# Purely Relational FLWORs

Torsten Grust
Technical University of Munich
Department of Computer Science, Database Systems
Munich, Germany
grust@in.tum.de

## ABSTRACT

We report on a compilation procedure that derives relational algebra plans from arbitrarily nested XQuery FLWOR blocks. While recent research was able to develop relational encodings of trees which may turn RDBMSs into highly efficient XPath and XML Schema processors, here we describe *relational encodings of nested iteration*, *variables*, and the *item sequences* to which variables are bound. The developed techniques are *purely relational* in more than one sense: (a) we rely on a standard (or rather: classical) algebra that is readily supported by relational engines, and (b) we use relational concepts like functional and multivalued dependencies to significantly simplify the emitted plans. This work blends well with the mentioned tree encodings and thus contributes a further important building block to investigations into XQuery processors based on relational database technology.

## 1. INTRODUCTION

In a sense, the relational query processing model may be described as rather simplistic, on multiple levels: on the logical level, sequences of tuples (tables) are fed into algebraic operators which generate such sequences again, and the implementations of these operators work best if the tuple sequences are stored in a physically contiguous (secondary) memory fragment. While this simplicity is the key to the efficiency of relational query engines, it also prescribes a rather restrictive mode of data access and processing.

It is all the more remarkable that the database research community succeeded in turning RDBMSs into efficient processors for originally non-tabular *tree-shaped data*: trees are relationally encoded such that core operations on this data type, *e.g.*, path traversals or the validation of a tree's structure, are mapped onto sequential tuple scans which relational engines inherently support well.

Such tree encodings have primarily been used to devise relational XPath processors [9,11], but they may also yield efficient implementations of XML Schema validators [6]. Among other benefits, the resulting systems inherit the scalability of the underlying relational back-ends [3]. It is legitimate to hope that this technology may be developed into full-fledged XQuery implementations, given that we can find relational ways to also express XQuery concepts beyond XPath axis traversals.

To this end, this paper does *not* talk about XPath evaluation at all but shifts focus to *the* central XQuery language feature, the `for-let-where-order by-return` (or FLWOR) block [2]. The presence of arbitrarily nested iteration as well as the possibility to bind and then refer to variables in expressions presents an interesting technical challenge since the compilation target—relational algebra—is a combinator-style language that lacks any notion of explicit iteration or variables.

In Section 2, we start off by sketching the ideas which underly the compilation of nested XQuery FLWOR blocks into relational algebra. The canonical compilation yields relational plans featuring particularly simple instantiations of relational operators, *e.g.*, all occurring joins are equi-joins, but of significant size and certain redundancy: the item sequence representation is *denormalized* (Section 3). Due to the inherent simplicity of the plans, however, we are able to use a simple form of plan analysis to detect, among other properties, induced functional and multivalued dependencies which we then use in highly effective peep-hole-style plan simplification (Section 4). We will also sketch how the analysis of such dependencies may enable non-syntactic XQuery join detection (Section 5). The resulting relational XQuery processor supports the XQuery dialect sketched in Table 1 and is able to evaluate the XMark benchmark [12] queries on document sizes beyond 1 GB in interactive time (Section 6).

## 2. RELATIONAL FLWORs

We will now describe techniques which enable the compilation of the mentioned XQuery dialect (Table 1) into the relational algebra shown in Table 2. Besides the *row numbering* operator $\varrho$[1], this constitutes a classical, rather restricted relational algebra variant. We will discuss the specifics of some operators as we go.

XQuery operates with *ordered, finite sequences of items* as the principal data type. *Items* either represent atomic values (of one of the XML Schema atomic types, *e.g.* `string`, `decimal`, `positiveInteger`) or nodes. For a relational representation of an atomic value $v$, we choose an implementa-

[1]Many relational engines implement $\varrho$ in terms of the `DENSE_RANK` operator defined by SQL:1999 [7].

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

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

| Operator | Semantics |
|---|---|
| $\sigma_{\mathsf{a}}$ | select all rows with column $\mathsf{a} = \mathit{true}$ |
| $\pi_{\mathsf{a,b:c,d:}v}$ | projection onto col.s a, b, d, no duplicate removal (rename c into b, new constant col. d of value $v$) |
| $\varrho_{\mathsf{a:(b,..,c)/d}}$ $\varrho_{\mathsf{a:(b,..,c)}}$ | (group rows by d,) new col. a numbers rows densely starting from 1 according to order given by b, . . , c |
| $\_ \times \_$ | Cartesian product |
| $\_ \bowtie_p \_$ | join with predicate $p$ |
| $\_ \dot{\cup} \_, \_ \setminus \_$ | disjoint union, difference |
| $\delta$ | duplicate elimination |
| $\circledcirc_{\mathsf{a:(b,..,c)}}$ | apply $\circ \in \{*, =, <, ..\}$ to b, . . , c, result in new col. a |

**Table 2: Operators of the relational algebra.**

tion type $t$ supported by the relational back-end such that the value domain of $t$ includes $v$ (or may encode $v$). A node $n$ is represented by a *surrogate value* $\gamma_n$ reflecting *node identity* and *document order*: we require $\gamma_{n_1} = \gamma_{n_2} \Leftrightarrow n_1$ `is` $n_2$ and $\gamma_{n_1} < \gamma_{n_2} \Leftrightarrow n_1$ `<<` $n_2$ for any two nodes $n_{1,2}$. A variety of such node surrogates have already been described, *e.g.*, preorder ranks [9] or ORDPATH labels [11]. In the context of this paper, we may treat atomic values and node surrogates alike (and will simply refer to *items* from now on).

