WPI-CS-TR-05-01 Feburary 2005

# Algebraic XQuery Decorrelation with Order Sensitive Operations

by

Song Wang
Xin Zhang
Elke A. Rundensteiner
Murali Mani

# Algebraic XQuery Decorrelation with Order Sensitive Operations

Song Wang, Xin Zhang, Elke A. Rundensteiner and Murali Mani
Department of Computer Science, Worcester Polytechnic Institute
Worcester, MA 01609, USA
(songwang|xinz|rundenst|mmani)@cs.wpi.edu

**Abstract**

XQuery, the defacto XML query language, is typically composed of highly nested expressions. Iterative execution of such expressions tends to be intuitive but inefficient. Instead, decorrelation of nested XQuery expressions opens up the opportunity for significant query optimization. Although several algorithms have been proposed to optimize nested XQueries, these works pay little attention to the ordered semantics of XQuery expressions. The appropriate extension of decorrelation to XQuery with multiple level orderby clauses and other order sensitive functions hence represents an important and non-trivial task. We propose an algebraic rewriting technique of nested XQuery expressions containing explicit orderby clauses. The proposed work successfully addresses the challenges caused by the hierarchical nature and the ordered semantics of XQuery expressions. Using a running example, our decorrelation algorithm is illustrated. Further, we show the performance gain achievable by our approach via an experimental study.

## 1 Introduction

The XQuery language [21] and the XML path language [20] have both been widely accepted for querying XML data. Several optimization techniques have been proposed for XPath expressions, such as XPath containment [7], answering XPath queries using views [1] and XPath satisfiability [10]. Beyond the features of XPath, XQuery provides more powerful query support by nested expressions having correlated variable bindings, and multiple level ordering overwriting the source document order. On the other hand, the direct applicability of the well known optimization techniques to the XQuery language [1] is precluded by these new features. How to enable the usage of existing optimization techniques to XQuery processing with nested and order sensitive semantics becomes an important and non-trivial task. Our work in this paper intends to provide a practical approach to fill the gap between the existing work of XPath query optimization and the XQuery optimization with order semantics.

XQuery expressions are typically composed of highly nested FLWOR (short for the *for*, *let*, *where*, *orderby* and *return*) blocks to retrieve and reconstruct hierarchical XML data. We call such nested XQuery expression *correlated* if an inner FLWOR block refers to a bound variable defined outside this block. The intuitive method

[1] In this paper, we use the term XQuery to refer to the complex XQuery expression that cannot be rewritten as an XPath expression.

of an iterative execution of correlated queries tends to be less efficient than an equivalent optimized set-oriented execution strategy. Decorrelation has been used as an effective approach for optimization of nested queries in relational databases [9, 4, 16].

Unlike in relational databases, order is important for XML queries. By default, both the XPath and XQuery languages are order sensitive. The XPath language has order sensitive functions such as $position()$, $first()$ and $last()$. All the order sensitive functions used in the XPath language work on the document order. In addition XQuery expressions may contain multiple *orderby* clauses that overwrite the document order. XQuery decorrelation needs to preserve the ordered semantics.

Some early works on XQuery unnesting were done by Fegaras [6] and May et al. [12]. Fegaras extended the rule-based unnesting algorithm used in Object Oriented databases to the XQuery language in stream processing without considering order. Inspired by [3], May et al. provided a rule-based rewriting solution for XQuery unnesting based on an order preserving algebra. However, they did not discuss order beyond the document order and how the order affects the unnesting. These previous works provided rewriting rules for only certain classes of nested XQuery expressions. These works did not solve the XQuery unnesting problem under the order semantics thoroughly, especially for the multiple level orderings and order sensitive functions.

Below we list the issues that must be considered for efficient XQuery decorrelation.

- Typically, an XQuery expression may have multiple level orderby clauses among subqueries. The decorrelation approach must preserve the order specifications in the original XQuery.
- In case of duplicate elements existing in variable bindings, there may be repeated computation. For better performance, the decorrelation approach must avoid such repeated computations.
- The Left Outer Join operator has been widely used to avoid incorrect removal of tuples of the outer query block after decorrelation. In XQuery decorrelation, existence of empty collection makes this problem much more common. The decorrelation approach must keep the original query semantics.

Considering these issues, in this paper we propose an algebra based decorrelation algorithm that not only works correctly for arbitrarily nested XQuery expressions with multiple orderby clauses, but also generates an efficient query plan. Our work can be easily extended to existing algebra based query engines. Our work is inspired by the magic decorrelation proposed by Seshadri et al. [16]. Here the authors proposed a SQL decorrelation method designed to generate more efficient rewritten queries for complex correlated SQL queries.

The basic idea in [16] is that first the duplicate free values of the outer referenced column are extracted. Then based on these values, all possible results from the subquery are computed once and then materialized. Thereafter materialized results are joined with the outer query block on the outer referenced column to restore the duplication.

Our approach, called *Magic branch*, is an extension and adaption of this technique towards more efficient XQuery decorrelation. The new challenges come from: (1) multiple level orderings in XQuery vs. single level ordering (at most) for SQL, and (2) the hierarchical nature of XML vs. the flat relational tables. For the first challenge, we treat the multiple level orderings as special "aggregate functions" and process accordingly. For the second challenge, some extensions are needed for correctness. Consider the following XQuery.

```
for $a in Doc("a.xml")/a
return <tag>
       {for $b in Doc("b.xml")/b
        where $b = $a/c
        return $b}
       </tag>
```

The reason of adding an extra *Distinct* to the Magic Set decorrelation in [16] is that the correlated column may not be the key of the table and duplications may exist. In above XQuery example, the navigation ($\$a/c$) may generate more complex situation due to the hierarchical nature and multi-set semantics of XML. Consider two instances of $\$a$ having distinct node ID $a_1$ and $a_2$ in the input XML document. Suppose that the navigation $\$a/c$ generates tuples: ($a_1, c_1$), ($a_1, c_2$), ($a_2, c_3$) and ($a_2, c_4$). Since the predicate of the Where clause in above XQuery is comparison on values, we need to retrieve the string value for each of the $\$a/c$ nodes. Assume that $c_1$, $c_2$ and $c_3$ have the same string value $v_1^c$, while $c_4$ has the string value $v_2^c$. To save repeated computation in the subquery, one naive way of using magic decorrelation here is adding an extra value based *Distinct* upon the $\$a/c$ and conducting a value based Join with $\$b$. Suppose $b_1$ has string value $v_1^c$ and $b_2$ has string value $v_2^c$. After rebuilding the duplications by a value based Join (or Left Outer Join if the count bug exists) with the whole $\$a/c$, the result tuples are: ($a_1, c_1, b_1$), ($a_1, c_2, b_1$), ($a_2, c_3, b_1$) and ($a_2, c_4, b_2$). According to the semantics of the example XQuery, two problems need to be fixed: (1) $b_1$ should not appear two times for $a_1$, even if it joins with different $\$a/c$ nodes; and (2) $b_1$ and $b_2$ should be combined together for $a_2$. Such situation cannot be handled by original Magic Set decorrelation.

