Rewriting Nested XML Queries Using Nested Views

Nicola Onose* Alin Deutsch* Yannis Papakonstantinou† Emiran Curtmola*

Computer Science and Engineering, University of California San Diego
{nicola,deutsch,yannis,ecurtmola}@cs.ucsd.edu

ABSTRACT

We present and analyze an algorithm for equivalent rewriting of XQuery queries using XQuery views, which is complete for a large class of XQueries featuring nested FLWR blocks, XML construction and join equalities by value and identity. These features pose significant challenges which lead to fundamental extension of prior work on the problems of rewriting conjunctive and tree pattern queries. Our solution exploits the Nested XML Tableaux (NEXT) notation which enables a logical foundation for specifying XQuery semantics. We present a tool which inputs XQuery queries and views and outputs an XQuery rewriting, thus being usable on top of any of the existing XQuery processing engines. Our experimental evaluation shows that the tool scales well for large numbers of views and complex queries.

1. INTRODUCTION

The ability to equivalently rewrite queries using views is required by multiple data management tasks. For example, a query processor can speed up the processing of a query when part of the computation needed by the query has already been performed by the materialized views or cached queries. Another application comes from the field of privacy-preserving data publishing, in which a data source answers only those client queries which can be rewritten using exclusively views the data owner agrees to publish [27]. Finally, in data integration views have been used to describe the source content (in Local-As-View architectures) and the source capabilities. We point the reader to the survey [17] for a comprehensive list of applications of rewriting queries using views.

The XML data model emerges as equally important to the relational model for many of the data management problems that require answering queries using views. At the same time, solving the problem in the context of XML and XQuery presents a set of novel challenges. First, data and queries are nested. Second, XQuery has *list* semantics, which degenerates to *bag* semantics if the **unordered** keyword is used, and to *set* semantics in the presence of the duplicate-eliminating primitive **distinct-values** (as in Example 1.1 below). Any rewriting algorithm must be equipped to uniformly test the equivalence of the produced rewriting to the original query under any of these semantics. Contrast this with prior work on XPath (and relational) rewriting, which use only *set* semantics. Third, XML elements may be compared for equality on either their value or their identity; no such distinction comes up in conjunctive queries and XPath. Finally, XQuery copies XML subtrees in the result construction phase, precluding the use of XML node ids for assembling the view data into the query result.

We provide an algorithm which, given an XQuery Q and a set of XQuery views $\mathcal{V} = \{V_1, \ldots, V_n\}$, discovers a rewriting query RW of Q using $\mathcal{V}$. We follow the classic definition of rewriting query [17]: RW is evaluated on the views and for all possible database instances D its result $RW(V_1(D), \ldots, V_n(D))$ coincides with the result $Q(D)$ of Q. The algorithm is sound for full XQuery queries and views, and is complete for an expressive subset of XQuery called OptXQuery [9], which includes nesting, optional use of duplicate elimination (using the **distinct-values** keyword), and allows in the **where** clause conjunctions of id-based and value-based equality conditions, as well as existential quantification (**some** clauses).

EXAMPLE 1.1. *Consider the sample data of Figure 1 and the following query Q that groups paper reviews by the papers' authors; it is a minor variation of query Q4 from W3C's XMP use case [30]. The* **distinct-values** *function eliminates duplicates, comparing elements by value-based equality [31].* [1] *The* **for** *loop binding* $a *(called the* $a *loop) has* set *semantics since the output list of* **distinct-values** *has no duplicates and the order of its elements is non-deterministic. The inner* **for** *loop has* bag *semantics i.e., duplicates are not removed but the order is non-deterministic since the loop is in the context of the unordered* $a *loop.*

```
let $doc := document("DBLP.xml")
for  $a in distinct-values($doc//paper[review]/author)
return ⟨evaluation⟩{ $a,                                        (Q)
       for  $p in  $doc//paper
            $r in  $p/review
       where some  $a1 in  $p/author
             satisfies  $a1 eq $a
       return  $r
       }⟨/evaluation⟩
```

Consider now the following view V, which outputs a list of feedback *elements, where each one contains a review and the list of authors of the corresponding paper.*

*Supported by the NSF under grants IIS-0415257 and IIS-0347968 (CAREER).

†Supported by the Gordon and Betty Moore Foundation and by NSF grants EAR-0225673 and ITR-313384.

SIGMOD 2006, June 27–29, 2006, Chicago, Illinois, USA.

[1] The query can be expressed in a shorter form by replacing its **where** clause with "**where** $a **eq** $p/author" or by replacing the inner **for** with "$doc//paper[author **eq** $a]/review". It is well known [21] how to reduce such syntactic sugar (use of "**eq** " or use of predicates in paths) to OptXQuery.

```
let $doc := document('DBLP.xml')
for  $p in $doc//paper, $r in $p/review
return ⟨feedback⟩{ $r,                                          (V)
                   ⟨authors⟩{ $p/author }⟨/authors⟩
       }⟨/feedback⟩
```

Since the view V returns all author and review information which is pertinent to the query Q, there is a rewriting RW of Q using V:[2]

```
let $doc := document(V)
for  $a2 in distinct-values($doc/feedback[review]/authors/author)
return ⟨evaluation⟩{ $a,                                        (RW)
       for  $f in $doc/feedback
            $r in $f/review
       where some $a1 in $f/authors/author
             satisfies $a1 eq $a
       return $r
       }⟨/evaluation⟩
```

Our query processor discovers rewritings that use the views in order to obtain the variable bindings generated by the query.

To the best of our knowledge this is the first work on rewriting using views for XQuery, which also provides formal guarantees of completeness for a large subset of XQuery.

Prior work on rewriting using views for XML queries (reviewed in Section 6) focused on the XPath language [3, 33]. The resulting algorithms involve detecting query subexpressions subsumed by the view. The subsumption test is enabled by a pattern-based representation of XPath expressions called *tree patterns* [2, 22], which reduce the test to matching the view pattern against the query pattern. The standard specification of XQuery semantics does not support the extension of XPath rewriting techniques as it provides no pattern-based query representation. We therefore adopted *NEsted XML Tableaux (NEXT)* [9], a pattern-based notation, which consolidates navigation in the minimum number of tree patters and hence maximizes rewriting opportunities. NEXT can represent a large subset of XQuery, called OptXQuery and introduced in [9]. OptXQuery is a natural boundary within which the rewriting algorithm is guaranteed to find rewritings whenever they exist. The OptXQuery subset includes the XPath language, and it fully subsumes the c-XQuery subset of conjunctive, nested XQueries whose containment is studied in [13].

Our contributions are:

1. An algorithm for rewriting queries from the OptXQuery subset of XQuery using exclusively views from the same sublanguage. The algorithm equivalently rewrites the variable binding stage of the query, which is where the large costs of navigations, joins and selections of data intensive applications are incurred. If the document order of the query result is immaterial (due to the use of **distinct-values** and **unordered** keywords), then the algorithm is *complete*, i.e., it always finds an equivalent rewriting, if such exists (Theorem 3.1).

2. We analyze the complexity of checking the existence of a rewriting, showing NP-completeness in the *width* of the query's NEXT pattern representation. This measure (defined in Section 3.5), is typically much smaller than the query size, depending only on the number of variables shared across nested query blocks and the number of variables involved in equality conditions. For instance, the width of query Q in Example 1.1 is 3 . Our rewriting algorithm is worst-case exponential in the query width, which is optimal behavior: it runs in PTIME when the query is acyclic, (for which it turns out that the width is 1) its performance degrading gracefully with increasing query width. The acyclic case includes XPath tree patterns.

3. We introduce a technique for extending the rewriting algorithm from OptXQuery to arbitrary XQuery, by identifying and rewriting the OptXQuery subexpressions. The technique is sound, in the sense that it creates only equivalent rewritings. It is not complete, but this is an unavoidable consequence of the undecidability of checking rewriting existence for (even slightly) more expressive queries and views than OptXQuery. Indeed, even relational rewriting algorithms perform a best-effort approach and are incomplete for full SQL. Our approach combines the benefits of complete rewriting feasible for NEXT/OptXQuery with the easy-to-engineer but incomplete techniques based on isomorphically matching common sub-expressions between query and view [12].

4. We provide a tool which inputs XQuery queries and views and outputs an XQuery rewriting, thus being usable on top of any of the existing XQuery processing engines. A demo of the tool is available at `http://db.ucsd.edu/reform`.

5. We report on our experimental evaluation, which measured rewriting times of around 1 second for as many as 128 views, for queries of up to 16 nesting levels, 48 variables per level.

The remainder of this paper is organized as follows. Section 2 describes the system architecture, OptXQuery and its corresponding NEXT notation. Section 3 presents the rewriting algorithm, its implementation, and analyzes its complexity. The extension of the rewriting algorithm to arbitrary XQueries is given in Section 4. We report on the experimental evaluation in Section 5. Related work is discussed in Section 6.

2. ARCHITECTURE AND FRAMEWORK

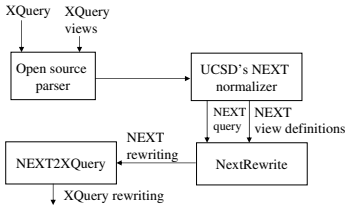

Figure 2: NEXT XQuery Processor Architecture

XML and Equivalence We model an XML document D as an ordered labeled tree of nodes $\mathtt{N}_{XML}$, edges $\mathtt{E}_{XML}$, a function $\lambda : \mathtt{N}_{XML} \rightarrow$ *Constants* that assigns a label to each node, and a function : $\mathtt{N}_{XML} \rightarrow$ *IDs* that assigns a unique id to each node.