| pos | item |
|---|---|
| 1 | $i_1$ |
| ⋮ | ⋮ |
| $n$ | $i_n$ |

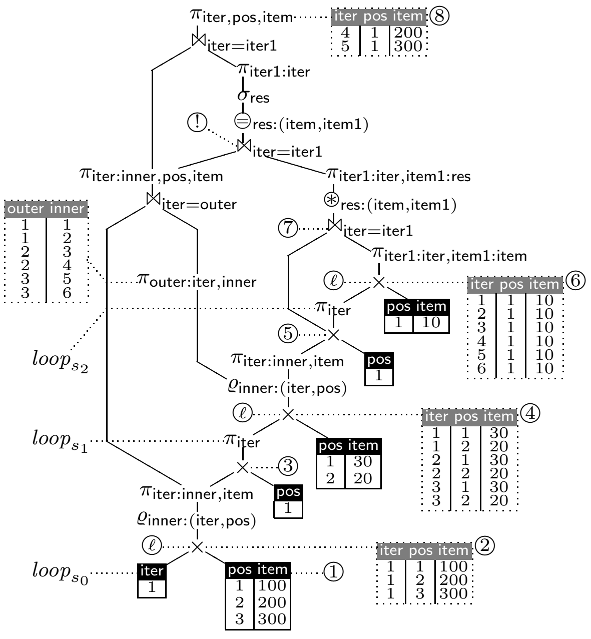
An XQuery *item sequence* (`(`$i_1$`,...,`$i_n$`)`) is encoded by the table with schema pos|item depicted here. Column pos maintains sequence order. The singleton sequence (`(`$i$`)`) and item $i$ have identical representations: a single-row table with schema pos|item; the empty sequence `()` maps into the empty table with the same schema.

## 2.1 Variables and Iteration

To render the following more vivid, let us discuss the compilation of query $Q_1$ as a running example:

$$s_0\left\{\begin{array}{l} \texttt{for \$x in }\overbrace{\texttt{(100,200,300)}}^{e_1}\texttt{ return} \\ s_1\left\{\begin{array}{l}\texttt{for \$y in }\overbrace{\texttt{(30,20)}}^{e_2}\texttt{ return}\\ s_2\{\ \texttt{if (\$x eq \$y * }\underbrace{\texttt{10}}_{e_3}\texttt{) then \$x else ()}\end{array}\right.\end{array}\right. \qquad (Q_1)$$

(the $s_i$ denote *iteration scopes* which we will introduce in a moment). $Q_1$ is ultimately compiled into the plan depicted in Figure 1. The plan carries several annotations (in ⋯⋯) of intermediate result relations which we will use to illustrate



$\pi_{\mathsf{iter,pos,item}}$
$\bowtie_{\mathsf{iter=iter1}}$
$\pi_{\mathsf{iter1:iter}}$
$\sigma_{\mathsf{res}}$
$\circleddash_{\mathsf{res:(item,item1)}}$
$\bowtie_{\mathsf{iter=iter1}}$
$\pi_{\mathsf{iter:inner,pos,item}}$
$\pi_{\mathsf{iter1:iter,item1:res}}$
$\bowtie_{\mathsf{iter=outer}}$
$\circledast_{\mathsf{res:(item,item1)}}$
$\bowtie_{\mathsf{iter=iter1}}$
$\pi_{\mathsf{iter1:iter,item1:item}}$
$\pi_{\mathsf{outer:iter,inner}}$
$\pi_{\mathsf{iter}}$
$\pi_{\mathsf{iter:inner,item}}$
$\varrho_{\mathsf{inner:(iter,pos)}}$
$loop_{s_2}$
$loop_{s_1}$
$loop_{s_0}$
outer inner
1 1
1 2
2 3
2 4
3 5
3 6
iter pos item
4 1 200
5 1 300
1 1 10
2 1 10
3 1 10
4 1 10
5 1 10
6 1 10
1 1 30
1 2 20
2 1 30
2 2 20
3 1 30
3 2 20
1 1 100
1 2 200
1 3 300
pos item
1 10
1 30
2 20
1 100
2 200
3 300
pos
1
iter
1
①
②
③
④
⑤
⑥
⑦
⑧
!
ℓ

**Figure 1: Initial relational algebra plan for XQuery query $Q_1$. Annotations in ○ explained below.**

our relational encoding of variables and iteration. A full and formal account of the compilation technique may be found in [7,8].

Once the relational representation of an item sequence is fixed, the encoding of a variable that gets bound to this sequence by an XQuery `for`-clause may be derived rather straightforwardly. In $Q_1$, variable `$x` gets bound to each of the items of sequence $e_1$ (whose encoding forms leaf ① of the plan in Figure 1) in three independent iterations. We capture these semantics via table $e_{\texttt{\$x}}$ with schema iter|pos|item as shown here: in the second iteration (iter = 2), for example, `$x` is bound to the single item 200. Note how the single-column table $loop_{s_1} = \pi_{\mathsf{iter}}(e_{\texttt{\$x}})$ can encode the fact that the outer `for`-clause results in *three* evaluations of the associated loop body.

| iter | pos | item |
|---|---|---|
| 1 | 1 | 100 |
| 2 | 1 | 200 |
| 3 | 1 | 300 |

**Table $e_{\texttt{\$x}}$.**

**Loop lifting.** The principal idea behind the compilation scheme is that *every XQuery subexpression occurs in the scope of an iteration.* The iterations in each such scope are encoded by a single-column table with schema iter. For $Q_1$, the three scopes have been marked $s_{0,1,2}$ and Figure 2 lists the corresponding *loop* tables. In case an XQuery expression $e$ occurs in iteration scope $s_i$, we compile $e$ in dependence of relation $loop_{s_i}$ to obtain its algebraic plan. We refer to this technique as *loop lifting.* In a nutshell, an item sequence (as well as a single item) is loop-lifted by *forming the Cartesian product of its relational representation with the current loop relation.* Application of this rule leads to the relational representation for sequence $e_1$ at ② in Figure 1: $e_1$ has been compiled in the outermost iteration scope $s_0$ and thus loop-lifted over $loop_{s_0} = \begin{array}{|c|}\hline \mathsf{iter}\\ \hline 1\\ \hline\end{array}$. The three places where loop-

