Reihe Informatik
TR-2007-007

University of Mannheim
Database Research Group
B6, 29
68131 Mannheim, Germany
brantner|kanne|moerkotte@informatik.uni-mannheim.de

Let a Single FLWOR Bloom

Matthias Brantner [*] Carl-Christian Kanne Guido Moerkotte

July 9, 2007

Abstract

To globally optimize execution plans for XQuery expressions, a plan generator must generate and compare plan alternatives. In proven compiler architectures, the unit of plan generation is the query block. Fewer query blocks mean a larger search space for the plan generator and lead to a generally higher quality of the execution plans. The goal of this paper is to provide a toolkit for developers of XQuery evaluators to transform XQuery expressions into expressions with as few query blocks as possible.

Our toolkit takes the form of rewrite rules merging the inner and outer FLWOR expressions into single FLWORs. We focus on previously unpublished rewrite rules and on inner FLWORs occurring in the `for`, `let`, and `return` clauses in the outer FLWOR.

1 Introduction

XQuery evaluators become more and more mature in terms of features and performance, and XQuery is being integrated into mainstream DBMS products as a native language [6, 10, 17]. However, XQuery processing research is still missing some fundamental tools to facilitate the development of industrial-strength XQuery optimizers. The goal of this paper is to fill one of these gaps: We provide a rewrite toolkit that allows to reduce the number of query blocks in a query expression. This widens the search space for plan generators by making more information visible to a single run of the plan generation algorithm. Let us begin by stressing the importance of our goal:

Industrial-strength query optimizers proceed in a two-phase manner. In a first phase, the query is translated into an internal representation, and heuristical rewrite rules are applied to simplify and normalize the query. In a second phase, a plan generator enumerates alternative execution plans, determines their costs, and chooses the optimal plan. Alternative plans can differ in the access paths used for the basic input sets (e.g. whether to use an index or not), in the order in which the basic input sets are joined, and in the position of other operators, such as grouping or sorting.

However, efficient plan generation algorithms cannot take arbitrary query structures as input. Instead, the unit of plan generation is the *query block*. Depending on the design of the query compiler, a query block can be represented in a variety of ways, for example as a source language construct (SELECT FROM WHERE in SQL, or FLWOR in XQuery), as a node in an internal graph representation (such as the Query Graph Model QGM [20]), or as an algebraic expression. Some queries exhibit a nested structure, where a query block references subquery blocks. In such cases, the plan generator is called in a bottom-up fashion, generating plans for all subquery blocks before the surrounding query block is processed. It is easy to see that in such cases, the search space examined by the plan generator is limited, because only locally good solutions are computed. For globally optimal plans, it is desirable to reduce the number of query blocks to have more information available in a single run of the plan generator, creating a larger search space of alternative plans. For this reason, in the first phase of optimization, queries are rewritten by merging as many query blocks as possible. This is state-of-the-art for SQL query processing (e.g. [4, 12, 21]), but not highly developed for XQuery.

For an industrial-strength approach to XQuery optimization, such a rewriting step to merge query blocks is particularly necessary:

[*]This work was supported by the Deutsche Forschungsgemeinschaft under grant MO 507/10-1.

- In XQuery expressions in real applications, a nested query structure is the norm rather than an exception. This is due to a number of reasons, including the construction of hierarchical XML results, the absence of a grouping construct, the generation of queries using visual editors, and, last but not least, the inlining of (non-recursive) XQuery functions that contain FLWOR expressions.
- XML query processing can benefit from holistic n-way joins [3] which perform single-pass tree-pattern matching instead of constructing results just using binary joins. The detection of tree patterns and the decision when to use regular joins and when to use pattern matching is a global decision during plan generation that requires access to as much of the query as possible.

An example for a highly nested query (inspired by XMark Query 3) is shown here:

```
let $auction := doc("auction.xml") return
  let $euro:=for $o in $auction/site/open_auctions/open_auction
             for $i in $auction/site/regions/europe/item/@id
             where $o/itemref/@item eq $i
             return $o
  for $a in $euro
  where zero-or-one($a/bidder[1]/increase/text()) * 2
        <= $a/bidder[last()]/increase/text()
  return
    for $p in $auction/site/people/person[profile/@income > 5000]
    for $w in $p/watches/watch
    where $a/@id = $w/@open_auction
    return <auction id="{$a/@id}">
            <increase first="{$a/bidder[1]/increase/text()}"
                      last="{$a/bidder[last()]/increase/text()}"/>
              <watched_by id="{$p/@id}"/>
           </auction>
```

The query body is constructed of four FLWOR expressions, three of which are nested inside other FLWORs. However, these are only the explicit FLWOR blocks. Depending on the compiler design, the number of nested query blocks may be even deeper. For example, with a plan generator that focuses on purely structural tree-pattern matching, nested value-based predicates such as `profile/@income > 5000` may be separate query blocks.

Without further processing, such a query is optimized using several runs of the plan generation algorithm, where each plan for a FLWOR expression is used in the plan for the surrounding FLWOR. This separate optimization of subqueries impedes the discovery of good overall execution plans. This is demonstrated by our example, in which there are two value-based joins, one joining the Open Auctions to the European Items, and one joining the Open Auctions to the Persons with an income higher than 5000. However, the join conditions in the `where` clauses are in different FLWORs, prohibiting the plan generator to see both of the joins and optimize their order. Join order optimization is a cornerstone of efficient relational query processing and just as important in XQuery processing [7].

As in many other cases, the nested structure of the query is not required to obtain the query result, but is used because this way, the query is simpler to write. In fact, the whole query above can be formulated using a single FLWOR block. One alternative to do so is shown below, with the results of each processing step bound to a separate variable:

```
let $auction := doc("auction.xml"), $x32 := $auction/site
for $o in $x32, $x13 in $o/open_auctions, $a in $x13/open_auction
for $i in $x32, $x15 in $i/regions, $x16 in $x15/europe
for $x17 in $x16/item, $x18 in $x17/@id
let $x4 := $a/itemref, $x19 := $x4/@item
let $x33 := $a/bidder[1], $x34 := $x33/increase, $x35 := $x34/text()
let $x36 := $a/bidder[last()], $x37 := $x36/increase, $x38 := $x37/text()
let $x39 := $a/@id
for $p in $x32, $x20 in $p/people, $x21 in $x20/person
for $w in $x21/watches, $x22 in $w/watch
let $x8 := $x21/@id, $x10 := $x22/@open_auction
let $x13 := $x21/profile, $x27 := $x13/@income
for $x1 in <auction id="{$x39}">
              <increase first="{$x35}" last="{$x38}"/>
              <watched_by id="{$x8}"/>
            </auction>
where zero-or-one($x35) * 2 <= $x38 and $x39 = $x10
      and $x27 > 5000 and $x19 eq $x18
return $x1
```

While this form of the query is less readable and more difficult to write, it is easier to optimize because all the basic operations, intermediate results, input sets, and their dependencies are uniformly represented in a single, top-level FLWOR construct.