Equivalent rewriting. We start from the classic definition of the equivalent rewriting problem (taken from [17]), adapting it for the XML case. Let Q be a query and $\mathcal{V} = \{V_1, \ldots, V_n\}$ be a set of view definitions. The query RW is an equivalent rewriting of Q using $\mathcal{V}$ if RW refers only to the views in $\mathcal{V}$ and for every input document D, Q and RW return isomorphic XML results: $Q(D)$ (the result of Q on D) is isomorphic to $RW(V_1(D), \ldots, V_n(D))$. We consider two flavors of isomorphism: unordered and ordered, leading to *ordered*, respectively *unordered rewritings*. Unordered isomorphism disregards the sibling ordering within each node, while ordered isomorphism preserves it. Unordered rewritings are appropriate whenever an XQuery contains a **distinct-values** or an **unordered** keyword, which cause the output XML tree to be nondeterministically ordered [31]. The **unordered** keyword is typically used in data-centric applications of XQuery (in which the document order is immaterial, and only the data contents matters).

[2]We support views which output a collection of XML elements, rather than a well-formed document with a single root element (we generate an artificial root). For presentation simplicity, we do not show the navigation to the root element.

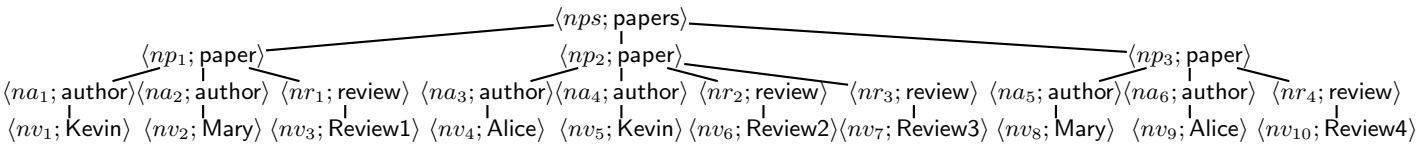

Figure 1: Input XML data

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

Figure 3: OptXQuery

The **distinct-values** keyword is unavoidable in XQueries performing duplicate elimination or grouping by value, as illustrated in Example 1.1. When neither keyword appears in the query and the ordering of its answer XML tree is relevant, we resort to ordered rewritings.

Architecture We have borrowed the normalization module of UCSD's NEXT processor [9], which inputs a query and views expressed in XQuery, applies a series of normalization rules (described in detail in [9]), and produces *Nested XML Tableaux (NEXT)* of the query and the views (see Figure 2). The NEXT representation consolidates all variable binding operations regardless of whether they appear in the **for** or the **where** clause and regardless of whether they are in the context of set (i.e., **distinct-values**) or bag semantics in the **in** clause. This consolidation facilitates the match of the navigation of the query with the navigation of the view. Not all XQuery expressions can be normalized into a NEXT representation. According to [9], this is possible if the given XQuery follows the OptXQuery syntax of Figure 3 and refrains from set/list/bag equality comparisons [31]. We sketch in Section 4 how arbitrary XQueries are handled. This involved extending the NEXT notation to accommodate all XQuery features, and detecting and rewriting only the maximal OptXQuery subexpressions.

The syntax of NEXT (see Figure 4) uses only a subset of OptXQuery features: (i) **in** clauses are (one-step) path expressions, (ii) **where** clause conditions are conjunctions of equality conditions involving variables and constants and (iii) there is no explicit **distinct-values**. These restrictions are compensated by a duplicate-eliminating projection operator, which in its full-blown version [9] is a group-by construct. However, the grouping functionality is not needed by the rewriting algorithm. We keep the name **groupby** for consistency with the NEXT terminology and we advise the reader to think of **groupby** as duplicate-eliminating projection, as described below.

The query Q and the view V from Example 1.1 are rewritten into the NEXT normal forms below. NEXT extends FLWR expressions with a **groupby** clause that consists of a list of group-by variables. A variable from the **groupby** list is called *groupby-id variable* if it appears within brackets (e.g., variable $\$p1$ of (NEXT_Q)), and is called *groupby-value variable* otherwise (e.g, variable $\$a$

$$
\begin{array}{rcll}
XQ & ::= & \langle n\rangle\{XQ_1,\ldots,XQ_m\}\langle/n\rangle & \text{(P1)} \\
 & | & Var & \text{(P2)} \\
 & | & \textbf{for } Var_1 \textbf{ in } Path_1,\ldots,Var_n \textbf{ in } Path_n & \text{(P3)} \\
 & & (\textbf{where } CList)? & \\
 & & \textbf{groupby } (Var'_1|[Var'_1])\ldots(Var'_k|[Var'_k]) & \\
 & & \textbf{return } XQ_1 & \\
Path & ::= & (\textbf{document } (\text{“}Constant''){|}Var)((/|//)Constant) & \text{(P4)} \\
CList & ::= & Cond\ (\textbf{and } Cond)* & \text{(P5)} \\
Cond & ::= & Var \textbf{ eq } (Var|Constant) & \text{(P6)} \\
 & | & Var \textbf{ is } Var & \text{(P7)}
\end{array}
$$

Figure 4: NEXT Query Syntax

of (NEXT_Q)). A **groupby** construct inputs the tuples of variable bindings produced by the preceding **for** and **where** clauses and projects the variables of the **groupby** list, eliminating duplicates. To detect duplicates, tuples are compared component-wise, looking for equal values on the bindings of the groupby-value variables and for equal id's for the groupby-id variables. The **return** clause is executed once for each tuple in the output of the **groupby** clause.

B_1^Q ($NEXT_Q$):

```
for $p in document("DBLP.xml")//paper,
    $r in $p/review,
    $a in $p/author
groupby $a
return ⟨evaluation⟩{ $a,
    B2^Q:
    for $p1 in document("DBLP.xml")//paper,
        $a1 in $p1/author,
        $r1 in $p1/review
    where $a1 eq $a
    groupby [$p1], [$r1]
    return $r1
}⟨/evaluation⟩
```

B_1^V ($NEXT_V$):

```
for $p in document("DBLP.xml")//paper,
    $r in $p/review
groupby [$r], [$p]
return ⟨feedback⟩{ $r,
    ⟨authors⟩{
    B2^V:
    for $a in $p/author
    groupby [$a]
    return $a
    }⟨/authors⟩
}⟨/feedback⟩
```

*The tree of **groupby** blocks.* We call **groupby** *block* a NEXT expression described by Production (P3) of Figure 4. For example, NEXT_Q contains blocks B_1^Q and B_2^Q as illustrated above. If a block B' is immediately (transitively) nested within block B, we say that B' is a child (descendant) of B and B a parent (ancestor) of B'. The **groupby** blocks of a query are therefore organized in a tree structure, called the **groupby** tree. The *bound variables* of a block B are those variables $\$v$ appearing in a "$\$v$ **in** *path*" expression. All other variables are *free* in B, but must be bound in some ancestor block of B. For instance, $\$a$ is bound in B_1^Q and free in B_2^Q.

Pattern-based graphical representation of NEXT. NEXT queries have a graphical, pattern-based notation that extends the well-known tree patterns used in XPath-related works. The representation depicts the tree of groupby blocks, labeling each block with a pattern, a list of groupby-value variables, a list of groupby-id variables, and a return function. A NEXT pattern $P = (F, EQ_{val}, EQ_{id})$ consists of a forest F of tree patterns, which capture navigation. As is common in tree pattern notations, each node is labeled with (i)

a variable and (ii) the label that this node matches to. Edges are labeled with / or // depending on whether the corresponding nodes are in a child or descendant relationship. EQ_{val} is the set of value-based equalities, denoted by dotted lines, and EQ_{id} is the set of id-based equalities (denoted by double dotted lines, though not needed in our example). The bound variables of a NEXT pattern are precisely the variables which are the target of some /- or //-edge. The return functions consist of element creation and concatenation and take as parameters variables and nested blocks. In the latter case, the meaning is that, upon instantiation, the parameter is replaced by the result of the corresponding block.

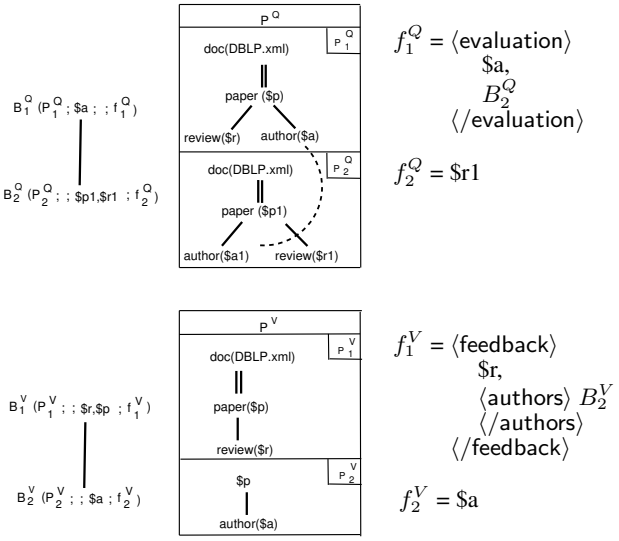

Figure 5: NEXT for Q and V from Example 1.1

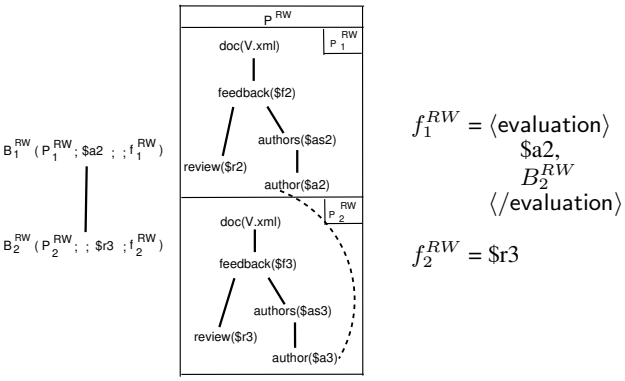

Figure 6: NEXT for RW from Example 1.1

EXAMPLE 2.1. *Figure 5 shows the NEXT representation for Q and V, and Figure 6 shows the same for RW. In each case, we see the tree of* **groupby** *blocks at the left, followed by the corresponding NEXT patterns, and to their right, the return functions. The second argument of the groupby blocks shows the list of groupby-value variables ($\$a$ for B_1^Q, empty for B_2^Q, B_1^V and B_2^V). The third argument gives the list of groupby-id variables (empty for B_1^Q, $\$p_1, \$r_1$ for B_2^Q).*

