The NEXT Framework for Logical XQuery Optimization

Alin Deutsch Yannis Papakonstantinou Yu Xu

University of California San Diego

Abstract

Classical logical optimization techniques rely on a logical semantics of the query language. The adaptation of these techniques to XQuery is precluded by its definition as a functional language with operational semantics. We introduce Nested XML Tableaux which enable a logical foundation for XQuery semantics and provide the logical plan optimization framework of our XQuery processor. As a proof of concept, we develop and evaluate a minimization algorithm for removing redundant navigation within and across nested subqueries. The rich XQuery features create key challenges that fundamentally extend the prior work on the problems of minimizing conjunctive and tree pattern queries.

1 Introduction

The direct applicability of logical optimization techniques (such as rewriting queries using views, semantic optimization and minimization) to XQuery is precluded by XQuery's definition as a functional language [30]. The normalization module of the NEXT XQuery processor enables logical optimization of XQueries by reducing them to *NEsted Xml Tableaux (NEXT)*[1], which are based on logical semantics. NEXT extend tree patterns [3, 21] (which have been used in XPath minimization and answering XPath queries using XPath views) with nested subqueries, joins, and arbitrary mixing of set and bag semantics.

As a proof-of-concept of NEXT's applicability to XQuery logical optimization, but also for its own importance in improving query performance, we developed and evaluated a query minimization algorithm that removes redundant navigation within and across nested subqueries. Minimization is particularly valuable in an XQuery context, since redundant XML navigation arises naturally and unavoidably in *nested* queries, where the subqueries perform navigation that is redundant relative to the query they are nested in. A common case is that of queries that perform grouping in order to restructure or aggregate the source data. The grouping is typically expressed using a combination of self-join and nesting, in which the navigation in the nested, inner subquery completely duplicates the navigation of the outer query (see Examples 1.1 and 1.2). Another typical scenario pertains to mediator settings, where queries resulting from unfolding the views [20, 17, 25] in the original client queries contain nested and often redundant subqueries (when the navigation in two view definitions overlaps). Finally query generation tools tend to generate non-minimal queries [31].

EXAMPLE 1.1 Consider the following query that groups books by authors (it is a minor variation of query Q9 from W3C's XMP use case [27]). The **distinct-values** function eliminates duplicates, comparing elements by value-based equality [30].

```
let $doc := document("input.xml")
for $a in distinct-values($doc//book/author)
return <result> { $a,                                    (X1)
        for $b in $doc//book
        where some $ba in $b/author satisfies $ba eq $a
        return $b }
        </result>
```

Notice that the **for** loop binding $\$a$ (from now on called the $\$a$ loop) has *set* semantics, all others have *bag* semantics i.e., duplicates are not removed.[2]

The straightforward nested-loop execution of this query is wasteful since the nested loops (the $\$b$ **for** loop and the $\$ba$ **some** loop) are redundant: the $\$a$ loop has already navigated to the corresponding book and author elements. In this case, we say that the redundant navigation appears *across* nested subqueries, where nesting is w.r.t. the **return** clause. The NEXT XQuery processor performs a more efficient execution (inspired by the OQL groupby operator [8]): eliminate the redundant navigation by scanning books and authors just once and then apply a group-by operation. ◇

Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004

[1]Both the plural and singular form are "NEXT".

[2]The query can be expressed in a shorter form by replacing its **where** clause with "**where** $\$a = \$b/author$" or by replacing the inner **for** with "$\$doc//book[author = \$a]$". It is well known [19] how to reduce such syntactic sugar (use of "=" or use of predicates in paths) to the basic XQuery constructs we use (see Figure 3).

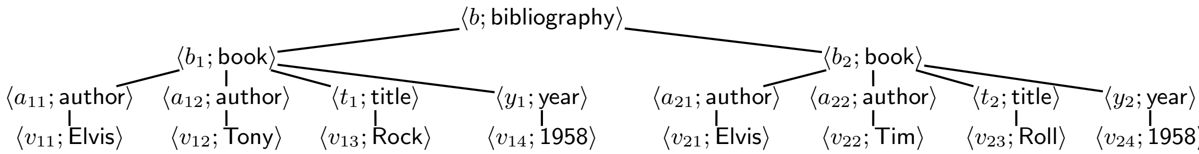


Figure 1: Data of Running Example

It turns out that, when attempting to perform grouping by more than one variable, the resulting XQueries contain redundant navigation both across and within subqueries.

EXAMPLE 1.2 The following nested XQuery groups on two variables: book titles are grouped by author and year of publication.

```
for $a in distinct-values($doc//book/author)          (X2)
    $y in distinct-values($doc//book/year)
where some $b3 in $doc//book,$a3 in $b3/author,
              $y3 in $b3/year
        satisfies $a eq $a3 and $y eq $y3
return <result> {$a, $y,
          for $b' in $doc//book
          where some $a' in $b'/author, $y' in $b'/year
                  satisfies $a'eq $a and $y'eq $y
          return $b'/title}
        </result>
```

The $\$doc$ variable is defined as in the first line of (X1) and its definition will be omitted from now on. Notice the use of join equality conditions on $author$ and $year$ in the **some** of the $\$b'$ loop. Once again, the navigation of the outermost subquery (the $\$a$ and $\$y$ loops) is duplicated by the nested subquery. In addition, redundant navigation occurs also within the outermost subquery: the **some** loop binding $\$b_3$ navigates to $book$, $author$ and $year$ elements, all of whom are also visited by the $\$a$ and $\$y$ loops. ⋄

The combined effect of the normalization and minimization modules of the NEXT XQuery processor removes the redundant navigation from the above examples. This minimization is beneficial regardless of the query execution model. In many XQuery processors, including our own, the matching of paths and equality conditions is performed by joins that outperform brute force loops. Minimization reduces the number of joins in such cases.

Section 2 describes the system architecture and NEXT and highlights NEXT's key logical optimization enabling feature: NEXT consolidate all navigation of the original query in the XTableaux tree pattern structure, regardless of whether navigation originally appeared in the **where** clause, within non-path expressions in the **in** clause, or even within subqueries that are within a **distinct-values** and hence follow set semantics.

Section 3 describes the normalization algorithm that reduces a wide set of XQueries, called OptXQuery, to NEXT. All example XQueries appearing in this paper fall in this class. Due to space limitations we only briefly discuss in Appendix D the processing of non-OptXQuery XQueries.

Section 4 describes a minimization algorithm that, given a NEXT, fully removes redundant navigation, in a formally

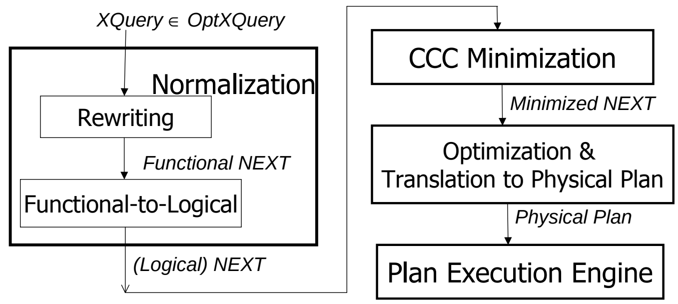


Figure 2: The NEXT XQuery Processor Architecture

defined sense. The expressiveness of OptXQuery raises the following novel challenges that fundamentally change the nature of the minimization problem, such that previous algorithms for the minimization of conjunctive queries [5, 2] and XPath queries [3, 23, 11], do not apply:

1. OptXQueries are nested (as opposed to conjunctive queries and tree patterns).
2. OptXQueries perform arbitrary joins (in contrast to tree patterns, which correspond to acyclic joins [12]).
3. OptXQueries freely mix bag and set semantics (as opposed to allowing either pure bag or pure set semantics in relational queries, and only set semantics in tree patterns).

Section 5 discusses the implementation of the minimization algorithm. Though the problem is NP-hard, as is the case for minimization of relational queries, the implementation reduces the exponentiality to an approximation of the query tree width [12] and results in fast minimization even for very large queries, as proven by our experimental results. We summarize the contributions of this work and provide future directions in Section 6. Related work is described in Section 7.

2 Framework and Architecture

XML We model an XML document D as a labeled tree of nodes N_{XML}, edges E_{XML}, a function $\lambda : \mathsf{N}_{XML} \rightarrow$ *Constants* that assigns a label to each node, and a function $\mathsf{id} : \mathsf{N}_{XML} \rightarrow$ *IDs* that assigns a unique id to each node. We ignore node order. The tree of Figure 1 serves as our running example.

OptXQuery The paper focuses on the OptXQuery subset of XQuery, which follows the syntax of Figure 3 and also satisfies the constraints described below. Notice that OptXQuery allows navigation along the children (/) and descendant (//) axes of XPath, existential quantification using **some**, arbitrary conjunctive conditions (as opposed to

$$
\begin{array}{lll}
\mathit{XQ} & ::= & \langle n\rangle\{\mathit{XQ}_1,\ldots,\mathit{XQ}_m\}\langle/n\rangle \\
 & | & \mathit{XQ}_1, \mathit{XQ}_2 \\
 & | & \textbf{for}\ (V\ \textbf{in}\ \mathit{XQ})+\ (\textbf{where}\ \mathit{CList})? \\
 & & \textbf{return}\ \mathit{XQ} \\
 & | & (\textbf{document}\ (\text{“}constant''\text{)}|\mathit{Var})((/|//)\mathit{Constant})* \\
 & | & \mathit{Constant} \\
 & | & \textbf{distinct-values}(\mathit{XQ}) \\
\mathit{CList} & ::= & \mathit{Cond}\ (\textbf{and}\ \mathit{Cond})* \\
\mathit{Cond} & ::= & \mathit{Var}_1\ \textbf{eq}\ (\mathit{Var}_2|\mathit{Constant}) \\
 & | & \textbf{some}\ (V\ \textbf{in}\ \mathit{XQ})+\ \textbf{satisfies}\ \mathit{CList}
\end{array}
$$

Figure 3: OptXQuery

acyclic conditions only [12]), element creation that may include nested queries (as opposed to tree conditions that return a single element or tuples of variable bindings, and duplicate elimination using the **distinct-values** function (which allows both bags and sets). The grammar can be trivially extended with additional constructs that have an obvious reduction to OptXQuery, such as predicates in path expressions.

OptXQuery's constraints rule out (i) queries that directly or indirectly test the equality of constructed sets (ii) implicit disjunctive conditions (aside from the explicit absence of **or**). Appendix C provides sufficient conditions for ruling out (i) and (ii). We limited the syntax and included the first constraint in order to be able to guarantee full minimization, as explained in Section 4, since it is well known from both relational and object-oriented query processing that minimization and containment problems become undecidable once set equality, negation and universal quantification are allowed. On the contrary, there is no theoretical reason against disjunctions and we can extend NEXT to incorporate them, but for simplicity we focus on purely conjunctive queries. Though only OptXQueries are guaranteed to be fully minimized, the processor may also input arbitrary XQueries and optimize them using minimization, as briefly discussed in Appendix D. The main body of the paper assumes that the input query is in OptXQuery.

Normalization and NEXT The normalization module of the NEXT processor (see Figure 2) inputs an OptXQuery, applies a series of rewriting rules, discussed in Section 3, and produces a *functional NEXT*, whose syntax (see Figure 4) extends a subset of OptXQuery with an OQL-inspired group-by construct [4].

Functional NEXT The functional NEXT syntax allows only path expressions in the **for** clause, while OptXQuery also allowed nested subqueries. Also, NEXT allows only variables in the condition, while OptXQuery also allowed **some**, which include existential navigation. It is the use of group-by that has enabled us to move all navigation to the path expressions of the **in** clauses. The *Functional-to-Logical* module performs a straightforward translation of its input into a *logical NEXT*, whose syntax extends tree

$$
\begin{array}{lll}
\mathit{XQ} & ::= \langle n\rangle\{\mathit{XQ}_1,\ldots,\mathit{XQ}_m\}\langle/n\rangle & \text{(P1)} \\
 & |\ V & \text{(P2)} \\
 & |\ \textbf{for}\ V_1\ \textbf{in}\ \mathit{Path}_1,\ldots,V_n\ \textbf{in}\ \mathit{Path}_n & \text{(P3)} \\
 & \quad (\textbf{where}\ \mathit{CList})? & \\
 & \quad \textbf{groupby}\ (V_1'|[V_1'])\ldots(V_k'|[V_k'])(\textbf{into}\ P)? & \\
 & \quad \textbf{return}\ \mathit{XQ}_1 & \\
\mathit{Path} & ::= (\textbf{document}\ (\text{“}Constant''\text{)}|\mathit{Var})((/|//)\mathit{Constant}) & \text{(P4)} \\
\mathit{CList} & ::= \mathit{Cond}\ (\textbf{and}\ \mathit{Cond})* & \text{(P5)} \\
\mathit{Cond} & ::= V_1\ \textbf{eq}\ (V_2|\mathit{Constant}) & \text{(P6)}
\end{array}
$$

Figure 4: Functional NEXT Syntax

patterns [21, 3, 23] to capture nesting, cyclic joins, and mixed set and bag semantics. There is an 1-1 correspondence between functional and logical NEXT expressions.

