 3.

Built-in function `fn:unordered()`**.** To evaluate the function application `fn:unordered(`e`)`, the XQuery processor may *disregard sequence order* in the value of expression e: any permutation of the items in the sequence is an acceptable result. Recast in terms of order interactions, `fn:unordered()` removes the loop *seq* $\circlearrowleft$ in Figures 2 and 3. If Expression (5) is evaluated as the argument of `fn:unordered()`, the result sequence `(1,20,2,10)` would now be admissible (there are 23 more possible results).

This language-level control over weakened order semantics clearly bears promising performance potential, provided that the query runtime can adapt its evaluation mode at the granularity of individual sub-expressions. One stumbling block is the fact that the effects of order indifference may only partially be understood in the language itself, as we will see now.

2.2. Understanding Order Indifference in XQuery Core

The W3C XQuery specification [7] defines the semantics of the language in terms of *normalization rules*, which, in a nutshell, define a mapping $[\![\cdot]\!]$ from XQuery surface syntax to a restricted XQuery Core dialect.

Following this route, we may try to describe the effect of `unordered{}` by

(a) pushing down `unordered{}` through `for` iterations, location steps and node set operations, and
(b) inserting calls to `fn:unordered()` in the appropriate places.

This leads to the normalization rules shown in Figure 4 (these rules emphasize order semantics and deliberately ignore most of the intricate details of XQuery normalization).

$$\frac{\textit{ordering mode} = \texttt{unordered} \qquad [\![e_1]\!] = e_1' \qquad [\![e_2]\!] = e_2'}{[\![\texttt{for } \$x \texttt{ in } e_1 \texttt{ return } e_2]\!] = \texttt{for } \$x \texttt{ in fn:unordered(}e_1'\texttt{) return } e_2'} \text{ (FOR)}$$

$$\frac{\textit{ordering mode} = \texttt{unordered} \qquad [\![e_1]\!] = e_1' \qquad [\![e_2]\!] = e_2'}{[\![e_1/e_2]\!] = \texttt{fn:unordered(}e_1'/e_2'\texttt{)}} \text{ (STEP)}$$

$$\frac{\textit{ordering mode} = \texttt{unordered} \qquad [\![e_1]\!] = e_1' \qquad [\![e_2]\!] = e_2'}{[\![e_1 \texttt{ union } e_2]\!] = \texttt{fn:unordered(}e_1' \texttt{ union } e_2'\texttt{)}} \text{ (UNION)}$$

Figure 4. An attempt to capture the effect of *ordering mode* `unordered` **in XQuery Core. Rules for** `except`**,** `intersect` **not shown (analogous to** UNION**).**

Under *ordering mode* `unordered`, for example, the order of variable bindings in a `for` iteration is non-deterministic. This is exactly what Rule FOR expresses. Similar rules could be used to capture further contexts of order indifference. We have, *e.g.*:

$$\frac{[\![e]\!] = e'}{[\![\texttt{fn:count(}e\texttt{)}]\!] = \texttt{fn:count(fn:unordered(}e'\texttt{))}} \text{ (FN:COUNT)}.$$

Equivalent rules may be formulated for further aggregate functions, as well as `fn:distinct-values()`, `fn:exists()`, `fn:empty()`, *etc.* The quantifier `some` (analogously for `every`) is indifferent to the sequence order of its domain:

$$\frac{[\![e_1]\!] = e_1' \qquad [\![e_2]\!] = e_2'}{[\![\texttt{some } \$x \texttt{ in } e_1 \texttt{ satisfies } e_2]\!] = \texttt{some } \$x \texttt{ in fn:unordered(}e_1'\texttt{) satisfies } e_2'} \text{ (QUANT)}$$

Note that no premise *ordering mode* = `unordered` is needed in the two latter rules: they apply in either *ordering mode* setting. Further XQuery constructs benefit from these introductions of `fn:unordered()`: the normalization of general comparisons is based on the `some` quantifier such that, *e.g.*, the expression e_1 `=` e_2 effectively normalizes to

`some` $\$x$ `in fn:unordered(`e_1`) satisfies`
`some` $\$y$ `in fn:unordered(`e_2`) satisfies` $\$x$ `eq` $\$y$.

The full impact of `unordered{}` and `fn:unordered()`, however, may only incompletely be described on the level of XQuery Core. To see this, consider the nested iteration

```
for $x in (1,2)
for $y in (10,20)          .          (6)
return <a>{$x, $y}</a>
```

In the scope of *ordering mode* `unordered`, two applications of Rule FOR yield the expression

```
for $x in fn:unordered((1,2))
for $y in fn:unordered((10,20)) .      (7)
return <a>{$x, $y}</a>
```

This rewrite fails to fully recognize the freedom introduced by the weakened order requirements. Although `fn:unordered()` introduces a certain degree of non-determinism in (7)—and thus desirable freedom of choice for the runtime system—this expression will, for example, never yield `(<a>1 10</a>,<a>2 10</a>,<a>1 20</a>,<a>2 20</a>)` which is one of the 24 acceptable results of the original Expression (6) under *ordering mode* `unordered`: in (7), variable `$x` remains the outer, `$y` continues to be the inner iteration variable. The semantics of XQuery's *ordering mode* `unordered` do not imply such a restriction, however [7].