Understanding of the rewriting algorithm is facilitated if one thinks of the result of NEXT query Q on an XML document D as being computed in two stages. First, the *binding stage* computes all variable bindings by matching the NEXT patterns against the XML input. Bindings are organized in a nested relation according to the nesting of **groupby** blocks. We call this nested table the *binding table*. Next, the *tagging stage* operates on the binding table, constructing an XML fragment for each tuple in the nested relation by calling the appropriate return function. Denoting with Q^{bind} the function (from XML documents to binding tables) implemented by the binding stage, and with Q^{tag} the function (from binding tables to XML documents) implemented by the tagging stage, we have $Q^{tag}(Q^{bind}(D)) = Q(D)$ for every XML document D and NEXT query Q.

Table 1: Result of NEXT$_Q$'s binding stage

B_1^Q		
$\$a$	B_2^Q	
	[$\$p1$]	[$\$r1$]
<author> Kevin </author>	np_1	nr_1
	np_2	nr_2
	np_2	nr_3
<author> Mary </author>	[$\$p1$]	[$\$r1$]
	np_1	nr_1
	np_3	nr_4
<author> Alice </author>	[$\$p1$]	[$\$r1$]
	np_2	nr_2
	np_2	nr_3
	np_3	nr_4

EXAMPLE 2.2. *For the sample input provided in Figure 1, NEXT$_Q$'s binding stage yields the nested relation of variable bindings shown in Table 1. Notice that the binding table's attribute names coincide with the variable names and the names of the nested* **groupby** *blocks. For each tuple, the attributes named after variables hold the corresponding bindings of groupby variables and the attributes named after the nested* **groupby** *blocks recursively hold the corresponding set of bindings, which is in turn a nested relation.*

3. THE REWRITING ALGORITHM

Given a NEXT query Q and a set of NEXT views $\bar{V}$, algorithm NEXTREWRITE equivalently rewrites the binding stage of Q to use solely the views. Since Q's tagging stage is independent of how the variable bindings were obtained, it is reused. NEXTREWRITE obtains a query RW, such that (i) RW^{tag} is Q^{tag}, (ii) the binding stage of RW is expressed only in terms of the views and returns the same bindings as the binding stage of Q for each XML document D (preserving their order if ordered rewritings are sought): $Q^{bind}(D) = RW^{bind}(D)$. Since $RW = Q^{tag} \circ RW^{bind}$ and $Q = Q^{tag} \circ Q^{bind}$, it follows that RW is equivalent to Q.

To find RW^{bind}, NEXTREWRITE proceeds in three phases.

Phase 1 identifies all variables x of Q whose bindings are also produced by some view V and can therefore be retrieved by navigation into V's output. This navigation is called an *alternate view access path* towards x's bindings. The view access paths are (redundantly) added to Q. Call the resulting expanded query Q_E.

Phase 2 restricts the expanded query Q_E by dropping all original query navigation and keeping only the view access paths added during the expansion. Query variables in the tagging and groupby components are appropriately replaced. Order-related checks are performed at this point when ordered rewritings are of interest. The result is the candidate rewriting *RW*, which performs a join of the alternate view access paths.

Finally, Phase 3 checks whether the candidate rewriting *RW* is truly equivalent to Q. Equivalence may fail if the join of the view access paths is lossy, i.e. returns too many bindings, or due to the loss of element identities caused by XQuery's copy semantics for result construction. This is not the case in our example.

3.1 Phase 1: Detecting View Access Paths

This phase detects alternate *access paths* through the views towards the bindings of the query variables. The access paths are detected via *mappings* from the views to the query and are implemented by navigation patterns which *invert the view return functions*. We detail these concepts next.

View access paths. Given a vector $\bar{x}_Q$ of Q's variables and an equal-arity vector $\bar{x}_V$ of V's variables, we say that there is an access path through $\bar{x}_V$ to $\bar{x}_Q$ iff (i) for all documents D, the bindings of $\bar{x}_Q$ are a subset of the bindings of $\bar{x}_V$, and (ii) the bindings of $\bar{x}_V$ are retrievable from the output of V.

Inverse of a return function. Views do not return their variable bindings directly, but instead construct new XML output from them. The view's return functions must hence be inverted to retrieve the bindings. Given the return function f of a NEXT block, we denote the *inverse* of f with *Inv*(f). For each tuple of variable bindings t, $f(t)$ outputs an XML fragment x, while *Inv*$(f)(x)$ retrieves from x a copy of t. The reason only a *copy* of t (as opposed to t itself) is accessible is that XQuery semantics specifies that a return function's output is constructed by copying the XML subtrees to which the view variables are bound, thus losing the identities of element nodes [31]. Phase 3 of the algorithm determines whether these copies suffice for rewriting. We illustrate the inversion below.

EXAMPLE 3.1. *In Example 1.1, there is an access path to Q's variables $\$r, \$a$ through V's variables $\$r, \$a$, whose bindings are retrievable by navigating into V's output. The navigation pattern of the access path is obtained from* $\text{Inv}(f_1^V)$ *and* $\text{Inv}(f_2^V)$ *(shown in Figure 7 (a)), by adorning with fresh variables each internal node of the tree corresponding to the return function's XML constructors. To capture the fact that the bindings of variable $\$a$ returned by f_2^V are children of the* `authors` *element constructed by f_1^V, we add a /-edge from $\$as$ to $\$a$. The resulting inverse functions specify that, in order to navigate to an* `author` *element in the view output, we need to first navigate to an* `authors` *element and then continue with a nested navigation to its children.*

Invertible NEXT views. In order to invert NEXT return functions, we sometimes need to go outside of the language and use navigation to certain positions in a list. Take for instance the query

for $\$p$ **in** doc(in.xml), $\$x$ **in** $\$p/c$, $\$y$ **in** $\$p/c$,
return $\langle r\rangle\langle a\rangle\$x\langle/a\rangle\langle a\rangle\$y\langle/a\rangle\langle/r\rangle$

in which navigation to a children of the constructed r elements does not disambiguate between the bindings of $\$x$ and those of $\$y$. However, we can do so with an inverse function which navigates to the first a child for $\$x$, and the second for $\$y$. Notice that if in the above query, $\$y$ was instead bound to $\$p/d$, we could still disambiguate by navigating along $r/a/c$ for the bindings of $\$x$ and along $r/a/d$ for $\$y$.

The return functions of arbitrary XQueries are not necessarily invertible (consider aggregates for instance, where one cannot navigate into the aggregate result to reconstruct the arguments). However, a sufficient condition for a NEXT query to be invertible is that for each of its return functions f, (i) all nested **groupby** block arguments B of f appear within an element constructor in f: $\langle a\rangle B\langle/a\rangle$, or (ii) f has at most one nested **groupby** block argument.

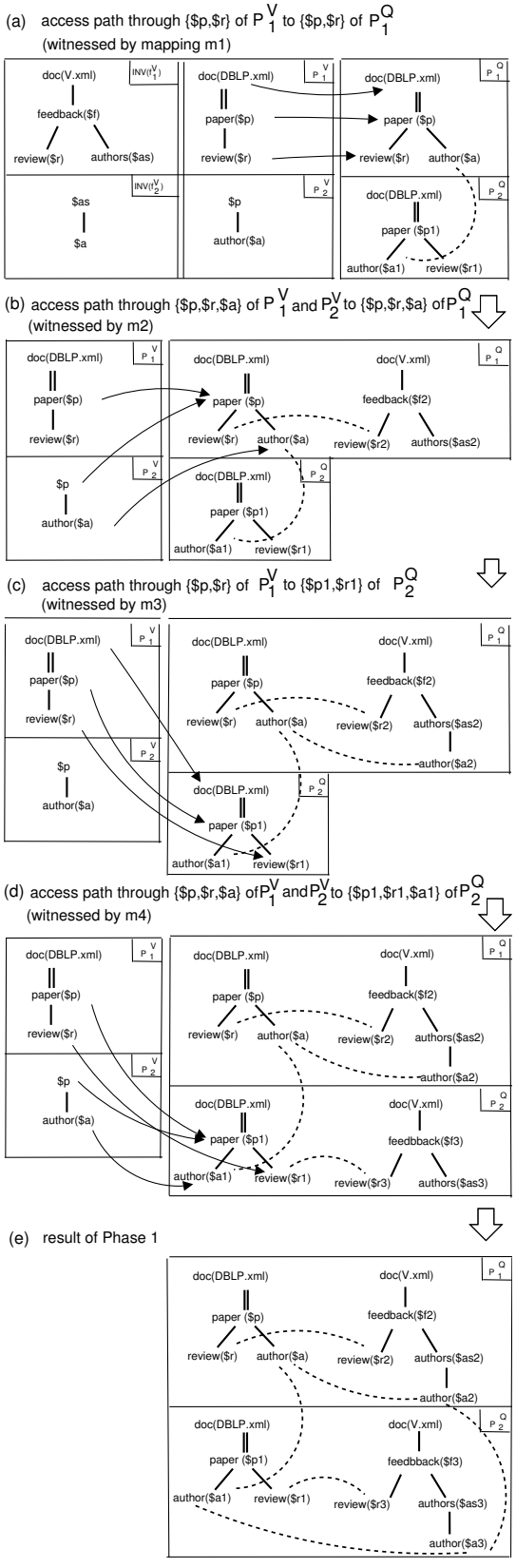

Figure 7: Phase 1 of NEXTREWRITE **for Q,V from Example 1.1**

Detecting access paths using mappings. We now focus on how access paths are detected. By definition, an access path through view variables $\bar{x}_V$ towards the query variables $\bar{x}_Q$ means that the

set of bindings of $\bar{x}_Q$ is contained in the set of bindings of $\bar{x}_V$. We adopt the containment test from [9], which characterizes containment by the existence of *NEXT pattern mappings*, which we shall henceforth call simply *mappings*. It follows from [9] that there is an access path to $\bar{x}_Q$ through $\bar{x}_V$ if there is a mapping from V to Q which maps $\bar{x}_V$ into $\bar{x}_Q$, and if V's return functions are invertible.