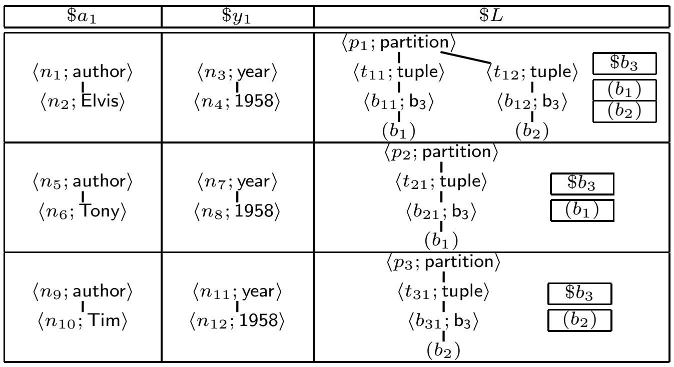
Group-By The arguments of group-by are a list of *groupby variables* $G_1,\ldots,G_k$, the name of an optional *partition* variable P, and the result expression. A group-by inputs the tuples of variable bindings produced by the **for** and **where** clauses and outputs a tuple set that has exactly one tuple for every set of tuples that have equal groupby variable bindings. Equality is identity-based if the groupby variable appears as $[G_i]$ or value-based if the variable appears as G_i. In OQL fashion, a new variable binding is created for the variable P and binds to a table that has the tuples that belong to this group. However, in order to stay within the XML data model, we emulate the nested table with a special partition element that contains tuple elements, which in turn contain elements named after the names of the aggregated variables, excluding $.

For example, consider the functional NEXT (X3), which groups book titles by author and year (indeed, it is the minimized form of XQuery (X2), and the corresponding logical NEXT will be seen in Figure 8(c)).

for $\$b_3$ **in** \$doc//book, $\$a_1$ **in** $\$b_3$/author, $\$y_1$ **in** $\$b_3$/year
groupby $\$a_1, \$y_1$ **into** \$L **return** (X3)
$\langle$result$\rangle$\{ $\$a_1, \$y_1$
for $\$b'$ **in** $\$L/tuple/b_3$ **groupby** $[\$b']$ **return**
for $\$t$ **in** $\$b'/title$ **groupby** $[\$t]$ **return** $\$t$ \}
$\langle$/result$\rangle$

The first table below illustrates the tuples generated by the outermost **for** clauses of (X3) when run on the data of Figure 1 and the next table illustrates the output of its first group-by. For illustration purposes, the bindings of the partition variable are also shown in nested table format. The notation (x) stands for the tree rooted at the node with id x. Notice that grouping by value results into creating copies for the bindings of the group-by variables in the result. For example, notice that the first binding of $\$a_1$ is neither (a_{11}) nor (a_{21}) but is a new object (n_1) that has equal value with (a_{11}) and (a_{21}). Efficient implementations of group-by can avoid to physically produce copies.

$\$a_1$	$\$y_1$	$\$b_3$
(a_{11})	(y_1)	(b_1)
(a_{12})	(y_1)	(b_1)
(a_{21})	(y_2)	(b_2)
(a_{22})	(y_2)	(b_2)



Logical Next The Functional-to-Logical module creates the logical NEXT that corresponds to its input. Figure 5 illustrates the functional and the logical NEXT that correspond to query (X2).

Logical NEXT reflect the nesting of group-by expressions using a *groupby tree* (see tree on the left side of the logical NEXT of Figure 5). Each node of the groupby tree corresponds to a **for** expression of the functional NEXT and the immediate nesting of two **for** expressions is represented by an edge between their nodes. We label a node N with $N(\mathbf{X}; G_i; G_v; \mathbf{f})$ (for example, $\mathsf{N}_1(\mathsf{X}_1; ; \$a_1, \$y_1; f_1(\$a_1, \$y_1, \mathsf{N}_2))$), where:

⇒ the *XTableau* $\mathbf{X} = (F, EQ_{val}, EQ_{id})$ consists of a forest F of tree patterns, which captures navigation, a set of value-based equality conditions EQ_{val} (represented by bubble-ended dotted lines) and a set of id-based equalities EQ_{id} (represented by arrow-ended dotted lines). The three shaded sections of the pattern in Figure 5 correspond to the Xtableaux of $\mathsf{N}_1, \mathsf{N}_2, \mathsf{N}_3$. The formal XTableau semantics extend the tree pattern semantics of [21] to account for the equality conditions and specify the set of bindings for the variables of the tree pattern $\mathbf{X}$. An alternate (and shorter) route towards specifying the bindings of the variables of the XTableaux is based on the 1-1 correspondence between logical and functional NEXT: Each node in the XTableau of group-by tree node N corresponds to a variable in the **for** expression that corresponds to N. Each edge corresponds to a navigation step to a child (graphically represented by a single edge) or a descendant (represented by a double edge). Nodes are labeled with the corresponding tag name tests, or $*$ if no such test is performed. Similarly, the equality conditions in the **where** clause correspond to the equalities of the XTableau. The set of variable bindings delivered by the XTableau is the set of bindings delivered for the variables of the corresponding **for** expression in the functional NEXT. In addition to prior tree pattern formalisms, we accommodate free and bound variables: since the nested queries may refer to variables bound in outer queries. For example, variable $\$b'$ is bound in N_2 and free in N_3. Tree patterns of a groupby node may be rooted at variable nodes bound in the tree pattern of an ancestor groupby node. Similarly, equalities may involve variables that are bound at ancestor groupby nodes. The equality $\$a_1$ **eq** $\$a'$ belongs to X_2 despite $\$a_1$ being free in X_2. Also, $\$b'$ belongs to X_2 (where it is bound), and it is free in X_3.

⇒ G_i and G_v are the vectors of groupby-id variables and groupby-value variables. For example, N_1 has an empty groupby-id list and its groupby-value variable list "$\$a_1, \$y_1$" specifies that the result expression f_1 will be invoked once for each unique pair of values of $\$a_1, \$y_1$, where uniqueness is based on value comparison. The variable list corresponds to the groupby list of the functional NEXT.

⇒ the *result* function $\mathbf{f}$ inputs the group-by variables' bindings and the results of the nested queries and outputs an XML tree. The result function may be the identity function or it may involve concatenation and/or new element creation. The function f_1 creates an element named result that contains $\$a_1$, $\$y_1$ and the result of N_2 (in this order). The function f_2 returns the result of N_3 and f_3 returns $\$t$. The specifics of the function are unimportant for minimization purposes, since it cannot be minimized; hence in the rest of the paper we refer to the result functions as $f_1, f_2, \ldots$.

Normalization Benefit Normalization reduces queries into the NEXT form, where all selections and navigations are consolidated in the XTableaux, regardless of whether navigation initially appeared in **some** loops, within **distinct-values** functions, or within subqueries nested in the **in** clause (see following example). This consolidation enables minimization to detect the opportunities for eliminating redundant navigation, regardless of the context in which navigation originally appeared. Normalization is crucial for maximizing the minimization opportunities and guaranteeing full minimization for the queries of OptXQuery. Example 2.1 below illustrates the need for the consolidation achieved through normalization. It shows a query that is semantically equivalent to (X2) but involves a more complex **in** clause. The combined action of normalization and minimization reduces it to the same minimal form with (X2). We will see how this query is normalized in Section 3.

EXAMPLE 2.1 While apparently more complicated than the query (X2), query (X5) below is what an XQuery expert would write, since it results in a more efficient execution plan, that avoids redundant navigation within the same subquery. In fact this is the most efficient way to perform grouping by multiple variables in XQuery.

for $\$p$ **in distinct-values**(
    for $\$b_1$ **in** $\$doc//book$,
        $\$a_1$ **in** $\$b_1/author$, $\$y_1$ **in** $\$b_1/year$
    return ⟨pair⟩⟨a⟩{$\$a_1$}⟨/a⟩⟨y⟩{$\$y_1$}⟨/y⟩⟨/pair⟩),
  $\$a$ **in** $\$p/a/author$, $\$y$ **in** $\$p/y/year$ (X5)
return ⟨result⟩ {$\$a$}{$\$y$}
    { **for** $\$b'$ **in** $\$doc//book$
      where some $\$a'$ **in** $\$b'/author$, $\$y'$ **in** $\$b'/year$
      satisfies $\$a'$ **eq** $\$a$ **and** $\$y'$ **eq** $\$y$
      return $\$b'/title$}
  ⟨/result⟩

The outermost **for** binds the variable $\$p$ to distinct pairs of $author$ and $year$ subelements of $book$ elements. For each

$$
N_1 \left\{ \begin{array}{l}
\textbf{for } \$b_1 \textbf{ in } \$doc//book, \$a_1 \textbf{ in } \$b_1/author, \\
\quad \$b_2 \textbf{ in } \$doc//book, \$y_1 \textbf{ in } \$b_2/year, \\
\quad \$b_3 \textbf{ in } \$doc//book, \$a_3 \textbf{ in } \$b_3/author, \$y_3 \textbf{ in } \$b_3/year \\
\textbf{where } \$a_1 \textbf{ eq } \$a_3 \textbf{ and } \$y_1 \textbf{ eq } \$y_3 \\
\textbf{groupby } \$a_1, \$y_1 \textbf{ return} \\
\langle \text{result} \rangle \{\$a_1, \$y_1, \\
\quad N_2 \left\{ \begin{array}{l}
\textbf{for } \$b' \textbf{ in } \$doc//book, \$a' \textbf{ in } \$b'/author, \\
\quad \$y' \textbf{ in } \$b'/year \\
\textbf{where } \$a_1 \textbf{ eq } \$a' \textbf{ and } \$y_1 \textbf{ eq } \$y' \\
\textbf{groupby } [\$b'] \textbf{return} \\
N_3 \left\{ \begin{array}{l}
\textbf{for } \$t \textbf{ in } \$b'/title \\
\textbf{groupby } [\$t] \textbf{ return } \$t
\end{array} \right.
\end{array} \right. \\
\} \langle / \text{result} \rangle
\end{array} \right. \qquad (X4)
$$

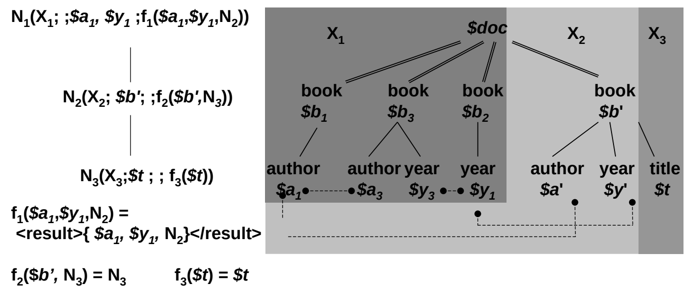


Figure 5: Logical and Functional NEXT corresponding to query (X2)

pair, the nested $\$b'$ loop retrieves the corresponding book elements. This loop is the unavoidable redundant navigation across subqueries. ⋄

Minimization Module Normalization does not solve the minimization problem by itself, as we still have to identify *which* navigations are reusable. The CCC algorithm minimizes the redundant navigation in a given NEXT query and provably finds the minimal equivalent XTableaux of its input NEXT. This requires detecting and eliminating redundant navigation *within* and *across* nested XTableaux.

For example, the NEXT of Figure 8(c) and its corresponding functional NEXT (X3) are the minimized form of XQueries (X2) and (X5). We navigate to books just once and the inner subqueries utilize the navigation of the outer level. Notice that the minimized NEXT of Figure 8(c) has fewer nodes and edges than the original NEXT of Figure 5(b). Indeed it is the minimum possible number of nodes and edges.

Executing NEXT Finally, the minimized NEXT is reduced to a physical plan, similar to the algebraic plans of [14, 15] and is executed. Our logical optimization steps can be easily incorporated in other implementations of XQuery as well by attaching a groupby clause to FLWR, i.e., by having the ability to execute the groupby of the functional NEXT. One can improve performance by removing trivial **groupby** 's, such as those of the inner **for** loops of (X3), and keeping only the essential ones, such as only the outermost **groupby** of (X3).

3 Normalization into NEXT

Figure 6 presents a set of rewrite rules which provably normalize any OptXQuery to a NEXT query (as shown by Theorem 3.1 below). Some of these rules are known simplification rules of XQuery; they are used extensively both in reducing XQuery to its formal core [29] as well as in query optimization [19]. We focus the presentation on the rules that are particular to **groupby**, such as Rules (G1), (G3), (G4) and (G5) and leave out the trivial standard normalization rules. Notice that, for simplicity of presentation, all rules are shown using **for** and **some** expressions that define exactly one variable. The extension to multiple variables is obvious.