The goal of this paper is to provide a toolkit for developers of XQuery evaluators to transform XQuery expressions into expressions with as few query blocks as possible. This toolkit takes the form of rewrite rules merging the inner and outer FLWOR expressions into single FLWORs. These unnesting rules are supplemented by some helpful normalization rewrites. We have chosen to present our rules using regular XQuery syntax because other representations (such as QGM or algebraic expressions) are less universal and would be more difficult to adapt to different evaluators. We do not use the XQuery Core sublanguage because it does not have a query block construct suitable for plan generation. It is, instead, inherently nested, even for quite simple XQuery expressions. Due to space constraints, we limit our presentation to previously unpublished rewrite rules and to inner FLWORs occurring in the `for`, `let`, and `return` clauses in the outer FLWOR.

2 Related Work

Michiels et al. [19] discuss rewrite rules on two levels. Starting from expressions in XQuery Core, they propose to first rewrite them into normal forms (still in XQuery Core) that make the subsequent stages robust against different syntactic formulations of the same query, and to support tree-pattern detection. They also simplify the query by removing unnecessary constructs introduced by Core normalization. Some of these simplification rewrites could be incorporated into our toolkit. The rewritten query is then translated into an algebra that includes a tree-pattern matching operator. These algebraic expressions are then rewritten using algebraic equivalences in order to merge simple path-navigation operators into holistic tree-pattern matching operators. The rewrite rules on the algebraic level are orthogonal to the ones presented in our toolkit and can be used by a plan generator to create execution plans based on tree-pattern matching.

The very thorough paper by Hidders et al. [5] has a similar aim, but directly translates a fragment of XQuery into tree patterns without an intermediate algebraic phase. In a first phase, the queries are annotated with properties such as result cardinality, ordering, and occurrence of duplicates. These properties are then used to control a rewriting of the query into the Tree Pattern Normal Form (TPNF), which is always possible for the language fragment under consideration. For TPNF, a direct mapping onto tree patterns is then described. Unfortunately, the language fragment does not cover important XQuery constructs, such as value-based predicates. Another problem is that the rewrite rules are based on XQuery Core, which is unsuitable as a plan generator input, for example because the absence of a `where` clause makes it difficult to identify applicable join conditions. However, the property annotations are not only useful for TPNF rewriting and can be used when implementing our rewrite toolkit. Further, the TPNF technique may be used by plan generators to identify parts of the query that can be evaluated using pattern matching.

May et al. [18] have presented unnesting strategies for XQuery. Their approach is based on algebraic equivalences to be applied after translation of XQuery into the NAL algebra of the Natix system. The main focus of that work is unnesting of selection predicates which correspond to `where` clauses on the source level. The paper also discusses unnesting the subscripts of map operators, which on source level corresponds to `let` clauses. However, the rules are exclusively for the conversion of implicit grouping into explicit grouping operators, and not for the general unnesting of `let`. Translated into the source form, the presented rewrite rules are complementary to the rules discussed in this paper.

3 Overview

The overall goal of this paper is to flatten an XQuery expression, i.e. merge as many query blocks (i.e. FLWOR expressions) as possible. To achieve this goal, we basically proceed in two phases: (1) Normalization and (2) FLWOR Merging. Fig. 1 gives an overview of our processing model.

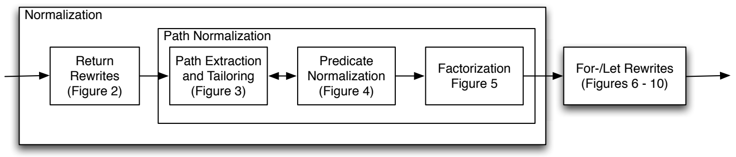

Normalization
Path Normalization
Return Rewrites (Figure 2)
Path Extraction and Tailoring (Figure 3)
Predicate Normalization (Figure 4)
Factorization Figure 5
For-/Let Rewrites (Figures 6 - 10)

Figure 1: Processing model

In both phases, we apply a set of rules based on XQuery syntax on a query. A separate figure presents one set of rules for each normalization and rewriting step. The Overview Figure 1 contains references to each of them.

3.0.1 Normalization

comprises two major subtasks:

1. All `ExprSingle` expressions[1] from the `return` clause are moved to the expression creating the binding sequence of a new `for` expression.
2. Path expressions are normalized (as far as possible). In particular, (1) all path expressions not directly associated with a `for` clause are bound to variables using `let`, (2) path expressions are taken to single steps, (3) predicates are moved into the `where` clause, and (4) common location steps are factorized.

3.0.2 FLWOR Merging

Starting from this normalized form, we remove as many query blocks (FLWOR expressions) as possible. Specifically, we present rewrite rules that eliminate or merge inner FLWORs occurring in the `for` or `let` clause, respectively.

3.0.3 Notation

Our rewrite rules are formulated using XQuery syntax [9]. However, to simplify the presentation, we use the following abbreviations for frequently used clauses:

```
ForOrLetClause    :=   ForClause | LetClause
ForOrLetClauses   :=   ForOrLetClause*
```

Moreover, we assume that all variable names are unambiguous. Since we sometimes introduce new variables or change the bindings of existing ones, we introduce a notation for variable substitution: `Expr`[\$x2 ← \$x1] denotes `Expr` with all free occurrences of \$x2 replaced by \$x1.

3.0.4 Running Example

We illustrate the application of our rules on the query from the introduction. Applying our rules to the example query yields a query which has a single FLWOR block.

The original example query (i.e. before rewriting) is repeated here for convenience.

```
let $auction := doc("auction.xml") return
  let $euro:=for $o in $auction/site/open_auctions/open_auction
             for $i in $auction/site/regions/europe/item/@id
             where $o/itemref/@item eq $i
             return $o
  for $a in $euro
  where zero-or-one($a/bidder[1]/increase/text()) * 2
        <= $a/bidder[last()]/increase/text()
  return
    for $p in $auction/site/people/person[profile/@income > 5000]
    for $w in $p/watches/watch
    where $a/@id = $w/@open_auction
    return <auction id="{$a/@id}">
             <increase first="{$a/bidder[1]/increase/text()}"
                       last="{$a/bidder[last()]/increase/text()}"/>
               <watched_by id="{$p/@id}"/>
           </auction>
```

[1]Note that `ExprSingle` is the expression produced by the grammar rules from [9].

In practice, this query could well be the result of an inlined XQuery function. XQuery functions are often used as *views* to increase data independence, or simply to make queries more readable, similar to views in SQL. In our case, the sequence bound to `$euro` could be an inlined function to retrieve European Auctions, whereas the bottommost FLWOR expression could be a function to retrieve all watchers for a given auction. The results of these functions are joined using the surrounding FLWOR block. In such a context, the application of our rewrite rules can also be described as *view merging*, allowing the plan generator to optimize join orders beyond view borders.

4 Normalization

Normalization does not decrease the FLWOR nesting level of a query. Instead, it transforms the query such that the unnesting rewrite rules can still be applied in case of minor syntactical variations. In addition to this preparatory character, normalization also directly helps to achieve our ultimate goal of preparing queries for plan generation: XQuery allows several different ways of formulating predicates (e.g. the `where` clause and XPath predicates). However, the plan generator requires a single unified formulation of all the constraints on all the variables in the currently considered query block to systematically explore the search space of alternative plans. Execution plan alternatives for value-based predicates include, but are not limited to, the placement of selection operators, the use of joins, and index selection. Which of these alternatives is used, and in which order the different predicates of a query are evaluated, should not depend on the nesting level or the placement of the predicates. This robustness is achieved by our normalization phase.