DEFINITION 3.1. *(***NEXT pattern mappings [9]***) A mapping m from NEXT pattern P_1 to NEXT pattern P_2 maps the variables of P_1 into variables of P_2 such that*

(i) variables map to variables of the same tag label,

(ii) the source and target variables of each /-edge in P_1 map to the source, respectively target variable of some /-edge in P_2,

(iii) the source and target variables of each //-edge in P_1 map to the source, respectively target of a path of /- and //-edges in P_2,

*(iv) for each id-equality v **is** u in P_1, the equality $m(v)$ **is** $m(u)$ is in the closure of P_2's id-equalities under reflexivity, symmetry, and transitivity, and*

*(v) for each value-equality v **eq** u in P_1, the equality $m(v)$ **eq** $m(u)$ is in the closure of P_2's value-equalities under reflexivity, symmetry, transitivity, and the rule x **is** $y \Rightarrow x$ **eq** y.* ◇

EXAMPLE 3.2. *Figure 7 illustrates Phase 1 of algorithm* NEXTREWRITE *on the running example. For every view access path to its variables, the query is expanded with the appropriate navigation given by the view inverses. Each snapshot shows the mapping (depicted by arrows) which detects an access path, and the successor snapshot shows the query after adding this access path. The value equality conditions (shown by dotted lines) record the correspondence between query variables and the view variables providing their alternate access path. Snapshots (a) and (b) show access paths detection for block B_1^Q, and similarly (c) and (d) for block B_2^Q. The result of Phase 1 is shown in Snapshot (e) which contains, besides the original query patterns, the NEXT pattern of the rewriting candidate RW (compare to Figure 6).*

Exploiting view block nesting for non-redundant mapping computation. According to XQuery semantics, when a nested block is correlated by shared variables to an ancestor block, the bindings of the nested block's free variables are provided by the ancestor block. This means that whenever a view block B^V provides an access path, the access to the bindings of B^V's free variables is provided by ancestor blocks of B^V. Therefore, the access path through B^V must be an extension of an access path through B^V's ancestors. Similarly, the mapping m witnessing an access path through B^V must be an extension of a mapping m_p witnessing an access path through the ancestors. By extension, we mean that m and m_p agree on shared variables. To avoid redundant rediscovery of ancestor mappings, Phase 1 visits the view's **groupby** tree in a top-down fashion, finding mappings m_p from ancestor blocks once, and recursively extending them to nested blocks if possible.

EXAMPLE 3.3. *Snapshot (a) in Figure 7 shows the mapping $m_1 = \{\$p \mapsto \$p, \$r \mapsto \$r\}$ of P_1^V into P_1^Q. Mapping m_2 in Snapshot (b) is the extension of m_1 to P_2^V using $\{\$a \in P_2^V \mapsto \$a \in P_1^Q\}$. To avoid clutter, we only show the arrows for the extension.*

Congruence Closure. The congruence closure operation is needed to expose implicit mapping opportunities. See for instance Figure 8 for patterns P^V and P^Q. P^V has no mapping into P^Q, as the only way to map $\$u_2$ is into $\$x_2$, which has no d-child to map $\$u_4$ into. However, from the value-equality of $\$x_2$ with $\$x_4$, we can infer the existence of a d-child under $\$x_2$, call it $\$x_6$, which is value-equal

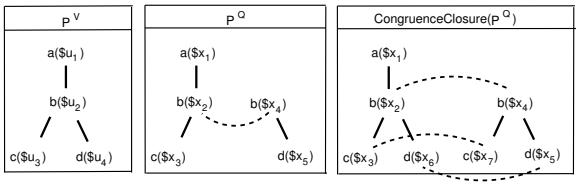

Figure 8: CONGRUENCECLOSURE **enables mappings**

to $\$x_5$ (and, symmetrically, that of a c-child $\$x_7$ which is value-equal to $\$x_3$). $\$x_6$ can now serve as target for $\$u_4$. Procedure CONGRUENCECLOSURE (shown in the pseudocode below) makes this kind of inferences by exposing additional mapping targets.

3.2 Phase 2: Candidate Rewriting

Phase 2 restricts the result of Phase 1, keeping only the view access paths. To this end, it identifies all variables that can be dropped from the query's group-by lists, substituting the remaining ones with view variables as dictated by the view access patterns. If no appropriate view variables are found, the rewriting fails. Additional failure cases apply when ordered rewritings are sought, or due to the copy semantics of XQuery result construction (explained below). The new return functions of the rewriting are obtained by applying the same substitution to the query's return functions.

Dropping ***groupby*** *variables.* To check whether a **groupby** variable x can be dropped, we test that (i) x does not appear in any query return function, and (ii) in the **groupby** list, x appears together with some other variable y which determines x's value or id (depending on whether x is a groupby-value of groupby-id variable). This test is necessary to preserve the cardinality of the groups output by the binding stage. For groupby-id variables, the bindings of x are determined by those of y if y binds to children of x. [3]

Replacing ***groupby*** *variables.* When replacing a groupby variable v of the original query with a variable u of the view, we must make sure that the rewriting generates the same groups of variable bindings as the query. To this end, the bindings of both variables should ideally be identical (same value and identity). This means that v can only be replaced with variables u which are at least value-equal to v. In addition, if v is a groupby-value variable, it may be replaced with u regardless of whether u is a groupby-value or groupby-id variable, since the id of u does not contribute to the **groupby** result. But if v is a groupby-id variable, it must be replaced only by a groupby-id variable u since groupby-value variables lose the identity (and cardinality) of their bindings.

EXAMPLE 3.4. *Figure 6 displays the NEXT form of the candidate rewriting obtained for the query and view from Example 1.1 by restricting the Phase 1 result from Snapshot (e). The* ***groupby*** *tree block is isomorphic to that of Q and the patterns navigate exclusively into the views. The groupby-value variable $\$a$ has been replaced by $\$a_2$, since author values can be obtained from the view as well, as witnessed by the value-equality between $\$a$ and $\$a_2$. The groupby-id variable list $\$p_1, \$r_1$ has been first restricted to $\$r_1$, since it determines $\$p_1$ which is not used in the return function and the nested blocks. Then $\$r_1$ is substituted by $\$r_3$ according to the value-equality. All these replacements generate f_1^{RW} and f_2^{RW} in Figure 6, obtained from f_1^Q and f_2^Q in Figure 5 (by replacing variables $\$a$ with $\$a_1$ and $\$r_1$ with $\$r_3$).*

[3]Other cases requiring XML Schema information are: (i) when x and y are siblings, both non-optional, sharing their tag names with no sibling; (ii) when y binds to x's parent, x shares its tag with no siblings, and is not optional; (iii) in the presence of an XML key constraint.

Issues of copy semantics. The copy semantics of XQuery views adds an additional technical problem: since the view's output elements are copies of input elements, they lose the original identities and there is no hope to find a view groupby-id variable u whose bindings have the same identities as those of v. Fortunately, it is sufficient if the identities of u's bindings are in one-to-one correspondence with those of v's bindings. This ensures the same number of groups regardless of whether we group by v or u, and the same outcome of **is** tests if we substitute v for u in them. If V's output extracted by u was created by copying the same elements as v binds to, then this one-to-one correspondence is guaranteed by the XQuery copy semantics. The above restrictions for groupby-variable replacement are only necessary, not sufficient for preserving the number of groups produced by the binding stage. The final check is performed in Phase 3.

Assembling view access paths for ordered rewritings. The order of an XQuery's result corresponds to the order in which variable bindings are generated in the binding stage. Normalization preserves ordering: a clause **groupby** $\$x, \$y, \$z$ can only be obtained if, before normalization, the **for** loop binding $\$x$ appears before the $\$y$ loop, which in turn appears before the $\$z$ loop. Consequently, the order of the variable bindings is induced by the lexicographic ordering of the variables in the **groupby** lists. For instance, clause **groupby** $\$x, \$y, \$z$ orders the triples of bindings first by the document order of the bindings of $\$x$, breaking ties by the order of $\$y$ bindings, whose ties are broken by the order of the $\$z$ bindings.

When ordered rewritings are sought, we must preserve the initial ordering of the query's **groupby** variable list $\bar{x}_Q$ when replacing them with view variables $\bar{x}_V$. To this end, we search for an ordering of the view access paths which imposes on $\bar{x}_V$ the order desired for $\bar{x}_Q$. This is not always possible, as illustrated by Example 3.5, which shows a case with an unordered but no ordered rewriting.