$loop_{s_0}$

| iter |
|---|
| 1 |

$loop_{s_1}$

| iter |
|---|
| 1 |
| 2 |
| 3 |

$loop_{s_2}$

| iter |
|---|
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |

**Figure 2: Iteration scopes ($Q_1$).**

lifting occurs for the subexpressions $e_{1,2,3}$ of $Q_1$ have been marked by ⓛ in Figure 1. From the loop-lifted $e_1$ we derive the representation $e_{\texttt{\$x}}$ of `$x` at ③. Next, the subexpressions occurring in the body of the outermost `for`-clause are to be compiled in iteration scope $s_1$. The plan computes $loop_{s_1} = \pi_{\mathsf{iter}}(e_{\texttt{\$x}})$ which is then used to loop-lift the relational representation of $e_2$ to yield the intermediate result at ④: this relation encodes the item sequence `(30,20)` for each of the three iterations performed by the outer `for`-clause. The compilation scheme continues, computes the representation of `$y` at ⑤ and emits algebraic code to derive the relational encoding ⑥ of the atomic constant `10` in the innermost iteration scope $s_2$ (via loop lifting, here: $loop_{s_2} \times \begin{array}{|c|c|}\hline \mathsf{pos} & \mathsf{item} \\ \hline 1 & 10 \\ \hline \end{array}$): in each of the six iterations of the body of the inner `for`-clause, the constant assumes the value 10.

We really reap the benefits of this representation—which collects the values of a subexpression as well as the bindings of a variable *for all iterations* into single tables—when it comes to the bulk evaluation of XQuery functions and operators like, *e.g.*, `*` in $Q_1$. At ⑦, we assemble the arguments `$y` and `10` to operator `*` in each iteration via an equi-join on the iter values. The subsequent ⊛ operator then computes the overall six multiplications necessary to evaluate $Q_1$ *in bulk*. A similar remark applies to ⊜ further up in the plan and to all operator and function applications in general.

The final result of the plan is ⑧: the fourth iteration of the innermost loop body contributes the single item `200`, the fifth iteration contributes item `300` to the query result sequence `(200,300)`.[2]

In a nutshell, loop lifting is the key technique to compile explicit iteration—XQuery's `for`-clause—into efficient bulk-style applications of algebraic operators. We thus operate the relational engine in the mode it supports best.

## 3. PLAN SIZE AND DENORMALIZED ITEM SEQUENCES

It is a plus of this compilation scheme that it does not restrict the shape of the input query: expressions may nest arbitrarily given that the syntactical and typing rules of XQuery are obeyed. The emitted algebraic plans, however, may be large. Although a rather simple—and thus efficient—variant of relational algebra (Section 4) suffices to evaluate the plans, the significant number of operators may clearly have an impact on performance.

Note that the plans include several occurrences of common sub-plans such that a lot may already be gained by compiling into DAGs (instead of operator trees). The principal size problem remains, though. To exemplify, XMark query $Q8$ (involving XPath navigation, a value-based join, aggregation, and element construction) compiles into a plan DAG of 153 operators. The simple query $Q_2$

$$\texttt{for \$x in (}k\texttt{,...,2,1)} \quad \texttt{return \$x * 5} \qquad (Q_2)$$

initially yields the plan of Figure 4(a) and thus a total of 13 operators.

[2]A final simple back-mapping step (not shown in Figure 1 for space reasons, but discussed in [8]) canonicalizes the values in columns iter and pos.

**Denormalized item representation.** Loop lifting effectively collects all values an expression $e$ assumes during the evaluation of enclosing `for`-clauses into a single table. Should $e$ be (a) *constant* or (b) *invariable* with respect to its enclosing iterations, loop lifting consequently leads to a *fully denormalized representation* for $e$ and thus to—at least potentially—significant data redundancy: $e$'s representation will be identical for all iterations (*i.e.*, groups of identical iter values).

| iter | pos | item |
|---|---|---|
| 1 | 1 | 5 |
| ⋮ | ⋮ | ⋮ |
| $k$ | 1 | 5 |

(a)

| iter | pos | item |
|---|---|---|
| 1 | 1 | $i_1$ |
| ⋮ | ⋮ | ⋮ |
| 1 | $n$ | $i_n$ |
| ⋮ | ⋮ | ⋮ |
| $k$ | 1 | $i_1$ |
| ⋮ | ⋮ | ⋮ |
| $k$ | $n$ | $i_n$ |

(b)

**Figure 3: Denormalized item sequences.**

Note the loop-lifted representation of the atomic item `5` in $Q_2$ shown in Figure 3(a). Similarly, inside a loop of $k$ overall iterations, the constant or invariable item sequence $(i_1, \ldots, i_n)$ has the encoding depicted in Figure 3(b). Depending on the values of $k$ and $n$, this redundancy may, again, severely impact plan performance.

XQuery's orthogonality clearly is one of its virtues. We thus designed the compiler to be completely *compositional*—which is crucial and typical for a compiler of a functional-style language: a subexpression is compiled independent of its embedding expression. Plan size and data redundancy seem to be prices we pay for this orthogonal treatment of, *e.g.*, items and item sequences.

## 4. PLAN ANALYSIS AND SIMPLIFICATION

In the sequel, we argue that the just mentioned issues need not be showstoppers for loop-lifting-based XQuery compilation. The concluding Section 6 briefly reviews performance results which clearly indicate the same.

Despite their significant size, the generated plans come with an effective optimization hook: the operators are particularly simple, restricted variants of the classical relational algebra operators, *e.g.*,

- all $\bowtie$-operators are equi-joins,
- the projection operator $\pi$ does not remove duplicates,
- all invocations of $\dot\cup$ receive disjoint argument relations.

