Optimization of Nested XQuery Expressions with Orderby Clauses

Song Wang [*], Elke A. Rundensteiner, Murali Mani

Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609, USA

Abstract

XQuery, the defacto XML query language, is a functional language with operational semantics, which precludes the direct application of classical query optimization techniques. The features of XQuery, such as *nested* expressions and *ordered* semantics, further aggravate this situation. The appropriate extension of existing optimization techniques to XQuery processing hence represents an important and non-trivial task. We propose an algebraic rewriting technique of nested XQuery expressions containing explicit orderby clauses. Unlike prior work, this technique enables the optimization of nested XQuery expressions not only with set but also with ordered sequence semantics. Our technique is based on two steps. First, we perform order-sensitive algebraic query unnesting. Second, we apply query minimization techniques that exploit pairwise XPath set containment after pulling up order-sensitive operations. We illustrate how our proposed technique is able to not only successfully tackle the XQuery logical optimization problem solved in the NEXT framework, but also to correctly support ordered semantics. We have implemented the proposed optimization techniques on top of the XAT algebraic framework in our RainbowCore project. We show the performance gain achievable by our approach using an experimental study with the RainbowCore engine.

Key words: XQuery Processing, Ordered Semantics, XQuery Optimization

1 Introduction

The XQuery language [23] and the XML path language [22] have both been widely accepted for querying XML data. XPath expressions specify patterns to be matched

[*] Corresponding author.
Email address: `songwang@cs.wpi.edu` (Song Wang).

in the XML document, and return a sequence of XML elements. Beyond the semantics of pattern matching, XQuery [1] expressions are capable of performing complex querying and customizing output result construction. XQuery expressions utilize nested query blocks and explicit ordering clauses to achieve these features.

XQuery expressions are typically composed of highly nested FLWOR (short for the *for*, *let*, *where*, *orderby* and *return*) blocks to retrieve and reconstruct hierarchical and ordered XML data. An XQuery expression is said to be *correlated* if an inner FLWOR block refers to a bound variable defined outside this block.

Unlike in relational databases, order is an important issue 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 functions used in the XPath language work on the document order. Informally, document order is the order defined by a pre-order, depth-first traversal of the nodes in an XML document. In addition XQuery expressions may contain the *orderby* clause as part of a FLWOR expression that overwrites the document order for XML fragments generated by that XQuery expression based on explicit sorting. XML result structures generated by XQuery expressions can also be partially ordered. For example, the parent "books" are ordered by their publishing year, but the children "authors" inside each "book" element may not be ordered.

Many optimization techniques have been proposed for XPath expressions in recent years. Among them, the logical level XPath expression optimization includes XPath containment [9], answering XPath queries using views [2] and XPath satisfiability [13]. These logical optimization techniques tremendously improve the performance of XPath evaluation. Intuitively applying these techniques to XQuery expressions, which include multiple XPath expressions in general, will be particularly beneficial. However, the direct applicability of these techniques to XQuery expressions is precluded by the features of the XQuery language, such as the nesting of the FLWR clauses and the *orderby* clauses. The extend of existing query optimization techniques to handle such complex XQuery expressions becomes an important and non-trivial task.

In this paper, we discuss how to optimize query expressions that contain *orderby* clauses in the nested XQuery context. We propose an algebraic rewriting technique of nested XQuery expressions containing explicit orderby clauses. Our technique is based on two steps. First, we perform algebraic query unnesting based on the principles of magic decorrelation [25]. Second, we apply query minimization techniques that exploit pairwise XPath set containment after pulling up order-sensitive operations. Thus the ordered semantics of the XQuery expressions are isolated from the XPath expressions. This enables existing optimization techniques to be applied.

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

In the NEXT framework [5], the authors propose a new nested Xtableaux approach for logical XQuery optimization. We now go beyond this work, while using a more traditional algebraic rewriting and unnesting approach that follows well established formal principles as well as practice in industrial query engines. Using our approach, we are able to not only achieve the optimization specified in the NEXT framework but now also to correctly support ordered semantics.