EXAMPLE 3.5. *View V_1 below provides an access path for variables $\$x, \$z$ of query Q, and V_2 an access path for variable $\$y$. Q has no ordered rewriting using V_1 and V_2, since the two possible orderings of the view access paths yield the* **groupby** *lists $\$x, \$z, \$y$ and $\$y, \$x, \$z$, but not the desired $\$x, \$y, \$z$.*

Q: **for** $x **in** $path_1$,
$y **in** $path_2$,
$z **in** $path_3$
groupby $x,$y,$z
return E($x,$y,$z)

V_1: **for** $x **in** $path_1$, $z **in** $path_3$
groupby $x,$z **return** ⟨a⟩$x,$z⟨/a⟩

V_2: **for** $y **in** $path_2$
groupby $y **return** ⟨b⟩$y⟨/b⟩

3.3 Phase 3: Equivalence Check

Since the views may be under-conditioned, the candidate rewriting may contain the bindings of the query, but is not guaranteed to be equivalent. The equivalence check is similar to the relational case [19]: First, unfold the views in the rewriting (for us, this means substituting for each inverse function the pattern corresponding to the same view block) so that the rewriting is expressed in terms of source documents. Next, check equivalence between Q and the unfolding of RW. The equivalence check was handled in detail in [9], where it was employed in minimization of NEXT queries. Its key point is that equivalence of nested blocks has to be judged in the context of their ancestor blocks, since ancestor blocks provide the bindings for the variables which are free in descendant blocks.

EXAMPLE 3.6. *The unfolding of RW is shown in Figure 9.*

3.4 Details and Formal Guarantees

The pseudocode of algorithm NEXTREWRITE is shown in Figures 10 and 11. For a given query Q and view V, EXPANDALLBLOCKS visits Q's tree of **groupby** blocks in a top-down manner, invoking EXPANDBLOCK at each block B^Q.

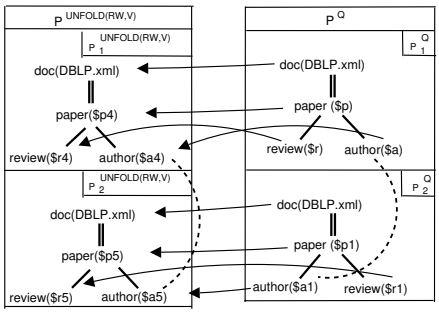

Figure 9: Sample mapping used in checking equivalence of Q with UNFOLD(RW, V)

Given a query block B^Q with pattern P^Q and a view block B^V with pattern P^V, EXPANDBLOCK searches for a subset of P^Q's variables to which P^V provides an alternate access path. They are detected via mappings (m'_p in line 4). Once a view access path is found, it is recorded by calling procedure ADDVIEWACCESS (line 5). The search continues recursively for access paths provided by blocks nested within B^V (lines 6–7). To avoid redundant computation of mappings from B^V during the visit of nested blocks, m'_p is passed as argument.

Detecting access paths within the query's ancestor context. Since a nested query block B^Q may have free variables correlating it with its ancestors, an access path may become possible only once we consider the extra constraints on the bindings of free variables, as provided by the patterns of B^Q's ancestors. Procedure EXPANDBLOCK takes this into account by mapping the view pattern P^V not just into P^Q, but into the *merged pattern* M^Q. M^Q is obtained by unioning together the edges and equalities of P^Q with those of its ancestor patterns, and inferring any additional implied equalities according to procedure CONGRUENCECLOSURE (lines 2–3).

Expanding P^Q with view access. Whenever P^V provides an access path to variables in P^Q, ADDVIEWACCESS expands P^Q with a copy of the navigation pattern corresponding to the inverse of the view return function, *Inv*(f) (line 2). *Inv*(f) may contain variables v which occur in P^V or in the ancestor patterns of P^V and are thus already mapped by m_p. These are the variables in domain$(m_p) \cap$ *Inv*(f) referred to in line 3. The image under m_p of each such v identifies a query variable $m_p(v)$ with an access path through v. This is recorded in line 4 by the value-equality between $m_p(v)$ and v_c, the access path variable introduced in line 2 to retrieve the v bindings for this access path.

EXAMPLE 3.7. *The mapping m_1 in Snapshot (a) witnesses the access path to $\$r \in P_1^Q$ through the inverse of f_1^V. Snapshot (b) shows the extension of P_1^Q with a copy of* Inv(f_1^V). *Line 3 of* ADDVIEWACCESS *computes $\{\$r\} =$ domain$(m_1) \cap$ Inv(f_1^V). Line 4 adds the equality between the copy of variable $\$r$ (which is $\$r_2$) and $m_1(\$r)$ (which is $\$r \in P_1^Q$).*

Correlating view access paths. The argument m_r of ADDVIEWACCESS is used to handle free root variables $\$v_a$ in *Inv*(f). These variables also occur in the inverse function *Inv*(f_a) of some ancestor block B_a^V of B^V, thus correlating *Inv*(f) and *Inv*(f_a). By the time ADDVIEWACCESS is invoked for B^V, it has already executed on B_a^V and has introduced a copy of $\$v_a$ into B_a^V. Upon reaching B^V, ADDVIEWACCESS must preserve the correlation by using the same copy of $\$v_a$. To this end, it consults the mapping m_r which records the correspondence between correlation variables in *Inv*(f) and their copies in *Inv*(f_a) (lines 5–6). m_r is then extended to m'_r (line 7) to record the new correspondences between correlation variables in *Inv*(f) which appear in nested blocks of B^V (line 8).

NEXTREWRITE($Q, \bar{V}$)

▷ *Phase 1: expand Q with alternate view access paths:*
1 **for** each $V \in \bar{V}$
2 **do let** B^Q, B^V be the root **groupby** blocks of Q, V
3 $X =$ EXPANDALLBLOCKS(B^Q, B^V)
▷ *Phase 2: construct candidate rewriting using only view access:*
4 $RW =$ KEEPVIEWSONLY($X, \bar{V}$)
▷ *Phase 3: check if candidate rewriting is equivalent to Q:*
5 **if** UNFOLD($RW, \bar{V}$) is not equivalent to Q
6 **then** report "no rewriting exists"
7 **else** minimize redundant access paths in RW and return result

EXPANDALLBLOCKS(B^Q, B^V)
1 EXPANDBLOCK($B^Q, B^V, m_\emptyset, m_\emptyset$) ▷ $m_\emptyset$: *empty mapping*
2 **for** each child B_c^Q of B^Q
3 **do** EXPANDALLBLOCKS(B_c^Q, B^V)

EXPANDBLOCK(B^Q, B^V, m_p, m_r)
▷ m_p: *mapping of pattern vars from B^V's ancestors into B^Q*
▷ m_r: *mapping of inverse function vars from B^V's ancestors into access path vars in expansion of B^Q and ancestors*
1 **let** P^Q, P^V be the patterns of B^Q, B^V
2 **let** M^Q denote the pattern obtained by merging P^Q with the patterns of all ancestor blocks of B^Q
3 CONGRUENCECLOSURE(M^Q)
4 **for** each mapping $m'_p : P^V \to M^Q$ which extends m_p
5 **do** $m'_r =$ ADDVIEWACCESS(P^Q, B^V, m'_p, m_r)
6 **for** each child B_c^V of B^V
7 **do** EXPANDBLOCK(B^Q, B_c^V, m'_p, m'_r)
8 CONGRUENCECLOSURE(P^Q)

ADDVIEWACCESS(P^Q, B^V, m_p, m_r)
▷ *adds to P^Q the inverse of B^V's return function*
1 **let** f be the return function of B^V, and *Inv*(f) its inverse (*Inv*(f) is a tree pattern)
2 add to P^Q a copy of the edges in *Inv*(f) (for each variable $v \in$ vars(*Inv*(f)) denote its copy with v_c)
▷ *record equality between query variables*
▷ *and their corresponding view access path variables:*
3 **for** each $v \in$ vars(*Inv*(f)) $\cap$ domain(m_p),
4 **do** add to P^Q the value-equality $m_p(v)$**eq** v_c
▷ *link free variables in the copy of* Inv(f) *to*
▷ *their bound occurrence in ancestor (given in m_r):*
5 **for** each $v \in$ vars(*Inv*(f)) $\cap$ domain(m_r),
6 **do** replace v_c with $m_r(v)$ in P^Q
7 construct extension m'_r of m_r that maps each $v \in$ vars(*Inv*(f)) $\setminus$ domain(m_r) into v_c
8 **return** m'_r

CONGRUENCECLOSURE(P) ▷ *side-effects P*
1 extend P with the symmetric, transitive closure of its equalities
2 **for** each value-equality v **eq** $u \in P$ and each child v' of v
3 **if** u has no child u' with v'**eq** u' **then**
4 add a copy of the subtree rooted at v' as a child of u
5 add a value-equality between each copied variable and its copy
6 analogous to lines 2–5 for id-equalities v **is** $u \in P$

Figure 10: Phase 1 of Algorithm NEXTREWRITE

EXAMPLE 3.8. *In Figure 7, Snapshot (a) $m_r = m_\emptyset$, and* Inv(f_1^V) *has no free variables. According to line 7 in procedure* ADDVIEWACCESS, $m'_r = \{\$f \mapsto \$f_2, \$r \mapsto \$r_2, \$as \mapsto \$as_2\}$ *is constructed to record the actual names of the variables introduced as copies of those in* Inv(f_1^V). *In Snapshot (b)* ADDVIEWACCESS *is called with m'_r.* ADDVIEWACCESS *now extends P^Q with a copy of* Inv(f_2^V), *yielding the pattern from Snapshot (c). Notice that this copy is added below the $\$as_2$ variable due to lines 5–6 in* ADDVIEWACCESS*; m'_r has recorded that $\$as_2$ is the copy of the occurrence of $\$as$ in* Inv(f_1^V), *and the same copy is consistently used for the occurrence of $\$as$ in* Inv(f_2^V).