The normalization process is stratified in two stages. First, all standard XQuery rewriting rules are applied in any order. Next, the **groupby**-specific rules are used. Rule (RG1) may be applied in both stages. In Appendix A we prove:

Theorem 3.1 *The rewriting of any XQuery Q with the rules in Figure 6 terminates regardless of the order in which rules are applied, i.e. we reach a query T for which no more rewrite rule applies. If Q is an OptXQuery, then T is guaranteed to be a NEXT query.* ⋄

EXAMPLE 3.1 Recall query (X2) from Example 1.2. In the first phase of the normalization of (X2), Rules (R1), (R13), (R14) and (R8) apply, yielding the query (X6).

$$
\begin{array}{l}
\textbf{for } \$a \textbf{ in distinct-values}(\qquad\qquad\qquad\qquad (X6) \\
\quad \textbf{for } \$b_1 \textbf{ in } \$doc//book \textbf{ return for } \$a_1 \textbf{ in } \$b_1/author \\
\quad \textbf{return } \$a_1) \\
\textbf{return for } \$y \textbf{ in distinct-values}(\\
\textbf{for } \$b_2 \textbf{ in } \$doc//book \textbf{ return for } \$y_1 \textbf{ in } \$b_2/year \textbf{ return } \$y_1) \\
\textbf{where some } \$b_3 \textbf{ in } \$doc//book \textbf{ satisfies} \\
\quad \textbf{some } \$a_3 \textbf{ in } \$b_3/author \textbf{ satisfies some } \$y_3 \textbf{ in } \$b_3/year \\
\quad \textbf{satisfies } \$a \textbf{ eq } \$a_3 \textbf{ and } \$y \textbf{ eq } \$y_3 \\
\textbf{return } \langle \text{result} \rangle \{\$a, \$y, \\
\quad \textbf{for } \$b' \textbf{ in } \$doc//book \\
\quad \textbf{where some } \$a' \textbf{ in } \$b'/author \textbf{ satisfies} \\
\quad \textbf{some } \$y' \textbf{ in } \$b'/year \textbf{ satisfies } \$a' \textbf{ eq } \$a \textbf{ and } \$y' \textbf{ eq } \$y \\
\quad \textbf{return for } \$t \textbf{ in } \$b'/title \textbf{ return } \$t\} \\
\quad \langle /\text{result} \rangle
\end{array}
$$

The second phase of the normalization applies **groupby** rewriting rules to (X6). A rewrite step with Rule (G1) applied to the outermost **for** replaces the **distinct-values** function with a **groupby** clause which groups by the value of variable $\$a$. Similarly, Rule (G3) turns the inner **for** expression, which does not involve **distinct-values**, into a **for** expression that involves grouping by identity. By applying Rule (G4) the **some** structures are eliminated. Notice that the variables defined in **some** do not participate in the groupby variable lists. Rule (G5) removes nested subqueries from generator expressions. Rule (G6) substitutes $\$a_1$ for $\$a$ and $\$y_1$ for $\$y$. Rule (G10) collapses **groupby** 's. The transformations reduce the query (X2) to the NEXT (X4). ⋄

$(R1)$ **for** $\$V_1$ **in** $E_1, \ldots, \$V_n$ **in** E_n **where** C **return** E
$\mapsto$ **for** $\$V_1$ **in** E_1 **return for** $\$V_2$ **in** E_2 **return** ... **for** $\$V_n$ **in** E_n **where** C **return** E
$(R2)$ **for** $\$V$ **in** (**for** $\$V_1$ **in** E_1 **return** E_2) **return** $E_3 \mapsto$ **for** $\$V_1$ **in** E_1 **return for** $\$V$ **in** E_2 **return** E_3
$(R3)$ **for** $\$V$ **in** $\langle e\rangle E_1\langle /e\rangle$ **return** $E_2 \mapsto \theta_{\$V\mapsto\langle e\rangle E_1\langle /e\rangle}(E_2)$ (* $\theta_{\$V\mapsto E_1}(E_2)$ substitutes E_1 for $\$V$ in E_2 *)
$(R4)$ **for** $\$V_1$ **in** $\$V_2$ **return** $E \mapsto \theta_{\$V_1\mapsto\$V_2}(E)$ (*if $\$V_2$ is not defined by **let** *)
$(R5)$ **for** $\$V_1$ **in** E_1 **return for** $\$V_2$ **in** $\langle e\rangle E_2\langle /e\rangle$ **where** C **return** $E_3 \mapsto \theta_{\$V_2\mapsto\langle e\rangle E_2\langle /e\rangle}$(**for** $\$V_1$ **in** E_1 **where** C **return** E_3)
$(R6)$ **for** $\$V_1$ **in** E_1 **return for** $\$V_2$ **in** $\$V_3$ **where** C **return** $E_3 \mapsto \theta_{\$V_2\mapsto\$V_3}$(**for** $\$V_1$ **in** E_1 **where** C **return** E_3)
$(R7)$ **for** $\$V$ **in** (E_1, E_2) **return** $E_3 \mapsto$ (**for** $\$V$ **in** E_1 **return** E_3), (**for** $\$V$ **in** E_2 **return** E_3)
$(R8)$ **some** $\$V_1$ **in** $E_1, \ldots, \$V_n$ **in** E_n **satisfies** C
$\mapsto$ **some** $\$V_1$ **in** E_1 **satisfies some** $\$V_2$ **in** E_2 **satisfies** ... **some** $\$V_n$ **in** E_n **satisfies** C
$(R9)$ **some** $\$V$ **in** (**for** $\$V_1$ **in** E_1 **return** E_2) **satisfies** $C \mapsto$ **some** $\$V_1$ **in** E_1 **satisfies some** $\$V$ **in** E_2 **satisfies** C
$(R10)$ **some** $\$V$ **in** $\langle e\rangle E_1\langle /e\rangle$ **satisfies** $C \mapsto \theta_{\$V\mapsto\langle e\rangle E_1\langle /e\rangle}(C)$
$(R11)$ **some** $\$V_1$ **in** $\$V_2$ **satisfies** $C \mapsto \theta_{\$V_1\mapsto\$V_2}(C)$ (* if $\$V_2$ is not defined by **let** *)
$(R12)$ **some** $\$V$ **in distinct-values** (E) **satisfies** $C \mapsto$ **some** $\$V$ **in** E **satisfies** C
$(R13)$ $\$V(/|//)C \mapsto$ **for** $\$V_1$ **in** $\$V(/|//)C$ **return** $\$V_1$ (* if $\$V/C$ does not appear in "$\$X$ **in** $\$V/C$"*)
$(R14)$ $\$V(/|//)C_1 \ldots (/|//)C_n \mapsto$ **for** $\$V_1$ **in** $\$V(/|//)C_1$ **return** ... **for** $\$V_n$ **in** $\$V_{n-1}(/|//)C_n$ **return** $\$V_n$ (* for $n \geq 2$ *)
$(R15)$ **distinct-values** $(\$V|\langle e\rangle E_1\langle /e\rangle|$**distinct-values** $(E)) \mapsto \$V|\langle e\rangle E_1\langle /e\rangle|$**distinct-values** (E) (*if $\$V$ is not defined by **let** *)
$(RG1)$ $\langle e\rangle E_1, \ldots, E_n\langle /e\rangle /c \mapsto \sigma_c(E_1), \ldots, \sigma_c(E_n)$
$\sigma_c(\langle c\rangle E\langle /c\rangle) \mapsto \langle c\rangle E\langle /c\rangle$ $\quad \sigma_c(\langle a\rangle E\langle /a\rangle)\,(*a \neq c*) \mapsto ()$ $\quad \sigma_c(\$V) \mapsto \$V \quad (*if(tagName(\$V) = c)*) \quad () \quad (*else*)$
σ_c(**for** $\$V_1$ **in** E_1 **return** E_2) $\mapsto$ **for** $\$V_1$ **in** E_1 **return** $\sigma_c(E_2)$ $\quad \sigma_c(E(/|//)c) \mapsto E(/|//)c$ $\quad \sigma_c(E(/|//)a) \mapsto ()(*a \neq c*)$
$\sigma_c(E_1, E_2) \mapsto \sigma_c(E_1), \sigma_c(E_2)$ $\quad \sigma_c$(**distinct-values**(E)) $\mapsto$ **distinct-values**$(\sigma_c(E))$

Group-By Rewriting Rules

$(G1)$ **for** V **in distinct-values**(E_1) **return** $E_2 \mapsto$ **for** V **in** E_1 **groupby** V **return** E_2
$(G2)$ **distinct-values**$(E_1) \mapsto$ **for** V **in** E_1 **groupby** V **return** V (*for **distinct-values**(E_1) which does not appear in "$\$X$ **in distinct-values** (E_1)"*)
$(G3)$ **for** V **in** E_1 **return** $E_2 \mapsto$ **for** V **in** E_1 **groupby** $[V]$ **return** E_2
$(G4)$ **for** V_1 **in** E_1**where some** V_2 **in** E_2 **satisfies** C **groupby** G **return** E_3
$\mapsto$ **for** V_1 **in** E_1, V_2 **in** E_2 **where** C **groupby** G **return** E_3
$(G5)$ **for** V_2 **in** (**for** V_1 **in** E_1 **groupby** G_1 **return** E_2) **groupby** V_2 **return** E_3
$\mapsto$ **for** V_1 **in** E_1, V_2 **in** E_2 **groupby** V_2 **return** E_3
$(G6)$ **for** X **in** $(X' \mid \langle c\rangle E\langle /c\rangle)$**groupby** G **return** $E_r \mapsto \theta_{X\mapsto(X' \mid \langle c\rangle E\langle /c\rangle)}(E_r)$
$(G7)$ **for** V **in** E **groupby** G_1 **return for** X **in** $(X' \mid \langle c\rangle E_2\langle /c\rangle)$**where** C **groupby** G_2 **return** E_r
$\mapsto \theta_{X\mapsto(X' \mid \langle c\rangle E_2\langle /c\rangle)}$(**for** V **in** E **where** C **groupby** G_1 **return** E_r)
$(G8)$ **for** V_1 **in** E_1, X **in** $(X' \mid \langle c\rangle E\langle /c\rangle)$ **groupby** G **return** $E_r \mapsto \theta_{X\mapsto(X' \mid \langle c\rangle E\langle /c\rangle)}$(**for** V_1 **in** E_1 **groupby** G**return** E_r)
$(G9)$ **for** V **in** $\langle e\rangle E_1, \ldots, E_n\langle /e\rangle /c$ **groupby** $[V]$ **return** E_r
$\mapsto$ (**for** V **in** $\sigma_c(E_1)$ **groupby** $[V]$ **return** E_r), ..., (**for** V **in** $\sigma_c(E_n)$ **groupby** $[V]$ **return** E_r)
$(G10)$ **for** V_1 **in** $E_1, \ldots, V_n$**in** E_n **groupby** G_1 **return for** V_1' **in** $E_1', \ldots, V_k'$ **in** E_k' **groupby** G_2 **return** E_r
$\mapsto$ **for** V_1 **in** $E_1, \ldots, V_n$ **in** E_n, V_1' **in** $E_1', \ldots, V_k'$ **in** E_k' **groupby** G_1, G_2 **return** E_r (*if G_1 and G_2 only contain grouping by value variables*)
$(G11)$ **groupby** $E \mapsto$ **groupby** $strip(E)$
$strip(\langle\text{tag}\rangle E\langle\text{/tag}\rangle) \mapsto strip(E)$ $\quad strip(E_1, E_2) \mapsto strip(E_1), strip(E_2)$
$strip([E]) \mapsto [strip(E)]$ $\quad strip(\$V, \$V) \mapsto strip(\$V) \mapsto \$V$

Figure 6: Rules for rewriting OptXQuery into NEXT

Example 3.2 illustrates the normalization of (X5), which is the efficient variant of query (X2).

EXAMPLE 3.2 Recall from Section 1 (X5), the expert's choice of writing query (X2). Standard XQuery normalization rules (R1),(R13), (R14), (R8) and (R2) are applied. Then **groupby**-specific rules (G1,G3, G4, G5, G6, G8, G11) and RG1 are applied and the final result is the NEXT query shown below.

for $\$b_1$ **in** \$doc//book, $\$a_1$**in** $\$b_1/author, \$y_1$**in** $\$b_1/year$
groupby $\$a_1, \$y_1$**return**
  ⟨result⟩{$\$a_1, \$y_1,$
  for \$b′ **in** \$doc//book, \$a′ **in** \$b′/author,\$y′ **in** \$b′/year
  where $\$a'$ **eq** $\$a_1$**and** $\$y'$ **eq** $\$y_1$
  groupby [\$b′] **return**
    for \$t **in** \$b′/title **groupby** [\$t] **return** \$t }
  ⟨/result⟩

◇

4 Minimization of NEXT Queries

The minimization algorithm focuses on the Xtableaux, which describe the navigation part of NEXT queries, in order to eliminate redundant navigation. The algorithm we present here does not incorporate knowledge about the semantics of the result functions, treating them as uninterpreted symbols.[3] It is easy to see that under this assumption, two equivalent NEXT queries must have isomorphic group-by trees, where the corresponding (according to the isomorphism) nodes of the two group-by trees have identical (up to variable renaming) groupby lists and result functions. However, this does not constrain the Xtableaux associated with the corresponding group-by nodes in any other way than having to deliver the same set of bindings for their variables.