Example: The following XQuery expression sorts part of the authors by their last name and groups books together with their first author. Then it sorts each author's book by its publishing year. This query is adapted from W3C XQuery Use Cases XMP Q4 [21] 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>
```

Fig. 1. Motivation XQuery Example: Q_1.

In this example XQuery expression, the outer *for* clause binds $a to a sequence of authors appearing in the XML document. The outer *orderby* clause sorts this sequence by the authors' last name. For each instance of $a, the inner query block is then evaluated. Such an intuitive iterative execution tends to be less efficient than an equivalent collection-oriented execution strategy, since for every binding of $a, many operation steps are repeated in the inner sub-query. For efficient execution of such XQuery expressions, decorrelation is necessary. After decorrelation, a join will be generated to connect the outer and inner query blocks. In the decorrelated query, a one time navigation into the XML document for the inner sub-query is sufficient. While we briefly sketch the decorrelation process in Section 4, details of this process can be found in [20,24].

After decorrelation, a closer inspection of the example XQuery reveals that we can even do better: the navigations in the "outer" and "inner" query blocks turn out to be rather similar. The set of author nodes in $\$b/author[1]$ are contained in the author nodes in $\$a$ under set semantics. These navigations however differ in that the author nodes in $\$a$ are sorted by their last names, whereas the ones in $\$b/author[1]$ are sorted by the books' year. Even though these two navigations are not identical, they are similar enough so that one of the two navigations could be saved. We thus suggest that a more "optimal" query plan for this example query will be: 1) get all the books; 2) get the first author associated with each book; 3) sort by the author's last name (major order) and the book publication year (minor order); and 4) group all the book titles by authors. Our experiments in the Section 7 indeed confirm that such a plan can be 2-3 fold faster compared to the unoptimized one. In this paper, we will show a systematic approach for achieving such optimized query plan.

Such XQuery expressions are not rare; rather such cases will always occur when a nested XQuery expression is used for reconstructing the given XML into some new format. If we do not discover that the two navigations are similar, the query plan would include a join between these two navigations. Instead our approach enables the elimination of such redundant navigations whenever possible. In this paper, we will describe how to adapt known XPath containment algorithms to apply them for the reduction of such redundant XPath navigations in XQuery expressions containing *orderby* clauses.

We have implemented the proposed optimization techniques on top of the XAT algebraic framework in our RainbowCore [26] project. The XAT algebra extends the relational algebra by allowing collection-valued columns and by being order-preserving. It also introduces new operators to express necessary XQuery semantics. However, our approach is general and could easily be applied to other XML algebras like NAL [16] and SAL [3].

Our work brings forth the following novel contributions to XQuery optimization:

- To the best of our knowledge, we are the first to provide a practical approach for handling XQuery logical minimization with sequence semantics.
- Our magic branch approach inherits the advantages of magic decorrelation and opens the opportunities for further optimizations using existing algebraic techniques.
- We implement the magic branch decorrelation and the algebraic tree minimization techniques in our XQuery engine.
- We conduct experimental studies showing the performance improvements achievable by our proposed optimization techniques.

This paper is organized as follows. We first give a description of the related work in Section 2 and then briefly describe the algebraic framework used in this paper in Section 3. The magic branch decorrelation approach is illustrated in Section 4. The ordered semantics of XQuery and our algebraic minimization techniques are discussed in Sections 5 and 6 respectively. We present our experimental results in Section 7, while Section 8 concludes this paper.

2 Related Work

Modern database systems [12,7,20] attempt to merge sub-query blocks into the outer query block, thereby eliminating correlations and avoiding nested iterative evaluation. Such "decorrelation" is typically done by introducing outer join and grouping operations.

More recently, methods that focus on the efficiency of decorrelated sub-queries have been proposed. In [20], the authors proposed a technique called magic decorrelation for nested SQL queries. By materializing results from sub-queries and postponing the Outer Join, this approach produces a typically more efficient query plan. Our proposal is conceptually inspired by this technique.

Our decorrelation technique is inspired by the magic decorrelation proposed in [20]. Our approach, called the *Magic Branch*, is a natural extension and adaption of this technique towards more efficient XQuery decorrelation.

Decorrelation of XQuery expressions has also been studied in relationship to native XML query engines. One effort is by Paparizos et al. [17] in the TIMBER system. There 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. Their grouping operator is defined on sets of trees. One 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 [16]. Also they do not consider ordering, as now tackled by our solution.

Fegaras [8] and May et al. [16] have studied XQuery unnesting based on the unnesting techniques from object-oriented query languages [4,7]. However, these works do not discuss decorrelation of XQuery expressions containing *orderby* clauses, which is the main focus of our work.

The work that is most closely related to ours is the NEXT [5] framework, where the authors study minimization of nested XQuery expressions under "mixed set and bag semantics". Here the authors introduce new syntactic constructs to the XQuery language. Compared to this, we use a more traditional algebraic approach for decorrelation. In fact, we demonstrate that our classical algebraic rewriting achieves the same XQuery minimization as in the NEXT framework. Further our approach extends the problem tackled by their work and now solves it under sequence semantics, that is, by considering nested XQuery expressions with explicit orderby clauses. In addition we show how to utilize existing XPath containment and matching techniques to achieve query minimization in the ordered context.

Query containment has been studied in depth for the relational model [14]. Query containment for XPath expressions has been discussed for various axes and quantifiers [9], tag variables and equality testing [1], etc. In [6] the authors study the containment problem for nested XQuery expressions with different fanout. However none of these works consider the order semantics in XQuery; they do not even consider document order in XPath expressions. In short, our work provides a practical approach that fills the gap between the existing works of query containment and XQuery minimization with order semantics.

3 Preliminaries of Algebraic XQuery Processing

