Efficiently Supporting Order in XML Query Processing

Maged El-Sayed [a,*], Katica Dimitrova [b,**], Elke A. Rundensteiner [a]

[a] *Department of Computer Science*
Worcester Polytechnic Institute Worcester, MA 01609

[b] *Microsoft Corporation*
One Microsoft Way, Redmond, WA 98052

Abstract

XML is an ordered data model and XQuery expressions return results that have a well-defined order. However, little work on how order is supported in XML query processing has been done to date. In this paper we study the issues related to handling order in the XML context, namely challenges imposed by the XML data model, the variety of order requirements of the XQuery language, and the need to maintain order in the presence of updates to the XML data. We propose an efficient solution that addresses all these issues. Our solution is based on a key encoding for XML nodes that serves as node identity and at the same time encodes order. We design rules for encoding order of processed XML nodes based on the XML algebraic query execution model and the node key encoding. These rules do not require any actual sorting for intermediate results during execution. Our approach enables efficient order-sensitive incremental view maintenance as it makes most XML algebra operators distributive with respect to bag union. We prove the correctness of our order encoding approach. Our approach is implemented and integrated with *Rainbow*, an XML data management system developed at WPI. We have tested the efficiency of our approach using queries that have different order requirements. We have also measured the relative cost of different components related to our order solution in different types of queries. In general the overhead of maintaining order in our approach is very small relative to the query processing time.

Key words: XML Query, XQuery, Order in XML, Query Algebra, XML Data Management Systems.

* Corresponding author. *Email address*: maged@cs.wpi.edu (M. El-Sayed).

** All work done by Katica Dimitrova for this paper has been accomplished while she was a graduate student in the Department of Computer Science at Worcester Polytechnic Institute, Worcester, MA.

1 Introduction

1.1 XML and Order

XML has been widely accepted as data format for modeling and exchanging data for internet applications. Unlike most common data models including semi-structured, relational and object-oriented data models, XML data is order-sensitive. Supporting XML's ordered data model is crucial for many domains. An example is content management where document data is intrinsically ordered and where queries often need to rely on this order [20]. For example, if Shakespeare's plays are modeled as XML documents, the order among acts in plays is relevant. Then queries asking for a certain act in a play given its order must be supported.

XQuery [26], a World Wide Web Consortium (W3C) working draft of an XML query language, has been proposed as standard for querying XML. By the W3C specifications [26], XQuery expressions return results that have a well-defined order, unless otherwise is specified. The result of a path expression is always returned in document order [24]. The order in the result of a $FLWOR$ expression can in addition be imposed by the expression itself in many ways, as we will describe next. Hence, the result of an XQuery expression reflects in an interrelated manner both the implicit XML document order and the explicitly imposed order by the XQuery expression.

1.2 Problem Description

Support for such order when processing XQuery queries can severely affect query optimization opportunities. Thus, a major performance hit may result [26]. For this reason the XQuery language provides a function, named $unordered()$, that can be used for those expressions where the order of the result is not significant [26]. This allows us to turn sequences processed during query execution into sets. Set-oriented processing is known to offer potential opportunities for optimization.

One challenge in handling XML order is that the order of the result of an XQuery expression may follow (1) document order, (2) query order imposed by the *order by* clause, (3) query order imposed by the nesting of the query for and let clauses, and (4) query order imposed by the query $return$ clause or by the new result construction, or (5) a combination of any of the above.

The problem of incremental XML view maintenance poses unique challenges compared to the incremental maintenance of relational or even object-oriented views. XML views have to be refreshed correctly not only concerning the view content but also concerning the order of the view result document. In the relational context,

for example, order is of interest only if the *Order By* operation is explicitly present in the view definition. Even then, a possible solution is to maintain an unordered auxiliary view, and only recompute the ordered view on demand on the final output data. This is because all ordering is done uniformly based on sorting on some attribute value at the end of query processing. Such approach does not apply to the XML context, where all operations have to be order sensitive. Even if explicit reordering occurs (for example, due to an $OrderBy$ clause in the view definition) it does not necessarily completely reorder the XML view result. The internal elements (i.e., children/descendants elements) of the element(s) on which the ordering was performed still might be returned in document order.

Given that the order cannot always be ignored, efficient techniques for handling order in XML query processing must also be devised. That is, we need to have the ability to support order for processing queries and updates on data and on materialized views. At the same we need to minimize the overhead that comes with handling order. Some work has been proposed for supporting order in XML query processing [6,10,14,20] yet these solutions did not support all types of XQuery order or came with high overhead cost.

1.3 Our Approach

In this paper, we provide a general solution to the open problem of efficiently handling order in XML query processing and view maintenance. The work presented here has been conducted as part of the Rainbow system [27], an integrated XML data management system that supports XQuery. The *Rainbow Storage Manager* supports efficient retrieval and updates for both base XML data and derived intermediate XML data fragments. *Rainbow* uses a unique node identifier encoding for efficient reference-based query execution and efficient view maintenance.

Our solution supports all different types of XQuery order, as decried above. It also migrates the ordered bag semantics of intermediate query results into a non-ordered bag semantics. This way our approach removes, for most query operators, the overhead of maintaining order at the level of individual algebra operators. At the same time the order of intermediate results is preserved implicitly using the proposed order-encoding scheme. This opens up additional opportunities for query optimization. Sorting in our approach is necessary only when de-referencing the final XML result. Even then, typically, only partial sorting is required. This is mainly because the storage mechanism we use returns many parts of the result in the desired order, as we will describe later.

The contributions of this paper include: (1) We identify the challenges associated with handling order in the context of XML query processing and view maintenance. (2) We propose an efficient order encoding strategy that preserves order in

XML algebraic query processing. This strategy removes the overhead for each individual algebra operator to have to maintain order, It also removes the need for unnecessary sorting of intermediate data. In other words it migrates the ordered bag semantics of intermediate data into non-ordered bag semantics. (3) We prove the correctness of the proposed approach in that it ensures ordered semantics. (4) We have implemented and integrated our order strategy into the Rainbow XML data management system [27]. (5) We show, via an experimental study, that our order preserving approach comes with a relatively small overhead on query execution and view maintenance.

1.4 Paper Outline

The reminder of this paper is organized as follows. Section 2 describes related work. Section 3 introduces the XML algebra. In Section 4 we classify the challenges of maintaining order in XML query processing, while Section 5 describes our order solution. Section 6 discusses the cost and implications of our proposed order solution. Section 7 analyzes the results of our experiments, while Section 8 concludes this paper.