Another observation about the naive application of magic decorrelation in the above example is that the Left Outer Join with $\$a/c$ may not remove all the "count bugs". It can only deal with the situation that some $\$a/c$ instances may not have any matched $\$b$ instance[2]. The semantics of the example XQuery require the empty tags in

[2]Since there is no aggregation in this particular query, the Left Outer Join is actually a Join only.

the result even if some instances of $\$a$ do not have any $\$a/c$ instance at all. Thus another Left Outer Join with the whole sequence of $\$a$ may be necessary.

Our work successfully addresses these challenges caused by the hierarchical nature of XML documents. By employing grouping operations (see Section 4.4), repeated computation is avoided correctly; and by utilizing a ID based Outer Join with the binding variable only when necessary, the correctness and efficiency are achieved. Our work brings forth the following novel contributions to XQuery decorrelation:

- We extend and adapt the magic decorrelation method from SQL to the XQuery language, and propose an algebra based decorrelation algorithm.
- Our approach preserves the correct order semantics across multiple level nested XQuery expressions.
- We propose to use groupings to avoid repeated computations in XQuery expressions.
- Our algorithm proposes a hybrid evaluation distinguishing ID based operators and string value based operators for optimization purposes.

We employ an algebraic framework discussed in Section 3. This algebra extends relational algebra by allowing collection-valued columns and being order-preserving. It also introduces new operators to express necessary XQuery semantics. However, the main idea of our approach is generic and can be applied to other similar algebras like NAL [12] and SAL [2].

This paper is organized as follows. We first give a description of the related work in Section 2 and briefly describe the algebraic framework used in this paper in Section 3. A running example illustrating the magic branch decorrelation algorithm and a detailed explanation of our algorithm is discussed in Section 4 and 5. In Section 6 we focus on several additional issues specific to complex XQuery decorrelation. We turn to implementation and present experimental results in Section 7, while Section 8 concludes this paper.

## 2 Related Work

Modern database systems [9, 5, 16] attempt to merge subquery blocks into the outer query block, thereby eliminating correlations and avoiding nested iteration evaluation. Such “decorrelation” is typically done by introducing outer join and grouping operations.

More recently, methods that focus on the efficiency of decorrelated subqueries have been proposed. In [16], the authors proposed a technique called magic decorrelation for nested SQL queries. By materializing results from subqueries and postponing the Outer Join, this approach produces a typically more efficient query plan. Our proposal is conceptually inspired by this technique.

Decorrelation of XQuery expressions has also been studied in relationship to native XML query engines. One effort is by Paparizos et al. [13] in the TIMBER system. Here the authors pointed out the implicit use of grouping constructs in the XQuery's result construction. Recognizing and explicitly adding the grouping operation can lead to unnesting of XQuery expressions. Their work is based on the tree algebra in TIMBER. However they do not consider ordering and their grouping operator is defined on sets of trees, which makes it expensive. Another drawback of this approach is that their transformation from the XQuery language to the TAX tree is complex and not complete, as pointed out in [12].

Fegaras [6] and May et al. [12] have studied XQuery unnesting based on the unnesting techniques from object-oriented query languages [3, 5]. These works do not discuss decorrelation of XQuery expressions containing *orderby* clauses, which is now tackled by our work.

There are also some works in the literature about publishing XML documents from relational data that mention decorrelation in processing. In [17], magic decorrelation is extended and used in XPERANTO to decorrelate the XML view query and user XQuery. This approach is closest to our proposal, while the major difference is that the correlated attribute eventually comes from a column of the underlying flat relational table, instead of XML nodes of subtrees. Thus certain challenges we identified, such as the order issue have not been observed.

## 3 Preliminaries

**XQuery:** In this paper, we consider a subset of the XQuery language [21] defined by the grammar in Fig. 1. This subset, plus some extensions of user-defined functions, suffices to express the XMark benchmark query set [15]. Besides the basic *FLWOR* clauses, the XQuery fragment we consider also includes order-related functions (e.g., the position function), and quantifiers. Our approach is not applicable to recursively nested XQuery expressions.

**XA Algebra:** Our XML algebra (*XA*) expresses the subset of the XQuery language shown in Fig. 1. XA is an order-preserving extension of the relational algebra designed to handle ordered XML data. For the purpose of decorrelation, this algebra is similar to NAL [12], SAL [2] and the algebra proposed in [14]. Hence our approach can be easily extended to these algebras.

XA extends the relational algebra mainly in two key aspects: order-preservation and nested tuples. XA works on sequences of tuples where every column corresponds to a variable or to a generated column of an intermediate

| | | | |
|---|---|---|---|
| $Expr$ | ::= | $c$ | //atomic constants |
| | | $\$var$ | //variable |
| | | $(Expr, Expr)$ | //sequence construction |
| | | $Expr/a :: n$ | //navigation step (axis a, node test n) |
| | | $tag(Expr)$ | //element constructor: tagger |
| | | $FLWOR$ | //query block |
| | | $QExpr$ | //expression with quantifier |
| | | $CompExpr$ | //comparison expression for predicates |
| | | $OrderExpr$ | //order-sensitive function. eg. position() |
| $FLWOR$ | ::= | $(For \mid Let)^+$ $[Where]$ $[Orderby]$ return $Expr$ | |
| $For$ | ::= | for $\$var$ in $Expr$ | |
| $Let$ | ::= | let $\$var$ := $Expr$ | |
| $Where$ | ::= | where $Expr$ | |
| $Orderby$ | ::= | order by $Expr$ | |
| $QExpr$ | ::= | (some \| every) $\$var$ in $Expr$ satisfies $Expr$ | |
| $CompExpr$ | ::= | $Expr$ $CompOp$ $Expr$ | |
| | | //CompOp is any comparison operator. eg. "=" | |

Figure 1: Syntax of XQuery Subset

result. To avoid ambiguity, we assume all the variable and column names are unique in processing of the XQuery. We use the *XATable* to represent such ordered sequences of tuples. The input and output of each operator are both XATables. An XATable may contain nested tuples, that is, the content of a column may be a sequence of zero or more tuples. In other words, XATable allows collection-valued columns. The collection-value can be set, multi-set (bag) or sequence of atomic data types according to XQuery semantics. The formal definition of the XATable schema is given next.

**Definition 1** *Let $T = \{a_1, ..., a_n\}$ be a finite set of columns. XATable schema $R$ over $T$ is defined recursively as:*

*1. If $a_1, ..., a_n$ are atomic columns, then $R = (a_1, ..., a_n)$ is an XATable schema over $S$ having $depth(R) = 0$.*

*2. If $a_1, ..., a_n$ are atomic columns and $R_1, ..., R_m$ are XATable schemas having columns denoted as $A(R_1), ..., A(R_m)$, then $R = (a_1, ..., a_n, R_1, ..., R_m)$ is an XATable schema.*

Since XA is not designed for type inference purposes, we only have two kinds of atomic values in an XATable: the ID of an XML node, which globally identifies the XML node, and the string value of an XML node, which is the text content of the subtree rooted at the XML node. We distinguish the ID based operations from the string value based operations. The XML data storage provides conversion functions from the node ID of the input XML document to the associated string value. For newly constructed XML nodes, the storage has a skeleton function to assign unique ID to the nodes.