XQuery: In this paper, we consider a subset of the XQuery language [23] defined by the grammar in Fig. 2. This subset, plus some extensions of user-defined functions, suffices to express the XMark benchmark query set [19]. Besides the basic *FLWOR* clauses, the XQuery fragment we consider also includes order-related functions (e.g., the position function), and quantifiers.

We discuss our approach under the assumption that the query plan can be described as a tree. However XQuery also allows user-defined functions, and these functions can be recursive. Discussion of such recursive user-defined functions is beyond the scope of this paper.

$Expr$	::=	c	//atomic constants
		$\mid \$var$	//visible variable
		$\mid (Expr, Expr)$	//sequence construction
		$\mid Expr/a :: n$	//navigation step (axis a, node test n)
		$\mid tag(Expr)$	//element constructor: tagger
		$\mid FLWOR$	//query block
		$\mid QExpr$	//expression with quantifier
		$\mid BoolExpr$	//expression for predicate
		$\mid OrderExpr$	//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 $Expr$	
$BoolExpr$	::=	$CompExpr$ | not $BoolExpr$ | $BoolExpr$ (and | or) $BoolExpr$	
$CompExpr$	::=	$Expr$ $CompOp$ $Expr$	
		//CompOp is any comparison operator. eg. "="	

Fig. 2. Syntax of XQuery Subset

In this paper, we focus on nested XQuery optimization with orderby clauses instead of complex XPath processing. Evaluation algorithms for complex XPath expressions having arbitrary navigation axes and node tests [10,11] are orthogonal to XQuery decorrelation.

XAT Algebra: Our algebra (*XAT*) used in the RainbowCore project [26] expresses the subset of the XQuery language shown in Fig. 2. XAT 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 [16], SAL [3] and the algebra proposed in [18]. Hence our approach can be easily extended to these algebras.

We use the *XATTable* to represent ordered sequences of tuples. The input(s) and output of each operator are XATTables. An XATTable may contain nested tuples, that is, the content of an attribute may be a sequence of zero or more tuples.

Since XAT is not designed for type inference purposes, we only have two kinds of atomic values in an XATTable: the ID of an XML node and the string value of an 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 to the associated string value. For simplicity, we will not show such functions explicitly in our later discussions.

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

The XAT algebra inherits all operators from the relational algebra, such as *Select* (σ_p), *Project* (Π_{Attr}), *Join* ($\bowtie_p$), *Left Outer Join* (LOJ, $\overset{o}{\ltimes}_L$), *Natural Join* (NJ, $\bowtie$), *Cartesian Product* (CP, $\times$), etc. Except for the addition of order preserving semantics, these operators have the similar semantics as in the relational context. Below we define the Cartesian Product of two XATTables as an example showing order preserving semantics. (Let $r_L = h(R_L)$).

$$R_L \times R_R := (r_L \overline{\times} R_R) \oplus (t(R_L) \times R_R), where$$

$$r_L \overline{\times} R_R := \begin{cases} \epsilon & if R_R = \epsilon \\ (r_L \circ h(R_R)) \oplus (r_L \overline{\times} t(R_R)) & otherwise \end{cases}$$

Other Join operators can be similarly defined by augmenting their corresponding relational counterparts with order-preserving semantics.

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. Similarly we have an *Unordered* operator to represent the $unordered$ function in the XQuery, which makes the order insignificant for the following variable binding.

We also define the operators: *Orderby* and *Position*. The Orderby operator sorts the tuples in the input XATTable 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 XAT algebra also introduces new operators to represent the XQuery semantics, such as *Navigation* (ϕ_{xp}), *Tagger* ($Tag_{Pattern}$), *Nest* (N), *Unnest* (U), *Cat* (C), etc.

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 processing.

The Tagger operator accepts a pattern indicating where and which open tags and close tags to add around the content of certain columns in the input XATTable.

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

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

where $\vdash_{Attr}$ projects *out* the $Attr$ column from R and $R_{Attr}(h(R))$ retrieves the sequence of attribute values in $Attr$. The Nest operator is a inverse of Unnest and can be defined accordingly.

The *Cat* operator concatenates multiple columns together to form a single column. This operator is used to merge pieces of XML separated by comma in the return clause of XQuery expressions.

To clarify the translation of FLWOR expressions into the XAT algebra, we introduce the *Map* operator. The Map operator is a binary operator with the LHS input XATTable defining the for-variable and the RHS defining 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 the $Attr$ denotes the for-variable in the FLWOR expression and a is the new attribute whose value is calculated from expression e for every instance of $Attr$.

The last operator discussed here is the *Groupby* (GB) operator, which is denoted as $GB_{col_i;col_j;op}(R)$. This operator is introduced mainly 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 XATTable by the column col_i, then perform the operator *op* (e.g. the aggregation functions and the position function) on col_j of each group of tuples, finally concatenate all the groups together as output. The Groupby operator can also group on multiple columns.

For further detailed discussion of the XAT algebra, please refer to our technical report [27].

XQuery Normalization: Prior to translating the XQuery expressions into the XAT algebra expression, we use a source-level normalization step applied to the original XQuery expressions. Similar normalizations are also discussed in [15]. Our normalization does not aim to do optimization of the XQuery expressions, but rather provides a suitable format for easy generation of the XAT algebra tree.

Normalization Rule 1: The let-variables are treated as temporary variables. During normalization, they can be eliminated: the expression binding the let-variable is substituted for all occurrences of the let-variable. Note that in the implementation, the let-variable is calculated only once and is materialized for sharing among all the occurrences.

Normalization Rule 2: Since the *Map* operator is binary, the *For* clause defining more than one for-variable will be split into a sequence of nested For clauses. Each clause defines one for-variable only.

Translation of XQuery Expressions to XAT Algebra: Normalized XQuery expressions are translated into their corresponding XAT algebra representation in two steps: translating XPath expressions and translating the FWOR (without the Let clause) query expressions. As mentioned before, we simply translate each XPath expression into one *Navigation* operator.

The translation pattern of a flat FWOR query block to the XAT algebraic expression is illustrated in Fig. 3. 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 *Nest* operator on top of the Map is used to construct a sequence of all intermediate results. For those *where* clauses where no position function is used, the where clause can also be put in the LHS of the Map operator, just like the orderby clause.

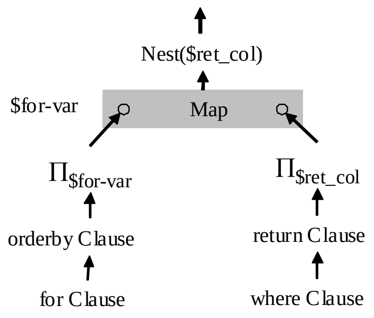

Fig. 3. Build Algebra Tree for XQuery FWOR Expression.

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

4 XQuery Decorrelation