2 Related Work

Many solutions for XML data management use relational database technology [7,19,20] as the underlying storage medium. Supporting the ordered nature of the XML data in the relational model context is an issue since order information is lost while converting from XML to the relational data representation [15]. Many solutions for semi-structured data have been extended to support XML data [8,12]. These solutions tend not to support XQuery, and more importantly, do not support order requirements of XQuery expressions.

Concurrently with these efforts to exploit existing database technologies, native XML storage manager systems [3,11] have also been proposed. An advantage of such native storage is that XML documents may be clustered in physical XML document order, thus facilitating efficient children/descendant access. [3] for example, supports four different clusterings based on different document order at the index storage. Such tree navigation is very frequent in XML query processing [10]. [21] shows that customized XML storage solutions perform better than other storage solutions when dealing with XML documents without DTDs or with documents with DTDs that have cycles.

Object identity is widely used in semi-structured databases [12,13] and in object-oriented databases [5]. W3C recommends that each node in an XML document

should have a node identifier [26]. Some XML algebra operators might be able to perform functionalities like duplicate elimination using only the node identifiers without the need to access the actual data [26]. An alternative solution in XML is to use the *id* attribute to identify XML elements. This is not a good solution since (1) such attribute is optional and (2) it can only be defined for element nodes. Hence, XML systems often generate and assign node identifiers for all nodes in the XML tree. Some of these identifiers can serve both as node identification and node order at the same time, like in [10,20].

Several techniques have been proposed for encoding order of XML documents. [20] describes three order encoding methods: global, local and dewey encodings. In the global encoding method, each node is assigned a number that represents the node's absolute position in the document, while in the local encoding method each node is assigned a number that represents its relative position among its siblings. The dewey order encodes the full path from the root node to the current node. The dewey order is shown to outperform the other two on workloads composed of both queries and updates. The main disadvantage of all these order encoding methods is that in the presence of updates renumbering might be needed for certain portions of the XML tree. [3] proposed an order encoding for XML documents nodes (called *FlexKey*) which is based on dewey. This method avoids the problem of renumbering in the case of updates by using variable length byte strings instead of numbers. Another encoding technique, used in [1,10], associates a numeric $start$ and end label with each data node in the XML document. The intervals between these labels are defined such that every descendant node has an interval that is strictly included in its ancestors's interval. By adding $level$ to the label of each node this order encoding technique allows for parent-child and ancestor-descendant relationships to be found. One disadvantage of this method is that re-labeling of nodes might be required if a large number of insertions are taking place within the same small label range. In addition, it is not possible to derive directly the label of a parent (or an ancestor) of a node given only its label, unlike in the case of *Flexkey* [3] and Dewey [20].

The Agora system [14], which stores XML in relational tables, provides support for handling order-sensitive XQuery expressions. XQuery queries are first normalized, then translated and rewritten into SQL queries to be executed over the relational tables. However, this solution is limited to XQuery queries that semantically match SQL and can successfully be translated and rewritten into SQL. Additionally, order handling is an expensive process where an XQuery is translated into many SQL queries requiring several passes and materializing of intermediate XML results.

[2] and [18] introduce mechanisms to publish relational data and object-relational data as XML documents. These solutions provide support for mainly document order. The use of a sorted outer union approach is proposed to retrieve the relational data needed for constructing XML documents when the resulting XML document does not fit into main memory. However, this approach may perform unnecessary

additional work as it produces a total ordering even when only partial ordering is sufficient. [20] proposed a solution for supporting ordered XML query processing using the relational database technology. This solution mainly focuses on handling XPath expressions order, and provides support for some XQuery order-based functionalities like *Before* and *after* operators and the *range* predicate. The work in [20] focuses on document order and does not handle different types of order imposed by XQuery expressions. Timber [10], a native XML data management system, provides support for document order and query order. However, to preserve order, sorting for some of the intermediate results appears to be required during execution [10]. The order handling strategy in Timber is built on top of the node $start$-end-$level$ labeling described above. Hence, it suffers from the disadvantages of that labeling techniques described above. [6] introduces a solution for maintaining XML document order that works in both a static and dynamic database environment. However, re-labeling of nodes might be required in some cases. Also [6] does not address order imposed by XML queries.

Many incremental solutions have been proposed for the problem of maintaining semi-structured and XML views [13,16,29], none of these solutions have supported order-sensitive view maintenance. Our order solution establishes the foundation for the first order-sensitive view maintenance solution for XML views [4].

Our proposed order approach supports different types of XQuery order, namely document order and order imposed by the query itself in a variety of ways. A key point in our solution is that the order is implicitly encoded in the node identifier and in the intermediate result schema in a way that allows the migration of intermediate results from ordered bag semantics into non-order bag semantics. Unlike in [10] most of our operators no longer need to be aware of the order of data they process. Also we do not need to incorporate any sorting operations for intermediate results. Our operators are distributive with respect to bag union. This opens up more optimization opportunities and allows for efficient incremental view maintenance [4].

3 Background: XML Algebra

In this section we introduce the XML algebra of Rainbow [28]. While our order handling solution is illustrated using this algebra, its principles are generally applicable.

3.1 Basic Notations

We adopt the XML standard defined by W3C [22]. An XML node refers to either an element, attribute, or text in an XML document. XML nodes are considered

duplicates based on their equality by node identity denoted by $n1 == n2$ [25].

Definition 3.1 *Given m sequences of XML nodes, let $seq_j = (n_{1j}, n_{2j}, ..n_{k_jj})$, $1 \leq j \leq m$, $k_j \geq 0$, n_{ij} is an XML node, $1 \leq i \leq k_j$.* ***Order sensitive bag union*** *of such sequences is defined as: $\overset{\circ}{\uplus}_{j=1}^{m} seq_j \overset{def}{=} (n_{11}, n_{21}, ...n_{k_1 1}, n_{12}, ...n_{k_2 2}, ..., n_{1m}, ...n_{k_m m})$.* ***Union*** *of such sequences is defined as $\bigcup_{j=1}^{m} seq_j \overset{def}{=} \{c_1, c_2, ..., c_s\}$ $|(\forall n_{ij}, 1 \leq j \leq m, 1 \leq i \leq k_j)(\exists! c_l, 1 \leq l \leq s)(n_{ij} == c_l)$.*

Order sensitive bag union of sequences concatenates the sequences into one resulting sequence. Union creates a set of all the unique nodes contained in the input sequences, i.e., duplicates are removed. We use $\uplus$ to denote bag union of sequences of XML nodes and $\dot{-}$ to denote bag difference of sequences of XML nodes. When a single XML node appears as argument for $\overset{\circ}{\uplus}$, $\bigcup$, $\uplus$ or $\dot{-}$, it is treated as a singleton sequence [23].