KEEPVIEWSONLY($X, \bar{V}$)
1 **for** each block B^X in query X, let P^X be its pattern
2 Replace P^X with the restricted pattern P^{RW} which keeps from P^X only variables reachable along paths of /- and //- edges from roots mentioning view names. Keep the edges and equalities involving them.
3 Drop from B^X's **groupby** list all variables which are:
– uniquely determined by other variables in the list (due to parent-child relationship), and
– do not appear free in nested blocks
4 Replace the remaining variables v in B^X's **groupby** list with variables $u \in P^{RW}$ such that:
5 v **eq** $u \in P^X$;
6 if v is a groupby-id variable, u was introduced during Phase 1 as the correspondent of a view's groupby-id variable;
7 if v is a groupby-value variable, u was introduced due to either a view's groupby-value or groupby-id variable;
8 **if** not all groupby variables can be replaced,
then report "no rewriting found"
9 **if** an ordered rewriting is sought and no ordering of the view pattern roots preserves the pre-replacement ordering of **groupby** variables
then report "no ordered rewriting found"

Figure 11: Phase 2 of Algorithm NEXTREWRITE

THEOREM 3.1. *Let the input to* NEXTREWRITE *be a NEXT query Q and a set $\bar{V}$ of invertible NEXT views.*

1. (Soundness) *The NEXT query* RW *output by* NEXTREWRITE *(if any) is an equivalent rewriting of Q.*
2. (Unordered Completeness) *If Q's binding stage Q^{bind} has some equivalent unordered NEXT rewriting using only $\bar{V}$, then* NEXTREWRITE *is guaranteed to find an equivalent rewriting* RW *of Q using only $\bar{V}$.*

It should not be surprising that we cannot guarantee completeness for ordered rewritings: in this case, even the equivalence of ordered NEXT queries is undecidable (this is a corollary of results in [29]). Our approach performs a best effort in that case, remaining sound (the equivalence check is now only sufficient, not necessary) and in practice still finding ordered rewritings in many cases.

3.5 Finding Mappings Efficiently

We specify next how the algorithm NEXTREWRITING finds mappings. This is the most crucial step for the performance of the algorithm since (i) it is expensive (indeed, it is the only step that is not of polynomial-time complexity) and (ii) is invoked repeatedly.

We first present the prior work on finding mappings. For relational conjunctive queries, checking the existence of a mapping from V to Q is NP-complete in the number of variables of V [6]. The NP-hardness lower bound transfers to finding mappings from a NEXT pattern P^V into a NEXT pattern P^Q.[4] However, checking

[4]Sketch of Proof: Given a relational mapping problem, assume that the relations are encoded in XML using, for instance, the default encoding of [5]. Then encode the relational conjunctive queries as single-block (i.e. non-nested) NEXT queries over the default encoding. It follows that finding NEXT mappings is as hard as finding conjunctive query mappings.

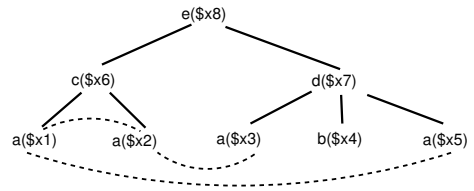

Figure 12: A NEXT Pattern ($\$x_5$ is the groupby-value variable)

containment mappings between simple tree patterns without value equalities used in prior XPath works is performed in polynomial time in the size of the source pattern [2, 26, 22, 16]. Notice that the differences between simple tree patterns and NEXT patterns are (i) the number of distinguished variables, i.e., variables that are projected in the output (tree patterns have only one, NEXT patterns have as many as there are groupby variables) and (ii) the number of variable equalities (tree patterns have none). It is therefore natural to seek an algorithm that runs in PTIME on tree patterns (no equalities, one distinguished variable) and whose performance degrades only with increasing number of groupby variables and/or equalities.

We start by analyzing the complexity of computing mappings from a pattern P. For each (node and corresponding) variable $v \in P$, define *gVars*(v) to be the set of grouping variables among the proper descendants of v in P. For the example pattern in Figure 12, assuming that $\$x_5$ is the only groupby-value variable, *gVars*$(\$x7) = \{\$x5\}$. Also, let $e(v)$ be all of v's proper descendants in P which are involved (directly or transitively) in equalities with variables who are not v's descendants. For instance, $e(\$x7) = \{\$x3, \$x5\}$. The symmetric, transitive closure of the equality conditions in P partitions its variables into *equality equivalence classes*. Given a set of variables $\mathcal{V}$ denote with *#eqCls*$(\mathcal{V})$ the number of distinct equality equivalence classes represented in $\mathcal{V}$. For instance, *#eqCls*$(\{\$x7, \$x3, \$x5\}) = 2$, since $\$x3$ and $\$x5$ are in the same equivalence class.

Define the *width* of v as

$$width(v) = \begin{cases} \#eqCls(gVars(v)), & \text{if } v \text{ is root} \\ \#eqCls(\{v\} \cup gVars(v) \cup e(v)), & \text{otherwise} \end{cases}$$

and define *width*(P) as the maximum width of a variable in P. For example, *width*$(\$x7) = 2$ and *width*$(\$x8) =$ *#eqCls*$(\{\$x5\}) = 1$.

THEOREM 3.2. *Let P^V, P^Q be NEXT patterns. Then checking the existence of a mapping $m : P^V \rightarrow P^Q$ is NP-complete in $width(P^V)$ and polynomial in the size of P^Q.*

This result follows from the fact that *width*(P) is related to the *tree width* [8, 15] of P when regarded as a graph whose edges consist of the /-, //-edges and the equalities.

For queries with bounded tree width, previous work [8, 15] presents polynomial-time evaluation algorithms corresponding to the generalization of Yannakakis' algorithm [1] from acyclic queries to bounded-tree-width queries. Our algorithm for computing mappings adapts these ideas. It is based on two key ideas. First, instead of computing mappings from P^V to P^Q one-at-a-time, we take a set-at-a-time approach by interpreting P^V as a query evaluated over the tree structure of P^Q.[5] P^V is compiled into a query execution plan in which we adapt from Yannakakis' algorithm [1] the idea of pushing projections before the join in the bottom-up evaluation of the plan. Our experiments show that the plan compilation and subsequent optimization require negligible time. In addition, this time is not critical as these tasks are performed off-line, before query arrival.

[5]This perspective stems from relational theory, in which mappings (homomorphisms) from a query V to a query Q are computed by regarding Q as a symbolic database and evaluating V over it [6].

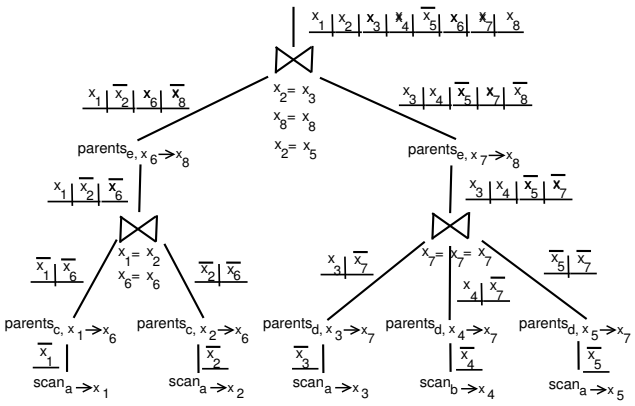

Figure 13: Algebraic plan for pattern in Figure 12

The operators used in the mapping computation plan output flat relational tables. We use relational join, selection, and projection operators, as well as two special operators that are evaluated over the structure of P^Q. First, $\mathsf{scan}_{l \mapsto v}()$ returns the nodes in P^Q whose label is l. The result is a set of unary tuples whose attribute is called v. Second, $\mathsf{parents}_{l, c \mapsto p}(R)$ takes as input a set of tuples R and extends each of them with a new attribute p. Given a tuple r with $r.c = n$, where n is a node of P^Q, the new value $r.p$ is the l-labeled parent n' of n, assuming n' and n are connected via an /-edge in P^Q. If no such parent exists, the tuple is dropped. Similarly for $\mathsf{ancestors}_{l, d \mapsto a}$, d is the attribute holding the descendant and a is the extension attribute holding an ancestor with label l.

The plan for a pattern P is obtained by using scan operators for the leaves of P, and join operators for its internal nodes. Moreover, for each operator o corresponding to a pattern variable v, add on top of o a projection operator whose projection list contains precisely the variables contributing to $width(v)$.

EXAMPLE 3.9. *We illustrate the translation into a plan on the pattern P from Figure 12. Given a pattern P^Q, the plan shown in Figure 13 finds all images of $\$x_5$ under mappings from P into P^Q. Each plan operator o is adorned with a schema S, whose overlined attributes A specify the projection list. The invariant maintained by the plan is that the sub-plan rooted at o computes the images of the A variables under all partial mappings of the S variables into P^Q. For instance, the topmost join operator finds the images of x_5 under the mappings of all eight variables of P.*

The join operators enforce the equality conditions, as well as the structural constraint requiring two variables to have the same parent variable. A condition $x = y$ evaluates to true on two tuples t_l (from the left join operand) and t_r (from the right join operand) if $t_l.x$ is equal to $t_r.y$ in P^Q; equality may be transitively implied by P^Q's equality conditions. The reader can check that in Figure 13, the number of overlined attributes for each operator o corresponds to the width of v.