After XQuery normalization and translation, the correlation in an XQuery expression is represented in the XAT tree by the *Map* operator and *linking* operators (operators in the inner query blocks referring to variables defined in the outer FLWOR query block). 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, removing the Map operator in the XAT tree transformation is the main goal of the decorrelation algorithm. Depending on the different semantics of the operators that the Map is pushed over, the Map operator will be pushed down along the RHS accordingly, until the linking operator is reached and the Map operator is rewritten as a join. Our techniques are an extension of magic decorrelation [20]. These extensions are sufficient to ensure efficient XQuery decorrelation. Please note that in this paper, we omit the detailed discussion about the empty collection problem, which is handled in the decorrelation algorithm by adding left outer joins. Since our example XQuery does not need left outer joins, we omit this step here. For the complete magic branch decorrelation algorithm, please refer to our technical report [27].

Below we will use the XQuery expression shown in Section 1 as the running example. The generated XAT tree for the example query is shown in Fig. 4. The I_1, I_2 and I_3 blocks are generated from the outer query block. They represent the *orderby* clause, *for* clause and *return* clause respectively. Similarly the J_1, J_2, J_3 and J_4 blocks are generated for the inner query.

We now discuss how the different operators affect the "pushing down" of the Map operator. For this, we first distinguish between *tuple-oriented* and *table-oriented* operators. The propagation of the Map operator down over *tuple-oriented* operators is different from that over *table-oriented* operators.

Definition 1. *A tuple-oriented operator is one that examines each tuple in the input XATTable(s) one at a time and generates a corresponding output tuple(s) as needed. A table-oriented operator, on the other hand, examines multiple and possibly all tuples in the input XATTable(s) for generating an output tuple(s).*

The table-oriented operators in our algebra include: *Nest*, *OrderBy*, *Groupby*, *Distinct* and all relational aggregation functions. Further, since the order semantics in XQuery have to be defined on a sequence of tuples, all order-sensitive operators such as *Position* are classified as table-oriented operators. Other operators such as Select, Project and Join correspond to tuple-oriented operators.

We show the *Position* operator below as an example of a table-oriented operator. The output of the *Position* operator depends on all the tuples in the input XATTable.

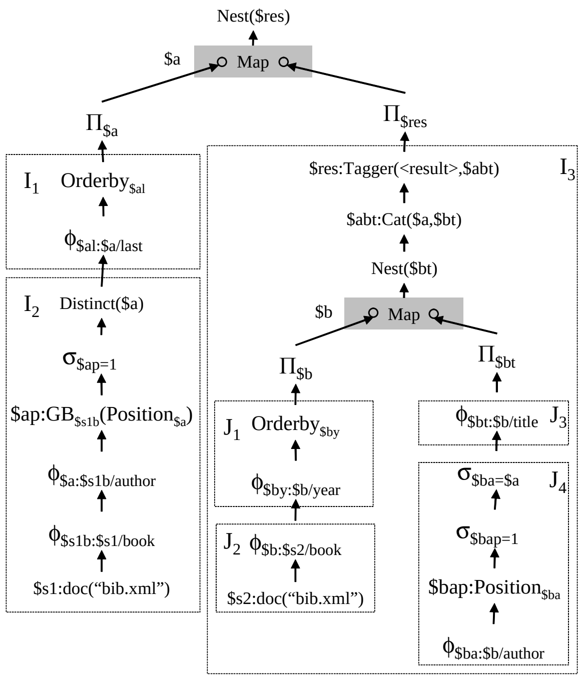

Fig. 4. The XAT Tree for the Example XQuery in Sec.1.

col_1	col_2
a1	b1
a2	b2

col_1	col_2	$position(col_2)$
a1	b1	1
a2	b2	2

For a tuple-oriented operator we can simply push the *Map* operator down over it. For table-oriented operators, we need to perform an extra rewriting for the operator. That is, we will generate a *Groupby* operator, which groups the input tuples by the for-variable introduced by the Map operator, and performs the original table-oriented operator for each group. Intuitively the added grouping operator separates the whole column used by the table-oriented operator into partitions according to the context variable. Thus each partition keeps the group boundary of the column correctly. We will now illustrate this decorrelation process for the XAT tree in Fig. 4 in a step-by-step fashion below.

Step 1: Considering the Map operator of the inner query block, we simply push the Map operator down the RHS until we reach the table-oriented position operator. For the position operator, a Groupby operator is generated and the position function becomes the embedded operation of the Groupby operator. We then continue pushing down the Map operator until the RHS becomes empty and the Map operator can be removed. This step is shown in Fig. 5.

Step 2: Next we consider the Map operator of the outer query block in Fig. 4. We simply push the Map operator down the RHS until we reach a Nest operator. The Nest operator is another table-oriented operator. Propagation of the Map over the Nest operator is shown in Fig. 6.

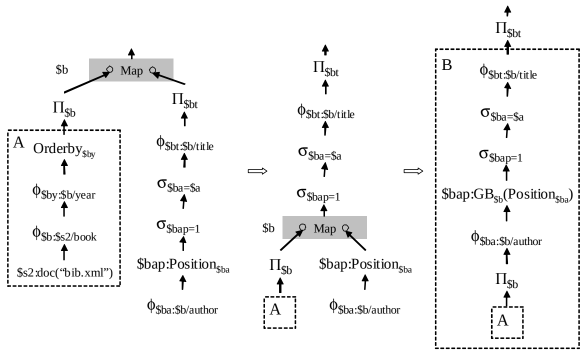

Fig. 5. Propagation of Map Operator for Inner Query Block

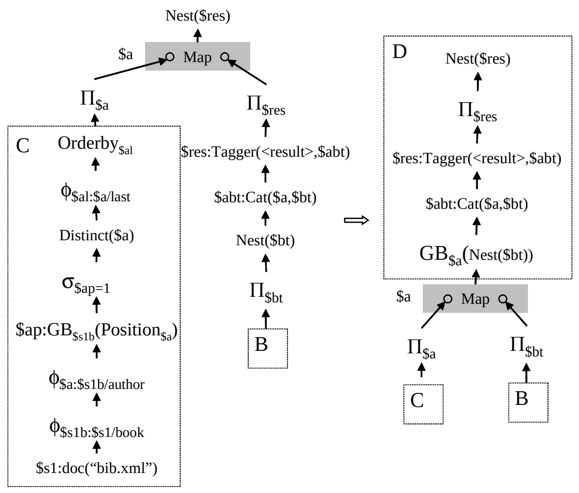

Fig. 6. Propagation of Map Operator for the Outer Query Block

Step 3: Continuing to push the Map operator of the outer query block down, now the linking operator $\sigma_{\$ba=\$a}$ becomes the right child of the Map operator. The last step of the propagation is to absorb the Map operator into the linking operator. A Join is formed to connect both the branches. This transformation of the XAT tree is shown in Fig. 7.

Finally, the decorrelated XAT tree is shown in Fig. 8. The LHS of the Join operator retrieves a distinct sequence of authors ordered by their last names. The RHS of the Join operator retrieves the sequence of $(book, author)$ ordered by the books' year. Here the $author$ is the first author of each $book$.

5 Order in XQuery Processing

As mentioned before, XML is an ordered data model. The result of an XPath expression is a sequence of elements matching the pattern, where the order of the elements is determined by the document order in the input XML. Beyond the de-

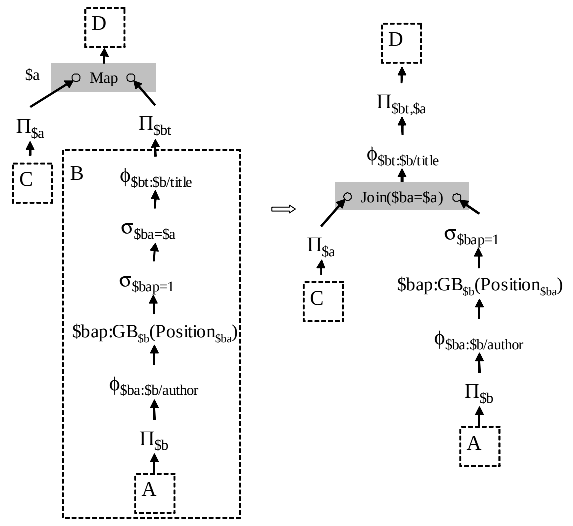

Fig. 7. Propagation of Map Operator for the Outer Query Block (Contd.)

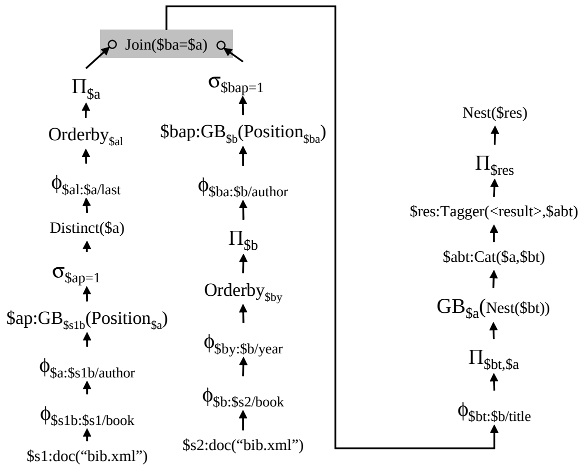

Fig. 8. The XAT of Example XQuery after Decorrelation.

fault document order, the XQuery expressions can have additional order semantics on the sequences. The main extensions are: (1) sorting the elements by certain attributes or sub-elements; (2) making the order of the sequence in the *for* clause not significant by a value based distinction or the "unordered" function ; and (3) constructing the output document order (may be partial) among multiple level of elements in the result XML hierarchy. The XAT algebra is an order sensitive algebra and captures all these order semantics in the intermediate XATTables.

5.1 Order in the XATTables

The XATTables can be treated as extended relational tables. Different with the relational tables, the order among the tuples in the XATTables may be significant. Beyond ordering, the grouping property is also important.

The complexity exists since 1) hierarchical XML intermediate results can be defined with multiple level orderings; and 2) arbitrary XQuery expressions can also define partial orders. Two examples of partial order in XML intermediate results are shown in Fig. 9. Here we use the XML schema graph to show the hierarchical structures. In Fig. 9(i), the $book$ nodes are not ordered under the root, while $author$ nodes are ordered by the $name$ attribute for each $book$ node. In Fig. 9(ii), the $book$ nodes are ordered consistently with the default document order in the input XML document. The $price$ children of $book$ nodes are not ordered, while $author$ children are ordered by their names locally.

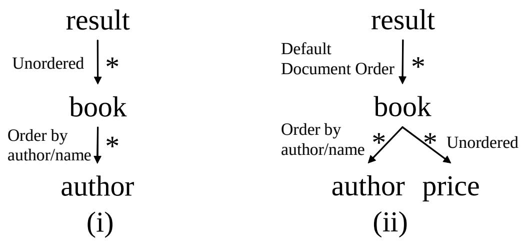

Fig. 9. Examples of Partial Order in XML Intermediate Results