We use the term **path** to refer to a path expression [26] consisting of any combination of forward steps, including $//$ and $*$. The sequence of children of the XML node n located by the path $path$ and arranged in document order is denoted as $\overset{\circ}{\phi}(path : n)$. The notation $\overset{\circ}{\phi}(path : n)[i]$ represents the i^{th} element in that sequence. The number of children of the XML node n that can be reached by following the path $path$ is denoted as $|\overset{\circ}{\phi}(path : n)|$. Hence, $\overset{\circ}{\phi}(path : n) \overset{def}{=} (n_1, n_2, ...n_k)|(n_i = \overset{\circ}{\phi}(path : n)[i], 1 \leq i \leq k) \wedge (k = |\overset{\circ}{\phi}(path : n)|)$. For example, for n being the XML node $prices$ from Figure 1 (a), and $path =$ "$//price$", then $\overset{\circ}{\phi}(path : n) = (<price>39.95</price>, <price>65.95</price>)$.

The sequence of extracted children located by the path $path$ from each of the nodes in the sequence $seq = (r_1, r_2, ...r_k)$ respectively is denoted as $\overset{\circ}{\phi}(path : seq)$. That is, $\overset{\circ}{\phi}(path : seq) \overset{def}{=} \overset{\circ}{\uplus}_{i=1}^{k} \overset{\circ}{\phi}(path : r_i)$. The notation $\overset{\circ}{\phi}(path : seq)[i]$ stands for the i^{th} element of that sequence. $|\overset{\circ}{\phi}(path : seq)| = \sum_{i=1}^{k} |\overset{\circ}{\phi}(path : r_i)|$. The notation $\phi(path : seq)$ stands for the corresponding unordered sequence. As $|\phi(path : seq)| = |\overset{\circ}{\phi}(path : seq)|$, for convenience we also use the notation $|\phi(path : seq)|$ for the cardinality of $\overset{\circ}{\phi}(path : seq)$ in later sections.

3.2 The XML Algebra XAT

We use XQuery [26], a World Wide Web Consortium working draft for an XML query language, as query language. Figure 1 shows our running example in particular (a) two source XML documents "bib.xml" and "price.xml", (b) an XQuery expression defined over these two documents and (c) the result "result.xml" generated by executing the XQuery expression over the source documents. The XQuery

Fig. 1. (a) Two source XML documents "bib.xml" and "price.xml", (b) XQuery expression and (3) XML result "result.xml"

expression is translated into an XML algebraic representation. Given that to date no standard XML algebra for XQuery processing has emerged, we use the XML algebra called XAT [28]. The Rainbow XML data management system [27] developed at WPI is based on this algebra. The data model for the XAT algebra is a tabular model called XAT table. Typically, an XAT operator takes as input one or more XAT tables and produces an XAT table as output. An **XAT table** R is an order-sensitive table of p tuples t_j, $1 \leq j \leq p$, $p \geq 0$ that is $R = (t_1, t_2, .., t_p)$. The column names in an XAT table schema of R represent either a variable binding from the user-specified XQuery, e.g., $\$b$, or an internally generated variable name, e.g., $\$col_1$. Each tuple t_j (1 ≤ j ≤ p) is a sequence of k cells c_{ij} (1 ≤ i ≤ k), that is $t_j = (c_{1j}, c_{2j}, ..., c_{kj})$, where k is the number of columns. Each cell c_{ij} (1 ≤ i ≤ k, 1 ≤ j ≤ p) with col_i in a tuple t_j, denoted by $t_j[col_i]$, can store an XML node or a sequence of nodes. Atomic values are treated as text nodes.

XAT Operators. In general, an XAT operator is denoted as $op_{in}^{out}(s)$, where op is the operator type symbol, in represents the input parameters, out the newly produced output column that is to be appended to the output table generated by the operator and s the input XAT table(s) (an exception is the $Source$ operator where s represents an XML document). Some of the XAT operators along side with their XAT tables are shown in Figure 2. The XAT algebra tree in Figure 2 is one possible execution plan for the query in Figure 1(b). Below we introduce the core subset of the XAT algebra operators [28].

A subset of the XAT operators corresponds to the relational complete subset of the XAT algebra including *Select* $\sigma_c(R)$, *Cartesian Product* $\times(R,P)$, *Theta Join* $\bowtie_c (R,P)$, *Left Outer Join* $\overset{\circ}{\bowtie}_{Lc}(R,P)$, *Distinct* $\delta_{col}(R)$, *Group By* $\gamma_{col[1..n]}(R,func)$, *Order By* $\tau_{col[1..n]}(R)$, where R and P denote XAT tables. Those operators are equivalent to their relational counterparts [1], with the additional responsibility to reflect the order among the tuples in their input XAT table(s) to the order among the tuples in their output XAT table. $Distinct$ and *Group By* are the only operators in the XAT algebra that output an unordered XAT table, following the specification in [26]. *Order By*, like its relational counterpart, orders the tuples by the values in the columns given as arguments.