THEOREM 3.3. *Let P^V, P^Q be NEXT patterns. Then the evaluation over P^Q of the plan for P^V produces all mappings $m : P^V \rightarrow P^Q$ and runs in time polynomial in the size of P^Q and exponential in $width(P^V)$.*

Since pure tree patterns contain no equalities and only one distinguished variable (corresponding in NEXT to one groupby-id variable), simple tree patterns have width 1, and the result on polynomial-time computability of tree pattern mappings [2] follows as a corollary of Theorem 3.2.

4. BEYOND NEXT REWRITING

We now extend the applicability of the NEXT rewriting algorithm to arbitrary XQueries. This involves (i) extending the NEXT notation to accommodate arbitrary XQueries and (ii) extending the rewriting algorithm accordingly.We sketch the required extensions in this section. We preserve the approach used for NEXT rewriting: represent queries using patterns and find view access paths by matching the view patterns against the query patterns.

Extending NEXT: NEXT+. The solution we adopt for (i) is to abstract the non-NEXT subexpressions as *uninterpreted functions*, i.e. functions about whose semantics we make no assumptions. This allows us to uniformly capture all XQuery primitives which are ruled out by the OptXQuery syntax: aggregate functions, built-in predicates other than equality, universal quantification, negation, disjunction, user-defined functions, etc. Each of these are treated as some function F whose arguments are in turn NEXT expressions enriched with uninterpreted function calls. We call this notation *NEXT+*. Its syntax corresponds to extending the NEXT grammar in Figure 4 with the productions

$FC ::= F(XQ_1, \ldots, XQ_l)$

$XQ ::=$ **for** Var_1 **in** $(FC_1 \mid Path_1), \ldots, Var_n$ **in** $(FC_n \mid Path_n)$ (**where** *CList*)? **groupby** $FC'_1, \ldots, FC'_k$ **return** FC''

Cond ::= *FC // where the function F is (coercible to) boolean*

NEXT+ contains NEXT in the particular case when only paths appear in the **in** clauses, $FC'_1, \ldots, FC'_k$ are calls of the identity function with variables as arguments, and the function in FC'' consists only of element construction and concatenation. The NEXT+ notation features additional nested block occurrences, where blocks are now not only by **groupby** blocks, but also by the arguments of function calls. For instance, in the above production for XQ,, the arguments of $FC_1, \ldots, FC_n, FC'_1, \ldots, FC'_k, FC''$ are function blocks nested within the block given by the outer **for** –**where** –**groupby** –**return** expression. Just like NEXT blocks, nested NEXT+ blocks may have free variables bound in their ancestor blocks. For example, consider the following query which returns the reviews of multi-author papers.

for $\$p$ **in** $\$doc//paper, \$r$ **in** $\$p/review$
where $count(\$p/author) > 1$ **return** $\$r$

Normalization into NEXT+ yields two blocks B_1, B_2:

$$B_1 \begin{cases} \textbf{for } \$p \textbf{ in } \$doc//paper, \$r \textbf{ in } \$p/review \\ \textbf{where } F(\underbrace{\textbf{for } \$a \textbf{ in } \$p/author \textbf{ groupby } [\$a] \textbf{ return } \$a}_{B_2}) \\ \textbf{groupby } [\$p], [\$r] \textbf{ return } \$r \end{cases}$$

where F is the boolean function $\lambda N.\ count(N) > 1$ (in lambda notation).

Extending NEXTREWRITE. Since algorithm NEXTREWRITE operates on nested blocks regardless of how these were created during normalization, only minimal changes are required. We still use mappings to detect access paths, still considering ancestor blocks to provide the context for the free variables. However, we must extend mappings to account for function calls. Our treating functions as uninterpreted requires extending Definition 3.1 such that function calls $F(a_1, \ldots, a_n)$ match only against calls of the same function $F(b_1, \ldots, b_n)$, and only if, recursively, a_i is equivalent to b_i for each i. In our example, the algorithm generates the following candidate rewriting using view V from Example 1.1:

for $\$f$ **in document** $(V)/feedback, \$r$ **in** $\$f/review$
where $count(\$f/authors/author) > 1$ **groupby** $[\$r]$ **return** $\$r$

Clearly, the more of the query is abstracted to functions, the less mappings will be discovered, resulting in fewer rewriting opportunities. We take particular care to maximize these opportunities by abstracting only the truly non-NEXT query primitives.

The advantage of our approach is twofold. First, we do not reject non-NEXT queries flat out, performing a best rewriting effort instead. Our approach combines the benefits of complete rewriting feasible for NEXT and the easy-to-engineer but incomplete techniques based on isomorphically matching common sub-expressions between query and view [12]. Second (and beyond the scope of this paper), we enable an extensible rewriting module, which can be incrementally enhanced by adding partial information about the uninterpreted functions (thus interpreting them partially).

5. EXPERIMENTAL EVALUATION

As shown in Section 3.5, the complexity of our algorithm is determined by the pattern width, a measure which is typically much smaller than the query size. Our experiments show that other factors such as query size and number of views do not affect the rewriting performance significantly, allowing algorithm NEXTREWRITE to scale up to large numbers of views and large queries.

We implemented a generator of synthetic queries and views which enables us to control the following parameters: the nesting depth d of the query, the breadth b (see below) of the patterns in each **groupby** block, and the number of views.

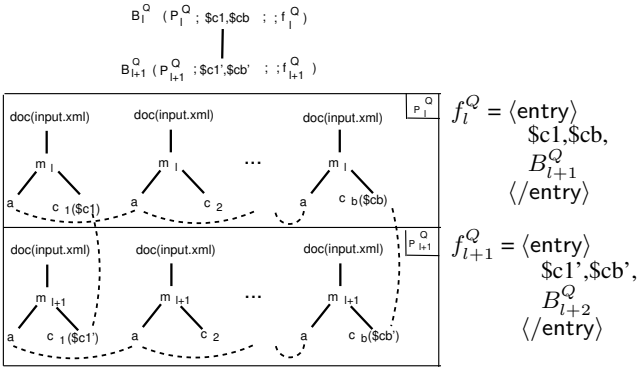

Figure 14: General form of synthetic queries and views

The queries. For a given value of d and b, the generator outputs a query $Q_{d,b}$ as follows. $Q_{d,b}$ has d **groupby** blocks $B_1^Q \ldots B_d^Q$, such that B_{l+1}^Q is nested within B_l^Q for each $l \in [1 \ldots d-1]$. The patterns P_l^Q and P_{l+1}^Q of blocks B_l^Q, respectively B_{l+1}^Q are shown in Figure 14, using the NEXT notation. They consist of b basic patterns, where the jth basic pattern navigates from the document root to an m_l child, from there to an a and a c_j child. In the figure, we only show the variable $\$c_j$ bound to this latter child. We call the *breadth of a block* the number of basic patterns it contains. The basic patterns are chained via value-equalities between the variables binding to a elements. Variables $\$c_1$ and $\$c_b$ are the groupby-value variables of block B_l^Q, and are also output by the return function f_l^Q. These variables also appear as free variables in block B_{l+1}^Q, which performs a value-equality join between them and its own groupby-value variables, $\$c'_1$ and $\$c'_b$.

The views. We note that the detection of views which are irrelevant to the query can be done very fast, by establishing the absence of mappings from view into query. This can be detected early, as soon as some internal operator of the view pattern plan returns an empty answer. Hence an evaluation of how the algorithm scales with the number of irrelevant views would produce excellent re-

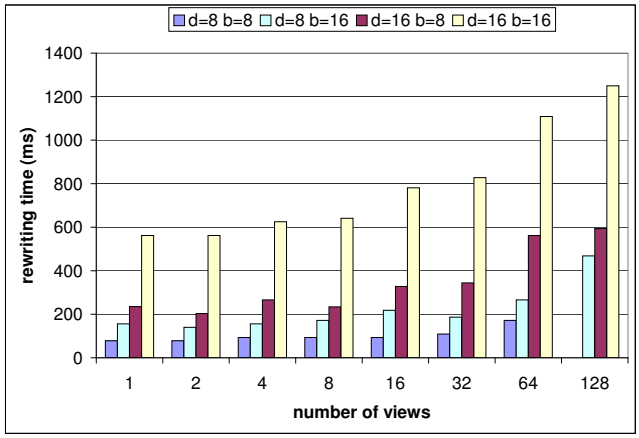

Figure 15: Rewriting times for increasing number of views.

sults but would not test the real challenges. We avoid irrelevant views by generating views exclusively from subpatterns of $Q_{b,d}$.

We start with a view that is identical to the query, and we recursively split it into smaller views by alternatively halving its depth (the depth of the **groupby** tree) and breadth (the number of basic patterns in each **groupby** block). At each recursive step, the pattern of the views at each level corresponds to a subpattern of the original query. Towards a realistic scenario, we force the patterns of the views to overlap: when obtaining two new blocks B_1 and B_2 by halving a view block on its breadth (B_1 is the left half), we extend B_1's pattern with a copy of the leftmost basic pattern of B_2.

By construction of the views, the query always has a rewriting.

The platform. Our experiments were all run on a Pentium 4 2.80GHz running Windows XP with 1GB RAM. The algorithm was implemented in Java.

The measurements. Our preliminary experiments show very encouraging results. Figure 15 shows the results of running experiments on patterns as described above for a number of views between 1 and 128. We display the results for four configurations, corresponding to the queries $Q_{8,8}, Q_{8,16}, Q_{16,8}, Q_{16,16}$. In particular, $Q_{16,16}$ contains 16 nested blocks, each of whose patterns contains 16 value-equality conditions and binds 48 variables of which 2 are groupby-value variables. In each configuration, we measure the rewriting time when increasing the number of views by successive splits as described above. For 128 views, the test runs in 594 ms for a query with 16 nested levels and a breadth of 8 basic patterns per level. For the query of depth 16 and breadth 16 and the same number of views, we measure a rewriting time of 1250 ms.