We say that NEXT query Q is *minimal*, if for any other

[3] Which means that $f_1(x, y)$ is equal to $f_2(u, v)$ iff f_1 and f_2 are the same function symbol and $x = u$ and $y = v$. Exploiting the semantics of the result functions in minimization is a future work direction.

NEXT query Q_o equivalent to Q, and for any group-by node N of Q, the node N_o of Q_o corresponding to N via the isomorphism has at least as many variable nodes in its Xtableau. Clearly, minimality rules out redundant navigation: if NEXT query Q performs redundant navigation, this can be removed, yielding an equivalent query with strictly less navigation steps, hence strictly less variables, so Q is not minimal.

Theorem 4.1 *Any NEXT query with uninterpreted result functions has a unique minimal form (up to variable renaming).*[4] $\diamond$

We present the *Collapse and Check Containment (CCC)* algorithm, which searches for this minimal form and is guaranteed to find it. Note that Theorem 4.1 implies that no other algorithm can further minimize CCC's output without manipulating the result functions. As a matter of fact, we conjecture that in the absence of any schema information, no manipulation of the result function can generate additional minimization opportunities. This conjecture and Theorem 4.1 imply that the CCC algorithm fully minimizes any NEXT query, regardless of its result function.

The CCC algorithm is shown in Figure 7. It minimizes a NEXT query Q by invoking min_query on the empty context and Q. min_query visits the group-by tree of Q in a top-down fashion. Let T be a subtree of Q's groupb-by tree and denote with N the root of T. T may have free variables whose bindings are provided by the *context* C, where C is the list of N's ancestors in Q's group-by tree. min_query(C, T) returns a minimized equivalent of T in context C as follows. First, the Xtableau X of N is minimized in context C by the min_tableau function (described shortly), which returns a minimized Xtableau X^{min} and a variable mapping θ. θ maps eliminated variables of X into retained variables – potentially variables provided by ancestor groupby nodes. This variable mapping is applied to the groupby lists and the arguments of the result function of N, yielding a new group-by tree node N'. The children of N' are set to the result of recursively applying min_query to each child of N under the appropriate context. Finally, the new group-by tree rooted at N' is returned.

Tableau Minimization The tableau minimization algorithm min_tableau is based on two key operations: collapsing variable nodes, and checking that this rewriting preserves equivalence.

The collapse step. Consider two variables x, y in the input tableau X. Assume that x is bound in X, while y may be either bound or free. Then *collapsing x into y* means substituting y for x in X. Notice that after a sequence of collapse steps, we may end up with two /-edges between

[4] Contrast this with the uniqueness problem for nested OQL queries, which is open, as a consequence of the open problem of deciding their equivalence [18]. We have developed a decision procedure for equivalence of NEXT queries with arbitrary nesting depth and uninterpreted result functions. This procedure is not needed in minimization, but its existence is crucial for the proof of minimal form uniqueness. Checking equivalence of NEXT queries is of independent interest for their optimization.

CCC(Q: NEXT query) := min_query(*empty context*, Q)

min_query (*Context*: group-by tree,

$$\begin{array}{ccc} & N(X; G_i; G_v; f) & \\ T_1 & \ldots & T_n \end{array} \quad : \text{group-by tree})$$

returns group-by tree

$(X^{min}, \theta) \leftarrow$ min_tableau$(Context, X, G_i, G_v)$
if *Context* is empty
  NewCtxt $\leftarrow N'(X^{min}; \theta(G_i); \theta(G_v); \theta(f))$
else /* *Context* is of the form $N_1^a(\ldots) - \ldots - N_m^a(\ldots)$ */

$$NewCtxt \leftarrow \begin{array}{c} Context \\ | \\ N'(X^{min}; \theta(G_i); \theta(G_v); \theta(f)) \end{array}$$

return

$$\begin{array}{ccc} & N'(X^{min}; \theta(G_i); \theta(G_v); \theta(f)) & \\ \text{min\_query}(NewCtxt, \theta(T_1)) & \ldots & \text{min\_query}(NewCtxt, \theta(T_n)) \end{array}$$

Figure 7: The CCC Minimization Algorithm

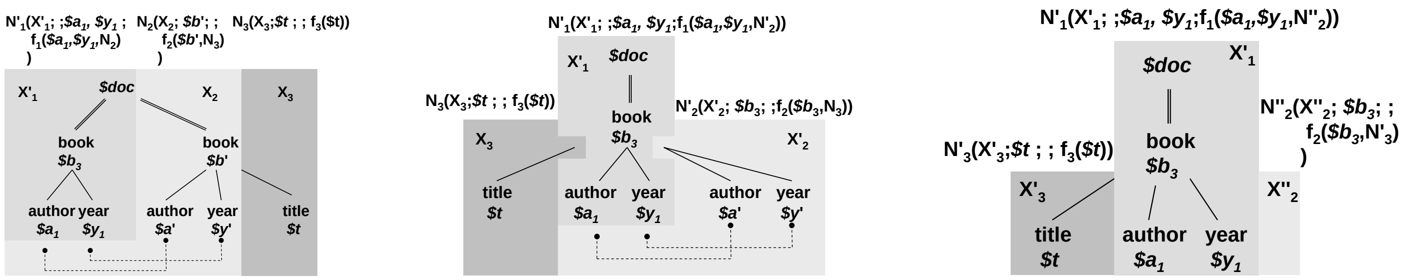
the same pair of variable nodes. In this case, we remove one /-edge. We also remove any //-edge $e = (s, t)$ such that there exists a path from s to t in X which does not include e. Clearly, the removed edges correspond to redundant navigation steps.

EXAMPLE 4.1 We illustrate the minimization of the NEXT of Figure 5. First we apply min_tableau to tableau X_1 of the root N_1 of the groupby tree. Since there is no ancestor context, it collapses only variables bound in X_1: $\$b_1$ into $\$b_3$, $\$b_2$ into $\$b_3$, then $\$y_3$ into $\$y_1$ and finally $\$a_3$ into $\$a_1$, to obtain the minimized groupby node N_1' in Figure 8 (a). Using the algorithm described later, min_tableau verifies that X_1 and X_1' (the Xtableau of N_1') are equivalent. Coincidentally, the variable mapping $\theta_1 = [\$b_1 \mapsto \$b_3, \$b_2 \mapsto \$b_3, \$y_3 \mapsto \$y_1, \$a_3 \mapsto \$a_1]$ does not affect the groupby lists and result function of N_1.

Next, N_2 is minimized under the context of N_1'. Now we can also collapse nodes across Xtableaux: we map $\$b'$ (from N_2) into $\$b_3$ (from N_1') to get the temporary Xtableau X_2' shown in Figure 8 (b). We continue collapsing $\$y'$ into $\$y_1$ and $\$a'$ into $\$a_1$ to obtain the groupby node N_2'' shown in Figure 8 (c). Notice that N_2'' has the empty Xtableau X_2'', which means that it performs no new navigation. Instead, it reuses the navigation in N_1' to get the bindings of $\$b_3$, on whose identity it then groups. It turns out that the above collapse steps are equivalence preserving, i.e., X_2 is equivalent to X_2'' in the context of N_1'.

The minimization of N_3 results in an identical N_3'. The overall effect is that the NEXT query (X4) has been optimized into the NEXT query of Figure 8 (c). $\diamond$

While not needed in the above example, there is one more case in which we try to collapse pairs of variables x, y, namely when they are both free in the Xtableau X. Collapsing them in X means adding the id-based equality x **is** y to X. The reason we consider such collapse steps on



(a) (b) (c)

Figure 8: (a) after minimization of Q_1 (b) after collapsing $\$b', \$b_3$ in Q_2 (c) the minimal form

free variables is subtle. The fact that X has a non-empty set of bindings may say something about the structure of the XML document which in turn may render the bindings of variable x reusable to obtain those of y. However, for documents where X has no bindings, the bindings of x and y may be unrelated. Therefore we need a way to say that x and y have related bindings *provided* X has bindings. The solution is to add the equality x **is** y to X (see Example 4.4).

Equivalence of group-by nodes in a context. After a collapse step of min_tableau has reduced the Xtableau X of a groupby node $N(X; G_i; G_v; f)$ into an Xtableau X' by deriving a mapping θ, it checks the equivalence of $N(X; G_i; G_v; f)$ to $N'(X'; \theta(G_i); \theta(G_v); \theta(f))$ in the context C provided by the ancestors of N. This means verifying that X and X' produce the same sets of bindings for the variables of the groupby lists when the bindings of their free variables are provided by the context C. The function min_tableau reduces the problem to checking containment of nodes without free variables (i.e., to equivalence of nodes in the absence of any context) and then solves the latter.

The reduction proceeds as follows: Let the context C be the list $N_1^a, \ldots, N_m^a$ of N's ancestors. Let $N_{C,N}$ be a new groupby node. Its groupby-id and groupby-value variables are the list of all group-by variables of $N_1^a, \ldots, N_m^a, N$. Its result function is the same as N's. Its Xtableau is obtained by merging the Xtableaux of $N_1^a, \ldots, N_m^a, N$ (put together all nodes and edges). Analogously, define $N_{C,N'}$. Then the following holds:

Proposition 1 *Group-by nodes N and N' are equivalent in context C if and only if the sets of bindings of the groupby variables of $N_{C,N}$ and $N_{C,N'}$ are contained in one another.*

EXAMPLE 4.2 By Proposition 1, the correctness of the collapse step of $\$b'$ into $\$b_3$ in Example 4.1 reduces to the equivalence of groupby nodes $N_{N_1',N_2}(N_1'\#N_2; \$b'; \$a_1, \$y_1; f_2(\$b', N_3))$ and $N_{N_1',N_2'}(N_1'\#N_2'; \$b_3; \$a_1, \$y_1; f_2(\$b_3, N_3))$. Here N_1', N_2, N_2' refer to Figure 8, and $X\#Y$ denotes the Xtableau obtained by merging Xtableaux X and Y. $\diamond$

While the reducibility of equivalence to containment is self-understood for conjunctive queries and tree patterns, it is a pleasant surprise for NEXT queries, as this is not true in general for nested OQL queries [18].[5]

Containment Mappings. Next we show how to check the containment of $N_{C,N}$ in $N_{C,N'}$ and vice versa. We will show in Proposition 2 below that containment is equivalent to finding a *containment mapping*, defined as follows. Let N, N' be two groupby nodes with identical result functions, with associated Xtableaux X, X', groupby-id variable lists G_i, G_i' and groupby-value variable lists G_v, G_v'. We omit the result functions from the discussion since they are identical (modulo variable renaming). A containment mapping from N to N' is a mapping h from the pattern nodes and constants of X to those of X' such that

1. h is the identity on constant values.
2. for any node n in X, n's tag is the same as that of $h(n)$.
3. for any /-edge $n \rightarrow m$ in X, there is a /-edge $u \rightarrow v$ in X' such that the conditions in X' imply the value-based equality of $h(n)$ with u and of $h(m)$ with v (by reflexivity, symmetry, transitivity, and the fact that id-equality implies value-equality). [6]
4. for any //-edge $n \rightarrow m$ in X, there are edges (regardless of their type) $s_1 \rightarrow t_1, \ldots, s_n \rightarrow t_n$ in X', such that the conditions in X' imply the value-based equality of t_i with s_{i+1} (for all $1 \leq i \leq n-1$), of s_1 with $h(n)$, and of t_n with $h(m)$.
5. for each equality condition x **eq** y in X (x, y are variables or constants) $h(x)$ **eq** $h(y)$ is implied by the conditions of X'. Analogously for x **is** y.
6. the value-based equality of vectors $h(G_v)$ and G_v' is implied by the conditions in X'.
7. the id-based equality of vectors $h(G_i)$ and G_i' is implied by the conditions in X'.

[5] [18] does show however that equivalence reduces to containment for nested OQL queries whose output is a VERSO relation [1]. It turns out that there is a close relationship between VERSO relations and NEXT queries: If we neglect the result functions of the groupby nodes and simply output tuples of bindings, the resulting nested relation is a VERSO relation.

[6] Checking that a certain equality is implied by the conditions in X' can be done in PTIME. It simply involves checking the membership of the equality in the reflexive, transitive closure of the equalities in X' (which is PTIME-computable).

The difference between the tree pattern containment mappings from [21] and the ones defined in this work is that the latter were designed to help reasoning about equality conditions, which are not allowed in tree patterns. For example, the intuition behind clauses 3. and 4. is that whenever two XML nodes are equal (by value or id), so are the subtrees T_1, T_2 rooted at them, so any path in T_1 has a correspondent in T_2.