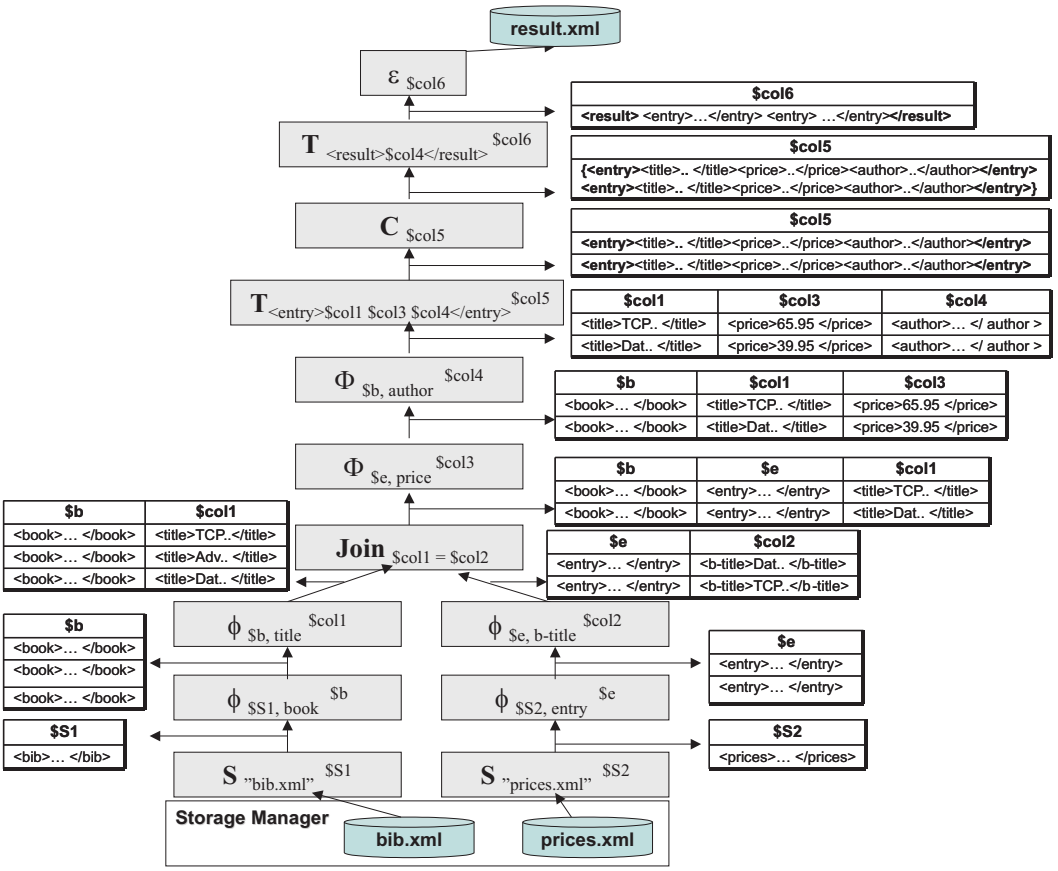


Fig. 2. The algebra tree for the XQuery in Figure1(b)

We now describe the XML-specific operators. The full description of the XAT algebra can be found in [28].

Source $S^{col'}_{xmlDoc}$ is always a leaf node in an algebra tree. It takes the XML document $xmlDoc$ and outputs an XAT table with a single column col' and a single tuple $tout_1 = (c_{11})$, where c_{11} contains the entire XML document.

[1] The operator *Group By* here is more powerful than its relational counterpart as it may take any arbitrary sub-query or function. This allows the *Group By* to perform nesting operations as well as grouping operations.

Navigate Unnest $\phi^{col'}_{col,path}(R)$ unnests the element-subelement relationship. For each tuple tin_j from the input XAT table R, it creates a sequence of m output tuples $tout_j^{(l)}$, where $1 \leq l \leq m$, $m = |\phi(path : tin_j[col])|$, $tout_j^{(l)}[col'] = \overset{\circ}{\phi}(path : tin_j[col])[l]$. The tuples $tout_j^{(l)}$ are ordered by major order on j and minor order on l. The $\phi^{\$col1}_{\$b,title}$ operator in Figure 2 generates one tuple for each "title" element we navigate to form the "book" elements in the input XAT table. This result in three tuples in the output XAT table, a tuple for each "title" element.

Navigate Collection $\Phi^{col'}_{col,path}(R)$ is similar to *Navigate Unnest*, except it places all the extracted children of one input tuple into one single cell. Thus it outputs only one single output tuple for each tuple in the input. For each tuple tin_j from R, it creates one output tuple $tout_j$, where $tout_j[col'] = \overset{\circ}{\phi}(path : tin_j[col])$.

Combine $C_{col}(R)$ groups the content of all cells corresponding to col into one sequence (with duplicates). Given the input R with m tuples tin_j, $1 \leq j \leq m$, *Combine* outputs one tuple $tout = (c)$, where $tout[col] = c = \overset{\circ}{\uplus}_{j=1}^{m} tin_j[col]$. $Combine$ has only column col in its output XAT table. The $C_{\$col5}$ operator in Figure 2 grouped all the "entry" elements in $\$col5$ tuples into one cell.

Tagger $T_p^{col}(R)$ constructs new XML nodes by applying the tagging pattern p to each input tuple. A pattern p is a template of a valid XML fragment [22] with parameters being column names, e.g., *<result>$col5</result>*. For each tuple tin_j from R, it creates one output tuple $tout_j$, where $tout_j[col]$ contains the constructed XML node obtained by evaluating the pattern p for the values in tin_j. For example, the $T^{\$col5}_{<entry>\$col1\$col3\$col4</entry>}$ in Figure 2 constructs a new "entry" node from the "title", "price", and "author" nodes for each input tuple.

XML Unique $\upsilon^{col'}_{col}(R)$ removes duplicate for sequences of XML nodes by node identifier. For each tuple tin_j from R, it creates one output tuple $tout_j$, where $tout_j[col']$ is a sequence containing the unique members in $tin_j[col]$ after removing duplicates by node identifier. **XML Union** $\overset{x\ col}{\cup}_{col1,col2}(R)$ is used to union multiple sequences into one sequence. For each tuple tin_j from R, it creates one output tuple $tout_j$, where $tout_j[col]$ is a sequence containing the members of the set $tin_j[col1] \cup tin_j[col2]$. The other two XML collection operators, **XML Intersection** $\overset{x\ col}{\cap}_{col1,col2}(R)$ and **XML Difference** $\overset{x\ col}{-}_{col1,col2}(R)$, perform intersection and difference between two sequences. Note that the operators *XML Union*, *XML Intersection* and *XML Difference* perform set operations on columns in a single XAT table, not on multiple XAT tables.

Expose $\epsilon_{col}(R)$ appears as a root node of an algebra tree. It outputs the content of column col into textual XML.

By definition, all columns from the input table are retained in the output table of an

operator. An additional column may be added the output table of an operator, except for some operators that do not require an additional column (e.g. the *Combine* operator). Such schema of a table is called *Full Schema (FS)*. However, not all the columns may be utilized by operators higher in the algebra tree. *Minimum Schema (MS)* of the output XAT table of an operator is defined as the subsequence of all columns, retaining only the columns needed later by the ancestors of that operator [28]. The process of determining the *Minimum Schema* is called **Schema Cleanup** and is described in [28].