Further, a rule similar to FOR cannot be found for iterations using positional variables (`at $p`). Expression

```
for $x at $p in fn:unordered(("a","b","c"))
return <e pos="{ $p }">{ $x }</e>
```

is *not* the equivalent of (4) under *ordering mode* `unordered`: the deterministic association of an item with its position in the binding sequence (*e.g.*, `"a"` occurs at position 1) is lost. Ultimately, this renders Rule FOR unsuitable to understand *ordering mode* `unordered`.

Language-level rewrites similarly fail to explain `fn:unordered()`. In general, "pushing down" `fn:unordered()` tends to strengthen originally weak order requirements: rewriting

```
fn:unordered(for $x in (1,2)
             return ($x, $x * 10))
```

into

```
for $x in fn:unordered((1,2))
return fn:unordered(($x, $x * 10))
```

only admits $1/6$ of the originally 24 possible sequence permutations.

Finally, XQuery Core-level query rewriting in the presence of `unordered { }` is context-dependent. Unfolding the `let` binding[3] in (as before, assume `$t` to be bound to the fragment of Figure 1)

```
let $c2 := $t//c[2]
return unordered{$c2}
```

to obtain

```
unordered{$t//c[2]}
```

illegitimately introduces non-determinism: while the former returns element c_2, the latter might evaluate to any of the two `c` elements.

Normalization goes some way to grasping the effects of order indifference in the language itself. The above-mentioned difficulties, however, are essential. This is also reflected in the W3C XQuery specification documents: a formal description of the impact of order indifference on the dynamic semantics of FLWOR blocks is explicitly omitted [7, § 4.8].

[3] This is a variation of an example due to Daniela Florescu.

Further, language definition aside, it still remains unclear *how* the query compiler and its runtime environment can specifically exploit the fact that a sub-expression occurs in the scope of weakened order requirements. This is what we shift focus to now.

3. Relational XQuery and Order Interaction

Undeniably, the correct and efficient implementation of weakened order semantics represents a challenge for any XQuery processor (Section 6). In the *purely relational* XQuery compiler *Pathfinder* [4] we indeed have the opportunity to perform *less work* whenever order is not observable by expressions. *Pathfinder* is purely relational in the sense that the principal structures of the XQuery data model—unranked trees and item sequences—as well as the dynamic semantics (evaluation) of XQuery are exclusively implemented in terms of relational idioms. Relational database kernels are probably the best understood as well as the best engineered query engines available today. The ultimate goal of the *relational XQuery* idea is to inherit this efficiency and scalability.

Here, we will review those bits of *Pathfinder* which are affected by order indifference. The subsequent developments directly plug into prior descriptions of this technology [10, 11].

Encoding trees and item sequences. *Pathfinder* can operate with any schema-oblivious XML tree encoding that assigns document order-preserving node identifiers. Viable candidates are, *e.g.*, ORDPATH labels [16] or preorder ranks [12]. Figure 5 depicts the XML fragment of Figure 1 in which nodes have been identified with their preorder rank, an integer $0, 1, 2, \ldots$. Element `b` precedes `d` in document order which is also witnessed by their preorder ranks: $1 < 3$.

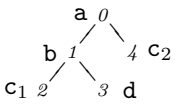


Figure 5. Encoded XML fragment (preorder ranks $0..4$).

To retain order in a purely relational setting, *Pathfinder* represents the item sequence $(i_1, i_2, \ldots, i_n)$, $n \geqslant 0$, in terms of a (possibly empty) two-column table exhibiting an explicit `pos` column. The items i_k either denote nodes (represented by their identifiers) or atomic values. In this simple model, the value of `fn:unordered((`$i_1, i_2, \ldots, i_n$`))` corresponds to a `pos item` relation in which column `pos` is arbitrarily populated with numbers $\{1, \ldots, n\}$ (or `pos` is missing at all).

pos	item
1	i_1
2	i_2
⋮	⋮
n	i_n

Dynamic semantics. *Pathfinder*'s code generator emits query plans that use the operators of a rather restricted variant of classical relational algebra (Table 1). The design of this algebra dialect has been guided by the processing capabilities of SQL-centric relational database kernels. For example, column projection π does not

remove duplicate rows and the row numbering primitive $\varrho_{\mathsf{a}:\langle \mathsf{b}\rangle\|\mathsf{c}}$ exactly mimics the functionality of the `ROW_NUMBER() OVER (PARTITION BY c ORDER BY b) AS a` ranking operator found in the SQL:1999 OLAP amendment. $\varrho_{\mathsf{a}:\langle \mathsf{b}\rangle\|\mathsf{c}}(q)$ groups input table q by column c and extends q by new column a containing a dense numbering $(1, 2, 3, \ldots)$ of the rows in each group. The row order in each group is determined by the sort criterion (column) b. Grouping is optional: $\varrho_{\mathsf{a}:\langle \mathsf{b}\rangle}(q)$ treats table q as a single large group.

Pathfinder-emitted algebraic code exhibits a number of restrictions which make the plans particularly amenable to analysis—enabling, *e.g.*, algebraic XQuery join recognition [9]—and simplification. A simple form of data flow analysis will be used in Section 4.

As is to be expected, the various interactions of orders in XQuery have a direct impact on the generated plans. We zoom in on two types of interaction below.

Order interaction ①: *doc* → *seq*. For XPath location step evaluation, *Pathfinder* relies on invocations of the step operator $\text{◰}_{ax::nt}(q)$: operator ◰ is parameterized by the XPath axis and node test $ax::nt$ and consumes a table q whose item column contains context node identifiers. Likewise, a duplicate-free table of encoded nodes constitutes the output of ◰. Several existing XPath step evaluation techniques may be plugged in to realize ◰, among these are *TwigStack* [5] or *staircase join* [12].