To define the order-preserving semantics of XA operators, we will use a sequence abstraction of the XATable. For an input XATable $R$, $h(R)$ denotes the first tuple (head) of the XATable and $t(R)$ denotes the remaining tuples (tail) of the XATable. The symbol $\oplus$ is used for the concatenation (ordered union) of two XATables. The concatenation of XATable columns is denoted by $\circ$. We define the algebraic operators recursively on their input XATable(s). For binary operators, we use left hand side (LHS) and right hand side (RHS) to distinguish between the two input XATables. We use $\epsilon$ to denote an empty XATable.

The XA algebra inherits all operators from the relational algebra, such as *Select* ($\sigma_p$), *Project* ($\Pi_{Attr}$), *Join* ($\bowtie_p$), *Left Outer Join* ($LOJ$, $\ltimes$), *Natural Join* ($NJ$, $\boxtimes$), *Cartesian Product* ($CP$, $\times$), etc. Except for the addition of order preserving semantics, these operators have similar semantics as in the relational context.

For the XQuery function *distinct-values()*, we introduce a value-based duplicate elimination operator $Distinct$. This operator is not order preserving and has semantics identical to its relational counterpart. We also have an ID-based duplicate removal operator $\delta_{ID}$. We also define the operators: *Orderby* and *Position*. The Orderby operator sorts the tuples in the input XATable by the string value of specified column(s). The Position operator gets the row number (beginning from 1) of each tuple and puts it as explicit value into a new column.

The XA algebra also introduces new operators to represent the XQuery semantics, such as *Navigation* ($\phi_{xp}$), *Tagger* ($Tag_{Pattern}$), *Nest* ($N$), *Unnest* ($U$), etc. We will define these operators below.

Since in this paper we do not focus on complex XPath processing, we use a “powerful” Navigation operator that can extract XML nodes and process XPath expressions over XML documents. We denote the Navigation operator as follows:

$$\phi_{\$col_j:xp(\$col_i)}(R) := (h(R) \times R_{Nav}) \oplus \phi_{\$col_j:xp(\$col_i)}(t(R))$$

where the schema of $R_{Nav}$ is $\{col_j\}$, $R_{Nav}$ is the sequence of extracted XML nodes from the XML node in $col_i$ of $h(R)$ by applying XPath $xp$ processing.

The Tagger operator accepts a pattern to add open tags and close tags around the content of certain columns in the input XATable. A pattern includes the tag’s name and associated column names. Unlike the Groupify-GroupApply operators [18], our Tagger operator is a simple operator. The Tagger does not build the result hierarchy; instead the result structure is built by a sequence of grouping, nesting and Tagger operators.

Given a tuple with a sequence-valued column $Attr$, we define the Unnest operator as:

$$U_{Attr}(R) := (h(R)_{\vdash_{Attr}} \times R_{Attr}(h(R))) \oplus U_{Attr}(t(R))$$

where $\vdash_{Attr}$ removes the $Attr$ column from $R$ and $R_{Attr}(h(R))$ retrieves the sequence of column values in $Attr$.

The Nest operator, the inverse of Unnest, can be defined accordingly. Note that the XQuery expressions have an interesting property that the result of each XQuery expression is merged into a single sequence by dropping empty elements. The Nest operator captures such semantics by removing null elements from input XATable.

To clarify the translation of FLWOR expressions into the XA algebra, we introduce the *Map* operator. The Map operator basically implements the for-iteration. It is a binary operator with the LHS input XATable capturing the query expression of the for-variable and the RHS an algebra expression $e$. The Map operator is defined as follows:

$$Map_{a:e(Attr)}(R) := (h(R) \circ a) \oplus Map_{a:e(Attr)}(t(R))$$

where $a$ is the new column whose value is calculated by applying expression $e$ to the instance of $Attr$ of $h(R)$. Intuitively the Map operator forces a nested loop evaluation strategy. At the algebraic level the goal of our work is to remove all Map operators and to do so correctly.

The last operator discussed here is the *Groupby* ($GB$) operator, denoted as $GB_{col_i;col_j;op}(R)$. This operator is introduced for the purpose of decorrelation. This GB operator is an extension of the groupby in the relational context. The Groupby operator will group the tuples of the input XATable by the column $col_i$, then perform the operator $op$ on $col_j$ of each group of tuples, finally concatenate all the groups together as output. The Groupby operator can group on multiple columns.

In our magic branch decorrelation, we use the Groupby operator for the following three purposes: (1) To keep the correct variable context during decorrelation. (2) To build hierarchical XML results. (3) To get duplicate-free columns in order to remove repeated computations.

Some operators are only ID based, eg. the Navigation and Map, while some operators are both ID-based and value-based. For example, the Groupby operator can grouping on the string values of the $col_i$, or on the ID value of the $col_i$. We will explicitly distinguish ID-based and value-based operators in this paper.

**XQuery Normalization:** Prior to translating the XQuery expressions into the XA algebra expression, we use a syntax-level normalization step applied to the original XQuery expressions. Similar normalizations are also discussed in [11]. To focus the discussions in this paper, the Let clauses are treated as a special case of the For clause. The For clause defining more than one for-variable will be split into a sequence of nested For clauses.

**Translating Normalized XQuery Expressions to XA Algebra:** Normalized XQueries are translated into their corresponding XA algebra representation in two steps: translating XPath expressions and translating the FWOR (without the Let clause) query expressions.

The translation pattern of a flat FWOR query block to the XA algebraic expression is illustrated in Fig. 2. A nested XQuery block can be translated recursively using this pattern. In this translation pattern, the Map operator introduces one for-variable from the for clause in the LHS expression. This for-variable can be referred to in the nested query blocks in the RHS. The Orderby operator sorts all the returns and the Nest operator on top of it is used to construct a sequence of all intermediate results. In this pattern, we put the Where clause to the RHS of the Map operator. Here we take the same pattern as the translation pattern of the correlated join in the XPERANTO system [17]. Such pattern provides a more general support for complex Where clause translation, e.g. a Where clause including position function. For the simple Where clause, the Where clause can also be put in the LHS of the Map operator, after the For clause.

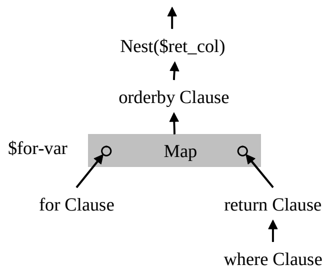

Figure 2: Build Algebra Tree for XQuery FWOR Expression.

The algebraic operators generated during translation form an XA algebra tree. We also allow the sharing of common subexpressions (e.g., multiple use of a let-variable) among multiple operators. This turns the XA tree into a DAG. In this paper, we do not emphasize the difference between them and just generally call them XA tree.

## 4 Magic Branch: Running Example

The flexibility of XQuery brings new challenges to XQuery decorrelation, such as: (a) complex nested subqueries can appear at any position in the FLWOR expression, (b) subqueries may have multiple correlated variables, (c) correlated variables can be referred to in multiple levels of query blocks, (d) subqueries may have universal quantifiers, aggregate functions, result constructors and order-sensitive functions. In this section we propose an algorithm for decorrelating nested XQuery expressions by transforming the corresponding XA algebra tree. After such rewriting, the original semantics of the XQuery (including the order semantics) are preserved.