Normalization proceeds in several consecutive steps, as shown in in Fig. 1. We first enforce a simple form for all `return` clauses before we break down complex location paths into primitives, with an emphasis on predicate normalization. Finally, we eliminate common subexpressions.

4.1 Return Normalization

In order to allow a uniform treatment of nested expressions in `return` and `let` clauses, we move all `ExprSingle` expressions from `return` clauses to `let` clauses (see Rewrite 1). This way, we can treat the unnesting of `return` and `let` uniformly and can always assume a `return` clause that consists of a single variable reference.

ForOrLetClauses WhereClause? OrderByClause? return ExprSingle$_1$	$\rightarrow$	ForOrLetClauses let $\$x1 :=$ ExprSingle$_1$ WhereClause? OrderByClause? return $\$x1$	(1)
ForOrLetClauses$_1$ let $\$x1 :=$ ExprSingle$_1$ ForOrLetClauses$_2$ WhereClause? OrderByClause? return $\$x1$	$\rightarrow$	ForOrLetClauses$_1$ for $\$x1$ in ExprSingle$_1$ ForOrLetClauses$_2$ WhereClause? OrderByClause? return $\$x1$	(2)
Condition: There are no other occurrences of $x1.			

Figure 2: Return rewrites

Other than normalizing the `return` clause, we can further prepare optimization by converting the new `let` clause into a `for` clause (see Rewrite 2). This is possible because on the right-hand side, the concatenation semantics of FLWOR blocks reestablishes the same result sequence as on the left-hand side of the rewrite, as long as $x1 is used nowhere but in the `return` clause.

Turning `let` into `for` expressions allows a significantly larger range of alternatives for plan generation. Evaluation of `for` clauses can be done in an iterative manner, generating the items of the binding sequence one by one, instead of computing and materializing the whole sequence at once. This allows efficient techniques such as pipelining and is the preferred style of implementation in database runtime engines [13].

4.1.1 Running Example

Applying the `return` elimination and `let` transformation rewrites (1 and 2) to the `return` expressions of our example query results in the following:

```
let $auction := doc("auction.xml")
for $x1 in let $euro:= for $o in $auction/site/open_auctions/open_auction
                        for $i in $auction/site/regions/europe/item/@id
                        where $o/itemref/@item eq $i
                        return $o
           for $a in $euro
           for $x2 in for $p in $auction/site/people/person[profile/@income > 5000]
                      for $w in $p/watches/watch
                      for $x3 in <auction id="{$a/@id}">
                             <increase first="{$a/bidder[1]/increase/text()}"
                                  last="{$a/bidder[last()]/increase/text()}"/>
                               <watched_by id="{$p/@id}"/>
                             </auction>
                      where $a/@id = $w/@open_auction
                      return $x3
           where zero-or-one($a/bidder[1]/increase/text()) * 2
                 <= $a/bidder[last()]/increase/text()
           return $x2
return $x1
```

4.2 Path Normalization

Path expressions are a crucial performance factor for the evaluation of almost every XQuery query. For efficiently evaluating path expressions, the plan generator makes cost-based decisions on algorithms that should be used to evaluate them. For example, an optimizer decides whether a holistic approach (e.g. [3, 16]) for evaluating multiple path expressions is superior to a fine granular approach that evaluates single steps individually (e.g. [8, 14]), probably with the help of an index. The plan generator requires a canonical form of the path expressions to make such decisions. Besides separating each processing step for plan generation, cutting path expressions involves two other advantages:

- It allows to move location step predicates from the middle of location paths into the `where` clause.
- Common subexpression elimination (see below) can be done on the granularity of steps.

4.2.1 Path Tailoring

`for` $x `in` `StepExpr/PathExpr`	$\rightarrow$	`for` $x1 `in` `StepExpr`
		`for` $x2 `in` $x1/`PathExpr`
			(3)

Condition: `StepExpr` must not produce duplicates.			

`for` $x `in` `StepExpr/PathExpr`	$\rightarrow$	`let` $x1 := `StepExpr`
		`for` $x2 `in` $x1/`PathExpr`
			(4)

`let` $x := `StepExpr/PathExpr`	$\rightarrow$	`let` $x1 := `StepExpr`
		`let` $x2 := $x1/`PathExpr`
			(5)

Figure 3: Path tailoring rewrites

In order to separate each processing step, we first extract all path expressions from the query which are not already binding expressions of `for` or `let`, and bind them to new `let` variables. We keep path expressions in `for` clauses because they need a different treatment in our predicate rewrites below.

Having extracted all path expressions, we cut them up into single location steps (see Fig. 3 for rewriting rules). Again to facilitate iterator-based evaluation, we attempt to avoid `let` clauses when possible (3 and 4) while breaking up path expressions in `for` clauses. Without further refinements, we can only cut those steps that do not produce duplicates (see [5, 15]). Of course, location steps assigned to a `let` variable remain in a `let` binding (5).

4.2.2 Predicate Normalization

The plan generator not only decides on the path evaluation algorithms and the order of joins based on structural predicates, but also on the order of regular, value-based joins and selections. Moving all non-structural predicates into the `where` clause makes such join and selection predicates explicitly available in a uniform manner. This allows a search space of plans that is robust against the syntactical placement of the predicate. Further, a unified `where` also allows predicate processing, which includes, but is not limited to, the inference of new predicates and the elimination of redundant ones.

$$\begin{array}{lcl}
\texttt{ForOrLetClauses}_1 & & \texttt{ForOrLetClauses}_1 \\
\texttt{for}\ \$x1\ \texttt{in}\ \texttt{StepExpr}[\texttt{Expr}_1] & & \texttt{for}\ \$x1\ \texttt{in}\ \texttt{StepExpr} \\
\texttt{ForOrLetClauses}_2 & \rightarrow & \texttt{ForOrLetClauses}_2 \\
\texttt{where}\ \texttt{Expr}_2 & & \texttt{where}\ \texttt{fn:boolean}(\$x1/(\texttt{Expr}_1))\ \texttt{and}\ \texttt{Expr}_2 \\
\texttt{OrderByClause?} & & \texttt{OrderByClause?} \\
\texttt{return}\ \texttt{ExprSingle}_1 & & \texttt{return}\ \texttt{ExprSingle}_1
\end{array} \qquad (6)$$

Condition: The value of $\texttt{Expr}_1$ must not depend on the context position or context size.

$$\begin{array}{lcl}
\texttt{ForOrLetClauses}_1 & & \texttt{ForOrLetClauses}_1 \\
\texttt{let}\ \$x1 := \texttt{StepExpr}[\texttt{Expr}_1] & & \texttt{let}\ \$x1 := \texttt{StepExpr} \\
\texttt{ForOrLetClauses}_2 & \rightarrow & \texttt{ForOrLetClauses}_2 \\
\texttt{where}\ \texttt{Expr}_2 & & \texttt{where}\ \texttt{fn:boolean}(\texttt{Expr}_1)\ \texttt{and}\ \texttt{Expr}_2 \\
\texttt{OrderByClause?} & & \texttt{OrderByClause?} \\
\texttt{return}\ \texttt{ExprSingle}_1 & & \texttt{return}\ \texttt{ExprSingle}_1
\end{array} \qquad (7)$$

Condition: The value of $\texttt{Expr}_1$ must not depend on the focus (context item, context position, or context size).