All the order information in the intermediate result has to be captured in the XATTables. The order context of an XATTable is denoted as $[\$col1^{O/G}, \$col2^{O/G}, ...]$. Each item can be either an ordering denoted as $\$col^O$ or a grouping denoted as $\$col^G$. For simplicity, we only consider grouping on a single column in this paper. The techniques can however be easily extended to grouping on multiple columns. The tuples of the XATTable are ordered (or grouped) first according to $\$col1$, with ties broken by $\$col2$, and so on. For each $\$col$, the ordering $\$col^O$ implies the grouping $\$col^G$ but not vice versa. Such annotation is sufficient to represent any partial orders in the XML intermediate results.

For example, suppose an XQuery first retrieves distinct books ($\$book$) from the $bib.xml$, then retrieves all the authors ($\$author$) for each book and sorts the authors by their names ($\$name$). The order context of the XATTable after these operations is denoted as: $[\$book^G, \$name^O]$, where the $\$book^G$ is implied by the distinct operation on books and the $\$name^O$ comes from the explicit sorting by $\$name$.

5.2 Ordered Semantics of the XAT Operators

The XAT algebra is an order sensitive algebra. Depending on how the tuple order of the input XATTable is changed by the operator and reflected in the output, the XAT operators can be divided into four categories: **order keeping**, **order generating**, **order destroying** and **order specific** operators.

- **Order-keeping** operators include most of the operators, such as Select, Project and Tagger. These operators inherit the order context of the input XATTable. For example, the tuple order among the input tuples of the Select operator will be kept in the output XATTable. Project and Tagger operators will behave similarly. Here the Project operator in XAT does not include the distinct semantics. If a column of the input XATTable which is part of the input order context is projected out, the column is marked and not really removed until the query plan cleanup after all query rewriting.

- **Order-generating** operators include the Orderby, Navigate and the Join operators. The Orderby operator will sort the input tuples by certain column(s). The Navigate operator will extract the document order of the elements of navigation and impose it into the respective orders of the tuples it generates. The Join operator will merge the order from its two inputs into a new order. These three operators are now be discussed in more detail below.

  An **OrderBy** operator sorting on $\$col1, \$col2, ...$ will generate a new order context $[\$col1^O, \$col2^O, ...]$. The order context of the input XATTable of the OrderBy operator may be overwritten, unless the input order context is compatible with the new one. Here "compatible" means the complete (or prefix) of the input XATTable is already included (or implied) by the new order context determined by the sorting.

  The compatibility can be determined by checking if the input order context is the "prefix" of the generated order context of the OrderBy operator. For example, $[\$col1^G, \$col2^G]$ is not compatible with the explicit sorting on $\$col2$. Thus the output order context will be $[\$col2^O]$ only. That is, the ordering is stronger than the grouping. But $[\$col1^G, \$col2^G]$ is compatible with ordering on $\$col1$ or on $(\$col1, \$col2, \$col3)$ with the output order context then being $[\$col1^O, \$col2^G]$ and $[\$col1^O, \$col2^O, \$col3^O]$ respectively.

  The **Navigate** operator extracts the document order and imposes it onto the output order context. The Navigate operator forwards the input order context to its output XATTable. If the input order context, including the trivial groupings, is not empty, the extracted document order will be attached to the end of the input order context. Otherwise the output order context is empty. The trivial groupings can be implied by the key constraints in the XATTable. Such key constraints can either be referred from the value-based distinction function or the id-based distinction from XPath semantics. One special case of such trivial grouping arises when the navigation is from the root of the XML document, since there is only one tuple in the input XATTable (for the root node).

  Different permutations of the same set of Navigates may result in different order contexts. For example, considering two Navigate operators from $\$a$: $\$a/b$ and $\$a/c$, if we perform $\$a/b$ before $\$a/c$, then the final order context will be $[\$a^O, \$b^O, \$c^O]$. If we perform the two Navigates in the opposite order, then the output tuple order will be different, namely $[\$a^O, \$c^O, \$b^O]$. Such rewriting among Navigation operators will thus be incorrect considering the order semantics.

  Suppose OC_L and OC_R denote the order contexts of the left and right input XATTables of a **Join** operator. Then the output order context inherits the OC_L. The OC_R is attached to the output order context if the OC_L is not empty. Otherwise, OC_R is discarded. Here all orderings and groupings properties in the left input XATTable, even if trivial, need to be included in OC_L for the empty test and order propagation. They may not be trivial anymore in the output XATTable. For example, suppose the left input XATTable has a unique identifier (key constraint) $\$col1$, then $\$col1^G$ is trivial since all groups consist of only one tuple. But it is no longer trivial in the Join output since a $1-m$ matching between the left and right input tuples may exist.

- **Order-destroying** operators include the Distinct operator and the Unordered operator, which represent the $distinct$ function and the $unordered$ function respectively. The value-based Distinct operator and the Unordered operator will destroy the order of the input tuples. That is, the output tuple order is considered to be not significant. Note that the Distinct operator will create a value based key constraint in the output XATTable. It thus implies a trivial grouping property.
- **Order-specific** operators include the Groupby operator. Similar with the OrderBy operator, the order context of the output XATTable of the GroupBy operator is determined by the compatibility of the input order context with the grouping semantics. The compatibility checking needs to consider functional dependencies in the input XATTable. For example, if the input tuples have been sorted on a column ($\$col1$) and the grouping is done on a column ($\$col2$), where $\$col2 \rightarrow \$col1$, then the order context generated by the GroupBy operator is compatible with the input order context. This order is preserved in the output. $\$col1$, $\$col2$ above can also be multiple columns each.

  For example, in Fig. 8 the Groupby operator preserves the order since $\$b \rightarrow \$by$ (there is one year for each book), and the input of the Groupby operator is a sequence sorted by the books' year. The Join operator produces a sequence of tuples with the major order of $\$al$ and minor order of $\$by$. This ordered sequence will be grouped by $\$a$ and all the book titles for each $\$a$ will be nested into a collection. Since $\$a \rightarrow \$al$ (there is one last name for each author), this Groupby operator will also preserve the order of the sequence.

6 Minimization of XAT Query Plan

In this section, we study how to remove redundant operations in the XAT tree. The goal is to rewrite it into an equivalent but smaller query plan with s smaller number of operators.