After XQuery normalization and translation, the correlation in an XQuery expression is represented in the XA tree by the *Map* operator and *linking* operators. The Map operator introduces the for-variable from the LHS For clause and the linking operator refers to it in the RHS. Intuitively the Map operator forces a nested loop evaluation strategy. Hence, eliminating the nested loop iteration, that is, the Map operator in the XA tree transformation is the main goal of the proposed decorrelation algorithm. Besides correctness, generating an efficient decorrelated XA tree is also important. As mentioned before, our techniques are an extension of magic decorrelation [16]; and these extensions are designed to ensure correct and efficient XQuery decorrelation.

Briefly speaking, our approach performs the following three optimization steps during decorrelation. (1) The Map operator on the top of the XA tree will be pushed down along the LHS whenever possible. (2) The Groupby operator is inserted to keep the correct context for special operators like Nest and aggregation functions. (3) The Groupby operator is also used to remove duplication from the input XATable of the linking operator and the Unnest is used to restore the original semantics.

We perform our tree transformation by pushing down or pulling up operators and by inserting or removing operators in the XA tree. Furthermore, each transformation step maintains the semantic consistency of the given query. Thus this algorithm can be stopped at any time generating a partially decorrelated query. This offers more control to the query optimizer to consider benefit and cost trade-offs of various decorrelation decisions determined by the physical plan of operators.

Below we show our magic branch decorrelation and optimization using a running example. The normalized XQuery in Fig. 3 will be used as our running example to illustrate the Magic branch decorrelation. The XA algebra tree for this query is shown in Fig. 4. In all the XA trees in this paper, we will use underlined operator to indicate value-based operations. Other operators are ID-based operations. For simplicity, we omit the transfer functions from ID to string value in the XA tree.

```
for $a in doc("a.xml")/a
order by $a/c
return <tag>
          {for $c in doc("c.xml")/c
           order by $c/e
           where $c=$a/d
           return $c/e}
       </tag>
```

Figure 3: Example XQuery having Correlated Subquery.

## 4.1 Pushing Computation out of the RHS

The idea here is to separate out computation from the RHS of the Map operator and perform it outside the RHS: either in the LHS or on top of the Map operator. If we think of the RHS computation of the Map operator as a function operating on an input variable $\$x$ that ends by applying a selection $\sigma$ on $\$x$, then this selection can be

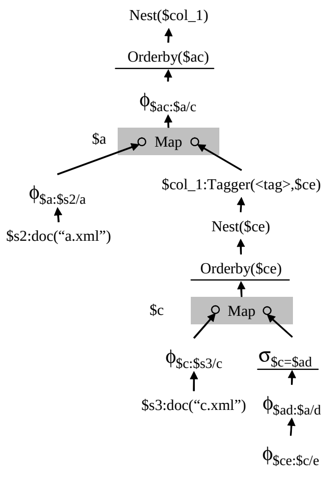

Figure 4: XA Tree for Example XQuery in Fig. 3.

applied in the LHS of the Map. We then can find a minimal subset of $\$x$, in terms of the selection, so that we can perform the RHS computation correctly.

Intuitively if a unary operator $Op_1$ in the RHS of the Map does not "use" any column generated by any of its descendant $Op_2$, and all operators "in between" $Op_1$ and the Map (ancestors of $Op_1$ but descendants of the Map) do not use any column of $Op_2$ either, then we can push $Op_1$ and all upper operators above the Map.

The illustration of this step is shown in Fig. 5. Here the RHS of the lower Map operator in Fig. 4 is pushed above the Map operator. As a consequence, the Map operator would have an empty RHS. Such Map operator can be removed.

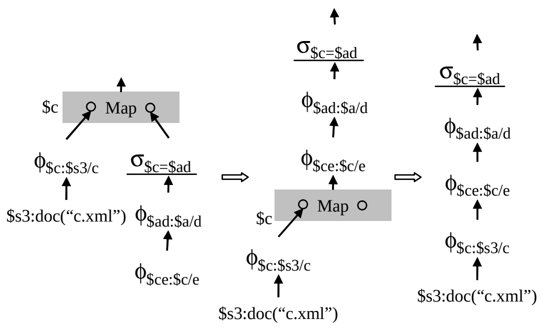

Figure 5: Removing Map after Pushing up Computation.

## 4.2 Generation of the Magic Branch

Now we focus on removing the Map operator whose LHS introduces the variable $\$a$. First we informally define the linking operator. We say that $Op$ is a *linking operator* for its ancestor $Map$ operator on variable $\$x$ if $Op$ "uses" this correlated variable $\$x$ to refer to a column in the LHS input XATable of $Map$. $Op'$ is (recursively) said to be *relatively correlated* to its ancestor $Map$ if a descendant of $Op'$ is a linking operator to $Map$.

In the running example, $\sigma_{\$c=\$ad}$ is a linking operator and all operators between it and the Map in the RHS are relatively correlated operators. In the case of multiple-level correlation, one Map operator may have multiple linking operators in the RHS. The linking operators and relatively correlated operators can be obtained by one Depth First Search (DFS) traversal of the XA tree.

In this generation step, we first make a copy of the LHS of the Map operator and then prepare to propagate this *Magic branch* to the linking operators. Here we traverse the XA tree in a DFS manner. The magic branch is propagated beginning from the "highest" relatively correlated operator and ends with the "lowest" linking operator. In Fig. 6, we add the generated magic branch to the XA tree using a new Map operator immediately on top of the current operator. To ensure correctness, we need a Select operator on top of the new Map. Otherwise the correspondence relationship between the correlated variables in the original XQuery will be lost. This Select and the original Map can be merged into a LOJ, which is used to restore the order of the result defined originally by the for-variable $\$a$.

Different with the original Magic decorrelation for nested SQL queries, we do not insert the Distinct operator in the branch below the Rename. As pointed out in Sec. 1, such Distinct may not preserve the original XQuery semantics. Instead, we use a Groupby operator and an ID-based duplicate removal operator to avoid repeated computations, as shown later in Fig. 9.

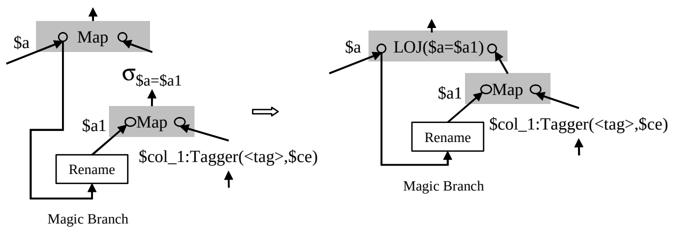

Figure 6: Generate the Magic Branch

## 4.3 Propagation of the Magic Branch

The intuitive idea of the magic branch decorrelation is to move all relatively correlated operators out of the correlation. As a result the linking operator and the Map operator will become parent and child. Then the linking operator can be rewritten into either a Join, Semijoin or Antijoin, depending on the query semantics. To achieve this, we need to propagate the Map operator down the XA tree to the linking operator.