$$\begin{array}{lcl}
\texttt{ForOrLetClauses}_1 & & \texttt{ForOrLetClauses}_1 \\
\texttt{for}\ \$x1\ \texttt{in}\ \texttt{StepExpr}[\texttt{Expr}_1\ \texttt{and}\ \texttt{Expr}_2] & & \texttt{for}\ \$x1\ \texttt{in}\ \texttt{StepExpr}[\texttt{Expr}_2] \\
\texttt{ForOrLetClauses}_2 & \rightarrow & \texttt{ForOrLetClauses}_2 \\
\texttt{where}\ \texttt{Expr}_3 & & \texttt{where}\ \texttt{fn:boolean}(\$x1/(\texttt{Expr}_1))\ \texttt{and}\ \texttt{Expr}_3 \\
\texttt{OrderByClause?} & & \texttt{OrderByClause?} \\
\texttt{return}\ \texttt{ExprSingle}_1 & & \texttt{return}\ \texttt{ExprSingle}_1
\end{array} \qquad (8)$$

Condition: The value of $\texttt{Expr}_1$ must not depend on the context position or context size.

$$\begin{array}{lcl}
\texttt{ForOrLetClauses}_1 & & \texttt{ForOrLetClauses}_1 \\
\texttt{for}\ \$x1\ \texttt{in}\ \texttt{StepExpr}[\texttt{Expr}_1\ \texttt{and}\ \texttt{Expr}_2] & & \texttt{for}\ \$x1\ \texttt{at}\ \$y1\ \texttt{in}\ \texttt{StepExpr}[\texttt{Expr}_2] \\
\texttt{ForOrLetClauses}_2 & \rightarrow & \texttt{ForOrLetClauses}_2 \\
\texttt{where}\ \texttt{Expr}_3 & & \texttt{where}\ \texttt{Expr}'_1\ \texttt{and}\ \texttt{Expr}_3 \\
\texttt{OrderByClause?} & & \texttt{OrderByClause?} \\
\texttt{return}\ \texttt{ExprSingle}_1 & & \texttt{return}\ \texttt{ExprSingle}_1
\end{array} \qquad (9)$$

Conditions: The value of $\texttt{Expr}_1$ depends on the context position, but not the context size. $\texttt{Expr}'_1 := \texttt{Expr}_1[\$fs:\mathrm{position} \leftarrow \$y1]$ and `StepExpr` must not consist of a reverse axis step (see text).

Figure 4: Predicate normalization rewrites

In Fig. 4, we present rules that get predicate expressions of location steps and move them into the `where` clause of the surrounding FLWOR block. For each extracted predicate expression, we have to set the context to the context defined by the according step. For example, if we move $\texttt{Expr}_1$ from a location step predicate into a `where` clause (see Rule 6), we have to guarantee that all context accesses are performed with respect to $\$x1$, which is why we prepend $\$x1$ to the predicate expression. Similarly, we can get comparison expressions that contain calls to the context position of a location step by creating a positional variable using the `for VarRef at VarRef` syntax and replacing accesses to the context position with the variable (see Rule 9). This is not strictly possible in XQuery syntax, but easily implemented in most evaluators because the context position is modeled as a special variable anyway. Our choice of variable name ($\$fs:\mathrm{position}$) follows the XQuery Formal Semantics, which also replaces context position by a special variable. Further, reverse axis steps cannot be handled this way, because the context position numbering is different from the order of the result sequence[2].

Note that for the sake of brevity, we assume that there is always a `where` clause in the outer expression. We treat outer FLWORs without a `where` clause as if there was a `where true` clause.

4.2.3 Common Path Elimination

To avoid redundant evaluation, we eliminate common paths, binding them to new `for` or `let` variables as needed. In Fig. 5, we present four rules for eliminating common location steps. However,

[2]If the rewrite is not done on source level, the internal representation may have a suitable special variable to bind for reverse axis numbering, making our rewrite possible again.

$$\begin{array}{l}\texttt{let } \$x1 := \texttt{StepExpr}_1 \\ \texttt{let } \$x2 := \texttt{StepExpr}_1/\texttt{StepExpr}_2\end{array} \rightarrow \begin{array}{l}\texttt{let } \$x1 := \texttt{StepExpr}_1 \\ \texttt{let } \$x2 := \$x1/\texttt{StepExpr}_2\end{array} \qquad (10)$$

$$\begin{array}{l}\texttt{for } \$x1 \texttt{ in StepExpr}_1 \\ \texttt{for } \$x2 \texttt{ in StepExpr}_1/\texttt{StepExpr}_2\end{array} \rightarrow \begin{array}{l}\texttt{let } \$x0 := \texttt{StepExpr}_1 \\ \texttt{for } \$x1 \texttt{ in } \$x0 \\ \texttt{for } \$x2 \texttt{ in } \$x0/\texttt{StepExpr}_2\end{array} \qquad (11)$$

$$\begin{array}{l}\texttt{let } \$x1 := \texttt{StepExpr}_1 \\ \texttt{for } \$x2 \texttt{ in StepExpr}_1/\texttt{StepExpr}_2\end{array} \rightarrow \begin{array}{l}\texttt{let } \$x1 := \texttt{StepExpr}_1 \\ \texttt{for } \$x2 \texttt{ in } \$x1/\texttt{StepExpr}_2\end{array} \qquad (12)$$

$$\begin{array}{l}\texttt{for } \$x1 \texttt{ in StepExpr}_1 \\ \texttt{let } \$x2 \texttt{ in } \$x1/\texttt{StepExpr}_2\end{array} \rightarrow \begin{array}{l}\texttt{let } \$x0 := \texttt{StepExpr}_1 \\ \texttt{for } \$x1 \texttt{ in } \$x0 \\ \texttt{let } \$x2 := \$x0/\texttt{StepExpr}_2\end{array} \qquad (13)$$

Figure 5: Common path elimination

elimination of common subexpressions is a complex process that cannot be sufficiently described using only those rules. We refer to [1] for algorithms on subexpression elimination.

4.2.4 Running Example

In the following, we present the query that is obtained by applying normalization, i.e. path extraction, path tailoring, predicate normalization, and common path elimination, to our example query.

```
let $auction := doc("auction.xml")
let $x32 := $auction/site
for $x1 in let $euro:= for $o in $x32, $x13 in $o/open_auctions
                        for $x14 in $x13/open_auction, $i in $x32, $x15 in $i/regions
                        for $x16 in $x15/europe, $x17 in $x16/item, $x18 in $x17/@id
                        let $x4 := $x14/itemref, $x19 := $x4/@item
                        where $x19 eq $x18
                        return $x14
           for $a in $euro
           let $x33 := $a/bidder[1], $x34 := $x33/increase. $x35 := $x34/text()
           let $x36 := $a/bidder[last()], $x37 := $x36/increase, $x38 := $x37/text()
           let $x39 := $a/@id
           for $x2 in for $p in $x32, $x20 in $p/people, $x21 in $x20/person
                      for $w in $x21/watches, $x22 in $w/watch
                      let $x8 := $x21/@id
                      let $x10 := $x22/@open_auction
                      let $x13 := $x21/profile, $x27 := $x13/@income
                      for $x3 in <auction id="{$x39}">
                                   <increase first="{$x35}" last="{$x38}"/>
                                   <watched_by id="{$x8}"/>
                                 </auction>
                      where $x39 = $x10 and $x27 > 5000
                      return $x3
           where zero-or-one($x35) * 2 <= $x38
           return $x2
return $x1
```

In this expression, for example, the XPath predicate `profile/@income > 5000` is removed from the location step and added to the `where` clause of the according FLWOR block. Moreover, we replaced the common path expressions from within the element construction and the `where` clauses (e.g. the path selecting the increases of the first and last bid) by single variables. Note that it is not possible to move the positional predicates into the `where` clause, as they occur in a `let` binding. Also note that for presentation purposes, we abbreviated consecutive occurrences of `for` and `let` expressions using commas. In the full representation of this query, `for` and `let` expressions that bind multiple variables are split into separate expressions.