EXAMPLE 4.3 Continuing Example 4.2, the mapping defined as $h = \{\$b_3 \mapsto \$b', \$a_1 \mapsto \$a', \$y_1 \mapsto \$y', \$a' \mapsto \$a', \$y' \mapsto \$y'\}$ is a containment mapping from $N_{N'_1,N'_2}(N'_1\#N'_2; \$b_3; \$a_1, \$y_1; f_2(\$b_3, N_3))$ into $N_{N'_1,N_2}(N'_1\#N_2; \$b'; \$a_1, \$y_1; f_2(\$b', N_3))$. Here the equality $h(\$a_1)$ **eq** $h(\$a')$ becomes $\$a'$ **eq** $\$a'$, which is trivially implied by the reflexivity of equality. ◇

Proposition 2 *$N_{C,N}$ is contained in $N_{C,N'}$ if and only if there is a containment mapping from $N_{C,N'}$ to $N_{C,N}$.*

By Propositions 1 and 2, all the CCC algorithm has to do to check the equivalence of nodes N and N' in context C is to find containment mappings in both directions between $N_{C,N}$ and $N_{C,N'}$. In fact, the nature of the collapse operation guarantees the existence of a containment mapping from $N_{C,N}$ to $N_{C,N'}$. Hence only the opposite mapping must be checked.

We prove the following result:

Theorem 4.2 *Let Q be a NEXT query. Then (a) the CCC algorithm finds the minimal form M, and (b) M is reached regardless of the order of collapse steps.* ◇

Remarks. 1. Note that collapse steps are quite different and more complex than the basic step used in tree pattern minimization, namely simply removing a variable node. This complexity is unavoidable: see Example 4.4 for a non-minimal NEXT query for which, if instead of collapsing nodes we only try removing them, no removal is equivalence preserving and we cannot modify the original query at all. Moreover, for the same query, if we do not collapse variables that are both free in a groupby node, confining ourselves to pairs with at most one free variable, we cannot reach the minimal form, and for two distinct sequences of collapse steps, we obtain two distinct, non-minimal queries.

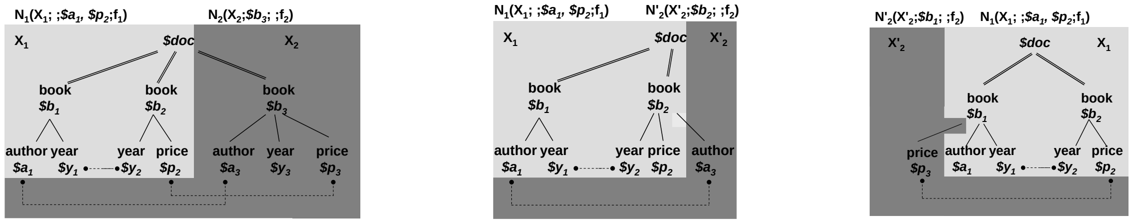
EXAMPLE 4.4 Consider the NEXT query in Figure 9 (a), where N_2 is a child of N_1 in the groupby tree. The navigation in N_2 binding variable $\$b_3$ can reuse from N_1 either the navigation for $\$b_2$ or that for $\$b_1$. We thus have a choice of collapsing $\$b_3$ into $\$b_2$ and then $\$y_3$ into $\$y_2$ and $\$p_3$ into $\$p_2$, obtaining the NEXT in Figure 9(b). Alternatively, we can collapse $\$b_3$ into $\$b_1$ and then $\$a_3$ into $\$a_1$ and $\$y_3$ into $\$y_1$, obtaining the NEXT query in Figure 9(c). In both cases, there are no more equivalence preserving collapse steps that involve at least one free variable, and we get "stuck" with either of the NEXT queries, depending on the initial collapse choice. However, note that we can continue by collapsing $\$b_1$ into $\$b_2$ in both versions of N'_2. Since in both versions these variables are free in N'_2, this means adding the id-based equality $\$b_1$ **is** $\$b_2$ to N'_2. This step in turn enables the collapse of all remaining nodes from N'_2 into nodes from N_1, leading in both cases to the same minimal NEXT query having a node N''_2 with an empty Xtableau. ◇

2. The CCC minimization algorithm applies directly also to queries Q containing $*$-labeled pattern nodes or id-based equality conditions. However, Theorem 4.2 fails in this case, i.e. the algorithm may not fully minimize Q, leaving some residual redundant navigation. But so will any other NP algorithm, unless $\Pi_2^p = NP$, for the following reason. The complexity of checking for the containment mapping is NP-complete in the number of variable nodes in the Xtableau. [10] shows that even for XQueries without nesting, but allowing either navigation to descendants and children of unspecified tag name, or id-based equality checks, equivalence is Π_2^p-complete. It follows that even if $N_{C,G}, N_{C,G'}$ are equivalent, the existence of the containment mapping is not necessary, i.e. the *only if* part of Proposition 2 fails. Consequently, the CCC algorithm might wrongly conclude that the collapse step leading to Q' is not equivalence preserving, and discard it.

From Logical NEXT to Functional NEXT. Notice that the translation of the logical NEXT output by the minimization algorithm into a functional NEXT must deal with a subtlety that minimization may have introduced: the translation of a groupby node N with a free variable $\$r$. Two cases may arise. First, $\$r$ may be among the groupby variables of some ancestor groupby node N^a (e.g. in the NEXT query from Figure 8 (c), $\$b_3$ appears in the groupby list of N''_2, and free in N'_3). Then in the translation of N we simply refer to $\$r$, using it as a free variable. Second, $\$r$ may not be in any groupby variable list (e.g. variable $\$b_3$ is free in N''_2 and not in any groupby list for the query in Figure 8 (c)). Then denote with N^a the groupby node in which $\$r$ is bound (N'_1 for $\$b_3$ in our example). The individual bindings for $\$r$ are collected in the nested relations created by N^a's groupby operation. To access these bindings, we add to the groupby construct in the translation of N^a the clause **into** $\$L$, with $\$L$ a fresh variable binding to the list of bindings of $\$r$. Now in the translation of N we add the loop **for** $\$r$ **in** $\$L/tuple/r$. The query in Figure 8 (c) translates to (X3).

5 Minimization Implementation Issues

The implementation of the minimization module sheds light on the cost of applying minimization and on the benefits of minimization in XQuery processing. The former was not a priori clear, since the CCC algorithm is based on repeatedly finding containment mappings, a step that is NP-complete in the general case. Notice that, in special cases when there are no equality conditions and no wildcard child navigation is allowed, the pattern of a NEXT query degenerates to the simple tree patterns of [3] for which containment is in PTIME.



(a) (b) (c)

Figure 9: Query with two distinct partial minimized forms

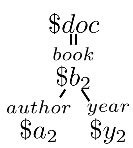


Figure 10: $N_2(X_2; ; \$a_2; f_2)$

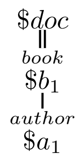


Figure 11: $N_1(X_1; ; \$a_1; f_1)$

We came up with an algorithm that behaves optimally on every input. The algorithm is based on the key observation that finding a containment mapping from groupby node N_1 to groupby node N_2 can be reduced to evaluating a boolean *relational* query obtained from N_1 on a small database computed from N_2. This allows us to exploit standard relational optimization techniques. In particular, the relational query corresponding to a simple tree pattern is *acyclic*. This class of queries can be evaluated in PTIME in the size of both N_1 and N_2 according to Yannakakis' algorithm [12]. We illustrate the reduction on an example.

EXAMPLE 5.1 Consider two NEXT queries Q_1, Q_2 whose groupby trees consist of one node each, N_1 respectively N_2 shown in Figures 11 and 10. We do not specify the result functions f_1, f_2 as they are ignored when checking for containment mappings. We represent N_2 internally as the relational "frozen" database D_{N_2} below, constructed in the spirit of [26]: we create a special constant $\overline{v}$ representing the equivalence class of variable $\$v$ with respect to the value-based equality conditions in N_2.

$\texttt{Child}_{N_2}$	sourceNode	targetNodeTag	targetNode
	$\overline{b_2}$	author	$\overline{a_2}$
	$\overline{b_2}$	year	$\overline{y_2}$

$\texttt{Desc}_{N_2}$	sourceNode	targetNodeTag	targetNode
	doc	book	$\overline{b_2}$

We also add relation RTC_{N_2} containing the reflexive, transitive closure of the union of $\texttt{Child}_{N_2}$ and $\texttt{Desc}_{N_2}$. We translate N_1 to the query

$$M_{N_1}() \leftarrow \texttt{RTC}_{N_2}(\$\texttt{doc}, \texttt{book}, \$b_1), \texttt{Child}_{N_2}(\$b_1, \texttt{author}, \overline{a_2})$$

Clearly, there is a containment mapping from N_1 into N_2 if and only if M_{N_1} returns a non-empty answer on D_{N_2}. $\diamond$

We emphasize that M_{N_1} in the above example is shown for brevity in conjunctive query syntax but it is implemented as an operator tree, in which selections and projections are pushed and joins are implemented as hash joins. Most importantly, the join ordering and pushing of projections are chosen according to Yannakakis' algorithm applied to the acyclic conjunctive query obtained if we ignore equality conditions in N_1 [12]. This approach results in a running time of $O(|N_2|^2 \times |N_1|)$ if there are no equality conditions in N_1 (where $|N|$ denotes the number of pattern nodes in the Xtableau of N)[7]. Moreover, it performs very well in practice in the general case. Our experimental evaluation shows that queries with up to 15 nesting levels and 271 path expressions are minimized in less than 100ms. Our experimental evaluation shows that such added optimization cost is clearly less than the benefit we obtain in query execution.

Note that in the CCC algorithm, the roles of N_1, N_2 are played by the queries $N_{C,N}$, respectively $N_{C,N'}$ from Proposition 1 , which change at every iteration, so D_{N_2} and M_{N_1} must be repeatedly recomputed. The most expensive operations are those of recomputing the equivalence classes of variables, and the transitive closure RTC_{N_2}. Fortunately, this does not have to be done from scratch if we recall that at every iteration, the Xtableau is changed by a simple collapse operation. We chose the following data structures which are easy to incrementally maintain with respect to collapse operations. For every Xtableau, we keep the equivalence classes of variables in a union-find data structure, so whenever node n is collapsed into m, we simply union the class of n with that of m in constant time. RTC_N is represented as an adjacency matrix in which $\mathsf{RTC}_N[x][y] = 1$ if and only if y is a descendant of x in the tree pattern of N. When n is collapsed into m, we set $\mathsf{RTC}_N[n][m] = \mathsf{RTC}_N[m][n] = 1$ and recompute the transitive closure by multiplying RTC_N with itself until we reach a fixpoint (guaranteed to occur in at most $\log |N|$ iterations, but much earlier in practice because of the small incremental change).

6 Conclusions and Future Work

We described the NEXT generalization of tree patterns, which enables logical optimization of XQuery and demonstrated its value by developing an effective technique for

[7] We make the standard assumption of $O(1)$ for indexing into the hash table when joining. Otherwise, an additional $\log |N_2|$ factor must be counted for sort-merge join.

minimization of nested XQueries, which removes redundancy across and within subqueries. A key ingredient of NEXT is the **groupby** operation, which reduces mixed (bag and set) semantics to pure set semantics that provides the typical framework for logical optimization such as minimization. Furthermore, it enables consolidation of all navigation in the XTableaux. The provided rewriting rules reduce any query from the OptXQuery subset of XQuery into a NEXT.

The minimization algorithm also capitalizes on the **groupby** of NEXT, which allows the navigation performed on a nesting level to reuse the navigation performed on higher levels. In addition, our minimization algorithm went fundamentally beyond prior minimization algorithms for tree patterns and conjunctive queries by introducing a new type of minimization step, called *collapsing*. The collapse step *adds* to a subquery identity-based equality conditions between its variables to state that their bindings are the same. Prior algorithms only *remove* variables [3, 23]. The removal step alone turns out to be insufficient for nested XQueries, as removal-based techniques not only fail to find a minimal form, but depending on the application order, they yield several distinct queries, each non-minimal. Indeed, we prove the existence of a unique minimal form for any NEXT query and show that our algorithm is guaranteed to find it regardless of the order in which it applies collapse steps (Theorem 4.2).

Minimization of queries from our XQuery subset is NP-complete, which is no surprise since even in the absence of XQuery's nesting, arbitrary (cyclic) joins, which one can write using XPath predicates, increase the complexity of minimizing XPath expressions described by tree patterns from PTIME [3, 23] to NP-hard [10]. Our minimization algorithm behaves optimally on every input: it runs in PTIME if the tree patterns have no cyclic joins and in NP in the presence of cyclic joins. As shown by our experimental evaluation, even in the NP-complete case optimization time is low (below 100ms for queries with up to 15 nesting levels and up to 271 path expressions, as explained in Appendix B) thanks to a careful implementation which reduces the exponential to an approximation of the tree width of the query [12] (small in practice), as opposed to the number of navigation steps (may by very large in practice). We incorporated minimization in our NEXT XQuery processor and provided experimental data points that prove the beneficial effect of minimization on the total execution time. Due to space constraints, the experimental evaluation is reported in the full paper, and included in Appendix 5 for the reviewer's convenience.

NEXT normalization and minimization can be used in any XQuery processor, regardless of its underlying execution model, as long as it supports an OQL-style **groupby** operator.