Conclusions. While we intend to conduct experiments on real-life queries and views (the challenge there is to collect query and view specimens actually deployed in applications), our preliminary experiments are quite encouraging. They show that the algorithm performs well for large queries and large numbers of overlapping views. The reasons for this performance is the exploitation of the underlying tree structure for quickly finding mappings. By not considering irrelevant views, we generated the worst case scenario for our algorithm in order to "stress-test" it. In practice, we expect much better behavior, as many views will be irrelevant and ruled out immediately.

6. RELATED WORK

In XQuery stream processing, [12] identifies common XQuery subexpressions and memoizes in a cache to avoid redundant evaluation. This can be seen as rewriting the original XQuery using the views in the cache. The test for pattern equivalence is based on expression isomorphism and thus trades rewriting opportunities for engineering simplicity. As we discussed, one can maximize rewriting opportunities within the NEXT/OptXQuery set, while retaining syntactic isomorphism for NEXT+/XQuery. NEXT rewriting allows us to match query and view navigation *across* XPath sub-expressions, regardless of whether it appears in the **for** or the **where** clause of a query, or as **distinct-values** argument. None of these matches are supported by pure syntactic isomorphism.

[3, 33] rewrite only the XPath subexpressions of XQueries using materialized XPath indexes and thus do not face the problems caused by nesting, equalities, XML construction in the view output, all of which we address in this paper.

The type of nesting we address here corresponds in the SQL relational case to the illegal nesting in the SELECT clause, and was therefore not a focus of SQL optimization. While this nesting is allowed in OQL, the development of complete algorithms for rewriting OQL queries using views is precluded by the fact that checking equivalence of nested OQL queries is an open problem even for the idealized conjunctive OQL sublanguage of [20]. We are aware of only one particular case of view-based OQL rewriting (which however does not involve nesting), namely rewriting OQL paths using path indexes [28, 18]. Note that other kinds of XQuery or OQL nesting (within the FOR and WHERE clauses) are easier to deal with, as it is in most cases translated away using normalization rules such as in Agora [21] (adopted in our work as well), and the normalization from [9] for moving nested **some** loops from the **where** clause into the **for** clause (see NEXT_Q).

[25] rewrites semistructured queries using semistructured views in the context of the semistructured OEM data model. The copy semantics of XQuery's construction operators rule out the use of database id's for assembling the view data into the query result, as done in [25].

In the context of data integration, [34] rewrites XQueries under open-world, "certain answer" semantics using source-to-target constraints. The work does not address equivalent rewritings in a closed world, which is our setting. The two settings requires very different algorithms even in the relational case.

In XML publishing, the Agora [21] and MARS [10] systems both rewrite XQueries using publishing views, but indirectly via a reduction to rewriting relational queries using relational views (Agora) or constraints (MARS). Consequently, they do not address list and bag semantics. The strength of the two approaches is to allow mixing of relational and XML models by reducing the XQuery treatment to a relational one. A benefit of our NEXT-based approach is that, by exploiting the underlying tree pattern structure, we achieve faster algorithms for pattern matching. The relational reduction misses this opportunity. Moreover, Agora and MARS first decorrelate queries into unnested queries. Each unnested query is then individually rewritten and evaluated using relational techniques and the results are put together using an outer join. Hence the solution forces the processor to use decorrelation and outer joins at the physical level. In contrast, we provide a tool which inputs XQuery and outputs an XQuery rewriting, thus imposing no requirements on the underlying engine, with obvious benefits of portability and of allowing the cost-based optimizer to subsequently choose a physical level plan.

The equivalence checker is a basic building block for any rewriting algorithm. There is a significant body of work on equivalence of XPath (tree patterns) ([22, 2, 26, 11, 32, 23, 14]). The only work we are aware of on equivalence of nested XML queries was reported in [9] and adopted in this paper. While not complete for checking equivalence of entire queries, the algorithm of [9] is complete for checking equivalence of *binding stages*, which is ultimately what

we want to rewrite. [13] completely solves checking containment for a class of nested XML queries subsumed by ours. However, it addresses a containment flavor based on homomorphic embedding of the resulting XML trees. Unfortunately, equivalence is not reducible to this kind of containment (two queries may be contained in each other without being equivalent). Moreover, the test for this containment flavor has inherently higher complexity (Π_2^p-complete if applied to NEXT patterns) than for the binding stage containment we use (NP-complete in the pattern width). One may wonder how we check binding stage equivalence under bag semantics, given that containment under bag semantics is open for conjunctive queries (but Π_2^p-hard) and undecidable for unions thereof [7]. Fortunately, in XML nodes have unique identities, and for a NEXT query, a bag of element values corresponds to a *set* of their identities, which allows us to reduce the test to set containment of tuples consisting of values and identities. Finally, we do not inherit the open problem of checking equivalence for OQL [20]. As shown in [20], equivalence and containment are not inter-reducible, except for OQL queries yielding VERSO relations [1]. These are essentially obtained as the result of grouping, and therefore are produced by the bind stage of NEXT queries.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we present a sound algorithm for rewriting using views for XQuery. The algorithm is complete for the large sublanguage OptXQuery, when unordered rewritings are of interest. We obtain good performance by exploiting the tree-like structure of OptXQuery navigation patterns.

Our rewriting algorithm was enabled by the integration of a proven relational approach (use a pattern-based representation of queries and views to detect subsumed query sub-expressions via pattern mappings) with solutions for the special XQuery challenges such as nested return clauses, two types of equalities, unrestricted mix of loops with set, bag and list semantics, and the copy semantics of XML construction. None of these challenges arose in prior XPath rewriting work.

A note about the copy semantics: our example shows that despite the fact that XQuery loses node identities when it constructs the result, there are interesting rewriting opportunities, which our algorithm discovers. Given (XQuery-inexpressible) views which preserve node identities, even more such opportunities would be exposed. A simple extension of our algorithm would easily exploit these opportunities: during query expansion (Phase 1), connect query and view variables with id-equality edges (as opposed to only value-equality in the current form). We can show that, with this modification, the previous soundness and completeness guarantees apply to id-preserving views.

Note that our algorithm yields a single rewriting, obtained by finding all (possibly redundant) alternate data accesses through views. This is already sufficient in certain scenarios, such as privacy-preserving publishing (where one only cares about the existence of a rewriting), or in distributed processing in which evaluating a redundant query over the local cache is likely preferable to sending even non-redundant queries to remote sites. We are working on integrating the rewriting algorithm with the NEXT query minimization techniques of [9] and the work on cost-based pruning during minimization described in the MARS system [10].

Finally, we are working on extending the rewriting in the presence of XML Schema and DTD constraints, which create even more rewriting opportunities. Our NEXT-based rewriting approach is compatible with this extension, exploiting pattern mappings to find the query sub-expressions affected by constraints.

8. REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

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

[3] A. Balmin, F. Ozcan, K. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized xpath views in xml query processing. In *VLDB*, 2004.

[4] K. Beyer, D. Chamberlin, L. Colby, F. Ozcan, H. Pirahesh, and Y. Xu. Extending XQuery for analytics. In *SIGMOD*, 2005.

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

[6] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[7] S. Chaudhuri and M. Y. Vardi. Optimization of real conjunctive queries. In *PODS*, 1993.

[8] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In *ICDT*, 1997.

[9] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT logical framework for XQuery. In *VLDB*, 2004.

[10] A. Deutsch and V. Tannen. MARS: A system for publishing XML from mixed and redundant storage. In *VLDB*, 2003.

[11] A. Deutsch and V. Tannen. Reformulation of xml queries and constraints. In *ICDT*, 2003.

[12] Y. Diao, D. Florescu, D. Kossmann, M. Carey, M. Franklin. Memoization in a streaming xquery processor. In *XSym* 2004.

[13] X. Dong, A. Y. Halevy, and I. Tatarinov. Containment of nested XML queries. In *VLDB*, 2004.

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

[15] J. Flum, M. Frick, and M. Grohe. Query evaluation via tree-decompositions. In *ICDT* 2001.

[16] G. Gottlob and C. Koch. The complexity of XPath query evaluation. In *PODS*, 2003.

[17] A. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4):270–294, 2001.

[18] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *VLDB*, 1990.

[19] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, 1995.

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

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

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

[23] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, dtds and variables. In *ICDT*, 2003.

[24] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogenous information sources. In *ICDE*, 1995.

[25] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *SIGMOD*, 1999.

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

[27] S. Rizvi, A. Mendelzon, S. Sudarshan, P. Roy. Query Rewriting Techniques for Fine-Grained Access Control. In *SIGMOD* 2004.

[28] P. Valduriez. Join indices. *ACM TODS*, 12(2):218–452, June 1987.

[29] S. Vansummeren Deciding Well-Definedness of XQuery Fragments. In *PODS* 2005.

[30] W3C. XML Query Use Cases . Available from http://www.w3.org/TR/xmlquery-use-cases/.

[31] W3C. XQuery: A query Language for XML. Available from http://www.w3.org/TR/xquery.

[32] P. T. Wood. Containment for XPath fragments under DTD constraints. In *ICDT*, 2003.

[33] W. Xu and Z. M. Özsoyoglu. Rewriting XPath Queries Using Materialized Views. In *VLDB*, 2005.

[34] C. Yu and L. Popa. Constraint-based XML query rewriting for data integration. In *SIGMOD*, 2004.