5 Merging FLWOR Blocks

After finishing the normalization phase, the query is prepared for the core rules of our toolkit, the `for` and `let` merging rewrites. The ultimate goal of the rewrites presented in this section is to reduce the number of query blocks as much as possible.

Reconsider our normalized example query shown above. This formulation of the query contains several nested FLWOR expressions. The FLWOR nesting depth in line 3 is three. The `for`-clause binding $o is nested in a `let` clause which, in turn, is nested in the outer most `for`-clause binding

$x1. Moreover, the query contains a `for` clause defining $x2 whose binding sequence is generated by another `for` clause.

In the following, we introduce rewrite rules that remove such nested expressions. Applying them to our example query eliminates all nested FLWORs.

We start with rewrites that remove FLWORs nested in `for` clauses (see Fig. 6) and then proceed to `let` clauses (see Fig. 7).

5.1 For Rewrites

The semantics of a `for` clause is to iterate over items of the binding sequence, binding the `for` variable to every item in this sequence. The remaining FLWOR expression is evaluated for each such binding, and the individual result sequences are concatenated. We are interested in a `for` clause if its binding sequence is created by a nested FLWOR expression. In some cases, we can lift the inner FLWOR to the outer level. This rewrite opportunity results from the fact that sequences in the XQuery data model are never nested. Hence, it often does not matter on how many levels an implicit concatenation of `return` sequences occurs because the result is always a flat sequence.

```
ForOrLetClauses₁
for $x1 in (ForOrLetClauses₂                     ForOrLetClauses₁
          for $x2 in ExprSingle₁                 ForOrLetClauses₂
          ForOrLetClauses₃                       for $x1 in ExprSingle₁
          where ExprSingle₂            →         ForOrLetClauses′₃                        (14)
          return $x2)                            ForOrLetClauses₄
ForOrLetClauses₄                                 where ExprSingle₃ and ExprSingle′₂
where ExprSingle₃                                return VarRef₁
return VarRef₁
```

Conditions: $\texttt{ForOrLetClauses}'_3 := \texttt{ForOrLetClauses}_3[\$x2 \leftarrow \$x1]$ and $\texttt{ExprSingle}'_2 := \texttt{ExprSingle}_2[\$x2 \leftarrow \$x1]$

```
ForOrLetClauses₁
for $x1 in (ForOrLetClauses₂                     ForOrLetClauses₁
          let $x2 := ExprSingle₁                 ForOrLetClauses₂
          ForOrLetClauses₃                       let $x2 := ExprSingle₁
          where ExprSingle₂            →         ForOrLetClauses₃                         (15)
          return $x2)                            for $x1 in $x2
ForOrLetClauses₄                                 ForOrLetClauses₄
where ExprSingle₃                                where ExprSingle₃ and ExprSingle₂
return VarRef₁                                   return VarRef₁
```

Figure 6: For rewrites

For example, consider the left-hand side of the first `for` Rewrite 14. In this rewrite, the variable $x1 is iteratively bound to each item returned by the inner FLWOR. The result of the inner FLWOR is generated by the `return` clause. Note that in our case, the `return` clause consists only of a variable reference, i.e. variable $x2. To merge the two blocks, we have to guarantee that the outer `for` variable $x1, after merging, is still bound to the same items, i.e. those generated by variable $x2. To this end, we replace the nested FLWOR with the expression responsible for binding $x2. In the rewrite, this expression is called $\texttt{ExprSingle}_1$ and bound by a `for` clause. The remaining (optional) clauses are moved into the outer FLWOR block. Specifically, $\texttt{ForOrLetClauses}_2$ and $\texttt{ForOrLetClauses}_3$ are pulled up one level. $\texttt{ExprSingle}_2$ from the inner `where` clause is conjunctively connected to the expression in the outer `where` clause [3]. After relocating the inner expressions, we have to replace free occurrences of the previous inner variable $x2 with $x1.

Similarly, we merge two query blocks if the binding sequence is created by a nested `let` variable (see our Rewrite Rule 15). Note that the right-hand side of Rule 15 may still contain a FLWOR nested in a `let` clause. This case is unnested by Rule 16, which is presented in the next section.

Other rules for FLWORs nested within `for` clauses are discussed in the extended version of this paper [2], including cases with positional variables and `order by` clauses.

[3]As before, expressions without `where` are treated as if a `where true` clause was added.

5.1.1 Running Example

On our example query, we can apply Rewrite Rule 14 twice. First, to eliminate the inner `for`-clause binding $x2, as this variable is returned to create the binding sequence for $x1. Second, we apply this rule to eliminate the `for` expression binding $x3. This results in the following expression:

```
let $auction := doc("auction.xml")
let $x32 := $auction/site
let $euro:= for $o in $x32, $x13 in $o/open_auctions , $x14 in $x13/open_auction
            for $i in $x32, $x15 in $i/regions , $x16 in $x15/europe
            for $x17 in $x16/item, $x18 in $x17/@id
            let $x4 := $x14/itemref , $x19 := $x4/@item
            where $x19 eq $x18
            return $x14
for $a in $euro
let $x33 := $a/bidder[1], $x34 := $x33/increase , $x35 := $x34/text()
let $x36 := $a/bidder[last()], $x37 := $x36/increase , $x38 := $x37/text()
let $x39 := $a/@id
for $p in $x32, $x20 in $p/people, $x21 in $x20/person
for $w in $x21/watches, $x22 in $w/watch
let $x8 := $x21/@id, $x10 := $x22/@open_auction
let $x13 := $x21/profile , $x27 := $x13/@income
for $x1 in <auction id="{$x39}">
            <increase first="{$x35}" last="{$x38}"/>
            <watched_by id="{$x8}"/>
           </auction>
where zero-or-one($x35) * 2 <= $x38 and $x39 = $x10 and $x27 > 5000
return $x1
```

5.2 Let Rewrites

`let` clauses require separate rewrites because they bind a variable to the result of its associated expression, i.e. without iterating over this result. Fig. 7 presents three rewrite rules to eliminate FLWORs nested in `let` clauses.

```
ForOrLetClauses1                    ForOrLetClauses1
let $x1 := ExprSingle1              ForOrLetClauses2
ForOrLetClauses2                    for $x2 in ExprSingle1
for $x2 in $x1              →       ForOrLetClauses3                    (16)
ForOrLetClauses3                    where ExprSingle2
where ExprSingle2                   return VarRef
return VarRef
```
Condition: There are no other occurrences of $x1.

```
ForOrLetClauses1
let $x1 := (ForOrLetClauses2              ForOrLetClauses1
            for $x2 in ExprSingle1        ForOrLetClauses2
            where ExprSingle2       →     for $x2 in ExprSingle1                    (17)
            return $x2)                   where ExprSingle3 and ExprSingle2
where ExprSingle3                         return $x2
return $x1
```
Condition: There are no other occurrences of $x1.

```
ForOrLetClauses1
let $x1 := (ForOrLetClauses2              ForOrLetClauses1
            let $x2 := ExprSingle1        ForOrLetClauses2
            where ExprSingle2       →     let $x2 in ExprSingle1                    (18)
            return $x2)                   where ExprSingle3 and ExprSingle2
where ExprSingle3                         return $x2
return $x1
```
Condition: There are no other occurrences of $x1.