In the XAT algebra tree shown in Figure 2 we $navigate$ to "title" elements from "book" elements in "bib.xml" and $navigate$ to "b-title" elements from "entry" elements in "prices.xml". We then perform a $join$ operation based on the "title" and "b-title" elements values. We then $navigate$ to the "price" elements from the "entry" elements and $navigate$ to the "author" elements from the "book" elements. Next we $construct$ new "entry" nodes from the "title", the "price", and the "author" elements. We place all the created "result" node in a collection, using the *Combine* operator, and tag this collection of "entry" nodes using the "result" tag. Finally we use the $expose$ operator to extract the result as an XML document ("result.xml"). In Figure 2 we only show the columns that are in the *Minimum Schema* for each table. All other columns have been removed due to *Schema Cleanup* [28].

4 Challenges of Handling in XML Query Processing

4.1 Challenges Posed by the Data Model

The query execution model of ordered-sensitive XML views can be seen as a *sequence of sequences*, where each of the sequences can have one or more XML nodes. An XML node in a sequence can be a simple node like an attribute or a text node or it can be an XML tree (an element node). In terms of our data model, the XAT table corresponds to the container sequence and the tuples in that table are the sequences inside the container sequence. Each cell (in a tuple) can store a single node or a sequence of nodes. Given such a data model, three order levels exist:

1) Order among processed sequences (tuples in an XAT table).

2) Order among nodes in a processed sequence of XML nodes (nodes in a cell in an XAT table).

3) Order among internal nodes (children/descendants) of processed XML nodes.

The processed nodes themselves may be either original nodes from the source document or nodes constructed during query execution. And the order defined for any

of those three levels may follow the source document order or may follow a new order imposed by the query. In some cases order might not be of importance.

4.2 Challenges Posed by the Different Order Requirements of the XML Query Language

We classify the order that an XQuery expression can reflect to its result into four main types:

1) **Document Order.** Document order is the order of nodes as they appear in the source XML documents. XQuery expressions typically return result in document order unless otherwise is specified by the query. This order might be present in base nodes exposed in the result. It also might be present in constructed nodes that follow the document order of the base nodes they are derived from.

2) **Query Order Imposed By the Query *order by* Clauses.** The query might have one or more *order by* clause(s) imposing new order to certain parts of the returned result.

3) **Query Order Imposed by the Nesting of Variable Binding in the Query** for **and** let **Clauses.** Nesting of variables in the for and let clauses in an XQuery $FLWOR$ expression also imposes a certain order based the order of the variables. For example, for a $FLWOR$ expression embedded into another $FLWOR$ expression we expect that, in general, a variable in the outside for clause places a major influence on the order while a variable in the inside for clause places a minor influence on the order. The same order semantics applies to the order among multiple variable bindings in the same for or let clause.

4) **Query Order Imposed by the Query** $return$ **clauses and by the New Result Construction.** The order in which variables are specified in the return clause determines the order of data bound to these variables.

Often the XQuery result reflects a mixture of more than one of the order types listed above. This makes handling XQuery order a complex issue.

On the query algebra level, different operators in the XML algebra deal with order in a different way. Here are some examples:

- The operator *Navigate Collection* $\Phi^{col'}_{col,path}(R)$ processes one tuple at a time, without requiring to access other tuples nor modifying the order among the tuples. Moreover, for each tuple in the input table it produce exactly one tuple in the output table.
- The operator *Tagger Tagger* $T^{col}_{p}(R)$ also preserves the relative order among the tuples it process. In addition it defines order among its internal nodes.

- The operator *Combine* $C_{col}(R)$ destroys the order among the tuples it process. It groups all the nodes from its input column in one cell and outputs only one tuple in the output XAT table that contains that cell. This raises the issue of maintaining order between those nodes.
- For the *Join* operator $\bowtie_c (R, P)$, the order of tuples in the output table of the *Join* operator depends on the order of tuples in its input tables. The order in its output table follows the order of the left input table R as a major order and the right table P as a minor order.
- The *Order By* operator $\tau_{col[1..n]}(R)$ destroys the order of the input table and imposes a new order based on a certain criteria. Hence the output table will have a new computed order based on the order of some column values.
- The operator $Expose\ \epsilon_{col}(R)$ outputs an XML document rather than an XAT table. This document is extracted as a tree from a col in the input XAT table. The extracted tree has order among its elements that reflects all previous order decisions.

We will discuss how different operators handle order in more detail in Section 5.

4.3 Challenges Posed by Order-sensitive View Maintenance

The problem of the incremental maintenance of XML views poses additional challenge. View maintenance of ordered XML data is difficult for two reasons [13]: (1) positions of the element may change dynamically during update time and (2) positions of elements may be different in views and in the source data.

It is essential to have a mechanism for encoding source XML nodes in a way that avoids reordering (re-labeling) source nodes on updates. It is also essential to maintain the order among the propagated nodes and sequences. This issue is similar to the that of maintaining order among processed nodes and sequences discussed above. Two other issues appear here (1) how to derive and maintain the relative order of the propagated updates to the order of the previously processed data, and (2) how to avoid re-ordering of nodes in the result when applying propagated update to the view result. Without an efficient solution, materialization of large auxiliary data structures and expensive scans of them might be needed to enable order-sensitive view maintenance. For example, to determine the order of an inserted tuple in an XAT table we might need to materialize and to scan the input or output tables to determine the right order of the inserted tuple. Our goal here is to provide an order handling technique that facilitates not only efficient order-sensitive query processing but also efficient order-sensitive view maintenance.

5 Maintaining XML Order

The requirement of preserving order, as described in Section 4, makes the XML query execution and view maintenance significantly different from the relational case. The two obvious solutions are: (1) relying on the physical sequential storage medium to be always ordered, or (2) assigning order values to processed sequences and nodes. Both solutions are not efficient especially in the presence of incremental updates.

Our solution for handling order relies on three main principles: (1) the underlying *Storage Manger* is capable of returning source document nodes in document order, (2) order is ignored when processing XML intermediate results, and (3) at the end of query processing and when generating the final result sorting is performed (typically only partial sorting) to return the result in the desired order. Our *Storage Manager* relies on the *MASS* system [3], also developed at WPI, for providing scalable storage and indexing for XML data with efficient update performance. The *Storage Manager* provides interfaces for storing and retrieving XML nodes (both original nodes and constructed nodes). *MASS* guarantees that when retrieving descendants of original XML nodes they are returned in document order, eliminating the need for sorting them at the result generation time. *MASS* provides scalable I/O performance for all *XPath* axes. Moreover, it provides an integrated indexing support for *XPath* node tests, position predicates and count aggregations.