In Fig. 8, a close inspection shows that the LHS and the RHS of the Join operator have similar XPath navigations to the *author* node. But they use different Orderby operators: the authors in the LHS are ordered by their last names and the RHS is ordered by the books' year. Hence when we consider order semantics, the two sequences do not match. To share the navigation computations among the two input branches of the Join operator, we first need to rewrite the query plan by pushing down the navigations and pulling up the orderby operators. Thus existing XPath matching and sharing algorithms can be applied.

Given the sharing of the XPath navigations takes place, then we find that since the Join is an equi-join on the now shared XPath navigation ($\$b = \$ba$), and thus the Join can even be removed. Below we will discuss these two types of rewritings in more detail.

6.1 Finding the Minimal Order Context

The XAT tree may include operators having various ordering properties. To perform algebraic rewriting while correctly maintaining the order semantics, we first propose a systematic way to determine the minimal ordered semantics.

This process includes two steps: a bottom-up tree traversal recording the order context of the XATTable; and a top-down tree traversal removing any overwritten order contexts. In the first step, the order context of the XATTable is generated according to the ordering property of each operator. In the second step, all the order context columns overwritten by upper operators (and thus not essential) will be removed. After this process, every intermediate XATTable will be associated with an ordered sequence, denoting the order context. The result order context associated with the XATTables after the process describes the **minimal ordered semantics** in the XAT tree. These order contexts must be kept during the algebraic rewriting to assure correctness with respect to order.

We show these two steps using the previous XAT tree in Fig. 10, that is, with the partial XAT tree that is sufficient for the purpose of explaining the main concepts above.

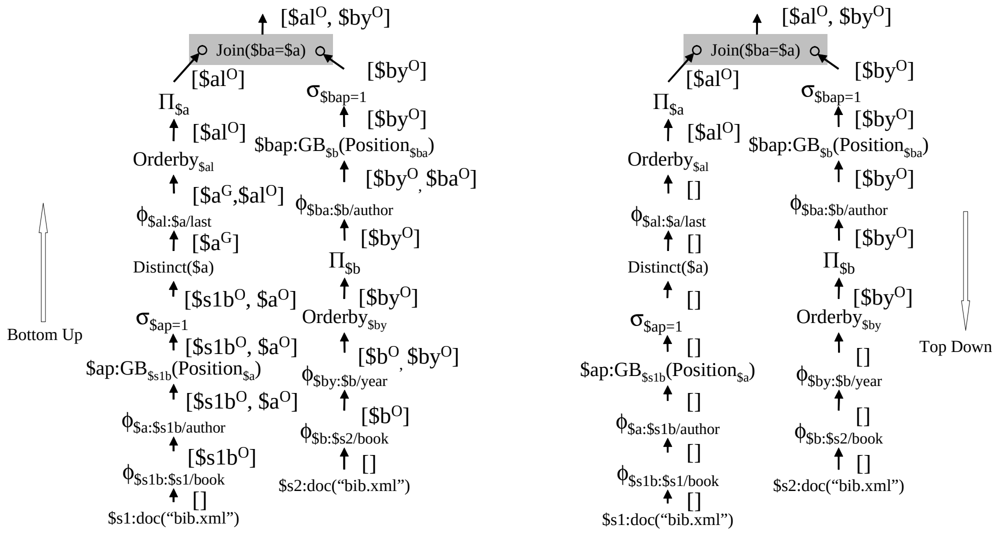

Fig. 10. The Process of Finding the minimal Order Context.

Note that during the first step, the Distinct operator in the LHS of the Join generates a value-based key constraint on $\$a$. The order context $\$a^G$ is trivial for the output

XATTable of the Distinct operator, but it is not trivial for the following Navigation operator. In the example query plan, there are two implicit functional dependencies: $\$a \rightarrow \$al$ and $\$b \rightarrow \$by$. Otherwise the two Orderby clauses in the example XQuery expressions would be ambiguous. Since $\$b \rightarrow \$by$, the Groupby operator grouping on $\$b$ will preserve the sorted order from $\$by$.

In the second step, we perform a top-down tree traversal of the XAT tree. The order context of every input XATTable will be truncated from tail to head in a step-by-step fashion. Such truncation will be stopped when a different output order context is being generated. In the case that the output order context is empty, the input order context is also set to empty. The remaining order context of the input XATTable will be the minimal order context. For example, for the Orderby operator in the LHS of the Join operator, we have:

$$\begin{aligned} &\xrightarrow{[\$a^G, \$al^O]} Orderby_{\$al} \xrightarrow{[\$al^O]} \\ \Rightarrow\ &\xrightarrow{[\$a^G]} Orderby_{\$al} \xrightarrow{[\$al^O]} \\ \Rightarrow\ &\xrightarrow{[]} Orderby_{\$al} \xrightarrow{[\$al^O]} \end{aligned}$$

The minimal input order context of the Orderby operator will be truncated to $[]$.

6.2 Orderby Pull up

Correct query rewriting under ordered semantics must guarantee that the order context of the result XATTable will not change after rewriting. To achieve this, we first define the correct rewriting of XAT trees below.

Definition 2. *For an XAT tree, suppose the minimal order context of the output XATTable of the root of the tree is C. If C remains unchanged after a given rewriting inside the tree, we call such a rewriting an* **order-preserving** *rewriting.*

We first intuitively describe the cases that we may encounter concerning order-sensitive rewriting. Intuitively, pulling up the Orderby operator over an order-keeping operator is always allowed. Pulling it over an order-generating operator is prohibited, since the upper Orderby operator can overwrite the lower Orderby operators. For the order-destroying operators, the lower Orderby operator can be removed. For the order-specific Groupby operator, we need to check the tuple order and the grouping column in order to make a correct rewrite.

In general, this now leads us to the following four rewriting rules for the pulling up of the Orderby operator.

Rule 1. *An Orderby operator and its associated Navigation operator (if any), which retrieves the column to be sorted on, can be pulled up together over an* *order-keeping operator.*

Proof. Without loss of generality, we need to show that the following rewriting does not change the ordered semantics. ($\$col$ can also represent multiple columns.)

$$Orderby_{\$col} \rightarrow OrderKeeping \Rightarrow OrderKeeping \rightarrow Orderby_{\$col}$$

Considering the minimal order context of the intermediate XATTables, we have:

$$\xrightarrow{OC1} Orderby_{\$col} \xrightarrow{OC2} OrderKeeping \xrightarrow{OC2},$$

where OC_i denotes the order contexts. According to the ordering property of the Orderby operator, $OC1 := []$ or $OC1 = OC2$. Otherwise $OC1$ will not be the minimal order context. Thus we have either

$$\begin{aligned} &\xrightarrow{[]} Orderby_{\$col} \xrightarrow{OC2} OrderKeeping \xrightarrow{OC2} \\ \Rightarrow &\xrightarrow{[]} OrderKeeping \xrightarrow{[]} Orderby_{\$col} \xrightarrow{OC2} \end{aligned}$$

or

$$\begin{aligned} &\xrightarrow{OC2} Orderby_{\$col} \xrightarrow{OC2} OrderKeeping \xrightarrow{OC2} \\ \Rightarrow &\xrightarrow{OC2} OrderKeeping \xrightarrow{OC2} Orderby_{\$col} \xrightarrow{OC2} \end{aligned}$$

In both cases, the final output order context $OC2$ is unchanged. □

Rule 2. *Consider pulling up the Orderby operator above a binary Join operator $\$o$.*

- *If the LHS of $\$o$ is ordered by $\$l$ and the RHS of $\$o$ is not ordered, then the Orderby operator can be pulled up.*
- *If the RHS of $\$o$ is ordered by $\$r$ but the LHS of $\$o$ is not ordered, then the Orderby operator cannot be pulled up.*
- *If the LHS of $\$o$ is ordered by $\$l$ and the RHS is ordered by $\$r$, then both Orderby operators in the LHS and RHS can be pulled up and merged into one single Orderby operator. This new operator sorts the XATTable using $\$l$ as the major order and $\$r$ as the minor order.*

We illustrate the three cases of Rule 2 using the Join operator in Fig. 11.

Proof. We only show the proof of the first case. The proofs of the other two cases are similar. We need to show that the following rewriting does not change the ordered semantics.

$$(Orderby_{\$col} \rightarrow Join \leftarrow) \Rightarrow (\rightarrow Join \leftarrow) \rightarrow Orderby_{\$col}$$

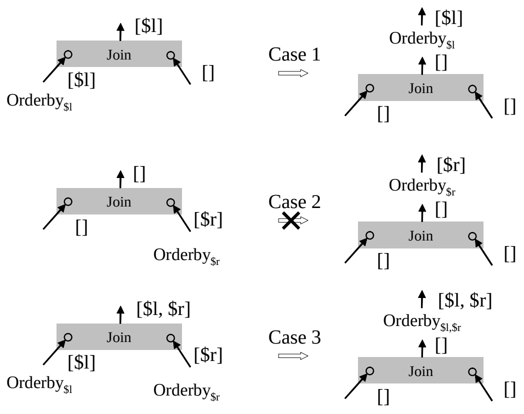

Fig. 11. The Three cases of Rule 2 on Pulling up Orderby over Join.

Considering the minimal order context of the intermediate XATTables, we have:

$$(\xrightarrow{OC1} Orderby_{\$col} \xrightarrow{OC2} Join \xleftarrow{[]}) \xrightarrow{OC2}$$

Similar to the proof of Rule 1, we have $OC1 := []$ or $OC1 = OC2$. So

$$(\xrightarrow{[]} Orderby_{\$col} \xrightarrow{OC2} Join \xleftarrow{[]}) \xrightarrow{OC2}$$
$$\Rightarrow (\xrightarrow{[]} Join \xleftarrow{[]}) \xrightarrow{[]} Orderby_{\$col} \xrightarrow{OC2}$$

or