The propagation of the magic branch over *tuple-oriented* operators is different from that over *table-oriented* operators (See Definition 2 in Section 5). A table-oriented operator is defined on groups of tuples and its result cannot be calculated without the group boundary. All the operators inherited from relational algebra except aggregations, as well as many other operators such as Tagger, are tuple-oriented. Aggregations, Nest and order-sensitive position functions are table-oriented.

For a tuple-oriented operator we can simply push the *Map* down over it as shown in Fig.7 for the Tagger.

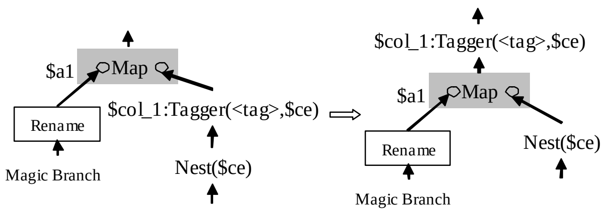

Figure 7: Propagation over Tuple-Oriented Operator: Tagger

For table-oriented operators, we need to perform an extra generation of a magic branch as done in Section 4.2. The propagation step for the table-oriented operator Nest is shown in Fig. 8. The intuition is to insert another LOJ to keep completeness of the for-variable column in the output XATable. Without such LOJ the empty collection bug may occur, which is similar but more general to the count bug already studied in the literature [8]. By repeatedly performing the generation (if needed) and propagation as we traverse the XA tree downwards, we will eventually push all the relatively correlated operators out of the RHS of the Map.

Now the *Orderby* operator coming from the orderby clause become the RHS child of the Map operator. The Orderby operator is a special table-oriented operator that will sort all the tuples within the input table. As for other table-oriented operators, propagating the Magic Branch will add a Groupby operator and a LOJ operator to the query tree. Different with other table-oriented operators, the added LOJ operator will also be used to restore the correct tuple order from the LHS magic branch, since the original tuple order may be destroyed by the Groupby operator to remove repeated computations.

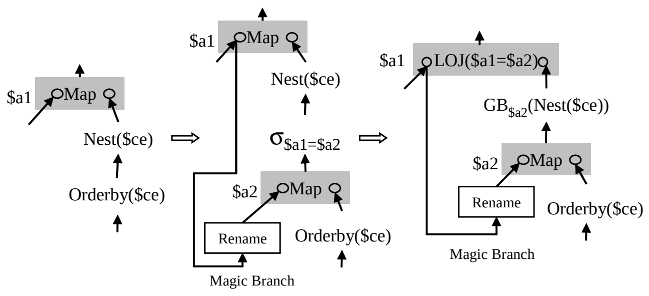

Figure 8: Propagation over Table Oriented Operator: Nest

## 4.4 Absorption of Magic Branch

Now the linking operator $\sigma_{\$c=\$ad}$ becomes the child of the Map operator. The last portion of the propagation is to absorb the magic branch into the linking operator. A Join (or Semijoin, Antijoin) is formed to connect both the branches. To form the branches, other operators may also be moved accordingly. For example, the $\phi_{\$ad:\$a3/d}$ is moved from RHS to LHS in Fig. 9. In order to avoid repeated computation by the Join operator, a Groupby operator grouping on the referred variable in the Join and a ID-based duplication removal operator are inserted. At the same time, an Unnest is also inserted to keep the semantic consistency. This step guarantees that no repeated computation exists for the Join operator. This transformation of the XA tree is shown in Fig. 9.

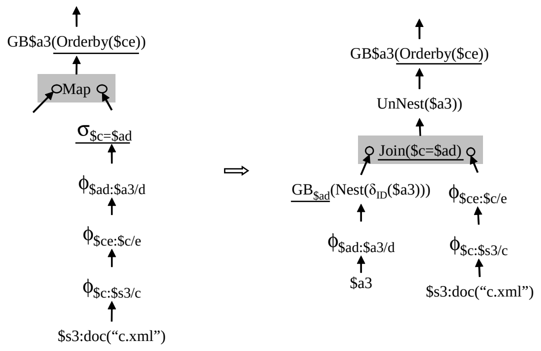

Figure 9: Absorption of Magic Branch by Linking Operator

## 4.5 Removal of Redundant Left Outer Join

During the magic branch generation and propagation, the LOJ operator is generated to rebuild the tuple order defined by the for-variable and to maintain completeness of the for-variable column in the XATable. Some of the LOJs may be redundant and could be removed. We perform a top down traversal of the XA tree and remove the redundant

LOJ operators. In the example XA tree, the top LOJ can be removed since the lower LOJ already would restore the correct tuple order and the Tagger operator is order preserving. As to the other two LOJs, there is no empty collection associated with a certain instance of $\$a$ that would get lost. Either one is sufficient to avoid the empty collection issue.

We obtain the final decorrelated XA tree as shown in Fig. 10. During execution, the content of column $\$a$ and its alias $\$a2$ are all XML node IDs. Thus a cheap ID-based LOJ can be achieved.

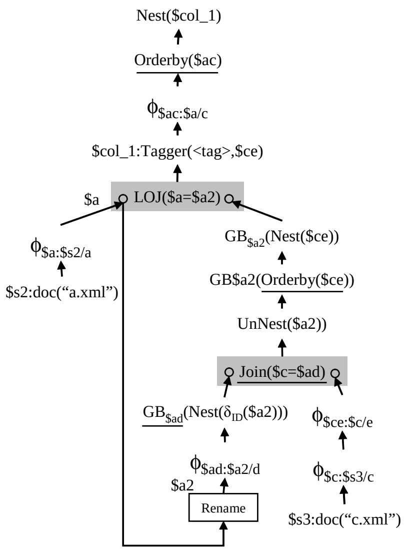

Figure 10: Decorrelated XA algebra tree

## 5 Magic Branch: The Basic Algorithm

In Section 4, we have illustrated our Magic branch decorrelation algorithm using a running example. For description purposes we follow an algorithm treating all nested XQueries uniformly. That is, we first generate all LOJs and then later remove unnecessary ones. Actually in the implementation, these two steps are done together without generating redundant LOJs.

### 5.1 Pushing Computation out of the RHS of Map Operator

The idea here is to factor out computation from the RHS of the Map operator and perform it outside the RHS: either in the LHS or on top of the Map operator. For this purpose, we first identify the subset of the columns in the input XATable that an operator or block of operators needs to function correctly. We call the XATable schema composed of this subset of columns the *min-schema* for that operator or the block of operators. The min-schema indicates all the input columns used by the operator(s), directly or indirectly. The min-schema can be computed recursively as follows:

- For each leaf operator, the min-schema is the correlated for-variable column or empty (if this operator refers to an XML document).
- For any other operator, min-schema includes the column name(s) it uses in its operation and the union of the min-schema(s) of the operator(s) generating the column(s).
- For a block of operators, min-schema is the union of the min-schema of every operator in the block minus the columns generated by the operators in the block.

Now we can move a block of operators out of the RHS of the Map using the following rule.