5.1 Node Identifier and Node Order

In many cases the order among processed XML nodes and collections depends on the order of their source document. In other words, the order among the tuples in an XAT table (and among nodes in a cell) depends on the source document order of the XML nodes present in these tuples (cell). Our query processing model uses node identifiers during query execution time. Hence, a node identity that serves the dual purpose of node identifier and order encoding is beneficial for both query processing and order handling. We also require the node identity of a base node to encode the unique path of that node in the XML tree and captures the order at each level along the path. We have thus considered techniques proposed in the literature for encoding order in XML data [1,3,10,20]. The lexicographical order encoding technique proposed in [3] that does not require reordering on updates is proposed. It is analogous to the Dewey ordering [20], except rather than using numbers in the encoding, it uses variable length strings. First, for each document node a variable length byte string key is assigned, such that lexicographical ordering of all sibling nodes yields their relative document ordering. The identity of each node is equal to the concatenation of all keys of its ancestor nodes and of that node's own key (see Figure 3). This order-reflecting node identity encoding is called **FlexKey**. We use

the notation $k_1 \prec k_2$ to note that *FlexKey* k_1 lexicographically precedes *FlexKey* k_2.

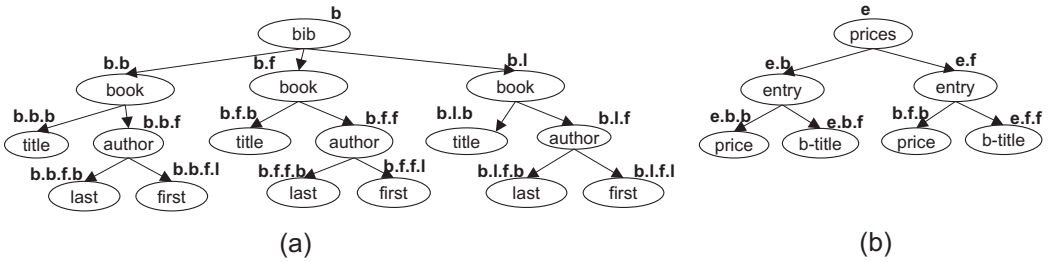


Fig. 3. Lexicographical order encoding of the two XML documents "bib.xml" and "prices.xml" presented in Figure 1(a).

The **FlexKey** encoding is well suited for query execution and for view maintenance because it has the following properties:

- It identifies a unique path from the root to the node. Hence the parent-child and ancestor-descendant containment relationships between nodes can easily be determined without the need to access the actual data. Accessing this relationship is a frequent operation in XML query execution.
- It embeds the relative order among nodes in the same XML tree in each node. Hence the order between any nodes can easily be determined (regardless of the level) by comparing their *FlexKey*s lexicographically.
- It does not require reordering on updates because of the use of variable length strings instead of numbers for encoding order. We can always create new gaps by extending the string by adding more letters. We discuss that in more detail in Section 6.

Base Nodes. We use *FlexKey*s for encoding the node identities of all nodes in the source XML document. That is, we assume that any given XML document used as source data has *FlexKey*s assigned to all of its nodes. For reducing redundant updates and avoiding duplicated storage we mainly store references (*FlexKey*s) in the XAT tables rather than actual XML data. This is sufficient as the *FlexKey*s serve as node identifiers and also capture the order. From here on, when we refer to a cell in a tuple we mean the *FlexKey* or the collection (or sequence) of *FlexKey*s stored in that cell. The actual XML data is stored only once in the *Storage Manager*. Figure 4 illustrates the usage of *FlexKey*s as references to source XML nodes. *FlexKey*s are used for accessing that data when needed by some operator. For example, the *Navigate* operator $\phi^{\$col1}_{\$S1,book}$ retrieves the "book" children of the root node of "bib.xml" from the *Storage Manager*, and places their *FlexKey*s in the output XAT table.

Constructed Nodes. We also use *FlexKey*s to encode the node identity of any constructed nodes either in the intermediate result or in the final extent. The *FlexKey*s assigned to constructed nodes are locally unique. Rather than instantiating the actual XML fragments in our system, we only store a skeleton representing their

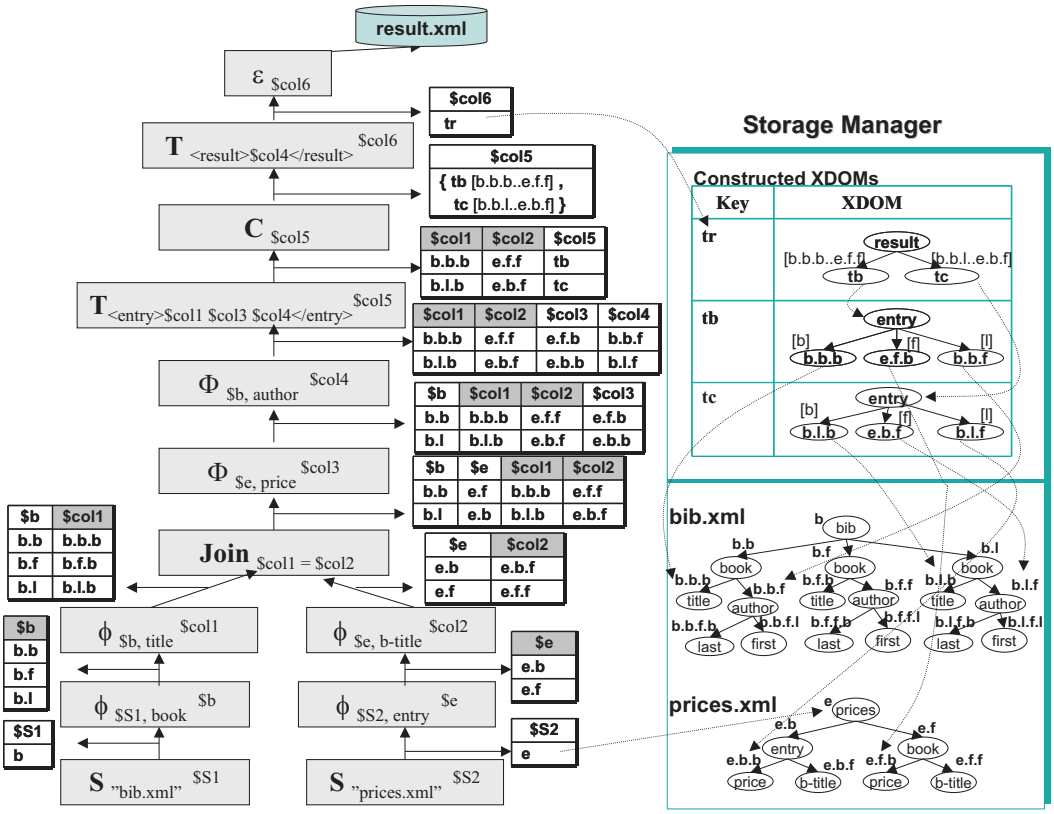


Fig. 4. Execution using *FlexKey*s for the XQuery expression in Figure1(b). Shaded columns represent *Order Schema*.