$$(\xrightarrow{OC2} Orderby_{\$col} \xrightarrow{OC2} Join \xleftarrow{[]}) \xrightarrow{OC2}$$
$$\Rightarrow (\xrightarrow{OC2} Join \xleftarrow{[]}) \xrightarrow{OC2} Orderby_{\$col} \xrightarrow{OC2}$$

In both cases, the final output order context $OC2$ is unchanged. □

Rule 3. *An Orderby operator can be removed if there is an order-destroying operator above it immediately.*

This rule is straightforward and we omit the proof here.

Rule 4. *An Orderby operator that sorts on $\$b$ can be pulled above a Groupby operator that groups on $\$a$ if $\$a \rightarrow \$b$.*

Proof. Given that $\$a \rightarrow \$b$, the order context $[\$b^O]$ is compatible with $[\$b^O, \$a^G]$. The Groupby operator will not destroy the input order context. Similarly the order context $[\$a^G]$ is compatible with $[\$b^O, \$a^G]$. Then we have:

$$\xrightarrow{[]} Orderby_{\$b} \xrightarrow{[\$b^O]} Groupby_{\$a} \xrightarrow{[\$b^O,\$a^G]}$$
$$\Rightarrow \xrightarrow{[]} Groupby_{\$a} \xrightarrow{[\$a^G]} Orderby_{\$b} \xrightarrow{[\$b^O,\$a^G]}$$
□

Proposition 1. *A series of algebraic query rewritings using Rules 1, 2, 3 and 4 in XAT trees form a rewriting that is order preserving.*

This proposition can be proved by induction on the rewriting steps.

In Fig. 12, the Orderby in the LHS of the Join can be pulled up above the Project, since the Project is a unary order-keeping operator. The Orderby in the RHS can also be pulled up above the Project, Groupby and Select. For the Groupby operator, since the Orderby operator sorts the tuples by $\$by$, which is functionally dependent on the grouping column $\$b$, the tuple order before and after the pulling up of the Orderby operator are identical. The LHS and the RHS Orderby operators can be pulled up above the Join and be merged into one single Orderby operator that sorts tuples by $\$al$ (major), and by $\$by$ (minor).

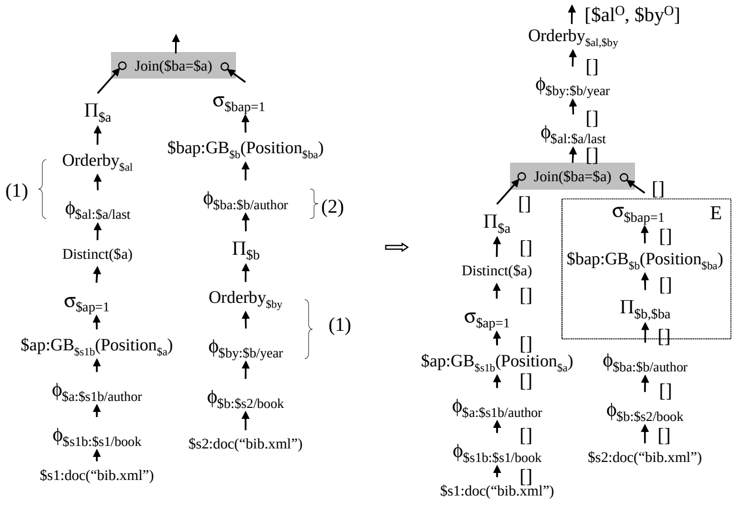

Fig. 12. Orderby Pull up

After pulling up the Orderby operators, the XQuery minimization problem is reduced from the ordered sequence matching problem to the well studied XPath matching under set semantics. To "gather" all the XPath expressions, we push down all the navigations to the bottom of the XAT tree. During this pushing, the Project operator needs to be changed accordingly as shown in Fig. 12.

6.3 XPath Matching and Redundancy Removal

In the example XAT tree, after pulling up the Orderby operators, the order context becomes null for the two branches below the Join operator. Then the optimization problem is reduced to the optimization under unorder semantics. Various query plans can be generated and the optimal can be picked.

By utilizing existing XPath matching algorithms [2], we find that the $\$a$ in the LHS of the Join operator and the $\$ba$ in the RHS both come from the same XPath expression $bib.xml/book/author$. We can remove such redundant navigation using the following rewriting rule.

Rule 5. *Consider an equi-join operator with $\$a = \$b$ with $\$a$ introduced from the LHS and $\$b$ from the RHS of the operator. We can remove the equi-join and the LHS if the following conditions hold:*

- *$\$b \subseteq \$a$ under set semantics, and*
- *$\$a$ is a set with no duplicates.*

The proof of Rule 5 is straightforward and we omit the proof here.

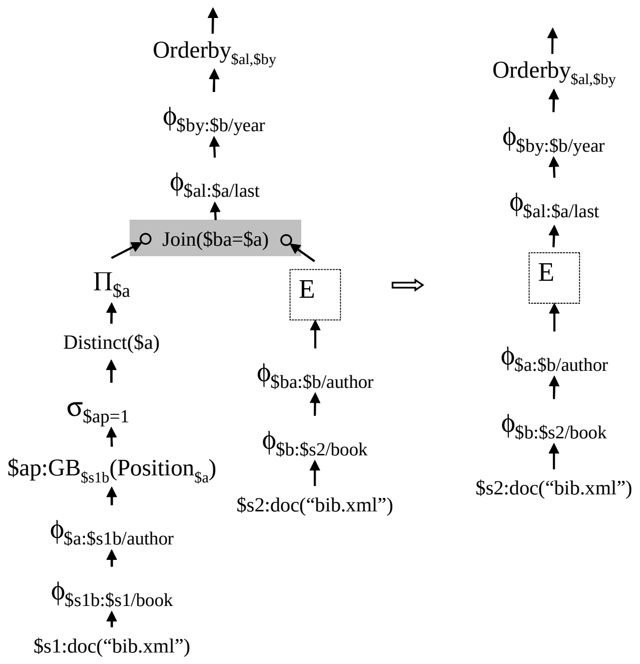

Fig. 13. Removing Redundant Joins and Navigations.

In Fig. 13, every author in $\$ba$ appears in $\$a$; the schema of the LHS has only one column $\$a$; and $\$a$ has no duplicates after the Distinct operator. Therefore the equi-join operator and in fact the complete LHS branch can be removed according to the Rule 5. The final query plan is shown in Fig. 14.

7 Experimental Study

We have conducted experiments to illustrate the performance gains achievable by our approach. We have implemented the magic branch decorrelation and minimization algorithm in the RainbowCore project, a Java-based native XQuery engine using the XAT algebra developed at WPI [26]. Our experiments shown in this section focus on nested XQueries containing order related predicates and clauses that can be minimized after our order context processing. The benefit of such minimization

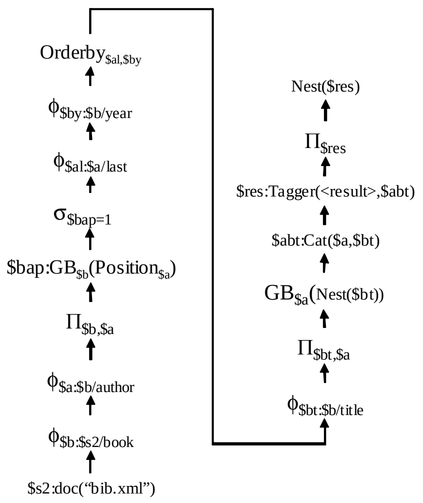

Fig. 14. The optimized XAT of the Example XQuery Q_1.

shown in the experiments comes from the saving of repeated computations. Such benefit varies with the amount of computations that can be saved.

In this section, we show the following three performance measurements: (1) The comparison of the execution time before and after the query decorrelation. This illustrates the potential benefit achieved by query decorrelation. (2) The comparison of the execution time before and after the query minimization. The varying benefit of the query minimization is shown by different XQueries. (3) The comparison of the query optimization time versus the query execution time. This illustrates the query decorrelation and minimization time is rather small compared to the execution time.

The input XML files are generated according to the schema of the "bib.xml" in the W3C XQuery Use Cases XMP [21]. The number of books in the XML file varies in the experiments. The number of authors per book ranges from 0 to 5, with uniform distribution. Each distinct author can be in the author list of 0 to 5 books. In other words, each author will appear 2.5 times on average in the XML file. The input XML files are stored as plain text files on the disk. We do not employ any pre-processed index on the XML files in these experiments. Furthermore, all the experiments are done in main memory using a simple iterative execution.

These experiments were performed on a 1.2GHz PC with 512MB of RAM running Windows 2000 using Java SDK 1.4.2.

7.1 Performance Analysis: Q_1

Our first experimental results based on the example XQuery described in Sec. 1 are shown in Fig. 15. We compare the query execution times of three query plans:

the original query plan with the nested sub-query shown in Fig. 4; the decorrelated query plan shown in Fig. 8; and the optimized query plan after removing redundant navigations and Joins depicted in Fig. 14. We have varied the input XML documents to have different numbers of book elements, as shown on the x-axis.

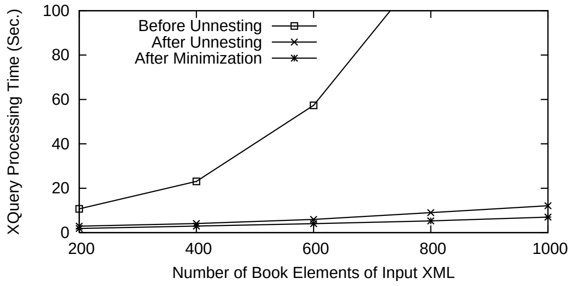

Fig. 15. Performance Comparison of Different Query Plans of Q_1.

We can see that the decorrelation step gives significant performance gains. One of the reasons is that in our experiment we do not employ any storage manager, so 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 sub-query will be saved. Thus the total I/O costs will decrease dramatically. The XAT minimization also brings significant performance improvements in the order of 30%-40%. This is due to the successful removal of the redundant Navigations and the costly Join operations. The performance gain of the XAT minimization is shown more clearly in Fig. 16 by focusing on the results before and after minimization.

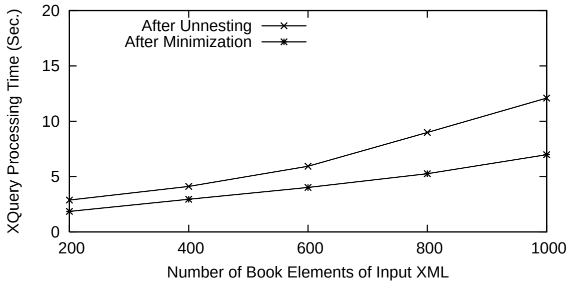

Fig. 16. Performance Gain of XAT Minimization of Q_1.

7.2 Performance Analysis: Q_2

In the following performance analysis, we will no longer show the execution of the query plan before the decorrelation. Instead we focus on the comparison of the execution of the query plans before and after our query minimizations.

The second experiment we conduct is based on the query Q_2, which is a variation of the previous XQuery Q_1 by removing the position function in the inner query block.