These restrictions render the operators efficiently implementable inside a relational engine and—more importantly in this context—enable rather straightforward plan analysis to infer a variety of *properties of intermediate plan results*. We then use the properties to spot those places in the plans which are subject to simplification or even pruning. As we will see, the simplifications can effectively cut down plan size and help to remove inefficiencies due to denormalization through loop-lifting.

Because the relational plans have been generated by an XQuery compiler (as opposed to an SQL compiler) the simplification process has a somewhat different "taste" than regular algebraic optimization.

### 4.1 Plan Properties and Inference

Table 3 lists the properties we will use during plan analysis and simplification. Properties *icols* and *ocols* (*input/output columns*) are associated with operators, the remaining properties relate to intermediate plan results, *i.e.*, relations.

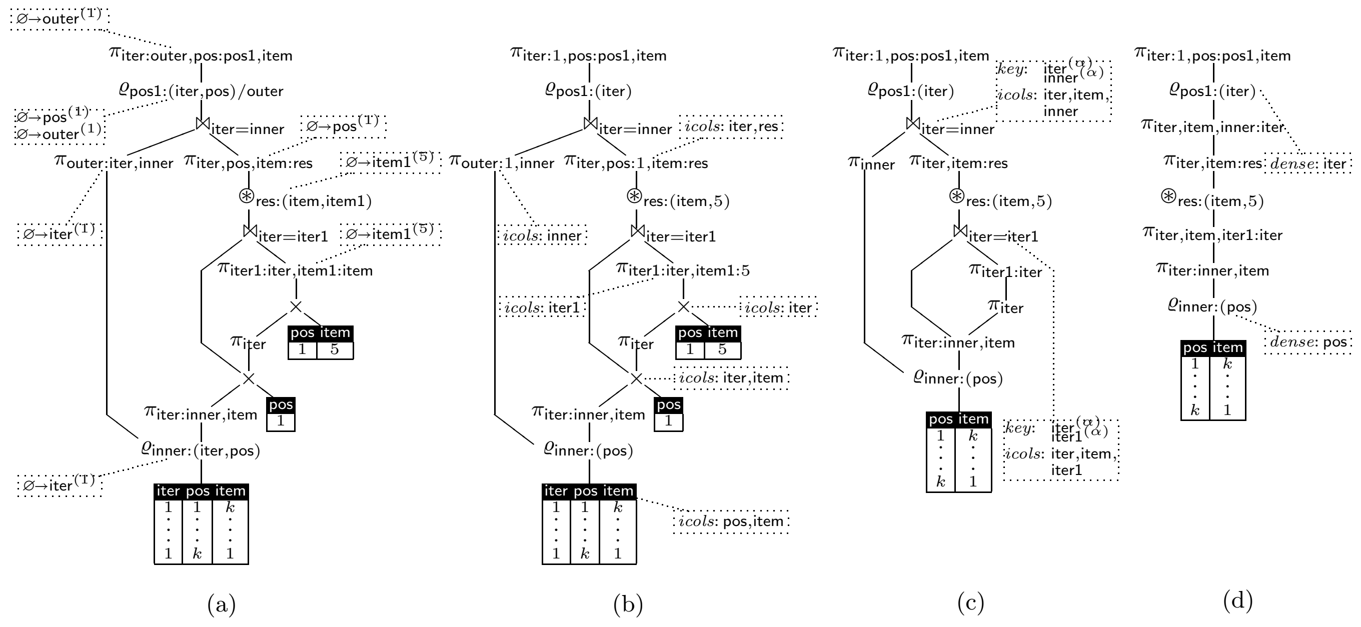


**Figure 4: Plan simplification based on inferred properties of intermediate results and operators (query $Q_2$).**

| Property | Description |
|---|---|
| $card_0$, $card_1$ | relation has cardinality 0 (or 1) |
| $key$: $\mathsf{a}^{(\alpha)}$ | column $\mathsf{a}$ is key with value domain $\alpha$ |
| $dense$: $\mathsf{a}$ | col. $\mathsf{a}$ is numbered densely starting from 1 |
| $\varnothing \rightarrow \mathsf{a}^{(v)}$ | $\mathsf{a}$ is constant col. of value $v$ |
| $\varnothing \twoheadrightarrow \mathsf{a}_1^{(\alpha_1)}, .., \mathsf{a}_n^{(\alpha_n)}$ | remaining col.s are independent of $\mathsf{a}_1, .., \mathsf{a}_n$ |
| $icols$: $\mathsf{a}_1, .., \mathsf{a}_n$ | col.s $\mathsf{a}_1, .., \mathsf{a}_n$ are required to evaluate this operator (or an ancestor operator) |
| $ocols$: $\mathsf{a}_1^{(\alpha_1)}, .., \mathsf{a}_n^{(\alpha_n)}$ | col.s $\mathsf{a}_1, .., \mathsf{a}_n$ appear in operator output |

**Table 3: Properties of relations (and operators).**

The two former properties are primarily used to detect that operators consume or produce obsolete result columns—a situation that is commonly created by preceding simplification steps. An operator $\omega$ inherits property *icols* from its parents in the plan DAG: if column $\mathsf{a}$ is required to evaluate a parent and $\omega$ does not produce output column $\mathsf{a}$ itself, then $\omega.icols$ will contain column $\mathsf{a}$ (and $\omega$ thus requires this column to be contained in one of its arguments). If, for example, $\omega = \pi_{\mathsf{a:b,c}}$ or $\omega = \circledast_{\mathsf{a:(b,c)}}$, $\omega.icols$ would contain columns $\mathsf{b}, \mathsf{c}$ instead (but not column $\mathsf{a}$).