**Rule 1** *A block of operators can be pushed out of the RHS of the Map operator, if: 1) the min-schema of this block of operators only includes the correlated for-variable column, 2) any output column generated by the operators does not appear in the min-schema of other operators outside the block in the RHS, and 3) there is no table-oriented operator in this block.*

The above rule enforces that no table-oriented operator can be moved outside the RHS of Map. This prevents the empty collection bug that could result from such transformation. We have to use the magic branch to decorrelate such operators, as shown previously in Fig. 8. After pushing computation out of the RHS of the Map, the RHS XA tree might be empty. In that case the Map operator can be removed by the following rule.

**Rule 2** *If the RHS subtree of a Map operator is empty, this* `Map` *operator can be removed.*

For a simple XQuery having flat FLWR clauses only, a linear XA tree is expected after using Rules 1 and 2. Example is shown in Fig. 5.

## 5.2 Generating and Propagating the Magic Branch

The main task of our decorrelation algorithm is to generate the magic branch and to propagate the branch down the RHS subtree. Introducing an extra Select and Map pair to the RHS of the Map operator will happen in the generation step. Similarly introducing the Select and Map pair also happens when propagating the magic branch down a table-oriented operator. Eventually the propagated magic branch reaches the linking operator and can be rewritten into a Join or a variant of the Join operator. During the magic branch decorrelation, extra LOJs will be generated to restore the order of tuples and to avoid the empty collection bug. We show the correctness of the generation and propagation steps below.

**Lemma 1 (Generation Lemma)** *Consider a query $Q$ having a Map $M_{a:e_R}(\$v_0)$ with $e_{L_0}$ as LHS appearing under a Nest. Now consider another Map $M'_{a:e_R}(\$v_1)$ with $e_{L_1}$ as LHS and $e_{L_1}$ is a magic branch renaming $\$v_0$ as $\$v_1$. Introducing $M'$ into $e_R$ and a Select $\sigma_{\$v_0=\$v_1}$ as parent of $M'$ does not change the evaluation of $Q$. In other words:*

$$Nest(...M_{a:e_R}(\$v_0)) = Nest(...\$v_0 \; LOJ_{\$v_0=\$v_1} \; M'_{a:e_R}(\$v_1)).$$

Intuitively Lemma 1 is correct since: (1) the LOJ inserted is ID-based operator, and both $\$v_0$ and $\$v_1$ are ID distinct; (2) the upper Nest removes null elements from input XATable.

In the propagation step of the magic branch, we need to deal with tuple-oriented and table-oriented operators differently.

**Definition 2** *A tuple-oriented operator is one whose output corresponding to a tuple in the input XATable depends only on that tuple. Any operator that is not tuple-oriented is said to be table-oriented. In other words, the output of a table-oriented operator depends on multiple tuples of the input XATable.*

The table-oriented operators in our algebra include: Nest, OrderBy, Groupby, Distinct and all relational aggregation functions. Since “order” semantics in XQuery have to be defined on a sequence of tuples, all order-sensitive operators are classified as table-oriented operators also.

Table-oriented operators need to be handled differently during propagation of the magic branch. We use the for-variable defined in the magic branch as the correlation context for the table-oriented operator.

**Lemma 2 (Propagate Lemma)** *The Map $M_{a:op(e_R)}(\$v_0)$ appearing under a Nest, where $op$ is a table-oriented operator whose correlation context is $\{\$v_0\}$ can be rewritten as:*

$$Nest(...M_{a:op(e_R)}(\$v_0))$$

$$= Nest(...\$v_0 \; LOJ_{\$v_0=\$v_1} \; (GB_{\$v_1;a;op} \; M'_{a:e_R}(\$v_1))).$$

*with $M'$ having $e_{L_1}$ as LHS and $e_{L_1}$ is a magic branch renaming $\$v_0$ as $\$v_1$.*

Intuitively the added grouping operator in Lemma 2 separates the whole column used by the table-oriented operator $op$ into partitions according to the context variable $\$v_1$. Thus each partition keeps the group boundary of the column correctly. For the tuple-oriented operators, no partition of the used column need to be kept and thus the magic branch can be simply propagated down.

## 5.3 Remove the Redundant Left Outer Join Operator

The main reason for introducing the additional LOJ operator during decorrelation are: 1) to restore the order of the result defined originally by the for-variable, and 2) to avoid the empty collection bug caused by the table-oriented operators.

In SQL query decorrelation, whenever aggregation functions occur between query blocks, the unnested join query needs to consider unmatched outer tuples to avoid the "Count Bug". The unmatched tuples of the outer query will generate a subquery result equal to the empty set. Whenever such an empty set is significant in the predicate expression or the result construction, we need to introduce a LOJ to handle these unmatched outer tuples.

In the XQuery language, such problem will happen not only if aggregate functions occur between query blocks, but also in many other cases as well. We call this the "Empty Collection Bug". For example, consider the following XQuery.

```
<result>{
for $book in (doc("bib.xml") //book)
return <book>$book[2]</book>
}</result>
```

Since the position function will always return "1" for every iteration of $\$book$, the expected result is a collection of empty "<book/>" tags. This number of empty tags should be equal to the number of instances of the $\$book$ variable. Without using `LOJ`, the `Select` operator upon the `Position` operator will remove all the tuples in the XA table and the final result will be wrong. We have to use `LOJ` to find all those $\$book$ instances in order to get the correct number of "<book/>" tags.

In the previous running example, we use a conservative approach of adding `LOJ` for every table-oriented operator to guarantee the correctness of the transformation. All of these LOJ operators have the magic branch as left hand side input. Some of these LOJ operators are redundant and can be removed as discussed below.

We first traverse the XA tree in a top-down manner to find all redundant LOJs. A LOJ $L_1$ is redundant with respect to a LOJ $L_2$ in the RHS of $L_1$ if: (1) $L_1$ and $L_2$ have the same magic branches with the same variable (possibly renamed); (2) all the operators above $L_2$ in the RHS of $L_1$ will not remove any instance of the variable introduced by the LHS of $L_2$; and (3) all the operators above $L_2$ in the RHS of $L_1$ will not make use of the possibly empty collection inserted by $L_2$. When we also consider the order preserving purpose of using the LOJ operator, the following additional condition needs to be checked: (4) all the operators above $L_2$ in the RHS of $L_1$ is order preserving.

The redundant LOJs are removed as follows. We first form a undirected graph with the LOJs as nodes. We define an edge between $L_i$ and $L_j$ if $L_i$ is redundant with respect to $L_j$. Then, from each component of this graph, we keep only the topmost LOJ in the query tree, and discard all the other LOJs.

In XA algebra, the Select, Join and variants of Join are example operators that may remove the instance of the variable introduced by the magic branch; and the Tagger operator is one example operator using the empty collections generated by the lower LOJ.

This step needs to traverse the XA tree multiple times. In the worst case, the complexity of the step will be $O(ln)$, where the $l$ denotes the number of the LOJs and the $n$ denotes the number of nodes in the XA Tree.

## 6 Complex XQuery Decorrelation

In this section we will discuss the application of the magic branch decorrelation on complex XQueries. These XQueries may have aggregations, order-sensitive functions, quantifications and multiple level nested subqueries.