```
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 = $a
                order by $b/year
                return $b/title
               }
       </result>
```

The authors used in the inner query can be any authors in the input XML file, which is different from Q_1. Thus any book whose author is the first author of at least one book will be returned in the inner query block. Recall the Rule 5 after decorrelation of the query plan of Q_2, the LHS and RHS of the Join operator cannot be merged and the Join operator cannot be removed. However, the same navigations occurring in both the LHS and RHS of the Join can be shared. The minimized query plan for Q_2 is shown in Fig. 17.

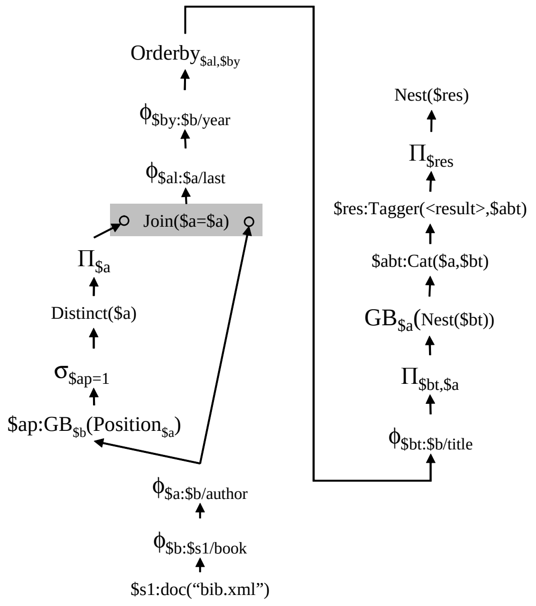

Fig. 17. The optimized XAT of the XQuery Q_2.

After minimization, the query plan for Q_2 merges the matching navigation and materializes the result of the navigation for the Groupby and the Join operators. Similar with the first experiment, we have varied the input XML documents to have different numbers of book elements. The results are shown in Fig. 18.

From Fig. 18, we can clearly see the effect of the XQuery minimization on query execution time. The minimization brings performance improvements in the order of 20%-30% for Q_2. The benefit of XQuery minimization is a little smaller than that for Q_1 since fewer operators are removed by the minimization.

The corresponding query optimization times including the decorrelation time and the minimization time for Q_2 are shown in Fig. 19. Both query decorrelation and minimization only take very small amount of time compared to the actual query

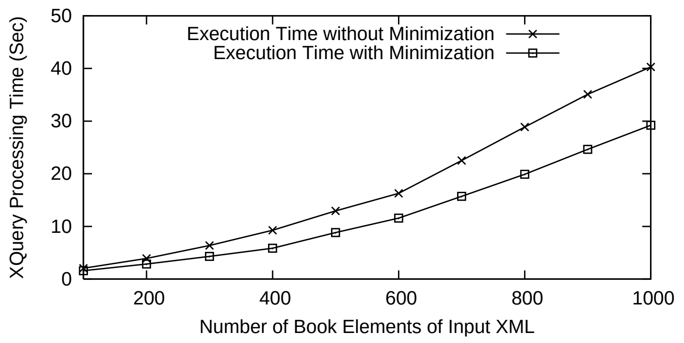

Fig. 18. Performance Comparison of Different Query Plans of Q_2.

execution time, especially when the input XML files are large.

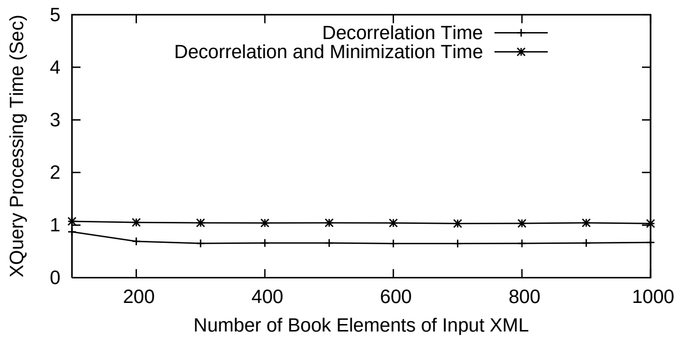

Fig. 19. Query Optimization Time of Different Query Plans of Q_2.

7.3 Performance Analysis: Q_3

The third experiment we conduct is based on Q_3, which is a variation of the previous XQuery Q_1 by dropping both the position functions.

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

The variation of Q_3 is used to test the effectiveness of the query minimization without the position function. In this case, more tuples will be in the input table of the Join operator without the query minimization. Thus the Join cost will be more significant and query minimization will result in a much better query plan in terms of execution time. The minimized query plan for Q_3 is shown in Fig. 20.

Similar with the other experiments, we have varied the input XML documents to have different numbers of book elements. The results are shown in Fig. 21. The execution time without query minimization increases quadraticly with the size of the

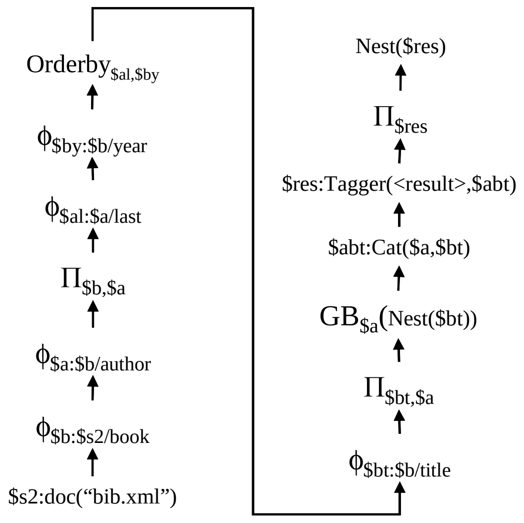

Fig. 20. The optimized XAT of the XQuery Q_3.

XML files. However, the execution time after the minimization increases linearly and much slower.

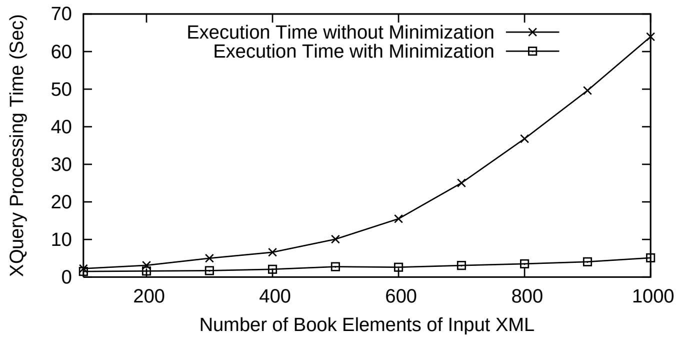

Fig. 21. Performance Comparison of Different Query Plans of Q_3.

7.4 Summary of the Experiments

We define the performance improvement rate of the XQuery minimization as:

$$\frac{(execution\_time\_without\_mini - execution\_time\_with\_mini)}{execution\_time\_without\_mini}$$

The average improvement rate of the minimization over Q_1, Q_2 and Q_3 is summarized in Fig. 22.

	Q_1	Q_2	Q_3
Ave. Impro. Rate	35.9013%	29.8444%	73.3869%

Fig. 22. Average Performance Improvement Rate of Different Query after Minimization.

Depending on the operators that the minimization can remove from the query plans, the XQuery minimization can achieve significant performance improvements. In some cases, such performance improvement can be greater than 50% of the execution time.

8 Conclusions

In this paper we propose an algebraic rewriting technique for nested XQuery expressions containing explicit orderby clauses. The proposed technique is based on the principles of magic decorrelation. Unlike prior work, this technique enables the optimization of nested XQuery expressions not only with set but also with ordered sequence semantics. We illustrate how our proposed technique is able not only to successfully tackle the same XQuery logical optimization problem solved in the NEXT framework, but to go one step beyond and support ordered semantics.

Our work extends previous work primarily in two aspects. First, to the best of our knowledge, we are the first to provide a practical approach handling XQuery logical minimization with sequence semantics. Second, our magic branch approach inherits the advantages of magic decorrelation, including the opening of the opportunities for further optimizations. 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 order-sensitive query plans as well as optimization of the operators using it.

References

[1] A. Deutsch and V. Tannen. Containment of Regular Path Expressions under Integrity Constraints. In *8th Int. Workshop on Knowledge Representation Meets Databases (KRDB), Rome, Italy*, pages 1–11, June 2001.

[2] 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.

[3] 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.

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

[5] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The NEXT Logical Framework for XQuery. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 29–41, 2004.

[6] X. Dong, A. Y. Halevy, and I. Tatarinov. Containment of Nested XML Queries. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 132–143, 2004.

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

[8] 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.

[9] 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.

[10] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 95–106, 2002.

[11] G. Gottlob, C. Koch, and R. Pichler. XPath Query Evaluation: Improving Time and Space Efficiency. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 379–390, 2003.

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

[13] 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.

[14] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering Queries Using Views. In *PODS, San Jose, CA*, pages 95–104, June 1995.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] W3C. XML Path Language (XPath)Version 2.0. W3C Working Draft. http://www.w3.org/TR/xpath20, November 2003.

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

[24] S. Wang, X. Zhang, E. A. Rundensteiner, and M. Mani. Algebraic XQuery Decorrelation with Order Sensitive Operations. Technical report, Worcester Polytechnic Institute, 2005. to appear.

[25] X. Zhang, K. Dimitrova, L. Wang, M. El-Sayed, B. Murphy, B. Pielech, M. Mulchandani, L. Ding, and E. A. Rundensteiner. Rainbow: Multi-XQuery Optimization Using Materialized XML Views. In *ACM SIGMOD Demo*, page 671, 2003.

[26] 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.

[27] X. Zhang and E. A. Rundensteiner. XAT: XML Algebra for the Rainbow System. Technical Report WPI-CS-TR-02-24, Worcester Polytechnic Institute, July 2002.