All other properties may be synthesized during a single bottom-up walk of the DAG. An excerpt of the property inference rules can be found in Figure 5. Property $e.card_0$ ($e.card_1$) holds if sub-plan $e$ yields a relation of exactly zero (one) tuple(s). From this we may infer further interesting relation characteristics, *e.g.*, the preservation of keys (see below). The presence of such a key column $\mathsf{a}$ is recorded in property $key$: $\mathsf{a}^{(\alpha)}$. A superscript $(\alpha)$ denotes a *domain identifier*: $\alpha$ represents the active domain of its column, *i.e.*, all values actually contained in column $\mathsf{a}$. We obviously cannot maintain the value domain itself due to its potential size, but two columns $\mathsf{a}^{(\alpha)}, \mathsf{b}^{(\alpha)}$ with identical inferred domain identifiers are guaranteed to share one active domain. A device similar to domain identifiers has also been used in [10]. In a relation $e$ with property $e.dense$: $\mathsf{a}$, the active domain of column $\mathsf{a}$ is $1, 2, \ldots, n$ where $n$ is the cardinality of $e$.

The two remaining properties are probably the most important ones in our context. If $e.\varnothing \rightarrow \mathsf{a}^{(v)}$, then all tuples in $e$ contain value $v$ in column $\mathsf{a}$. We denote this by a degenerate *functional dependency* (fd) with an empty left-hand side [1,5,10] for uniformity reasons which will become clear in a moment. The degenerate *multivalued dependency* (mvd) $e.\varnothing \twoheadrightarrow \mathsf{a}_1^{(\alpha_1)}, .., \mathsf{a}_n^{(\alpha_n)}$ indicates that the column group $\mathsf{a}_1, .., \mathsf{a}_n$ is independent of all remaining columns of $e$. For the relation in Figure 3(a), for example, we have $\varnothing \rightarrow \mathsf{item}^{(5)}$. For the relation in Figure 3(b), we have $\varnothing \twoheadrightarrow \mathsf{iter}^{(\alpha)}$ where $\alpha$ represents the domain $1, \ldots, k$.

In Figure 5, Rule (5) uses inferred functional dependencies for constant folding with respect to the binary operator $\circ$, while Rule (1) introduces such fds. Result column $\mathsf{a}$ will be constant in both cases. Rule (2) propagates key column $\mathsf{b}$ through a selection but introduces a fresh domain identifier $\beta$ for $\mathsf{b}$: the selection potentially restricts active domain $\alpha$; we know, however, that $\beta \subseteq \alpha$. This domain inclusion relationship is recorded for later perusal. Finally, an mvd is inferred by Rule (9). The Cartesian product guarantees the column values in $e_1$ and $e_2$ to be independent or orthogonal [5]. (Obviously, a symmetric inference rule in which $e_{1,2}$ swap roles is also correct.) Note that such mvds are preserved under certain preconditions (Rules (10)–(12)).

## 4.2 Peep-Hole Plan Simplification

The substantial size of the initially emitted plans make a classical approach to query rewriting a rather hopeless case: rewrite rules would need to be either (a) very specific, and thus carry huge DAG contexts, or (b) very generic which makes their applicability hard to detect. Instead, we use a collection of *peep-hole*-style equivalences (Figure 6) whose applicability may be decided by looking at the inferred properties of a *single* plan DAG node. Specific necessary properties thus appear in the premises of the algebraic equivalence rules.

$$\frac{}{(\sigma_{\mathsf{a}}(e)).\varnothing\to\mathsf{a}^{(\mathit{true})}}\;(1)\qquad \frac{e.\mathit{key}\colon \mathsf{b}^{(\alpha)}}{(\sigma_{\mathsf{a}}(e)).\mathit{key}\colon \mathsf{b}^{(\beta)}\wedge\beta\subseteq\alpha}\;(2)\qquad \frac{}{\begin{array}{l}(\varrho_{\mathsf{a:(b,..,c)}}(e)).\mathit{dense}\colon \mathsf{a}\\ \wedge\,(\varrho_{\mathsf{a:(b,..,c)}}(e)).\mathit{key}\colon \mathsf{a}^{(\alpha)}\end{array}}\;(3)\qquad \frac{e.\varnothing\to\mathsf{d}^{(\alpha)}}{\begin{array}{l}(\varrho_{\mathsf{a:(b,..,c)/d}}(e)).\mathit{dense}\colon \mathsf{a}\\ \wedge\,(\varrho_{\mathsf{a:(b,..,c)/d}}(e)).\mathit{key}\colon \mathsf{a}^{(\beta)}\end{array}}\;(4)$$

$$\frac{e.\varnothing\to\mathsf{b}^{(v_1)}\quad e.\varnothing\to\mathsf{c}^{(v_2)}}{(\circledcirc_{\mathsf{a:(b,c)}}(e)).\varnothing\to\mathsf{a}^{(v_1\circ v_2)}}\;(5)\qquad \frac{e_1.\varnothing\to\mathsf{a}^{(v)}\quad e_2.\varnothing\to\mathsf{a}^{(v)}}{(e_1\,\dot\cup\,e_2).\varnothing\to\mathsf{a}^{(v)}}\;(6)\qquad \frac{e_1.\mathit{card}_1\quad e_2.\mathit{key}\colon \mathsf{a}^{(\alpha)}}{(e_1\times e_2).\mathit{key}\colon \mathsf{a}^{(\alpha)}}\;(7)\qquad \frac{e_1.\mathit{key}\colon \mathsf{a}^{(\alpha)}\quad e_2.\mathit{key}\colon \mathsf{b}^{(\beta)}\quad \beta\subseteq\alpha}{(e_1\Join_{\mathsf{a=b}}e_2).\mathit{key}\colon \mathsf{a}^{(\beta)},\mathsf{b}^{(\beta)}}\;(8)$$