structure in the *Storage Manager*. References (*FlexKey*s) to other source data or to constructed nodes that are included in the newly constructed node are kept. For example, in Figure 4, although the constructed node tr is representing the whole output view extent, it is only stored as *<result>tb tc</result>*. When the constructed node is created, the *FlexKey* assigned to it reflects only its identifier and does not reflect its order. This is because the order of a constructed node at its creation time is just an intermediate order at a certain point of query execution. It does not necessarily reflect the desired final result order. We assign the order information to the constructed node at a later stage (when the constructed nodes are placed into a sequence with other nodes or when it becomes part of other constructed nodes). When defining the order of a constructed node we use an additional key for encoding order that we attach to the *FlexKey* of the constructed node. We call such additional key *Overriding Order*. We will discuss the *Overriding Order* encoding in more detail in Section5.3.

Composed Keys. In addition to the *FlexKey*s described above, we may also use *FlexKey*s created as a composition of other *FlexKey*s. This is mainly for maintaining any order that is different than the document order in sequences of XML nodes (see Section 5.3). For example, the *FlexKey* $k = \text{“}b.b.b..b.b.d\text{”}$ is a composition of the *FlexKey*s $k_1 = \text{“}b.b.b\text{”}$ and $k_2 = \text{“}b.b.d\text{”}$, where “..” is used as delimiter. We denote

this by $k = compose(k_1, k_2)$.

Now we discuss in detail how we maintain the order of processed XML data. We organize our discussion based on the query execution data model into (1) order among sequences of XML nodes, (2) order among nodes in a sequence of XML nodes, and (3) order among internal (children/descendant) nodes of processed nodes (XML fragments).

5.2 Maintaining Order Among Sequences of XML Nodes

We observe that the order among the tuples (sequences of XML nodes) in an XAT table can be determined, in some cases, by comparing the *FlexKey*s stored in cells corresponding to some of the columns. For two tuples in an XAT table, we define the expression $before(t_1, t_2)$ to be $true$ if the tuple t_1 should semantically be ordered before the tuple t_2, $false$ if t_2 is semantically before t_1 and $undefined$ if the order between the two tuples is irrelevant. For example, consider the tuples t_1 = (b.b.b, e.f.f, e.f.b, b.b.f) and t_2 = (b.l.b, e.b.f, e.b.b, b.l.f) in the input XAT table of the operator $T^{\$col5}_{<entry>\$col1\$col3\$col4</entry>}$ in Figure 4. Here t_1 should be before t_2, that is $before(t_1, t_2)$ is true. This can be deduced by comparing the *FlexKey*s in $t_1[\$col1, \$col2]$ and $t_2[\$col1, \$col2]$ lexicographically. We will show that this is not a coincidence. That is, the relative order among the tuples in an XAT table is indeed encoded in the keys contained in certain columns. Thus it can be determined solely by comparing those *FlexKey*s. Such columns are said to compose the *Order Schema* of the table. For any two tuples in the output XAT table of the *Distinct* operator the relative order is undefined.

Definition 5.1 *The **Order Schema** $OS_R = (on_1, on_2, ...on_m)$ of an XAT table R in an algebra tree is a sequence of column names on_i, $1 \leq i \leq m$, computed following the rules in Table 1 in a postorder traversal of the algebra tree.*

Two tuples are compared lexicographically as follows.

Definition 5.2 *For two tuples t_1 and t_2 from an XAT table R with $OS_R = (on_1, on_2, ...on_m)$, the comparison operation $\prec$ is defined by: $t_1 \prec t_2 \Leftrightarrow (\exists j, 1 \leq j \leq m)(((\forall i, 1 \leq i < j)(t_1[on_i] == t_2[on_i])) \wedge (t_1[on_j] \prec t_2[on_j]))$*

The rules in Table 1 guarantee that cells corresponding to the *Order Schema* never contain sequences, only single keys. The rules are derived from the semantics of the operators and rely on the properties of the *FlexKey*s.

For example, let us consider the rule for computing the *Order Schema* of the operator *Navigate Unnest* $\phi^{col'}_{col,path}(R)$, when the column col is the last column in the

[2] The column col'' by definition is responsible for holding keys such that (I) and (II) hold.

Cat.	Operator op	OS^*_Q
I	$T_p^{col}(R)$, $\Phi_{col,path}^{col'}(R)$, $\overset{x\,col}{\cup}_{col1,col2}(R)$, $\upsilon_{col}^{col'}(R)$, $\overset{x\,col}{\cap}_{col1,col2}(R)$, $\overset{x\,col}{-}_{col1,col2}(R)$, $\sigma_c(R)$	OS_R
II	$S_{xmlDoc}^{col'}$, $C_{col}(R)$, $\delta_{col}(R)$, $\gamma_{col[1..n]}(R, fun)$	$\emptyset$
III	$\times(R,P)$, $\bowtie_c(R,P)$, $\overset{\circ}{\bowtie}_{Lc}(R,P)$	$(on_1^{(R)}, on_2^{(R)}, ...on_{mr}^{(R)}, on_1^{(P)}, on_2^{(P)}, ...on_{mp}^{(P)})$ $mr = \|OS_R\|$, $mp = \|OS_P\|$
IV	$\phi_{col,path}^{col'}(R)$	$(on_1^{(R)}, on_2^{(R)}, ...on_p^{(R)}, col')$ if $on_m^{(R)} = col$ then $p = m-1$, else $p = m$.
V	$\tau_{col[1..n]}(R)$	(col''), col'' is new column [2]
VI	$\epsilon_{col}(R)$	N/A
* $Q = op_{in}^{out}(R)$, $OS_R = (on_1^R, on_2^R, ...on_m^R)$		

Table 1
Rules for computing *Order Schema*

Order Schema of the input XAT table R. By the semantics of this operator presented in Section 3, it processes one tuple at a time. However, it may produce zero or more tuples in its output XAT table Q for each tuple in R. The order of any two tuples in Q derived from two different tuples in R should be same as of those they are derived from in R. The order among two tuples derived from the same tuple in R should correspond to the document order of the nodes present in their cells corresponding to col'.