Consider variable `$t` to be bound to the root element `a` (preorder rank *0*) of the XML fragment in Figure 5. To evaluate the step `$t//(c|d)`, the compiled plan will make use of ◰, consuming as input the unary context node table

item
0

. The output will be the table shown on the left, containing the preorder ranks of the three element nodes $\mathtt{c}_1, \mathtt{c}_2, \mathtt{d}$ in *some* order that is determined by the actual XPath step evaluation technique. Under XQuery *ordering mode* `ordered`, document order determines sequence order after step evaluation. To adhere, *Pathfinder* wraps the invocation of ◰ in a call to ϱ to derive sequence order (*i.e.*, column pos) from the order-preserving node identifiers returned by the step:

item
4
2
3

$$\varrho_{\mathsf{pos}:\langle \mathsf{item}\rangle}\left(\begin{array}{|c|}\hline \mathsf{item} \\ \hline 4 \\ 2 \\ 3 \\ \hline \end{array}\right) \quad = \quad \begin{array}{|c|c|}\hline \mathsf{pos} & \mathsf{item} \\ \hline 3 & 4 \\ 1 & 2 \\ 2 & 3 \\ \hline \end{array} \; .$$

The *Pathfinder* compiler is specified in terms of inference rules which collectively define function $\cdot \Rightarrow \cdot$ (read: *compiles to*) from XQuery expressions to algebraic code. To compile an XPath location step $e/ax::nt$, Rule LOC is invoked (disregard the grouping ‖iter for now):

$$\frac{\textit{ordering mode} = \texttt{ordered} \qquad e \Rightarrow q_e}{e/ax::nt \Rightarrow \varrho_{\mathsf{pos}:\langle \mathsf{item}\rangle\|\mathsf{iter}}(\pi_{\mathsf{iter},\mathsf{item}}(\text{◰}_{ax::nt}(q_e)))} \;(\textsc{Loc})$$

The complete inference rule set is developed and discussed in [11].

Order interaction ③: *seq* → *iter*. To exploit the parallelism inherent in XQuery's functional iteration style (Section 2), *Pathfinder*-emitted code collects all bindings of a `for`-bound variable into a *single* table. In each iteration, an XQuery variable is bound to a single item i, which corresponds to one row $\langle 1, i\rangle$ in the pos item tables. Much like column pos maintains sequence order, we use an additional column iter to keep track of *iteration order*. The complete relational encoding of variable `$x` in Expression (4) thus is the table shown here.

iter	pos	item
1	1	"a"
2	1	"b"
3	1	"c"

These iter pos item tables are the prevalent representation of evaluated expressions (*i.e.*, item sequences) in *Pathfinder*'s XQuery compilation scheme: in what follows, a row $\langle i, p, v\rangle$ in the table representing the value of an expression e may invariably be read as "*in iteration i, e assumes item value v at the sequence position corresponding to p's rank in column* pos."

To properly implement the *seq* → *iter* order interaction in `for` $\$x$ `in` e_1 `return` e_2, *Pathfinder* generates code that derives iteration order (column iter) from the order in the binding sequence e_1. Assume $e_1 \Rightarrow q_{e_1}$. The relational encoding of the bindings of variable `$x` may then be computed by the mini-plan embedded in compilation Rule BIND below. Again, order interaction is realized in terms of the row numbering primitive ϱ:

$$\frac{\textit{ordering mode} = \texttt{ordered} \qquad e_1 \Rightarrow q_{e_1}}{\$x \;\texttt{in}\; e_1 \Rightarrow \begin{array}{|c|}\hline \mathsf{pos} \\ \hline 1 \\ \hline \end{array} \times \pi_{\mathsf{iter}:\mathsf{bind},\mathsf{item}} \;\; \varrho_{\mathsf{bind}:\langle \mathsf{iter},\mathsf{pos}\rangle}\; q_{e_1}} \;(\textsc{Bind})$$

Applied to `$y` and its binding sequence e_1 = `(10,20)` in the inner `for`-iteration of Expression (6) we have

$$e_1 \Rightarrow \begin{array}{|c|c|c|}\hline \mathsf{iter} & \mathsf{pos} & \mathsf{item} \\ \hline 1 & 1 & 10 \\ 1 & 2 & 20 \\ 2 & 1 & 10 \\ 2 & 2 & 20 \\ \hline \end{array} \quad \text{and} \quad \$y \;\texttt{in}\; e_1 \Rightarrow \begin{array}{|c|c|c|}\hline \mathsf{iter} & \mathsf{pos} & \mathsf{item} \\ \hline 1 & 1 & 10 \\ 2 & 1 & 20 \\ 3 & 1 & 10 \\ 4 & 1 & 20 \\ \hline \end{array} \; .$$

This meets the XQuery semantics of nested iteration: in iterations 1 and 3 of the `return` clause in (6), `$y` is bound to item 10, in iterations 2 and 4, `$y` is bound to 20.