Figure 7: Let rewrites

Rewrite Rule 16 tackles a frequently used case. There, a `for` iteration is used to enumerate all items contained in a `let` variable. This technique is used in our example query and may, for example, result from inlining an XQuery function as explained at the beginning of this section. The rules suggest to eliminate the `let` variable if it is used only once and inline the associated expression (i.e. $\texttt{ExprSingle}_1$). On this result, the rewrites of the previous section (see Fig. 6) can be applied and eliminate the nesting.

Fig. 7 also contains two rewrites that remove nested `for` (see Rule 17) and `let` (see Rule 18) expressions, respectively. Without loss of generality, the outer `let` clause in both rules is immediately followed by the `where` clause. If there was another `for` or `let` clause, it would not contain occurrences of x1 and, hence, could be moved above the `let` clause binding x1.

5.2.1 Running Example

The result of applying the `let` Rewrite Rule 16 and the `for` Rewrite Rule 14 to our example is the following query finally consisting of a single query block.

```
let $auction := doc("auction.xml"), $x32 := $auction/site
for $o in $x32, $x13 in $o/open_auctions , $a in $x13/open_auction
for $i in $x32, $x15 in $i/regions , $x16 in $x15/europe
for $x17 in $x16/item , $x18 in $x17/@id
let $x4 := $a/itemref , $x19 := $x4/@item
let $x33 := $a/bidder[1], $x34 := $x33/increase , $x35 := $x34/text()
let $x36 := $a/bidder[last()], $x37 := $x36/increase , $x38 := $x37/text()
let $x39 := $a/@id
for $p in $x32, $x20 in $p/people, $x21 in $x20/person
for $w in $x21/watches, $x22 in $w/watch
let $x8 := $x21/@id, $x10 := $x22/@open_auction
let $x13 := $x21/profile , $x27 := $x13/@income
for $x1 in  <auction id="{$x39}">
              <increase first="{$x35}" last="{$x38}"/>
              <watched_by id="{$x8}"/>
            </auction>
where zero-or-one($x35) * 2 <= $x38 and $x39 = $x10
      and $x27 > 5000 and $x19 eq $x18
return $x1
```

Note how in this form, all value-based join and selection predicates are available in a unified `where` clause. This allows a plan generator to decide on index access and join orders.

6 Intricacies

In the last section, we presented rewrite rules to merge FLWOR expressions. However, all of the presented elimination rewrites were limited in terms of FLWORs that do not contain positional variables or `order by` clauses. Merging FLWORs contains one of these constructs requires more sophisticated rewrite techniques or more restrictive conditions. In this section, we present new rewrite rules that are abutted to the previously presented rules for FLWORs but can also contain one of these intricacies.

6.1 Positional For Rewrites

A `for` expression (having a nested FLWOR expression) binding a positional variable is called a *positional* `for` expression. Fig. 8 presents two rewrites that are abutted to the `for` rewrites from Fig. 6. Both have on the left-hand side a positional `for` expression. In order to merge the outer and the inner FLWOR into one, the inner FLWOR must not contain a `where` or `order by` clause. Our solution for merging such FLWOR expresisons is shown on the right-hand side of Rewrite Rules 19 and 20.

```
ForOrLetClauses₁                                      ForOrLetClauses₁
for $x1 at $y1 in (for $x2 in ExprSingle₁             for $x1 at $y1 in ExprSingle₁
                   return $x2)                   →    ForOrLetClauses₂                  (19)
ForOrLetClauses₂                                      where ExprSingle₂
where ExprSingle₂                                     return VarRef₁
return VarRef₁
```

```
ForOrLetClauses₁                                      ForOrLetClauses₁
for $x1 at $y1 in (let $x2 := ExprSingle₁             for $x1 at $y1 in ExprSingle₁
                   return $x2)                   →    ForOrLetClauses₂                  (20)
ForOrLetClauses₂                                      where ExprSingle₂
where ExprSingle₂                                     return VarRef₁
return VarRef₁
```

Figure 8: Positional for rewrites

6.2 Order-by

A second restriction on the rewrites from the previous section is that neither the outer nor the inner FLWOR must contain an `order by` clause. In this section, we extend our merging rewrite rules to expressions containing an `order by` clause.

For keeping the required order in the unnested case, we introduce a technique that we call *Canonical Order By*. This technique modifies `for` clauses to bind a positional variable using the `for VarRef at VarRef` syntax and add an `order by` clause to the corresponding FLWOR expression having the new positional variables in its `OrderSpecList`. If there are multiple `for` and/or `let` clauses in one FLWOR expression the order of the positional variables in the `OrderSpecList` is determined by the order these clauses occur in the FLWOR expression. Doing so, we guarantee the query result to be in correct order.

In the following, we show how this technique can be applied to merge FLWOR expressions that contain an `order by` clause in the outer and/or the inner FLWOR. Analogously to the previous section, we start with `for` rewrites and then present `let` merging rewrite rules.

6.2.1 For Rewrites

Fig. 9 presents four rules to tackle queries whose inner or outer FLWOR contains an `order by` clause, respectively. The left-hand side (lhs) of Rules 21 and 22 correspond to the lhs of Rules 14 and 15 but with the inner FLWOR expression containing an `order by` clause. Similarly, the lhs of Rules 23 and 24 also correspond to the rules from Fig. 6 but with the outer FLWOR expression containing an `order by` clause. The right-hand side (rhs) of all four rewrite rules shows how the outer and inner FLWOR expression can be merged, despite not changing the order of the result.

On the rhs of each rule, we annotated expressions that need positional variables. For example, the expression

$$\texttt{ForOrLetClauses}_2 \text{ (with positional variables } \$\mathrm{y}2_1, \ldots, \$\mathrm{y}2_n)$$

means that the i-th `for` clause in $\texttt{ForOrLetClauses}_2$ is extended to bind the positional variable $\$\mathrm{y}2_i$ using the `for VarRef at VarRef` syntax. The `order by` clause in all rewrite rules presented in Fig. 9 is simplified such that the `order by` expression could also be an `stable order by` expression. Rules 21 and 22 we could omit the positional variables $\$\mathrm{y}4_1, \ldots, \$\mathrm{y}4_m$ and instead use `stable order by`. However, the latter could be less efficient as it probably requires sorting on uneccesary attributes.

In case both, the outer and inner FLWOR contain an `order by` clause, we proceed as shown in Fig. 6. However, we also merge the two `order by` clauses such that sorting is done first for the `OrderSpecList` of the outer FLWOR and then for the `OrderSpecList` of the inner FLWOR.