An extension of NEXT, called NEXT+, allows the normalization of arbitrary XQueries, which may be outside the OptXQuery set, into NEXT+ queries. Guaranteeing full minimization for NEXT+ is either impossible (e.g., it is straightforward to show that no algorithm can guarantee the full minimization of XQueries involving negation) or requires various extensions to NEXT and the minimization algorithm (e.g., extra minimization can be achieved by algorithms that understand the semantics of aggregation functions.) Nevertheless, the minimization algorithm can be applied to the NEXT subexpressions of NEXT+ queries and guarantee their full minimization (which, as said, does not imply the full minimization of the NEXT+ query). Space constraints relegate this discussion to Appendix D.

Looking beyond minimization, we plan to employ the NEXT notation to address , in the context of our mediator efforts (which include the Local-As-View approach), an answering-queries-using-views algorithm for XQuery.

7 Related Work

There is an extensive body of work on nested query optimization, for relational (SQL) and object-oriented (OQL [4]) queries. See [6], respectively [8] and the references within. For both OQL and SQL, the main effort is that of unnesting nested queries (merging query blocks), not their minimization. The group-by operation is crucially exploited to this end, by evaluating a nested query using an outerjoin followed by a group-by operation. See [16, 13] for the relational query evaluation, [8] for the object-oriented case, and [20, 24] for XML query evaluation. Such rewrites have only limited applicability when bag and set semantics are mixed [22] or the nesting occurs in the `select` clause. Our techniques succeed in these situations. One of our rewrite rules introduces group-by operations with every **for** loop, exploiting the well-known fact that the **distinct-values** operation is a special case of group-by [6]. Another common fact we exploit was recognized in [22], namely that quantifiers are not affected by duplicates. There is an interesting duality between our technique and the generalization of predicate pushdown [26] to nested (SQL) queries in [17]. The latter pushes conditions from the **where** clause of a query into its nested subqueries. Our technique pulls **for** loops up from nested queries. Existing algorithms for the minimization of tree patterns consider no nesting, no arbitrary joins, and only set semantics [3, 23]. Group-by detection is particularly important in XQuery, where surface syntax does not include a group-by construct. [24] uses algebraic rewriting for nested queries that perform grouping. Our algorithm solves this problem as a special case of minimization. [7] is the first work that introduces *Generalized Tree Patterns (GTPs)* that model nested queries and reduce the problem of evaluating a nested query into one of finding matches for its GTP. In addition, [7] shows a translation of GTPs to a physical plan algebra, which we have adopted, with minor modifications. There is an interesting correspondence as well as subtle differences between GTPs and NEXTs and the corresponding modules, stemming from NEXT's orientation towards problems such as minimization and answering queries using views. First, we make a distinction between optXQuery/NEXT and full XQuery/NEXT+. OptXQuery scopes the area where minimization (and, we conjecture, answering queries using views) is guaranteed to find optimal plans. OptXQuery/NEXT omits XQuery features that make minimization undecidable (e.g., negation and universal quantification) or too complex (e.g., aggregate functions). Such features are allowed in NEXT+, where we do not guarantee optimality of the resulting plan. Finally, note we have introduced a distinction between grouping-by-id and grouping-by-value since we find multiple aggregation examples in mediation. (A similar extension for [7] is possible.)

[25] addresses minimization of nested XQueries in the context of Peer-to-Peer systems, where scalability is an acute problem. They develop a PTIME algorithm, trading completeness of minimization for scalability. The algorithm is incomparable to ours: on one hand, it changes the structure of the group-by tree, which we do not do, as we treat result functions as uninterpreted. On the other hand, it only minimizes the nested subqueries in the context of their ancestor subqueries, but it does not attempt to reuse the navigation of the ancestors. No grouping is used, and the only step considered is removal of variables, which leaves even the simple XQuery from Example 1.1 unchanged. The key to our technique's success is precisely the sophisticated collapse step which goes beyond node removal, as well as the essential use of grouping.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] A. V. Aho, Y. Sagiv, and J. D. Ullman. Efficient optimization of a class of relational expressions (abstract). In *SIGMOD*, 1978.

[3] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *SIGMOD*, 2001.

[4] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, San Mateo, California, 1996.

[5] Ashok Chandra and Philip Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[6] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, 1998.

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

[8] S. Cluet and G. Moerkotte. Nested queries in object bases. In *DBPL*, 1993.

[9] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT Framework for Logical Query Optimization (Extended Version). In *http://www.db.ucsd.edu/People/alin/papers/vldb-2004-full.ps*.

[10] A. Deutsch and V. Tannen. Containment and integrity constraints for xpath fragments. In *KRDB*, 2001.

[11] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of XPath queries. In *VLDB*, 2003.

[12] J. Flum, M. Frick, and M. Grohe. Query evaluation via tree-decompositions. In Jan Van den Bussche and Victor Vianu, editors, *ICDT*, 2001.

[13] R. A. Ganski and H. K. T. Wong. Optimization of nested SQL queries revisited. In *SIGMOD*, 1987.

[14] H.V.Jagadish, S.Al-Khalifa, A.Chapman, L.V.S.Lakshmanan, A.Nierman, S.Paparizos, J.Patel, D.Srivastava, N.Wiwatwattana, Y.Wu, and C.Yu. Timber:a native xml database. *VLDB Journal*, 11(4), 2002.

[15] H. V. Jagadish, Laks V. S. Lakshmanan, D. Srivastava, and k.Thompson. Tax: A tree algebra for XML. In *DBPL*, 2001.

[16] W. Kim. On optimizing an sql-like nested query. *TODS*, 7(3):443–469, 1982.

[17] A. Y. Levy, I. S. Mumick, and Y. Sagiv. Query optimization by predicate move-around. In *VLDB*, 1994.

[18] A. Y. Levy and D. Suciu. Deciding containment for queries with complex objects. In *PODS*, 1997.

[19] I. Manolescu, D. Florescu, and D. Kossman. Answering XML Queries on Heterogeneous Data Sources. In *VLDB*, 2001.

[20] M.Carey, J. Kiernan, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Middleware For Publishing Object-Relational Data as XML Documents. In *VLDB*, 2000.

[21] G. Miklau and D. Suciu. Containment and equivalence for an xpath fragment. In *PODS*, 2002.

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

[23] P. Ramanan. Efficient algorithms for minimizing tree pattern queries. In *SIGMOD*, 2002.

[24] S.Paparizos, S. Al-Khalifa, H.V. Jagadish, L. Lakshmanan, A. Nierman, D.Srivastava, and Y. Wu. Grouping in XML. In *EDBT Workshop on XML Data Management (XMLDM'02)*, 2002.

[25] I. Tatarinov and A. Y. Halevy. Efficient query reformulation in peer-data management systems. In *SIGMOD*, 2004.

[26] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

[27] W3C. XML Query Use Cases . W3C Working Draft 15 November 2002. Available from `http://www.w3.org/TR/xmlquery-use-cases/`.

[28] W3C. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft 12 November 2003. Available from `http://www.w3.org/TR/xpath-functions`.

[29] W3C. XQuery 1.0 Formal Semantics. W3C Working Draft 07 June 2001. Available from `http://www.w3.org/TR/query-semantics/`.

[30] W3C. XQuery: A Query Language for XML. W3C Working Draft 12 November 2003. Available from `http://www.w3.org/TR/xquery`.

[31] Y.Papakonstantinou, M. Petropoulos, and V.Vassalos. QURSED: querying and reporting semistructured data. In *SIGMOD*, 2002.

A Details on Normalization into NEXT

OptXQuery We define the *OptXQuery* subset of XQuery [30] as the class of queries which are accepted by the grammar in Figure 3 and in addition satisfy the semantic constraints presented shortly after we provide some notations and the intuitions for the semantic constraints.

A query Q is not a NEXT query if Q has one of the following expressions which in general case cannot be rewritten into NEXT form: "**some** $\$V$ **in** (E_1, E_2) **satisfies** C", "**distinct-values**(E_1, E_2)", "$\langle e\rangle E\langle /e\rangle //c$", "$\langle e_1\rangle E_1\langle /e_1\rangle$ **eq** $(\$V|\langle e_2\rangle E_2\langle /e_2\rangle|Constant)$". Although the last two types of expressions are not allowed in the grammar shown in Figure 3, both types of expressions could appear after application of some rewriting rules in Figure 6 (R3 and R5).

A.1 Notations

We first define the notations used in semantic constraints. The *definition* $def(\$V)$ of a variable $V in "$\$V$ **in** E" is defined as $def(\$V) = f(E)$ where $f(E) = E$ if E is $\langle c\rangle XQ\langle /c\rangle$, or a path expression; $f(E) = f(E_1)$ if E is **distinct-values**(E_1); $f(E_1, E_2) = f(E_1), f(E_2)$; $f(E) = f(E_r)$ if E is a FLWR expression and E_r is the return clause of E. Note the recursive definition in the case of FLWR expressions.

A variable $\$V$ *directly depends* on $\$V'$ if $\$V'$ appears in $def(\$V)$. We say that $\$V$ *depends* on $\$V'$ if it directly or indirectly (via other variables) depends on $\$V'$. A variable $V is called a *simple variable* if the definition of $\$V$ and the definition of any variable that $\$V$ depends on contain no element constructor nor concatenation. An element constructor $\langle e\rangle E\langle /e\rangle$ is called a *simple element constructor* if the element $\langle e\rangle$ is created by the query via repeated application of element constructors to constants, simple variables and simple element constructors. Note the recursive definition and the fact that complex expressions such as path expressions or **for** loops are disallowed.

A variable $V is called a *tuple variable* if $def(\$V)$ is a simple element constructor. A variable $V whose definition is "$\$X/c_1/.../c_n$" is a tuple variable if all of the following conditions are met: $X is a tuple variable; every navigation step in the path expression is "/"; the evaluation result of $def(\$X)/c_1/.../c_n$ (Rule RG1) is a simple element constructor; and every intermediate result of evaluating $def(\$X)/c_1/.../c_n$ (e.g, $def(\$X)/c_1$, $def(\$X)/c_1/c_2$, ..., $def(\$X)/c_1/.../c_{n-1}$) is a simple element constructor. All other variables are not tuple variables.

A variable $V whose definition is "$\$X/c_1/.../c_n$" is called a *input simple variable* if all of the following conditions are met: $X is a tuple variable; every navigation step in the path expression is "/"; the evaluation result of $def(\$X)/c_1/.../c_n$ (Rule RG1) is a simple variable; and every intermediate result of evaluating $def(\$X)/c_1/.../c_n$ (e.g, $def(\$X)/c_1$, $def(\$X)/c_1/c_2$, ..., $def(\$X)/c_1/.../c_{n-1}$) is a simple variable or a simple element constructor. Obviously a tuple variable is not an input simple variable, but the variables depending on it may be input simple variables. Notice that an input simple variable binds to single elements from the input just like a simple variable. Simple variables and input simple variables together are called *input element variables*. All other variables are not input element variables.

In Query X2, all variables are simple variables. In Query X5, $\$b_1$, $\$a_1$, $\$y_1$, $\$b'$, $\$a'$, $\$y'$ are simple variables; $\$p$ is a tuple variable; $a and $y are input simple variables, but not simple variables since they depend on $p, a tuple variable, whose definition contains element constructors.

A.2 Semantic Constraints for OptXQuery

The semantic constraints for OptXQuery are:

- for each $V defined by "$\$V$ **in** E" in a **some** clause, $V and every variable $X in E and every variable that $X depends on are simple variables.
- For each **distinct-values**(E), if we define $V as "$\$V$ **in distinct-values**(E)", $V must be an input element variable or tuple variable; for each path expression "$\$V(/|//)c_1 \dots (/|//)c_n$" where $V depends on a variable $X defined in "$\$X$ **in distinct-values**(E), if we define $\$V_i$ as "$\$V_i$ **in** $\$V(/|//)c_1 \dots (/|//)c_i$", $\$V_i$ must be a tuple variable or an input element variable ($i = 1, \dots, n$).
- $V is an input element variable if $V starts a path expression containing "//".
- $V is an input element variable if $V appears in an equality condition.
- No "$\$V$ **in** $\$V'$" is allowed; at least one variable $\$V_i$ in "**for** $\$V_1$ **in** $E_1, \dots, \$V_n$ **in** E_n **where** C **return** E" must be a simple variable.

The first four constraints guarantee that none of the four types of non-NEXT expressions mentioned in the beginning of this section (respectively) would appear during rewriting. The last constraint's purpose is to avoid the need of introducing **if** construct into the grammar of NEXT as the following rule (Rule Rif) introduces **if** which is absent in OptXQueries: **for** $\$V_1$ **in** E **where** C **return** E_r $\mapsto$ $\theta_{\$V_1 \mapsto E}$(**if** C **then** (E_r) **else** $())$ where $E = \$V_2|\langle e\rangle E_1\langle /e\rangle$, and $\theta_{\$V \mapsto E_1}(E_2)$ substitutes E_1 for $\$V$ in E_2. However if the FLWR expression is immediately nested in another FLWR expression, the **if** construct is not needed because we can move up the condition as shown in Rules R5, R6 and G7. It is not difficult to see why the last constraint prevent the complete removal of the **for** loop of any FLWR expression. Consider any FLWR expression and assume $\$V$ is a simple variable defined in the **for** loop. If $\$V$ is defined by a path expression, clearly the **for** loop can not be completely removed because Rule Rif is not applicable. In the only other possible case where $\$V$ is defined by a FLWR expression, Rule R2 applies and leads to a longer chain of nested FLWR expressions. In Rule R2, either $\$V_1$ is a simple variable (then E_1 is either a path expression or a FLWR expression) , or E_2 is a FLWR expression. Rule R2 cannot apply infinitely as we will prove shortly. Because of the last constraint, finally there is a simple variable defined by a path expression which makes Rule Rif not applicable.