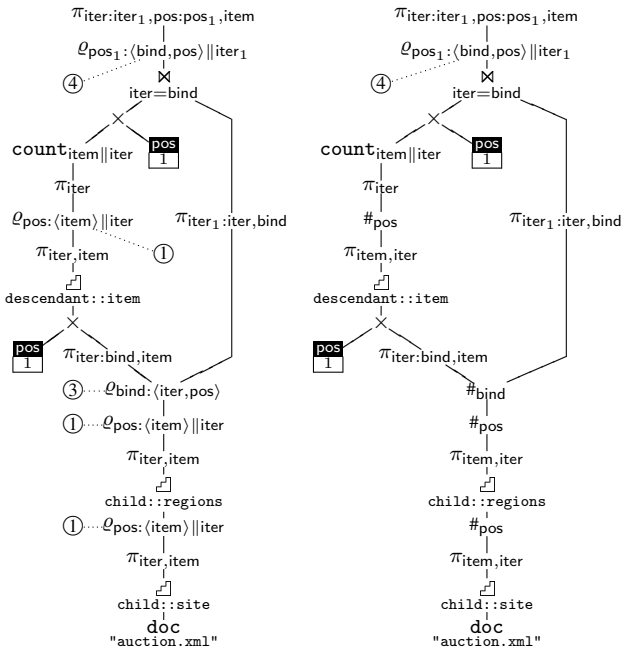
Impact of order interaction. *Pathfinder* applies the rules of compilation scheme $\cdot \Rightarrow \cdot$ in a bottom-up fashion to build a DAG of algebraic operators: the emitted code contains significant sharing opportunities [9, 10]. Figure 6(a) depicts the DAG for query *Q6* of the XMark benchmark [18]:

```
for $b in doc("auction.xml")/site/regions
return fn:count($b/descendant::item)
```
.(*Q6*)

The plan of Figure 6(a) faithfully implements the semantics of XQuery's *ordering mode* `ordered` which accounts for all five ϱ primitives among its overall 19 operators (the annotations in ◯ refer to the types of order interaction identified in Section 2, *e.g.*, ① indicates a *doc* → *seq* interaction). Since

$\pi_{a,b:c}$	project onto col.s a, c, rename c into b	$\varrho_{a:\langle b\rangle \| c}$	group rows by c, order by b, then number rows $(1, 2, \dots)$ in new col. a
σ_a	select rows with col. a $= \mathit{true}$	$\#_a$	unsorted (arbitrary) row numbering in new col. a
$\bowtie_{a=b}$	equi-join	$\circledcirc_{a:(b,c)}$	apply $\circ \in \{*, =, <, \dots\}$ to b, c, result in new col. a
$\times$	Cartesian product	$\texttt{count}_{a\|\|b}$	grouped row aggregation (counting) in new col. a
$\dot\cup$	disjoint union (append)	$\text{⌐}_{ax::nt}$	XPath step evaluation (axis ax, node test nt)
a b c	literal table with columns a, b, c	doc	XML document encoding access (fn:doc())

Table 1. Compilation target: relational algebra (excerpt).



(a) *Ordering mode* ordered. (b) *Ordering mode* unordered.

Figure 6. Plan emitted for *Q6* under varying *ordering mode* declarations.

an implementation of ϱ will typically require a blocking sort of its input, this gives a clear indication of the performance advantage that lies in wait in scopes of order indifference.

4. Algebraic Order Indifference

It is now only a small step to see how order indifference on the XQuery language level surfaces in the algebraic code:

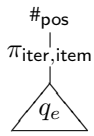
(a) if an expression is indifferent with respect to *sequence order*, its corresponding compiled code may populate column pos with *arbitrary* (although unique) values. Likewise,

(b) if an expression does not depend on *iteration order*, column iter may contain *arbitrary* unique values.

Note that we may not simply drop columns pos and iter. Function $\cdot \Rightarrow \cdot$ constitutes a *compositional* compilation scheme in which a sub-expression may be compiled independently of its enclosing expression: code for the latter will be generated under the assumption that the downstream plan generates columns pos and iter (Section 3). More importantly, however, the compiler acquires the ability to freely mix order-dependent as well as order-indifferent code, a feature that mirrors XQuery's capability to let go of order locally.

Whenever the compiler plans for the random population of a column c it places operator $\#_c$ (Table 1) in the DAG: $\#_c$ attaches new column c containing arbitrary unique numbers to its input table. $\#_c$ comes at negligible cost or may even be "for free": a table's hidden (or virtual) ROWID column which is typically maintained by relational database kernels makes for a perfect column c.

Compiling fn:unordered(). The compilation Rule FN:UNORDERED for fn:unordered(e) (Figure 7) directly reflects the above observation about algebraic *sequence order* indifference. The rule attaches the operator pair $\#_{pos}$—$\pi_{iter,item}$ on top of the plan q_e for e and effectively "overwrites" any sequence order information contained in its input q_e. Obviously, there is no need to generate column pos in q_e at all (this is addressed in Section 4.1).



Together with normalization rules like QUANT and FN:COUNT which insert calls to fn:unordered() on the language level (Section 2.2), Rule FN:UNORDERED suffices to uniformly introduce the order indifference inherent in these XQuery constructs into the algebraic code. No further specific treatment is required.