### 6.1 Aggregation and Position Functions

XQuery language supports all the aggregation functions supported in the relational context, such as *min*, *max*, *count*, etc. The aggregations attract special attention in the relational context because they are table-based operations and usually are associated with a grouping operation. In our magic branch decorrelation, all aggregation functions are classified as table-oriented operators and thus can be processed consistently. In order to keep the partition boundary of the aggregated column, we need to add an explicit grouping operator after the propagation. In order to avoid the infamous "count bug", we need to introduce a LOJ for aggregation functions.

Different from the relational and the object data model, the XML data model is order-sensitive. In the XQuery language, there are many order-sensitive functions to support this feature. Among them, the sequence-based XQuery expressions are popular, such as the Position function. We will use the Position function as an example to show how to use the magic branch to decorrelate order-sensitive functions in XQuery expressions.

According to Definition 2, the Position function is a table-oriented operator. Using parentheses in XQuery expression, the context of the Position function can be set differently. Consider the following two XQueries; they have different context for the Position function defined by the parentheses. In XQuery $Q1$ the second $name$ of every $author$ will be returned, while in XQuery $Q2$ the second $name$ node rooted by $book$ will be returned.

```
XQuery Q1:
for $b in doc("bib.xml")/book
return <name>$b/author/name[2]</name>
XQuery Q2:
for $b in doc("bib.xml")/book
return <name>($b/author/name)[2]<name>
```

The context of the Position function needs to be maintained before and after decorrelation. To achieve this, we need to propagate the magic branch over the GroupBy operator. The GroupBy operator is also a table-oriented operator. The XA trees before and after decorrelation of the GroupBy and Position functions are shown in Fig. 11 and 12 respectively.

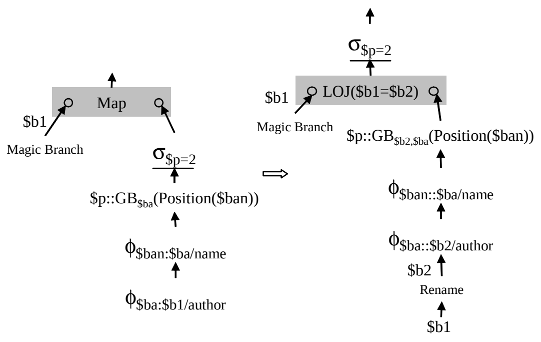

Figure 11: Decorrelate Position Function in XQuery Q1.

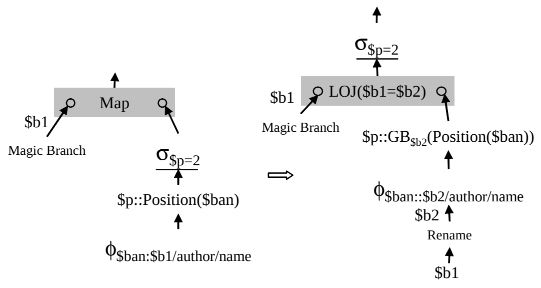

Figure 12: Decorrelate Position Function of XQuery Q2.

## 6.2 Existential and Universal Quantifications

In XQuery, we allow existential and universal quantifiers to appear in the predicates. The quantifiers used in the XQuery Where clause include *some* and *every*. The primitive *satisfies* is associated with the quantifiers. XQuery expressions can also use the relational-like *exists* primitive to test the emptiness of a collection.

All the predicates having quantifications can be rewritten using the aggregate function Count and the Select operator. Such XQuery expressions then can be decorrelated using the magic branch decorrelation. The XQuery having the *exists* primitive on a collection can also be rewritten using the Count operator. We use the following example to show the rewriting of the *some* quantifier. The XQuery $Q3$:

```
for $a in doc("a.xml")/a
where some $b in doc("b.xml")/b[c=$a/c]
      satisfies $a/b = $b
return <tag>$a</tag>
```

can be rewritten as $Q4$:

```
for $a in doc("a.xml")/a
where count(for $b in doc("b.xml")/b[c=$a/c]
            where $a/b = $b return $b) >0
return <tag>$a</tag>
```

If the query engine support the efficient evaluation of the Semijoin and the Antijoin operators, the XQuery $Q3$ can also be transformed during decorrelation using these Join variants.

## 6.3 Multiple Level Correlation

A complex XQuery can have multi-level nested subqueries having multiple linking operators. There are multiple Map operators in the translated algebra tree connecting the correlation variables and linking operators. In general, we will face a bushy Map tree. For such XQuery, we decorrelate all the Map operators recursively in a post-order traversal of the Map operators. That is, for each Map operator, we first decorrelate the LHS, then the RHS and lastly the subtree rooted by this Map operator. Such post-order decorrelation avoids pushing a Map operator over another Map operator.

# 7 Experimental Study

We have conducted experiments to illustrate the performance gains achieved by our approach. We have implemented the magic branch decorrelation and optimization algorithm in the RainbowCore project, a native XQuery engine based on the XA algebra developed at WPI [22].

We have run a set of experiments to evaluate the performance gain achieved by the magic branch unnesting algorithm. Experiments were run on a 1.2GHz PC with 512MB of RAM running Windows 2000. We used XML documents generated by the XMark benchmark [15] with 10 scale factors from 0.001 (113KB) to 0.01 (1.11MB). Each experiment was run five times, with lowest and highest results discarded and and the remaining three averaged.

We propose a general solution for nested XQueries, and could be applied in any algebraic based XQuery engine. Horizontal comparison with other engines will not be instrumental in evaluating the effectiveness. Thus, we conducted a vertical evaluation comparison before and after application of our techniques on the same system.

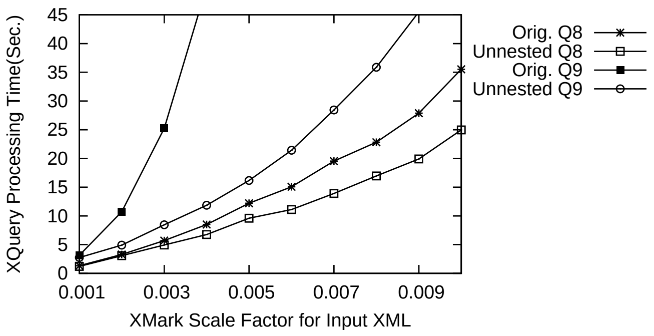

Figure 13: Comparison Before and After Decorrelation.

In our first set of experiments, we use two XQueries from the XMark benchmark, namely, *Q8* and *Q9*. Each represents a different category of complexity, namely, two query blocks with one level nesting, and three query blocks with multiple level nestings. The results are shown in Fig. 13. We also conduct experiments on XMark query *Q10*, *Q11* and *Q12* with results similar in overall trend confirming the effectiveness of our decorrelation. Other XQueries of XMark are either not nested or not supported by current RainbowCore due to the user defined functions.

We can see that the decorrelation step gives significant performance gains. One of the reasons is that in our experiment the navigations will be launched directly to the file for every instance of the LHS of the Map operators. After decorrelation, this repeated navigation in the subquery will be saved and the total I/O cost will decrease dramatically.