A.3 Normalization Rewriting Rules

Figure 6 presents a set of rewrite rules which provably normalize any OptXQuery to a NEXT query. Some of these rules are known simplification rules of XQuery; they are used extensively both in reducing XQuery to its formal core [29] as well as in query optimization [19]. We focus the presentation on the rules that are particular to **groupby**, such as Rules (G1), (G3), (G4) and (G5) and leave out the trivial standard normalization rules. Notice that, **where** clauses can be trivially added to FLWR expressions without **where** clauses in all rules except Rules R3,R4, and G7. The introduction of **where** clause to the three rules requires the introduction of **if** construct and would lead to three new rules. However because of the last semantic constraint, these three new rules are not needed for rewriting OptXQueries. The omission of **where** clauses is for the simplicity of presentation. For the same presentation reason, Rules G4, G5, G7 are shown using **for** loop that define exactly one variable. The extension to multiple variables is obvious.

The normalization process is stratified in two stages. First, all standard XQuery rewriting rules are applied in any order. Next, the **groupby**-specific rules are used. Rule (RG1) may be applied in both stages. We repeat here the following result (Theorem 3.1):

Theorem A.1 *The rewriting of any XQuery Q with the rules in Figure 6 terminates regardless of the order in which rules are applied, i.e. we reach a query T for which no more rewrite rule applies. If Q is an OptXQuery, then T is guaranteed to be a NEXT query.* $\diamond$

First, we prove that the rewriting terminates, then prove that OptXQueries are rewritten to NEXT queries.

Consider the first rewriting stage. We associate τ, $\langle sp, mp, var, elm, con, vp, some, for, dis, dm\rangle$, with each query Q, with sp being the most significant part of τ. Intuitively each component of τ indicates the degree the query violates the NEXT form in some aspect. Each rule decreases the value of τ and it is obvious to see each component of τ cannot be less than 0. Thus we prove rewriting in the first stage always terminates.

Several rules are worth to notice when defining τ. Rule 3 substitutes \$V with $\langle e\rangle E_1\langle/e\rangle$ in E_2. E_1 may be far from NEXT normal form, yet E_2 containing multiple occurrences of \$V may be in NEXT form. After substitution, τ may increase if τ is not properly designed. R7 duplicates one subquery (E_3). Rules R13 and R14, unlike all other rules, introduce new variables into the query.

For simplicity of presenting τ, we require input queries do not have a variable defined more than once, which can be achieved by variable renaming. The components of τ of a query Q are defined as:

- sp, the number of distinct "$\$V(/|//)c$" occurrences not in "$\$X$ **in** $\$V(/|//)c$" in Q.
- mp, the number of distinct "$\$V(/|//)c_1 \ldots /|//c_n$" occurrences in Q.
- $var = \sum_{distinct\ variable\ \$V\ in\ Q} var(\$V)$. If a variable \$V appears multiple times in Q, $var(\$V)$ is added to var only once. For $\$V$ in "$\$V$ **in** E", $var(\$V)$ = the number of distinct variables $\$V'$ defined in "$\$V'$ **in** E'" which is (anywhere) in E + $\sum_{distinct\ variable\ \$X in\ E} vars(\$X)$.
- elm, the number of distinct *direct element variables*. $\$V$ defined in "$\$V$ **in** E" is a direct element variable if $E = \langle e\rangle E_1\langle/e\rangle$ or if E is a concatenation expression and one of its concatenation component is an element constructor $\langle e\rangle E_1\langle/e\rangle$. $\$V$ defined in "$\$V$ **in** E" is a direct element variable if $E = \$V'$ and $\$V'$ is a direct element variable or if E is a concatenation expression and one of its concatenation component is $\$V'$ and $\$V'$ is a direct element variable.
- $con = \sum_{distinct\ variable\ \$V\ in\ Q} con(\$V)$. If a variable \$V appears multiple times in Q, the value $con(\$V)$ is added to con only once. For $\$V$ defined in "$\$V$ **in** E" , $con(\$V)$ = the number of concatenation anywhere in E.
- vp, the number of "$\$V$ **in** E" where E is not format of "$\$X(/|//)c$" anywhere in Q.
- $some$, the number of **some** clauses that define more than one variable anywhere in Q;
- for, the number of FLWR expressions that define more than one variable in the **for** clause anywhere in Q;
- dis, the number of occurrences of the **distinct-values** function anywhere in Q.
- dm is the number of "$\langle e\rangle E\langle/e\rangle/c$" occurrences in Q.

As examples, $\tau(X2) = \langle 1,2,0,0,0,2,2,1,2,0\rangle$, $\tau(X6) = \langle 0,0,4,0,0,2,0,0,2,0\rangle$, $\tau(X5) = \langle 1,2,9,0,3,3,1,1,1,0\rangle$, $\tau(X8)$= $\langle 0,0,15,0,1,1,0,0,1,0\rangle$.

The following table shows how each rule may change τ where $\uparrow$ means increase, $\downarrow$ decrease, - no change.

	sp	mp	var	elm	con	vp	some	for	dis	dm
R1	-	-	-	-	-	-	-	↓	-	-
R2	- ↓	- ↓	↓	-↑	- ↓	- ↓	-	-	-	-
R3	- ↓	- ↓	- ↓	↓	- ↓ ↑	- ↓↑	- ↓ ↑	- ↓ ↑	- ↓ ↑	- ↓↑
R4	-	-	- ↓	- ↓	-	↓	-	-	-	-
R5	- ↓	- ↓	- ↓	↓	- ↓ ↑	- ↓↑	- ↓ ↑	- ↓ ↑	- ↓ ↑	- ↓↑
R6	-	-	- ↓	- ↓	-	↓	-	-	-	-
R7	- ↓	-	-	-	↓	- ↑	- ↑	- ↑	- ↑	-↑
R8	-	-	-	-	-	-	↓	-	-	-
R9	-↓	-	↓	-↑	-↓	-↓	-	-	-	-
R10	-↓	-↓	- ↓	↓	- ↓ ↑	-↓ ↑	-↓ ↑	-↓ ↑	-↓ ↑	-↓ ↑
R11	-	-	- ↓	- ↓	-	↓	-	-	-	-
R12	-	-	-	-	-	-	-	-	↓	-
R13	↓	-	-↑	-	-	-	-	-	-	-
R14	-	↓	-↑	-	-	-	-	-	-	-
R15	-	-	-	-	-	-	-	-	↓	-
RG 1	-↓	-↓	-↓	- ↓	-↓	-↓	-↓	-↓	- ↓	↓

Similarly, we can prove rewriting in the second stage terminates and in fact the definition of τ is much simpler.

For an OptXQuery Q we prove that the rewriting ends up with a NEXT query. The rewriting in the first stage turns an OptXQuery Q into XNF form in Figure 12. Syntactically, there are three differences between OptXQueries and XNF queries. First, **for** or **some** loops in OptXQueries may define more than one variable. Second, path expressions in OptXQueries may be more than one step or appear outside of variable definitions. Third, variables in a XNF query can only be defined by single step path expressions in **some** clauses or in addition by the **distinct-values** function if variables are defined in **for** loops.

Assume the first stage rewriting stops and turns an OptXQuery Q into Q', then Q' must be in XNF form, which we prove by contradiction. If the first type of violation of XNF form exists, Rule 1 or Rule 8 is applicable. If the second type of violation of XNF form exists, R13 or Rule 14 is applicable. If the third type of violation of XNF form exists, consider a variable defined in "**for** $\$V$ **in** E" loop. If E is an element constructor, a FLWR expression, a path expression of more than one step, a concatenation expression, $\langle e\rangle E_1\langle/e\rangle/c$, or a variable, Rules R3, R2, R14, R7, RG1, R4 and R6 are applicable respectively, which contradicts the assumption that the rewriting has terminated. Notice that $\langle e\rangle E_1\langle/e\rangle/c$ is not allowed to define variables in OptXQueries but may appear in the definition of a variable after application of Rule R3, and it is the only type of expression that may be introduced by rewriting as the definition of a variable because of Rule R3 and R5. Notice that $\langle e\rangle E_1\langle/e\rangle//c$ cannot appear during rewriting because of the third semantic constraint. Similarly we can show that E defined in "**some** $\$V$ **in** E" can only be a single step path expression and cannot be the **distinct-values** function because of Rule R12. Since variables in equality conditions are required to be input element variables, equality conditions would not be affected (by Rule R3 and R5) and are still format of "$\$V$ **eq** $\$V'|c''$ after rewriting.

The second rewriting stage turns a query in XNF form resulting from the first stage into a NEXT query. Syntactically, there are three differences between XNF form and NEXT form. First, a query Q in XNF form may have **distinct-values** functions, which Rules G1 and G2 eliminate. Second, every FlWR expression in Q is added with the **groupby** clause. Third, Q may have **some** clauses which Rule G4 eliminates. Rules G1 and G2 introduce FLWR expressions to variable definitions which violates the NEXT form. When the rewriting terminates, each E

$$
\begin{array}{rcl}
XQ & ::= & \langle n\rangle\{XQ_1,\ldots,XQ_m\}\langle/n\rangle|FLWR|Constant \\
 & | & XQ_1, XQ_2|\textbf{distinct-values}\ (FLWR) \\
FLWR & ::= & \textbf{for}\ V\ \textbf{in}\ SP\ (\textbf{where}\ CList)?\textbf{return}\ XQ \\
SP & ::= & Path \mid \textbf{distinct-values}\ (FLWR) \\
Path & ::= & (\textbf{document}\ (\text{“}constant''\text{)}|Var)(/|//)Constant \\
CList & ::= & Cond\ (\textbf{and}\ Cond)* \\
Cond & ::= & Var_1\ \textbf{eq}\ (Var_2|Constant) \\
 & | & \textbf{some}\ V\ \textbf{in}\ Path\ \textbf{satisfies}\ CList
\end{array}
$$

Figure 12: XQuery Normal Form

in "$\$V$ **in** E" can only be a single step path expression, which again can be proven by contradiction. If E is a FLWGR expression, an element constructor, a variable, or $\langle e\rangle E_1\langle/e\rangle/c$, Rules G5, G6, G7, G8, G9 are applicable respectively, which contradicts the assumption that the rewriting has terminated. Unlike the proof in the first stage, E cannot be a path expression of more than one step which is not in XNF, and none of the rewriting rule introduces it. E cannot be a concatenation expression which is not in XNF and none of the rewriting rule introduces it because of the semantic constraints of OptXQueries. Note the element constructors in G6, G7 and G8 must be simple element constructors because of the second semantic constraint. In G9, the result of $\langle e\rangle E_1,\ldots,E_n\langle/e\rangle/c$ is either a simple element constructor or a variable because of again the second semantic constraint. Only Rules G6, G7 and G8 may make a **groupby** clause contain an expression other than variables and the expression can only be a simple element constructor. However G11 strips any simple element constructor and makes **groupby** clause contain only variables.

$$
N_1 \left\{
\begin{array}{l}
\textbf{for}\ \$b_1\ \textbf{in}\ \$doc//book, \$a_1\ \textbf{in}\ \$b_1/author, \\
\quad \$b_2\ \textbf{in}\ \$doc//book, \$y_1\ \textbf{in}\ \$b_2/year, \\
\quad \$b_3\ \textbf{in}\ \$doc//book, \$a_3\ \textbf{in}\ \$b_3/author, \$y_3\ \textbf{in}\ \$b_3/year \\
\textbf{where}\ \$a_1\ \textbf{eq}\ \$a_3\ \textbf{and}\ \$y_1\ \textbf{eq}\ \$y_3 \\
\textbf{groupby}\ \$a_1, \$y_1\ \textbf{return} \\
\langle\text{result}\rangle\{\$a_1, \$y_1, \\
\quad N_2 \left\{
\begin{array}{l}
\textbf{for}\ \$b'\ \textbf{in}\ \$doc//book, \$a'\ \textbf{in}\ \$b'/author, \\
\quad \$y'\ \textbf{in}\ \$b'/year \\
\textbf{where}\ \$a_1\ \textbf{eq}\ \$a'\ \textbf{and}\ \$y_1\ \textbf{eq}\ \$y' \\
\textbf{groupby}\ [\$b']\textbf{return} \\
N_3 \left\{
\begin{array}{l}
\textbf{for}\ \$t\ \textbf{in}\ \$b'/title \\
\textbf{groupby}\ [\$t]\ \textbf{return}\ \$t
\end{array}
\right.
\end{array}
\right. \\
\}\langle/\text{result}\rangle
\end{array}
\right. \qquad (X4)
$$

Example A.1 illustrates the normalization of an efficient variant of query (X2).