Compilation and *ordering mode* unordered. Recall that the compilation rules for XPath location steps and the generation of variable bindings in a for-iteration, Rules LOC and BIND (Section 3), use operator ϱ to implement the order interactions *doc* $\longrightarrow$ *seq* and *seq* $\longrightarrow$ *iter*, respectively. In Figure 7, we introduce the twin Rules LOC$^\#$ and BIND$^\#$ which carry the premise *ordering mode* $=$ unordered to ensure that the compiler picks the correct rule in dependence on the current *ordering mode*. Compared to their ordered counterparts, both rules trade ϱ for its unordered variant $\#$: (a) in Rule LOC$^\#$, sequence order is arbitrary ($\#_{pos}$) and does not depend on the document order of the nodes returned by ⌐, and (b) in Rule BIND$^\#$, iteration order is arbitrary since $\#_{bind}$ followed by $\pi_{iter:bind,\dots}$ effectively overwrites the iteration order information present in q_{e_1}.

In principle, the addition of these two rules is the only change needed to make the compiler aware of XQuery's *ordering mode*. Note that Rule LOC$^\#$ is not strictly

$$\frac{e \Rightarrow q_e}{\texttt{fn:unordered}(e) \Rightarrow \#_{\mathsf{pos}}(\pi_{\mathsf{iter},\mathsf{item}}(q_e))}\ (\textsc{Fn:Unordered})$$

$$\frac{\textit{ordering mode} = \texttt{unordered} \qquad e \Rightarrow q_e}{e/ax::nt \Rightarrow \#_{\mathsf{pos}}(\pi_{\mathsf{iter},\mathsf{item}}(\square ax::nt(q_e)))}\ (\textsc{Loc}^{\#})$$

$$\frac{\textit{ordering mode} = \texttt{unordered} \qquad e_1 \Rightarrow q_{e_1}}{\$x \texttt{ in } e_1 \Rightarrow \boxed{\mathsf{pos}\ 1} \times \pi_{\mathsf{iter}:\mathsf{bind},\mathsf{item}}(\#_{\mathsf{bind}}(q_{e_1}))}\ (\textsc{Bind}^{\#})$$

Figure 7. Extensions to make *Pathfinder*'s compilation scheme $\cdot \Rightarrow \cdot$ aware of order indifference.

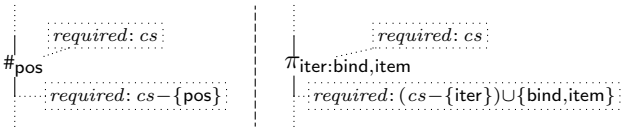


Figure 8. Top-down inference of strictly required input columns for # and π. DAG annotations in ⋯□.

needed, even: its effect is essentially the same as the composition of the Rules STEP (normalization, introduction of `fn:unordered()`) and FN:UNORDERED. Remember that a similar observation does not apply to `for`-iterations, though (Section 2.2).

With the declaration `declare ordering unordered` added to the prolog of XMark query *Q6*, the modified compiler emits the plan DAG of Figure 6(b). The effects of algebraic order indifference are clearly visible: all ϱ operators but one have been traded for #. The remaining ϱ operator implements the order interaction *iter* $\rightarrow$ *seq* which is not disabled by *ordering mode* `unordered` (but see Section 7).

4.1. Simple Column Dependency Analysis

Note that, although the # operators come at negligible cost, the plan still performs a significant share of wasted data movement: repeatedly, pos columns are introduced only to be disregarded and overwritten upstream (in Figure 6(b), note operators $\#_{\mathsf{pos}}$ indirectly followed by $\#_{\mathsf{pos}}$, or $\#_{\mathsf{pos}}$ followed by a Cartesian product that installs a constant pos column). As already indicated at the beginning of Section 4, this is a consequence of the compiler's compositional specification.

A simple form of column dependency analysis suffices to counter this effect. The compiler walks the plan DAG top-down to infer the set cs of *strictly required input columns* for each operator. At the root of the DAG, this inference process is seeded with the column set $\{\mathsf{pos}, \mathsf{item}\}$: these two columns are required to properly serialize the item sequence which forms the result of a query. Figure 8 depicts the inference for the operators # and π. Once these column dependencies have been recorded, the compiler may simplify or even prune operators which produce columns irrelevant for plan evaluation. This process is reminiscent of *projection pushdown* optimizations which are standard in relational query processors [13].

Column dependency analysis turns out to be quite effective to implement order indifference. For the plan of query *Q6* under *ordering mode* `unordered` (Figure 6(b)), note how the analysis finally realizes the order indifference we have introduced through Rules $\textsc{Loc}^{\#}$ and $\textsc{Bind}^{\#}$: order is (almost) no concern in the simplified plan of Figure 9. For XMark query *Q11* (Figure 11), which we will revisit in Section 5, the initial plan DAG of 235 operators is cut down to 141 nodes after the analysis.

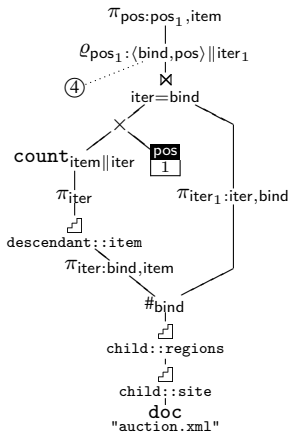


Figure 9. Plan of Fig. 6(b), column dependency analysis applied.