The second set of experiments try to show the effectiveness of decorrelation for XQuery with multiple level ordering and position functions. The following XQuery expression sorts part of the authors by their last name and groups books together with their first author, then sorts each author's book by publishing year. This query is adapted from W3C XQuery Use Cases XMP Q4 [19] by adding the position function and orderby clauses.

```
for $a in distinct-values(doc("bib.xml")/book/author[1])
order by $a/last
return <result>{$a,
                for $b in doc("bib.xml")/book
                where $b/author[1] = $a
                order by $b/year
                return $b/title
       }</result>
```

We have varied the input XML documents to have different numbers of book elements. The results are shown in Fig. 14. We clearly see the significant gains achieved by our unnesting algorithm, especially the more complex the nesting the larger the gain.

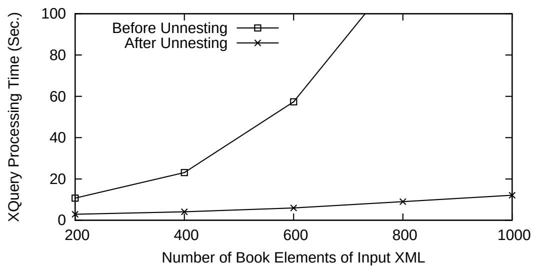

Figure 14: Performance Comparison of Different Query Plans.

The third set of experiments is designed to illustrate the performance gain of adding the Groupby operator to remove duplicates, we present a performance comparison with the naive magic decorrelation for XQuery expressions, where no duplicates are removed. In general, although our magic branch decorrelation introduces additional grouping operator, we expect our decorrelation to be beneficial when many duplicates existing in the variable referred to in the linking operator, or when there is considerable work performed for each variable instance in the linking operator. We measured the query execution times on the following XQuery.

```
for $a in doc("a.xml")/a
return {for $b in doc("b.xml")/b
        where $b=$a/b
        return $b}
```

The experiment has the following settings: the total number of $a$ nodes in the "a.xml" is set to 1K; the total number of $b$ nodes in the "b.xml" is also set to 1K. By varying the number of duplicates of $b$ nodes in $doc(a.xml)/a/b$, we get the performance comparison shown in Fig. 15. The Duplicates Ratio is defined as $1 - \frac{\#\_distinct\_b\_nodes}{\#\_total\_b\_nodes} \times 100\%$.

Our magic branch decorrelation results in a huge performance improvement, especially for the case where there is a large amount of duplicates.

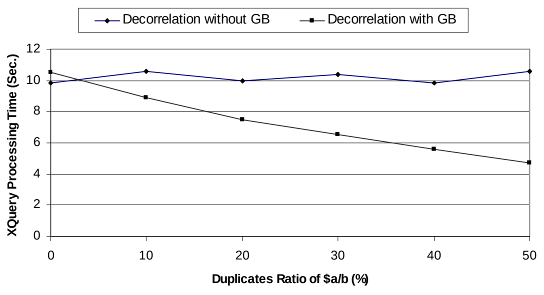

Figure 15: Decorrelation with/without Groupby I

We then lock the duplicates ratio of $\$a/b$ to $20\%$ and change the cardinality of $b$ node in "b.xml" from 1K to 6K, which increases the computation for each node in $\$a/b$. As shown in Fig. 16, the relative performance gain of our decorrelation increases accordingly.

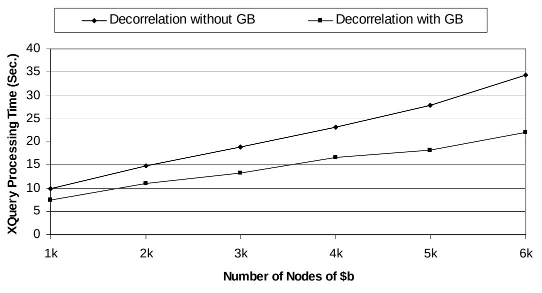

Figure 16: Decorrelation with/without Groupby II

## 8 Conclusion

In this paper we present a decorrelation algorithm, called *Magic branch* algorithm, based on an XQuery algebraic framework. Our work provides a deterministic algorithm for unnesting arbitrary nested XQuery with ordered semantics. Our work extends previous work in three aspects. First, our unnesting algorithm represents a uniform solution for removing correlated variables in a subquery. Second, our unnesting algorithm provides an efficient decorrelated query. This is possible by using a grouping operator to avoid repeated computations. Third, our unnesting algorithm preserves correct ordered semantics. The experimental studies illustrate the effectiveness of the proposed algorithm. As part of our future work, we plan to study the order inference of different operators in the order sensitive query plan as well as optimization of the operators using it.

## References

[1] A. Balmin, F. Ozcan, K. S. Beyer, R. Cochrane, and H. Pirahesh. A Framework for Using Materialized XPath Views in XML Query Processing. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 60–71, 2004.

[2] C. Beeri and Y. Tzaban. SAL: An Algebra for Semistructured Data and XML. In *ACM SIGMOD Workshop on the Web and Databases (WebDB)*, pages 37–42, 1999.

[3] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Proc. of the Int. Workshop on Database Programming Languages*, pages 226–242, 1993.

[4] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 197–208, 1987.

[5] L. Fegaras. Query unnesting in object-oriented databases. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 49–60, 1998.

[6] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. Query processing of streamed XML data. In *Proc. of the Int. Conf. on Information and Knowledge Management (CIKM)*, pages 126–133, 2002.

[7] G. Miklau and D. Suciu. Containment and Equivalence for an XPath Fragment. In *Symposium on Principles of Database Systems (PODS), Madison, Wisconsin*, pages 65–76, June 2002.

[8] W. Kießling. On Semantic Reefs and Efficient Processing of Correlation Queries with Aggregates. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 241–250, 1985.

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

[10] L. V. S. Lakshmanan, G. Ramesh, H. Wang, and Z. J. Zhao. On Testing Satisfiability of Tree Pattern Queries. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 120–131, 2004.

[11] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 241–250, 2001.

[12] N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 239–250, 2004.

[13] S. Paparizos, S. Al-Khalifa, H. Jagadish, L. Lakshmanan, A. Nierman, D. Srivastava, and Y. Wu. Grouping in XML. In *EDBT Workshops*, pages 128–147, 2002.

[14] C. Sartiani and A. Albano. Yet Another Query Algebra For XML Data. In *Proc. of Int. Database Engineering and Applications Symposium (IDEAS)*, pages 106–115, 2002.

[15] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 974–985, 2002.

[16] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 450–458, 1996.

[17] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML Views of Relational Data. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 261–270, 2001.

[18] G. M. Thorsten Fiebig. Algebraic XML Construction in Natix. In *World Wide Web Journal*, volume vol.4(3), pages 167–187, 2002.

[19] W3C. XML Query Use Cases, W3C Working Draft 02, May, 2003. http://www.w3.org/TR/xquery-use-cases.

[20] W3C. XML Path Language (XPath)Version 1.0. W3C Recommendation. http://www.w3.org/TR/xpath.html, March 2000.

[21] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, May 2003.

[22] X. Zhang, M. Mulchandani, S. Christ, B. Murphy, and E. A. Rundensteiner. Rainbow: Mapping-driven xquery processing system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, page 614, 2002.