EXAMPLE A.1 While apparently more complicated than the query (X2), query (X5) is what an XQuery expert would write, since it results in a more efficient execution plan, that avoids redundant navigation within the same subquery. In fact this is the most efficient way to perform grouping by multiple variables in XQuery.

Standard XQuery normalization rules (R1),(R13), (R14), (R8) (R2) apply, yielding the query X8 .

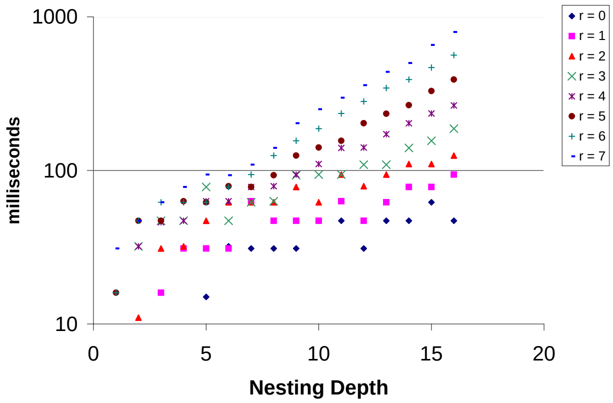


Figure 13: Minimization times as function of nesting depth d and redundancy r

for $\$p$ **in distinct-values**((X8)
  for $\$b_1$ **in** $\$doc//book$ **return**
  for $\$a_1$ **in** $\$b_1/author$ **return**
  for $\$y_1$ **in** $\$b_1/year$ **return** ⟨pair⟩⟨a⟩$\{\$a_1\}$⟨/a⟩⟨y⟩$\{\$y_1\}$⟨/y⟩⟨/pair⟩)
return
for $\$iv_1$ **in** $\$p/a$ **return** **for** $\$iv_2$ **in** $\$iv_1/author$ **return**
for $\$iv_3$ **in** $\$p/y$ **return** **for** $\$iv_4$ **in** $\$iv_3/year$ **return**
return ⟨result⟩ $\{\$iv_2, \$iv_4,$
  for $\$b'$ **in** $\$doc//book$
  where some $\$a'$ **in** $\$b'/author$ **satisfies some** $\$y'$ **in** $\$b'/year$
  satisfies $\$a'$ **eq** $\$iv_2$ **and** $\$y'$ **eq** $\$iv_4$
  return for $\$t$ **in** $\$b'/title$ **return** $\$t$ }
⟨/result⟩

Then **groupby**-specific rules (G1,G3, G4, G5, G6, G8, G11) and RG1 are applied and the final result is the NEXT query shown below.

for $\$b_1$ **in** \$doc//book, $\$a_1$**in** $\$b_1/author$, $\$y_1$**in** $\$b_1/year$
groupby $\$a_1, \$y_1$**return**
  ⟨result⟩$\{\$a_1\}\{\$y_1\}$
  { **for** \$b′ **in** \$doc//book, \$a′ **in** \$b′/author,\$y′ **in** \$b′/year
  where $\$a'$ **eq** $\$a_1$**and** $\$y'$ **eq** $\$y_1$
  groupby [\$b′] **return**
    for \$t **in** \$b′/title **groupby** [\$t] **return** \$t }
  ⟨/result⟩

◇

B Experiments

B.1 Minimization Time

We ran the following experiment to stress-test the CCC algorithm. We considered a family of synthetic NEXT queries $Q_{d,r}$, where the parameter d controls the nesting depth of the query, and r its intra-level redundancy. $Q_{d,0}$ are the queries with no redundancy within subqueries (there still is redundancy across them). Their general form is shown below, already in NEXT form, to give a better intuition on the grouping they perform:

for $\$x_1$**in** $\$doc//X, \$y_{1,1}$**in** $\$x_1/Y_1$
groupby $\$y_{1,1}$ **return**
$\langle T_1\rangle$ **for** $\$x_2$**in** $\$doc//X, \$y_{2,1}$**in** $\$x_2/Y_1, \$y_{2,2}$**in** $\$x_2/Y_2$
  where $\$y_{2,1}$ **eq** $\$y_{1,1}$
  groupby $\$y_{2,2}$ **return**
  ⟨T2⟩ **for** $\$x_3$ **in** \$doc//X, $\$y_{3,1}$ **in** $\$x_3/Y_1$,
    $\$y_{3,2}$ **in** $\$x_3/Y_2$, $\$y_{3,3}$ **in** $\$x_3/Y_3$
    where $\$y_{3,1}$ **eq** $\$y_{2,1}$ **and** $\$y_{3,2}$ **eq** $\$y_{2,2}$
    groupby $\$y_{3,3}$ **return**
    $\langle T_3\rangle$ ...

The nesting depth of group-by constructs goes all the way to d. Notice the redundant navigation across nesting levels: the bindings of $\$y_{n,i}$ are contained in those of $\$y_{n-1,i}$ for each nesting level n. Starting from $Q_{d,0}$, we add intra-level redundancy as follows: on each nesting level, we duplicate the tree pattern r times (relaxing every child step by turning it into a descendant step). For example, if $r = 1$, the second nesting level becomes

⟨T1⟩ **for** $\$x_2$ **in** \$doc//X, $\$y_{2,1}$ **in** $\$x_2/Y_1$, $\$y_{2,2}$ **in** $\$x_2/Y_2$
  $\$x'_2$ **in** \$doc//X, $\$y'_{2,1}$ **in** $\$x'_2//Y_1$, $\$y'_{2,2}$ **in** $\$x'_2//Y_2$
  where $\$y_{2,1}$ **eq** $\$y_{1,1}$
  groupby $\$y_{2,2}$ **return**
  ⟨T2⟩

and for $r = 2$, level 2 contains 9 variables. Notice that $Q_{d,r}$ is equivalent to $Q_{d,0}$ for every r. Indeed, all queries with the same d will be minimized to the same NEXT query. The queries we minimize have $1 + (r + 1)(d^2 + 3d)/2$ path expressions in them, which is a very large number for our maximal choices of d and r.

The measurements. Figure 13 depicts a family of curves. Each shows the minimization time as a function of the nesting depth d, for a fixed r. For example, a query of 15 nesting levels, with intra-level redundancy 7, has a total of 1081 variable bindings, and performs just as many individual navigation steps, which exceeds by far practical query sizes. The minimal form performs only 16 navigation steps. Minimization takes 656 milliseconds, which is an insignificant fraction of the running time even for much simpler queries, and a worthwhile effort to spend for such a significant reduction of navigation complexity.

The effect of bottom-up join evaluation and join/projection interleaving, according to Yannakakis' algorithm, was quite beneficial to our implementation. In a first, more brute-force implementation which performed the joins top-down, instead of according to Yannakakis' algorithm, we measured much slower minimization times, despite using the same efficient data structures for performing joins and unions. For example, $Q_{2,1}$ (11 nodes) and $Q_{4,0}$ would take more than 5 seconds (and so did all queries with larger d and r).

B.2 Effect of Minimization on Query Run Time

We measured the benefit of minimization on the overall query execution time of a set of OptXQueries. We used synthetic input documents containing books. Their sizes ranged from 1000 to 10000 in steps of 1000. For every

Query	Average Speedup	Optimized time	Unoptimized time
Query 1	1.5	60	90
Query 2	2.7	60	160
Query 3	1.7	60	102
Query 4	2.9	65	190
Query 5	10.5	133	1397
Query 6	5.5	65	360

Table 1: Average query running time ratio

size we had two files. In the first file the number of authors was roughly 1/100th of the number of books. In the second file the number of authors was roughly 100 always. In both files the number of years was 30 and the number of publishers was 1/100th of the number of books. All the experiments were executed on a 2Ghz CPU, 1GB of memory, and a 34GB drive running Windows 2000. The engine provides very competitive plans that make use of efficient join operators, in the spirit of [14]. For example, nested queries are not run in a naive way, where for each iteration of the outer query we ran the inside query from scratch. Instead, when the inner query has equality conditions with the outside query, the plan reads the data of the inner query just once and appropriately indexes them. Then it evaluates the outer query probing in each iteration the indexed table for matching data only of the inner query.

Table 1 shows the ratios of the average running time of the standard XQueries to their minimal NEXT query form. Query 1 is (X1) from Section 1. Queries 2 and 3 are the two equivalent queries (X2) and (X5). Query 4 is $Q_{2,0}$ in Figure 13, and it groups books by author at the first level and then by year at the second level. Query 5 is $Q_{3,0}$ in Figure 13, so it has one more nesting level than Query 4; it groups books at the third level by publisher. Query 6 is $Q_{2,1}$ in Figure 13, so it is similar to Query 4 but with intra-level redundancy.

The minimization time for all queries in Table 1 is less than 5 milliseconds. The minimal NEXT query is insensitive to the nesting depth.

C Constraints on OptXQuery

Below is a sufficient condition for disallowing set equality checks. This condition is not necessary and is relaxed in our formal specification of OptXquery. We say that a variable is an *input element* variable if it binds to single elements from the input. We say that it is a *simply created element* variable if it binds to elements created by the query via repeated application of element constructors to constants, input element and simply created element variables. Note the recursive definition and the fact that complex expressions such as path expressions or **for** loops are disallowed. We require that all variables appearing in equality conditions be input element variables, and all other variables be either input or simply created element variables. We check this by employing a simple type inference algorithm (presented in the full version of the paper), which identifies on the user query the variables violating the restrictions, warning the user that the query is not guaranteed to be fully minimized. Notice that this condition rules out from the where clause checks such as $\langle a\rangle \$x/b\langle /a\rangle$ **eq** $\langle a\rangle \$y/b\langle /a\rangle$, as well as $\langle a\rangle \$u\langle /a\rangle$ **eq** $\langle a\rangle \$v\langle /a\rangle$, where $\$u$ is bound to $\langle c\rangle \$x/b\langle /c\rangle$ and $\$v$ is bound to $\langle c\rangle \$y/b\langle /c\rangle$. Both compare the node sets $\$x/b$ and $\$y/b$ for equality. Also ruled out are from within a **distinct-values** function expressions such as $\langle a\rangle \$x/b\langle /a\rangle$, which compare the node sets obtained for various bindings of $\$x$.

A sufficient restriction for avoiding disjunctive conditions is (aside from the explicit absence of the keyword **or**): variables bound by **some** clauses shall not range over sets of nodes obtained by concatenating results of several navigations. Indeed, observe that **some** $\$x$ **in** (E_1, E_2) **satisfies** C is equivalent to (**some** $\$x$ **in** E_1 **satisfies** C) **or** (**some** $\$x$ **in** E_2 **satisfies** C).

Though only OptXQueries are guaranteed to be fully minimized, the processor may also input arbitrary XQueries and optimize them using minimization. We discuss the processing of arbitrary XQueries in Appendix D.

D Beyond OptXquery: NEXT+

The query processor also minimizes XQueries that are outside the optXQuery class. It first reduces such XQueries to the **NEXT+** form and then minimizes their NEXT components. In their functional representation, NEXT+ queries extend each component of NEXT's "**for** V **in** *Navigation* **where** *Condition* **groupby** *GroupBy List* **return** *Result Function*" in the ways described next.

First, the *Result Function* may be a XQuery expression that is disallowed in OptXQuery. For example, it may be a function specified in W3C's XQuery/XPath function and operator specification [28] (such as the **count** function), a non-OptXQuery XQuery/XPath expression such as navigation on the parent axis, or an XQuery-defined function [30] written by the user. The normalization reduces any non-OptXQuery result function to an uninterpreted function $f(Q_1, \ldots, Q_n)$, where $Q_1, \ldots, Q_n$ are NEXT queries or variables - as it is the case with NEXT result functions as well.

The *Navigation* part of the **for** clause may also be a function $f(Q_1, \ldots, Q_n)$. The uninterpreted function f appears in the Xtableau as a special function node, labeled by $f(Q_1, \ldots, Q_n)$. The node for the variable $\$V$ that binds to f connects to the f node via a special edge type. The equivalence step of the minimization algorithm is extended to require that two matching function nodes $f(Q_1, \ldots, Q_n)$ and $f(Q'_1, \ldots, Q'_n)$ have the same name and arity and Q_i is equivalent to Q'_i, for all $i = 1, \ldots, n$.

Next, the *Condition List* is a conjunctive normal form expression where terms of the conjunction may be $\mathbf{not}$ expressions or $\mathbf{or}$ expressions. Furthermore, predicates beyond equality are allowed (e.g. capturing "exclusive or" or universal quantification). The normalization algorithm captures all of the above by introducing uninterpreted boolean predicates $b(Q_1, \ldots, Q_n)$ in the conjunction (in addition to the equality predicates of NEXT). Two such predicates are considered equivalent only if they have the same name and equivalent arguments.

Finally, the *Groupby List* may also involve functions, i.e., non-variable components. (Notice that this may happen only by using **distinct-values** in the original XQuery.) In this case the minimization algorithm does not attempt to minimize the **for** expression; in effect, it treats it as an unintepreted function. However, it still minimizes the NEXT components of the function.