4.2. Effects of Order Indifference

Trading '|' for ',' . Recall our initial example expression `unordered{$t//(c|d)}` of Section 1. The compiler will (1) apply Rule STEP to rewrite the expression into `fn:unordered($t//(c|d))`. It is now explicit that the node sequence resulting from the XPath location step may be arbitrarily ordered. (2) In the algebraic plan, this order indifference is reflected by the application of Rule FN:UNORDERED which installs $\#_{\mathsf{pos}}$—$\pi_{\mathsf{iter},\mathsf{item}}$ on top of the plan for `$t//(c|d)` (Figure 10, left). This plan still exhibits instances of ϱ to implement order interaction $doc \rightarrow seq$ after location steps (□/) and the node sequence union '|' (□□). (3) Subsequent application of column dependency analysis detects column pos to be obsolete and leads to the final plan of Figure 10 (right). Note that $\dot{\cup}$ merely concatenates its argument relations (the algebraic equivalent of item sequence concatenation ','), such that the compiler indeed arrived at the algebraic equivalent of `unordered{$t//c}, unordered{$t//d}`.

5. Quantitative Assessment

Here we report on the performance advantage realized by the relational XQuery processor *Pathfinder* once we made it aware of order indifference: we modified normalization $[\![\cdot]\!]$ to introduce calls to `fn:unordered()` (Section 2.2), added the compilation Rules FN:UNORDERED, $\textsc{Loc}^{\#}$, and $\textsc{Bind}^{\#}$ as well as column dependency analysis (Section 4).

Pathfinder and MonetDB/XQuery[4] The experiments in the following were performed with *MonetDB/XQuery*, one of

[4] *Pathfinder* (packaged with *MonetDB/XQuery*) is available for download at `pathfinder-xquery.org`.

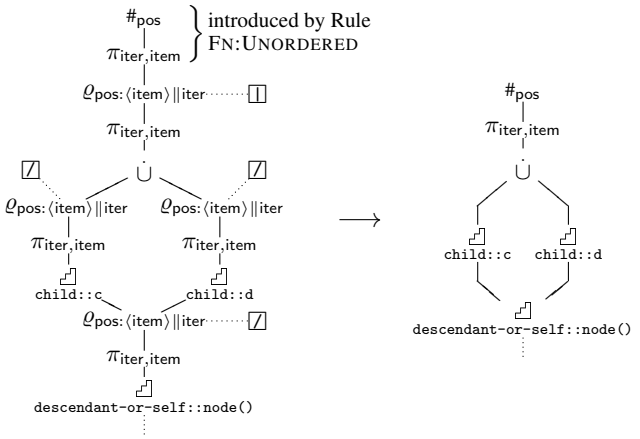


Figure 10. A document order-aware union '|' is cut down to sequence concatenation ',' (left: before, right: after column dependency analysis).

```
let $auction := doc("auction.xml")
for $p in $auction/site/people/person
let $l := for $i in $auction/site/open_auctions/
                      open_auction/initial
          where $p/profile/@income > 5000 * $i
          return $i
return <items name="{$p/name}">                    (Q11)
         {fn:count($l)}
       </items>
```

Figure 11. Query *Q11* of the XMark benchmark.

Pathfinder's relational back-end database systems. *MonetDB* [2] is an extensible database kernel whose internals have been optimized to perform query execution close to the CPU (*i.e.,* in the CPU caches or primary memory). In *MonetDB*, tables undergo full vertical fragmentation: any n-ary table is split into n binary tables (BATs). The narrow iter pos item tables emitted by compilation scheme $\cdot \Rightarrow \cdot$ fit the BAT model quite well. Further, a number of idioms which are common in *Pathfinder*-emitted plans—*e.g.*, occurrences of $\#_a$ and Cartesian products with one-column singleton tables like $\boxed{\text{pos}\ 1}$—operate on table descriptors rather than on individual rows and thus are almost for free in *MonetDB*. The system's implementation of the XPath location step operator ⏍ is based on *staircase join* [12]. *MonetDB/XQuery* itself is described in [3, 4].

The timings in this section were recorded on a Linux-based host, equipped with two 3.2 GHz Intel Xeon® CPUs, 8 GB of primary memory, and 280 GB of secondary memory residing on two SCSI hard disks.

Query Profiling. To quantify the potential of order indifference, we dissected the performance profile of query *Q11* of the XMark benchmark [18]. From the two nested `for`-iterations in Query *Q11* (Figure 11), *Pathfinder* derives that the query effectively performs a value-based join—using the general comparison operator `>`—between `person` and

Sub-expression	Time [ms]	%
`$auction/site/people/person`	107	< 1 %
`$auction/site/···/initial`	144	< 1 %
`···/@income, 5000 * $i` (+ atomization)	949	2 %
join (of `$p` and `$i`)	23,989	45 %
`return $i` (*iter* → *seq*)	23,861	45 %
`<items name=··· </items>`	627	1 %
`fn:count($l)`	3,367	6 %
	53,044	

Table 2. Profile breakdown for XMark Query *Q11*.

`initial` elements in an XMark document instance. This join recognition process is described in [9]. The execution time profile, summarized in Table 2, was recorded for an `auction.xml` instance of 558 MB (23, 513, 044 nodes). For this instance size, query *Q11* inspects and creates significant amounts of data: the join returns more than 315, 000, 000 `initial` elements (join selectivity is 4 %) and the outer `return` clause is evaluated 127, 500 times, *i.e.*, 127, 500 new `items` elements and `name` attribute nodes are created and as many invocations of `fn:count()` are processed.

Table 2 contains a breakdown—in milliseconds (ms) as well as fractions of overall execution time—of where time goes during evaluation of *Q11* if the compiler ignores order indifference. Note that the table shows aggregated execution times. The 3, 367 ms recorded for `fn:count()`, for example, summarize the time spent in all 127, 500 calls to this function. Since the implicit join in *Q11* has been picked up by *Pathfinder*'s code generator, the two path expressions which generate the bindings for variables `$p` and `$i` are evaluated once only.