$$\frac{e_2.\mathit{ocols}\colon \mathsf{a}_1^{(\alpha_1)},..,\mathsf{a}_n^{(\alpha_n)}}{(e_1\times e_2).\varnothing\twoheadrightarrow\mathsf{a}_1^{(\alpha_1)},..,\mathsf{a}_n^{(\alpha_n)}}\;(9)\qquad \frac{e_1.\varnothing\twoheadrightarrow\mathsf{a}^{(\alpha)}\quad e_2.\varnothing\twoheadrightarrow\mathsf{a}^{(\alpha)}}{(e_1\,\dot\cup\,e_2).\varnothing\twoheadrightarrow\mathsf{a}^{(\alpha)}}\;(10)\qquad \frac{e.\varnothing\twoheadrightarrow\mathsf{a}_1^{(\alpha_1)},..,\mathsf{a}_n^{(\alpha_n)}}{(\pi_{\mathsf{a}_1,..,\mathsf{a}_n,\mathsf{b},..,\mathsf{c}}(e)).\varnothing\twoheadrightarrow\mathsf{a}_1^{(\alpha_1)},..,\mathsf{a}_n^{(\alpha_n)}}\;(11)\qquad \frac{e.\varnothing\twoheadrightarrow\mathsf{d}^{(\alpha)}}{(\varrho_{\mathsf{a:(b,..,c)/d}}(e)).\varnothing\twoheadrightarrow\mathsf{d}^{(\alpha)}}\;(12)$$

**Figure 5: Property inference (excerpt of rules). Domain identifiers $\alpha,\beta$ introduced in a rule's consequence denote fresh (previously unused) domain identifiers.**

$$\frac{e.\varnothing\to\mathsf{b}^{(v)}}{\pi_{\mathsf{a},..,\mathsf{b},..,\mathsf{c}}(e)\equiv\pi_{\mathsf{a},..,\mathsf{b}:v,..,\mathsf{c}}(e)}\;(a)\qquad \frac{e.\varnothing\to\mathsf{b}^{(v)}}{\circledcirc_{\mathsf{a:(b,c)}}(e)\equiv\circledcirc_{\mathsf{a}:(v,\mathsf{c})}(e)}\;(b)\qquad \frac{e.\varnothing\to\mathsf{d}^{(v)}}{\begin{array}{l}\varrho_{\mathsf{a:(b,..,c)/d}}(e)\equiv\varrho_{\mathsf{a:(b,..,c)}}(e)\\ \wedge\,\varrho_{\mathsf{a:(b,..,d,..,c)/e}}(e)\equiv\varrho_{\mathsf{a:(b,..,c)/e}}(e)\end{array}}\;(c)$$

$$\frac{e.\mathit{key}\colon \mathsf{d}^{(\alpha)}}{\varrho_{\mathsf{a:(b,..,d,..,c)/e}}(e)\equiv\varrho_{\mathsf{a:(b,..,d)/e}}(e)}\;(d)\qquad \frac{e.\mathit{dense}\colon \mathsf{b}\quad (\varrho_{\mathsf{a:(b,..,c)}}(e)).\mathit{icols}\colon \mathsf{d}_1,..,\mathsf{d}_n}{\varrho_{\mathsf{a:(b,..,c)}}(e)\equiv\pi_{\mathsf{a:b},\mathsf{d}_1,..,\mathsf{d}_n}(e)}\;(e)\qquad \frac{(e_1\times e_2).\mathit{icols}\subseteq e_2.\mathit{ocols}\quad e_1.\mathit{card}_1}{e_1\times e_2\equiv e_2}\;(f)$$

$$\frac{e_1.\mathit{key}\colon \mathsf{a}^{(\alpha)}\quad e_2.\mathit{key}\colon \mathsf{b}^{(\beta)}\quad \beta\subseteq\alpha\quad (e_1\Join_{\mathsf{a=b}}e_2).\mathit{icols}\colon \mathsf{d}_1,..,\mathsf{d}_n\subseteq e_2.\mathit{ocols}}{e_1\Join_{\mathsf{a=b}}e_2\equiv\pi_{\mathsf{a:b},\mathsf{d}_1,..,\mathsf{d}_n}(e_2)}\;(g)\qquad \frac{e_1.\mathit{key}\colon \mathsf{a}^{(\alpha)}\quad e_2.\varnothing\twoheadrightarrow\mathsf{b}^{(\alpha)}}{e_1\Join_{\mathsf{a=b}}e_2\equiv e_1\times e_2}\;(h)$$

**Figure 6: Algebraic equivalences (excerpt) based on inferred properties.**

Some simplifications strive to minimize the dependence of an operator on specific input columns (in Rules (a) and (b), a reference to constant column b is traded for the constant value $v$). This removes column b from the *icols* property which, in turn, may render a Cartesian product (Rule (f)) or a join (Rule (g)) further down in the DAG obsolete. Another group of equivalences tries to simplify or prune occurrences of $\varrho$: the semantics of $\varrho$ are simple but most conceivable implementations require its input to be sorted. Rules (c) and (d) describe how a constant or key column d may be used to shorten the list of sort criteria. In Rule (d), column d is a key for $e$ and thus a decisive order criterion: further columns need not be inspected to decide the order of two rows. If a dense column b is the major sort criterion, we might as well derive the row numbering from b itself; $\varrho$ is pruned (Rule (e)). Rule (g) finally removes key joins in case of a key domain inclusion: every row in $e_2$ is guaranteed to find exactly one join partner in $e_1$. Note that we do not carry out the actual domain inclusion test here but rather check domain identifiers against the known inclusion relationships recorded by inference Rule (2).

Figure 4 exemplifies how these rules interact to simplify the initial plan for query $Q_2$ in several steps. Those nodes affected by the following simplification step are annotated with the properties which justify the simplification (remember that we are using peep-hole simplification: changes to the DAG are always in the direct vicinity of the affected nodes).