6.2.2 Let Rewrites

Similarly to the previous section, we also allow our `let` merging rewrite rules from Fig. 7 to contain `order by` clauses. The resulting rewrite rules are presented in Fig. 10.

Rewrite Rule 25 corresponds to Rule 16 but having an `order by` clause. Rules 26 and 27 contain an `order by` clause in the inner FLWOR, Rules 28 and 29 in the outer FLWOR, respectively.

7 Evaluation

A goal of this paper is to show how to rewrite a query into a form that consists of a single query block to give a single run of the plan generator as much uniformly structured information about the query as possible. We now elaborate on the importance of this goal by discussing the optimization of our example query during plan generation. We will see how more efficient plans can be generated only when the query has been reduced to a single block.

Due to space constraints, we do not explore the whole search space available, but focus on join ordering. We assume that the optimizer has decided on subplans to produce the sequences for Open

```
ForOrLetClauses_1
for $x1 in (ForOrLetClauses_2                ForOrLetClauses_1 (with positional variables $y1_1, ..., $y1_n)
           for $x2 in ExprSingle_1           ForOrLetClauses_2
           ForOrLetClauses_3                 for $x1 at $y2 in ExprSingle_1
           where ExprSingle_2                ForOrLetClauses'_3
           order by OrderSpecList_1    →     ForOrLetClauses_4 (with positional variables $y4_1, ..., $y4_m)   (21)
           return $x2)                       where ExprSingle_4 and ExprSingle'_2
ForOrLetClauses_4                            order by $y1_1, ..., $y1_n, $y2, OrderSpecList'_1, $y4_1, ..., $y4_m
where ExprSingle_4                           return VarRef_1
return VarRef_1
```

Conditions: ForOrLetClauses$'_3$:= ForOrLetClauses$_3$[\$x2 ← \$x1], ExprSingle$'_2$:= ExprSingle$_2$[\$x2 ← \$x1], and OrderSpecList$'_1$:= OrderSpecList$_1$[\$x2 ← \$x1]

```
ForOrLetClauses_1
for $x1 in (ForOrLetClauses_2                ForOrLetClauses_1 (with positional variables $y1_1, ..., $y1_n)
           let $x2 := ExprSingle_1           ForOrLetClauses_2
           ForOrLetClauses_3                 let $x2 := ExprSingle_1
           where ExprSingle_2                ForOrLetClauses_3
           order by OrderSpecList_1    →     for $x1 at $y2 in $x2                                             (22)
           return $x2)                       ForOrLetClauses_4 (with positional variables $y4_1, ..., $y4_m)
ForOrLetClauses_4                            where ExprSingle_4 and ExprSingle_2
where ExprSingle_4                           order by $y1_1, ..., $y1_n, $y2, OrderSpecList_1, $y4_1, ..., $y4_m
return VarRef_1                              return VarRef_1
```

```
ForOrLetClauses_1
for $x1 in (ForOrLetClauses_2              ForOrLetClauses_1
           for $x2 in ExprSingle_1         ForOrLetClauses_2 (with positional variables $y2_1, ..., $y2_n)
           ForOrLetClauses_3               for $x1 at $y2 in ExprSingle_1
           where ExprSingle_2              ForOrLetClauses'_3 (with positional variables $y3_1, ..., $y3_m)   (23)
           return $x2)              →      ForOrLetClauses_4
ForOrLetClauses_4                          where ExprSingle_4 and ExprSingle'_2
where ExprSingle_4                         order by OrderSpecList_1, $y2_1, ..., $y2_n, $y2, $y3_1, ..., $y3_m
order by OrderSpecList_1                   return VarRef_1
return VarRef_1
```

Conditions: ForOrLetClauses$'_3$:= ForOrLetClauses$_3$[\$x2 ← \$x1], ExprSingle$'_2$:= ExprSingle$_2$[\$x2 ← \$x1], and OrderSpecList$'_1$:= OrderSpecList$_1$[\$x2 ← \$x1]

```
ForOrLetClauses_1
for $x1 in (ForOrLetClauses_2              ForOrLetClauses_1
           let $x2 := ExprSingle_1         ForOrLetClauses_2 (with positional variables $y2_1, ..., $y2_n)
           ForOrLetClauses_3               let $x2 := ExprSingle_1
           where ExprSingle_2              ForOrLetClauses_3 (with positional variables $y3_1, ..., $y3_m)
           return $x2)              →      for $x1 at $y2 in $x2                                            (24)
ForOrLetClauses_4                          ForOrLetClauses_4
where ExprSingle_4                         where ExprSingle_4 and ExprSingle_2
order by OrderSpecList_1                   order by OrderSpecList_1, $y2_1, ..., $y2_n, $y2, $y3_1, ..., $y3_m
return VarRef_1                            return VarRef_1
```

Figure 9: Order-by for rewrites

Auctions, European Items, and Persons. The subplans may be based on pattern matching algorithms. Further, we assume that the predicate selecting the auctions according to their bids has been converted into a single predicate subplan. This predicate is, however, more expensive to evaluate than a simple value comparison, and its placement in the overall plan does affect performance significantly. Thus, finding an optimal plan includes finding an optimal position for this predicate. We now discuss execution plans for our example query in the form of algebraic expressions on an abstract level (see Fig. 11).

A straightforward translation of the original, nested, multi-block query looks like Fig. 11(a). Here, the FLWOR blocks are translated directly into separate subplans, and no global optimization takes place. For simplicity, we disregard the first line of the example query (the initial `let` clause for the document root). The top-level MapConcat operator represents the main FLWOR expression. Its operand generates the tuple stream and contains subplans for the European Auctions query block. The subplan connected to MapConcat by the dashed line represents the query block in the `return` clause (the last eight lines of the query). It has a free variable \$a in the subplan for the `people` sequence, and, hence, has to be reevaluated for every tuple of the MapConcat operand, as dictated by XQuery FLWOR semantics.

Fig. 11 shows four other execution plans based on the rewritten, single-block form of our example query. They can be enumerated by the plan generator because it has access to all value-based

```
ForOrLetClauses_1                       ForOrLetClauses_1
let $x1 := ExprSingle_1                 ForOrLetClauses_2
ForOrLetClauses_2                       for $x2 in ExprSingle_1
for $x2 in $x1                     →    ForOrLetClauses_3                                   (25)
ForOrLetClauses_3                       where ExprSingle_2
where ExprSingle_2                      order by OrderSpecList
order by OrderSpecList                  return VarRef
return VarRef
```
Condition: There are no other occurrences of $x1.

```
ForOrLetClauses_1
let $x1 := (ForOrLetClauses_2                 ForOrLetClauses_1 (with positional variables $y1_1, ..., $y1_n)
           for $x2 in ExprSingle_1            ForOrLetClauses_2
           where ExprSingle_2                 for $x2 at $y2 in ExprSingle_1
           order by OrderSpecList_1     →     where ExprSingle_3 and ExprSingle_2                      (26)
           return $x2)                        order by $y1_1, ..., $y1_n, $y2, OrderSpecList_1
where ExprSingle_3                            return $x2
return $x1
```
Condition: There are no other occurrences of $x1.

```
ForOrLetClauses_1
let $x1 := (ForOrLetClauses_2                 ForOrLetClauses_1 (with positional variables $y1_1, ..., $y1_n)
           let $x2 := ExprSingle_1            ForOrLetClauses_2
           where ExprSingle_2                 let $x2 in ExprSingle_1
           order by OrderSpecList_1     →     where ExprSingle_3 and ExprSingle_2                      (27)
           return $x2)                        order by $y1_1, ..., $y1_n, OrderSpecList_1
where ExprSingle_3                            return $x2
return $x1
```
Condition: There are no other occurrences of $x1.

```
ForOrLetClauses_1
let $x1 := (ForOrLetClauses_2               ForOrLetClauses_1
           for $x2 in ExprSingle_1          ForOrLetClauses_2 (with positional variables $y2_1, ..., $y2_n)
           where ExprSingle_2               for $x2 at $y2 in ExprSingle_1
           return $x2)                →     where ExprSingle_3 and ExprSingle_2                        (28)
where ExprSingle_3                          order by OrderSpecList_1, $y2_1, ..., $y2_n, $y2
order by OrderSpecList_1                    return $x2
return $x1
```
Condition: There are no other occurrences of $x1.