The lion's share of execution time, 90 %, is allocated to join evaluation and the proper realization of order interaction *iter* → *seq* (type ③). The 315, 000, 000 bindings for `$i`, the iteration variable of the *inner* `for`-block, have to be reordered after the join: the `ordered` XQuery semantics strictly prescribes that the primary sort criterion is the binding order in the *outer* `for`-block (cf. the discussion of Expressions (6) and (7) in Section 2.2).

While the join complexity is inherent in *Q11*, the enforcement of *iter* → *seq* clearly is wasted effort since the join result (bound to variable `$l`) is evaluated as an argument to `fn:count()`. In the modified compiler, normalization Rule FN:COUNT introduces a call to `fn:unordered()` which, via FN:UNORDERED, leads to the insertion of the operator pair $\#_{pos}$—$\pi_{iter,item}$ on top of the join code section. Subsequent column dependency analysis effectively removes the *iter* → *seq* interaction indicated in Table 2. In this case, the runtime system saves 45 % of the overall execution time.

XMark and Order Indifference. Substantial performance advantages are realizable throughout the complete XMark

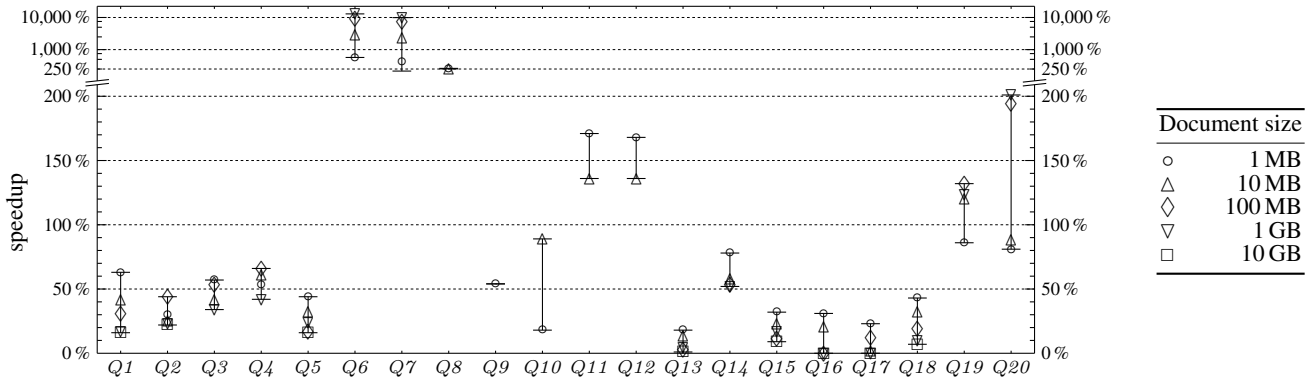


Figure 12. Observed impact of order indifference (speedup) on the XMark benchmark query set.

benchmark query set. Figure 12 reports on the speedup we observed if order indifference is enabled in *Pathfinder*. We measured wall clock execution times for the 20 XMark benchmark queries which were evaluated against XML document instances ranging from 1 MB up to 10 GB serialized size. (We recorded the timings for those queries which completed in interactive time—the cutoff time was set to 30 s; on the 10 MB instance, almost all queries ran in less than 1 s.) In this figure, a speedup of $100\,\%$ indicates that *Pathfinder* was able to generate algebraic code that executed twice as fast due to the exploitation of order indifference. The observed speedup falls into a range of 0 to $10,000\,\%$ (notice the logarithmic scale above the gap $\neq$ in the y-axis). The exceptional speedup for queries *Q6* and *Q7* is due to the removal of a ϱ operator that, in the initial plan, separated two step operators $\text{⌐}_{\texttt{descendant-or-self::node()}}$ and $\text{⌐}_{\texttt{child::}nt}$ where the context nodes for the first step were located close to the document root. After column dependency analysis, the now adjacent steps could be merged into $\text{⌐}_{\texttt{descendant::}nt}$.

6. Research in the Neighborhood

The exploitation of order indifference reaches quite deeply into the operation of an XQuery processor and, depending on the design of its internals, may call for substantial architectural changes. In fact, with the exception of Saxon [14], we have found no traces of order indifference in open-source XQuery engines (`fn:unordered()` is commonly implemented as the identity function).

Galax incorporates a limited notion of order indifference—based on the *duptidy* automaton of [8]—which is focused on the order of the node sequence that results from a series of XPath location steps.

The native XML database system *Timber* can correctly reflect local XQuery document and iteration order indifference (*Timber* does not support general item sequences or sequence order) since it has been extended with a new generic hybrid collection type [17]. For XQuery FLWOR blocks, *Timber* derives sort specifications which are used to instantiate a collection as a set or sequence and to parameterize operators of *Timber*'s query algebra. While restricted to a very limited XQuery dialect, *Timber*'s approach arguably seems more intricate and invasive than the simple pos-column based approach.

The normalization Rules STEP and UNION have counterparts in the specification of the XQuery Formal Semantics, where they are formulated in terms of the auxiliary function `fs:apply-ordering-mode()` [7, § 7.1.10]. Note that, for reasons explained in Section 2.2, no equivalent for Rule FOR is found in [7].