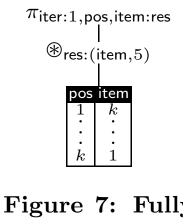

$\pi_{\mathsf{iter:1,pos,item:res}}$
$\circledast_{\mathsf{res:(item,5)}}$
pos
item
1
$k$
$k$
1

**Figure 7: Fully simplified plan for query $Q_2$.**

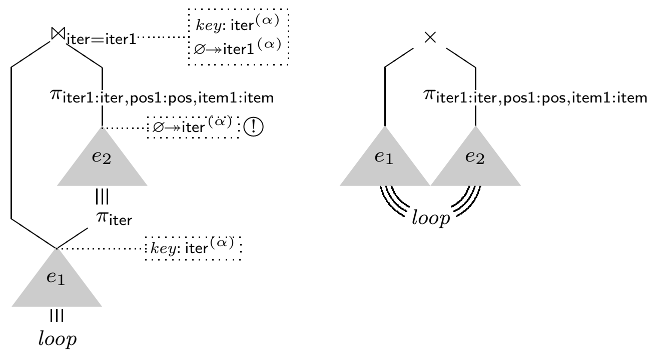
The bottom $\varrho$ node in Figure 4(a), for example, is simplified according to Rule (c): we have inferred that all rows carry the atomic value 1 in column item ($\varnothing\to\mathsf{item}^{(1)}$). Such a column does not make for a useful order criterion and is removed from the list of order criteria. The resulting, simpler row number operator $\varrho_{\mathsf{inner:(pos)}}$ numbers the incoming rows solely based on the dense column pos (Figure 4(d)). In such a situation, Rule (e) indicates that the dense column already provides the requested row numbering *for free*. This instance of $\varrho$ is thus replaced by a projection $\pi_{\mathsf{inner:pos}}$ which introduces column inner as a mere alias for column pos. In Figure 7, which depicts the fully simplified plan for query $Q_2$, even this projection has been pruned away.

Similarly, note how the loop lifting for the atomic item 5 is detected (and then rendered obsolete) with the help of property $\varnothing\to\mathsf{item}^{(5)}$ in the first two simplification steps in Figure 4.

## 5. MVDs FOR XQUERY JOIN DETECTION

An fd $\varnothing\to\mathsf{a}^{(v)}$ can detect the presence of a loop-lifted constant item in the plan DAGs. Likewise—and we feel that this is a quite elegant outcome of this research—we may use the more general mvd concept to detect the more general case of a loop-lifted invariable *item sequence*. To see this, note that the representation of the loop-lifted item sequence $(i_1,\ldots,i_n)$ (Figure 3(b)) is derived from the relational sequence encoding (see Section 2) by forming the Cartesian product with the current *loop* relation, *i.e.*, with a single-column relation with schema iter. But this *precisely* characterizes [1, 5] the presence of the degenerate mvd $\varnothing\twoheadrightarrow\mathsf{iter}^{(\alpha)}$ (where $\alpha$ is determined by the number of iterations to be performed) in the encoding of Figure 3(b). The property inference Rule (9) in Figure 5 makes this correspondence between loop-lifting and the presence of mvds available to the simplifier.

This observation paves the way for a form of XQuery join detection that is completely unaffected by XQuery's syntactic diversity. Figure 8(a) shows a prototypical plan in which the (arbitrary) subexpression $e_2$ has been lifted over the iteration scope opened by subexpression $e_1$ (which itself is to

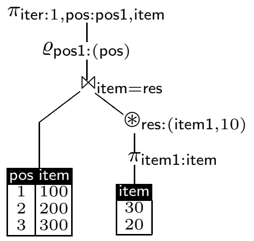


$\bowtie_{\text{iter=iter1}}$
*key*: $\text{iter}^{(\alpha)}$
$\varnothing \twoheadrightarrow \text{iter1}^{(\alpha)}$
$\pi_{\text{iter1:iter,pos1:pos,item1:item}}$
$\varnothing \twoheadrightarrow \text{iter}^{(\alpha)}$ !
$e_2$
$\pi_{\text{iter}}$
*key*: $\text{iter}^{(\alpha)}$
$e_1$
*loop*
$\times$
$\pi_{\text{iter1:iter,pos1:pos,item1:item}}$
$e_1$
$e_2$
*loop*

(a) Initial plan: sub-plan $e_2$ loop-lifted over $e_1$.

(b) Equivalent plan: sub-plan $e_2$ *not* loop-lifted over $e_1$.

**Figure 8: An inferred mvd (at !) helps to recognize an XQuery join scenario.**



$\pi_{\text{iter:1,pos:pos1,item}}$
$\varrho_{\text{pos1:(pos)}}$
$\bowtie_{\text{item=res}}$
$\circledast_{\text{res:(item1,10)}}$
$\pi_{\text{item1:item}}$
pos item
1 100
2 200
3 300
item
30
20

**Figure 9: Fully simplified plan for query $Q_1$.**

be evaluated inside an iteration encoded by relation *loop*).

Exactly if the source XQuery query describes a join between the two subexpressions $e_{1,2}$—*i.e.*, $e_2$ may be evaluated independently of $e_1$ before the two resulting item sequences are joined in some way—then we will be able to infer the mvd $\varnothing \twoheadrightarrow \text{iter}^{(\alpha)}$ (marked ! in Figure 8(a)) at the root of the sub-plan for $e_2$. We then simplify with the help of Rule (h) in Figure 6 and emit the plan of Figure 8(b) instead. In the simpler plan, note that $e_2$ is *not* loop-lifted over $e_1$ but over *loop*. We may now expect (the larger the intermediate result of $e_1$ the more significant) less data redundancy in $e_2$. More importantly, however, the new plan is nicely prepared to merge its root Cartesian product with a $\sigma$-operator to form a join. This algebraic join then implements the join that was present in the original XQuery query in some syntactic form.