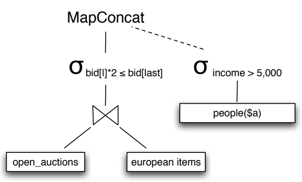
```
ForOrLetClauses_1
let $x1 := (ForOrLetClauses_2               ForOrLetClauses_1
           let $x2 := ExprSingle_1          ForOrLetClauses_2 (with positional variables $y2_1, ..., $y2_n)
           where ExprSingle_2               let $x2 in ExprSingle_1
           return $x2)                →     where ExprSingle_3 and ExprSingle_2                        (29)
where ExprSingle_3                          order by OrderSpecList_1, $y2_1, ..., $y2_n
order by OrderSpecList_1                    return $x2
return $x1
```
Condition: There are no other occurrences of $x1.

Figure 10: Order-by let rewrites

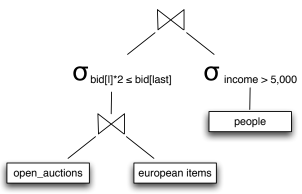
predicates of the query in a single `where` clause and can detect joins and determine an optimal order for them and the residual selections. We executed all five plans from Fig. 11 in our hybrid relational and XML DBMS Natix [11] on an XMark document with scaling factor one.

The experimental setup consisted of a PC with an Intel Pentium D CPU having 3.40GHz and 1GB of main memory, running on openSUSE 10.2 with Linux Kernel 2.6.18 SMP. To investigate the relative performance of the execution plans, we varied the selectivity of the predicate restricting the people by their income between 0.14 and 0. This corresponds to incomes between 60,000$ to 130,000$ instead of 5,000$ in the original query. Fig. 12 shows the result of this small performance study (execution time in seconds) for four plans from Fig. 11.

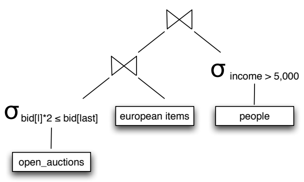
The experiment makes obvious why careful global plan generation based on single-block queries is crucial for efficient execution. The results of the nested-loop strategy of the straightforward translation are orders of magnitude slower (well beyond 100s) and have been left out of the graph. The join-based plans made possible by our rewritten single-block query show that an enumeration of alternatives is as important as in relational query processing: Depending on selectivity, the overall best plan varies. The plan according to Fig. 11(e) performs best with a very low selectivity, whereas the plan belonging to Fig. 11(b) outperforms the others with an increasing selectivity.



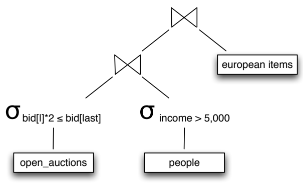
(a) Block-by-block translation



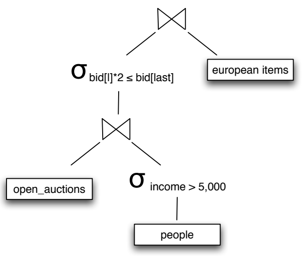
(b) Plan 1



(c) Plan 2



(d) Plan 3



(e) Plan 4

Figure 11: Alternative execution plans

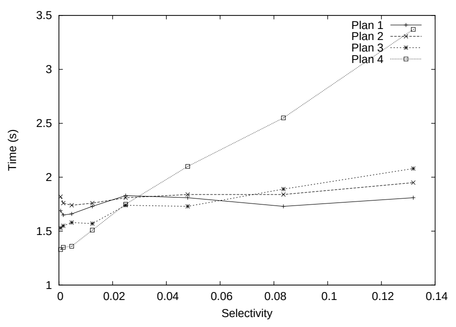


Figure 12: Performance results

References

[1] A. Aho, R. Sethi, and J.D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] M. Brantner, C-C. Kanne, and G. Moerkotte. Let a single flwor bloom. Technical report, University of Mannheim, 2007. http://db.informatik.uni-mannheim.de/publications/TR-07-001.pdf.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, pages 310–321, 2002.

[4] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proc. VLDB*, pages 197–208, 1987.

[5] J. Hidders et al. How to recognise different kinds of tree patterns from quite a long way away. In *Proc. PLAN-X*, 2007.

[6] K. S. Beyer et al. System RX: One part relational, one part XML. In *SIGMOD*, pages 347–358, 2005.

[7] N. May et al. XQuery processing in natix with an emphasis on join ordering. In *First International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P 2004)*, June 2004.

[8] S Al-Khalifa et al. Structural Joins: A primitive for efficient XML query pattern matching. In *ICDE*, pages 141–, 2002.

[9] S. Boag et al. XQuery 1.0: An XML query language. Technical report, World Wide Web Consortium, January 2007. W3C Recommendation.

[10] Shankar Pal et al. Indexing XML data stored in a relational database. In *VLDB*, pages 1134–1145, 2004.

[11] T. Fiebig et al. Anatomy of a native XML base management system. *j-VLDB-J*, 11(4):292–314, 2002.

[12] R. A. Ganski and H. K. T. Wong. Optimization of nested sql queries revisited. In *SIGMOD*, pages 23–33, 1987.

[13] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[14] T. Grust and M. v. Keulen. Tree awareness for relational dbms kernels: Staircase join. In *Intelligent Search on XML Data*, pages 231–245, 2003.

[15] J. Hidders and P. Michiels. Avoiding unnecessary ordering operations in xpath. In *Database Programming Languages*, pages 54–70, 2003.

[16] V. Josifovski, M. Fontoura, and A. Barta. Querying XML streams. *j-VLDB-J*, 14(2):197–210, 2005.

[17] Z. H. Liu, M. Krishnaprasad, and V. Arora. Native XQuery processing in Oracle XMLDB. In *SIGMOD*, pages 828–833, 2005.

[18] N. May, S. Helmer, and G. Moerkotte. Strategies for query unnesting in XML databases. *ACM Transactions on Database Systems*, 31(3):968–1013, 2006.

[19] P. Michiels, G. Mihaila, and J. Siméon. Put a tree pattern in your algebra. In *Proc. ICDE*, 2007.

[20] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *SIGMOD*, pages 39–48, 1992.

[21] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *Proc. ICDE*, pages 450–458, 1996.