In the completely vertically fragmented binary table (BAT) model of *MonetDB* [2], the column dependency analysis of Section 4.1 bears some close resemblance with the *dead code* (or *dead variable*) elimination found in programming language compilers [6]. Whenever *Pathfinder* infers that a column, say c, is not strictly required, this corresponds to dead code fragments in the procedural-style code generated for the *MonetDB* back-end: the code will create and populate the BAT for column c but the table will never be accessed later on. The column dependency analysis will remove all references to column c in the plan DAG will ultimately lead to the removal of the dead back-end code.

While this work purely argues on the *logical* algebra layer, the authors of [15] derive ordering and grouping information on the basis of *physical* plans. Physical plan optimization is orthogonal to the present work and may lead to additional enhancements in *Pathfinder*'s code generator. The techniques of [15] might infer, for example, that a particular sub-plan yields rows in $\langle \mathsf{b}, \mathsf{c} \rangle$ order (column b is the primary order criterion, column c secondary). This renders subsequent $\varrho_{\mathsf{a}:\langle \mathsf{b},\mathsf{c} \rangle}$ or $\varrho_{\mathsf{a}:\langle \mathsf{c} \rangle \| \mathsf{b}}$ operators as cheap as $\#_{\mathsf{a}}$.

Note that the present work, in a sense, promotes the concept of *interesting orders* as they were introduced in the seminal work on the System R optimizer [19]: order indifference leads to choice in the algebraic plans and row orders

which were formerly incompatible with the XQuery semantics may prove to be interesting.

7. Wrap-Up

It turned out that a fine-grained control over order indifference in XQuery indeed leads to the significant performance advantage anticipated by the W3C XQuery Working Group in [1]. We have seen, though, that such control has to be exercised below the XQuery language level in order to fully comprehend and then realize the impact of weakened order semantics. The prevalence of orders and their intricate interaction in XQuery makes this quite a challenging problem.

A key concept of the present work are the row numbering primitives ϱ and # which, in a sense, move the concept of order from the query runtime into the algebraic code generator. This brought forth the required hooks to detect that the enforcement of order is strictly needed or, far more interesting in this context, rendered obsolete in specific subexpressions. The required changes to the query compiler are limited: we extended normalization $[\![.]\!]$, added less than a handful of compilation rules, and introduced a simple form of column dependency analysis into the property inference framework that is in place in *Pathfinder* anyway.

Recall that in Figure 9 one row numbering operator, namely $\varrho_{\mathsf{pos}_1:\langle \mathsf{bind},\mathsf{pos}\rangle \| \mathsf{iter}_1}$, persisted even after column dependency analysis. At this point in the plan, *Pathfinder*'s property inference has derived that all entries in columns iter_1 are equal (the same holds for column pos) [9]. This (a) renders the grouping in the above ϱ obsolete and (b) identifies pos as a useless order criterion that may be removed. Further, column bind is found to contain arbitrary, although unique, entries: bind may thus be safely removed from the list of ϱ's order criteria as well. We are left with $\varrho_{\mathsf{pos}_1:\langle\rangle}$ which attaches a new, arbitrarily ordered, densely numbered column pos_1 to its input table (Section 3). Obviously, this operator comes "for free"—which ultimately removes any residual traces of order in the plan for *Q6*.

Finally, we read the execution profile in Table 2 as an indication of where research into the construction of XQuery processors should head: the evaluation of XPath location steps, for example, seems to be well understood by now—further XQuery concepts (*e.g.*, efficient atomization) seem to deserve as much attention in the future.

Acknowledgment. This research has been supported by the Deutsche Forschungsgemeinschaft (DFG) under grant GR 2036/2-1.

References

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

[2] P. Boncz. *Monet: A Next-Generation DBMS Kernel for Query-Intensive Applications*. PhD thesis, University of Amsterdam, The Netherlands, May 2002.

[3] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. Pathfinder: XQuery—The Relational Way. In *Proc. VLDB*, Trondheim, Norway, 2005.

[4] 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.

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

[6] J. W. Davidson and C. Fraser. The Design and Application of a Retargetable Peephole Optimizer. *ACM TOPLAS*, 2(2), 1980.

[7] D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3 Consortium, June 2006.

[8] M. Fernández, J. Hidders, P. Michiels, J. Siméon, and R. Vercammen. Optimizing Sort and Duplicate Elimination in XQuery Path Expressions. In *Proc. DEXA*, Copenhagen, Denmark, 2005.

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

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

[11] T. Grust and J. Teubner. Relational Algebra: Mother Tongue—XQuery: Fluent. In *Proc. of the 1st Twente Data Management Workshop (TDM)*, Enschede, The Netherlands, 2004.

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

[13] M. Jarke and J. Koch. Query Optimization in Database Systems. *ACM Computing Surveys*, 16(2), 1984.

[14] M. Kay. The Saxon XSLT and XQuery Processor. `http://saxon.sf.net/`.

[15] G. Moerkotte and T. Neumann. A Combined Framework for Grouping and Order Optimization. In *Proc. VLDB*, Toronto, Canada, 2004.

[16] 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*, Paris, France, 2004.

[17] S. Paparizos and H. V. Jagadish. Pattern Tree Algebras: Sets or Sequences? In *Proc. VLDB*, Trondheim, Norway, 2005.

[18] 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.

[19] P. G. Selinger, M. M. Astrahan, D. M. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proc. SIGMOD*, Boston, USA, 1979.