The initial plan (Figure 1) for query $Q_1$ is an instance of the prototypical plan of Figure 8(a). Note that the compiler can detect the presence of the mvd $\varnothing \twoheadrightarrow \text{iter}^{(\alpha)}$ at node ! in Figure 1 *while* it emits the code. At this point in time, the compiler holds references to the sub-plans for $e_{1,2}$ and can rather easily initiate the rewriting into the new DAG shape of Figure 8(b). The newly introduced Cartesian product is then merged with the two ancestor operators $\sigma$–$\circleddash$ to make the implicit value-based join expressed by XQuery query $Q_1$ explicit in the algebraic plan. Note that, depending on the XQuery comparison operator used in the source query, this join operator will be a $\theta$-join (not an equi-join). In [3], we discuss how the existential semantics of XQuery's general comparison operators (=, <, ...) may be compiled into efficient algebraic code. Figure 9 shows the fully simplified plan for query $Q_1$.

This variant of XQuery join detection is not affected by the shape of the source query. We will, in particular, infer the mvd $\varnothing \twoheadrightarrow \text{iter}^{(\alpha)}$ at ! in Figure 8(a) independent of how many iteration scopes, *i.e.*, nested `for`-clauses, separate $e_1$ and $e_2$.

## 6. PURELY RELATIONAL XQUERY

The XQuery compiler studied in this paper, dubbed *Pathfinder*, is part of the purely relational XQuery processor *MonetDB/XQuery* [3]. The back-end of this processor is *MonetDB* [4], an extensible open-source relational database kernel developed at CWI, Amsterdam. MonetDB's internals have been optimized for in-memory operation but the system is engineered to exploit the memory and file management services of the operating system to operate with data volumes that exceed main memory capacity. In particular, MonetDB efficiently supports *narrow tables* (three columns or less) which are pervasive in the plans emitted by our XQuery compiler.

To give an impression of where research into relational XQuery processors may lead, we reproduce a fragment of the XMark experiments reported in [3] in Table 4 (conducted on a 64-bit system with an 1.6 GHz AMD processor and 8 GB RAM). Pathfinder, faithfully operating MonetDB in the strict sequential access mode it was engineered for, indeed inherits the scalability of its back-end: XMark document instances beyond 1 GB serialized size are processed in interactive time, including the XQuery join queries $Q8$–$Q10$ (note that queries $Q11$, $Q12$ produce intermediate results quadratic in the size of the input document—any XQuery processor will thus struggle at this point).

| $Q$ | 110 MB | 1.1 GB |
|---|---|---|
| 1 | 0.41 | 1.2 |
| 2 | 0.30 | 2.4 |
| 3 | 1.51 | 12.5 |
| 4 | 0.45 | 3.8 |
| 5 | 0.16 | 1.2 |
| 6 | 0.05 | 0.3 |
| 7 | 0.07 | 0.4 |
| 8 | 0.75 | 10.4 |
| 9 | 0.87 | 12.9 |
| 10 | 5.31 | 55.0 |
| 11 | 3.48 | 960.9 |
| 12 | 1.66 | 431.3 |
| 13 | 0.22 | 1.3 |
| 14 | 2.20 | 21.3 |
| 15 | 0.28 | 1.7 |
| 16 | 0.26 | 1.8 |
| 17 | 0.30 | 2.8 |
| 18 | 0.13 | 0.9 |
| 19 | 0.55 | 5.3 |
| 20 | 0.62 | 4.9 |

**Table 4: XMark query evaluation time (seconds).**

Pathfinder and the MonetDB/XQuery system will be released into the open-source well before XIME-P 2005 (`www.pathfinder-xquery.org`).

## 7. REFERENCES

[1] C. Beeri, R. Fagin, and J.H. Howard. A Complete Axiomatization for Functional and Multivalued Dependencies in Database Relations. In *Proc. of the Int'l ACM SIGMOD Conference on Management of Data*, Toronto, Canada, August 1977.

[2] 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, February 2005. W3C Working Draft `http://www.w3.org/TR/xquery/`.

[3] P. Boncz, T. Grust, S. Manegold, J. Rittinger, and J. Teubner. Pathfinder: Relational XQuery Over Multi-Gigabyte Inputs in Interactive Time. Technical Report INS-E0503, CWI, Amsterdam, March 2005.

[4] P. Boncz and M.L. Kersten. MIL Primitives for Querying a Fragmented World. *VLDB Journal*, 8(2), March 1999.

[5] R. Fagin. Multivalued Dependencies and a New Normal Form for Relational Databases. *ACM Transactions on Database Systems (TODS)*, 2(3), September 1977.

[6] T. Grust and S. Klinger. Validation and Type Annotation for Encoded Trees. In *Proc. of the 1st Int'l Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, Paris, France, June 2004.

[7] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*, Toronto, Canada, September 2004.

[8] T. Grust and J. Teubner. Relational Algebra: Mother Tongue—XQuery: Fluent. In *Proc. of the Twente Data Management Workshop on XML Databases and Information Retrieval (TDM)*, The Netherlands, June 2004.

[9] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach a Relational DBMS to Watch Its Axis Steps. In *Proc. of the 29th Int'l Conference on Very Large Databases (VLDB)*, Berlin, Germany, September 2003.

[10] A. Klug. Calculating Constraints on Relational Expressions. *ACM Transactions on Database Systems (TODS)*, 5(3), September 1980.

[11] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. of the Int'l ACM SIGMOD Conference on Management of Data*, Paris, France, June 2004.

[12] A. Schmidt, F. Waas, M.L. Kersten, M.J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. of the 28th Int'l Conference on Very Large Databases (VLDB)*, Hong Kong, China, August 2002.