For any two tuples t_1 and t_2 in any XAT table in an XAT algebra tree, if tuple t_1 should semantically be before tuple t_2, then the lexicographical comparison from Definition 5.2 of the tuples always yields $t_1 \prec t_2$. On the contrary, if $t_1 \prec t_2$, then either t_1 should semantically be before t_2 or otherwise the order between these two tuples is irrelevant. This means that the relative order among the tuples is correctly preserved in the *Order Schema*, but the *Order Schema* may impose order among the tuples when such order is semantically irrelevant. In the following theorem, we state this observation more formally. We also prove its correctness.

All columns contained in the *Order Schema* of any table are also contained in the *Full Schema* of that table, except for the column in the *Order Schema* of the output table of the *Order By* operator. Thus, no extra computation is needed for evaluating the *Order Schema*. Moreover, they are often present even in the *Minimum Schema*. The order among the tuples in the output XAT table of the *Order By* operator depends on the values present in the tuples. Thus it is not captured by any of the *FlexKey*s present in the tuple. Thus we explicitly encode it in a new column created for that purpose.

Theorem 5.1 shows that the relative position among the tuples in an XAT table is correctly preserved by the cells in the *Order Schema* of that table.

Theorem 5.1 *For every two tuples $t_1, t_2 \in R$, where R is an XAT table in an XAT algebra tree, with $before(t_1, t_2)$ defined as in Section 3, (I) $before(t_1, t_2) \Rightarrow (t_1 \prec t_2)$, and (II) $(t_1 \prec t_2) \Rightarrow (before(t_1, t_2) \vee (before(t_1, t_2) = undefined))$.*

Proof: *We prove (I) by induction over the height h of the algebra tree, i.e., the maximum number of ancestors of any leaf node. To simplify the proof, we consider any algebra tree even if it does not have an $Expose$ operator as a root, i.e., a superset of what is necessary.*

Base Case: *For $h = 0$, the algebra tree has a single operator node, which is both a root and a leaf. That node must be a $Source$ operator, as each leaf in a valid XAT algebra tree is a $Source$ operator. As the input of $Source$ is an XML document, the output XAT table is the only table in the tree. Since the $Source$ operator outputs only one tuple t, the expression $before(t, t)$ is never* true. *Thus the theorem trivially holds.*

Induction Hypothesis: *For every two tuples $t_1, t_2 \in R$, where R is any XAT table in an XAT algebra tree with height l, $1 \leq l \leq h$, it is true that $before(t_1, t_2) \Rightarrow (t_1 \prec t_2)$.*

Induction Step: *We now consider an XAT algebra tree of height $h + 1$. Let op be the operator at the root of such algebra tree. All children nodes of the root must themselves be roots of algebra trees each of a height not exceeding h. By the induction hypothesis, (I) must hold for all XAT tables in those algebra trees. Thus, (I) holds for all the XAT table(s) that are sources for the operator op. It is only left to show that $before(t_1, t_2) \Rightarrow (t_1 \prec t_2)$ holds for any two tuples t_1 and t_2 in the output XAT table Q of the operator op.*

The operator op can be any XAT operator, excluding the $Source$ operator, as $h + 1 > 1$ and $Source$ can only appear as a leaf node in an XAT algebra tree. We proceed by inspecting the different cases depending on the type of the operator op, following the classification presented in Table 1.

Category I. *These operators process one tuple at a time, without requiring to access other tuples nor modifying the order among the tuples. Moreover, for each tuple in the input table they produce exactly one tuple in the output table, except for the $Select$, which may filter out some tuples. The later is not of significance, as only the relative order among tuples is addressed in this theorem. Hence, if the theorem holds for the tuples in their input XAT table R and $OS_Q = OS_R$, it must also hold for the tuples in their output XAT table Q.*

To prove that formally, we consider any two tuples $tout_1$, $tout_2 \in Q$. Let tin_1,

$tin_2 \in R$, *such that* $tout_1$ *derived from* tin_1 *and* $tout_2$ *derived from* tin_2. *By the induction hypothesis, (I) holds for any two tuples in* R, *hence also for* tin_1 *and* tin_2. *As* $before(tin_1, tin_2) \Leftrightarrow before(tout_1, tout_2)$, *in order to prove* $before(tout_1, tout_2) \Rightarrow (tout_1 \prec tout_2)$ *we only need to show that* $(tin_1 \prec tin_2) \Rightarrow (tout_1 \prec tout_2)$.

As the operators considered do not modify any values in the columns retained from the input tuple, but may only append new columns, it holds that $(\forall i, 1 \leq i \leq |OS_R|)$ $(tout_1[on_i] == tin_1[on_i])$. *Therefore, by Definition 5.2, we have* $(tin_1 \prec tin_2) \Rightarrow (tout_1 \prec tout_2)$.

Category II. *For the operator* $Combine$, *there is at most one tuple in the output XAT table. Hence the reasoning is same as presented for the operator* $Source$ *in the proof for the base case. The operator* $Distinct$ *by definition outputs an unordered XAT table* Q. *Hence for any two tuples* $t_1, t_2 \in Q$, $before(t_1, t_2) = undefined$. *Thus the left hand side of (I) is never* $true$, *so (I) trivially holds.*

Category III. *All the operators in this category belong to the Join family of operators and regarding order have the same behavior. Their output is sorted by the left input table* R *as major order and the right table* P *as minor order (see Section 3). Consider any two tuples* $tout_1$ *and* $tout_2$ *from the output XAT table* Q. *Let* $tout_1$ *be derived from* $tin_1^{(R)}$ *and* $tin_1^{(P)}$ *and* $tout_2$ *be derived from* $tin_2^{(R)}$ *and* $tin_2^{(P)}$, *where* $tin_1^{(R)}, tin_2^{(R)} \in R$ *and* $tin_1^{(P)}, tin_2^{(P)} \in P$. *Thus, by the definition of these operators:* $before(tout_1, tout_2) \Leftrightarrow before(tin_1^{(R)}, tin_2^{(R)}) \vee ((tin_1^{(R)} = tin_2^{(R)}) \wedge before(tin_1^{(P)}, tin_2^{(P)}))$. *Note that for the* $LeftOuterJoin$ *operator there could exist zero to many output tuples that are not derived from any tuple in* P. *But, as there could be at most one such tuple derived from each tuple in* R, *the above statement is still valid.*

There are two cases: (1) $tin_1^{(R)}$ *and* $tin_2^{(R)}$ *are two different tuples from* R, *or (2) both* $tout_1$ *and* $tout_2$ *are derived from the same tuple* $tin^{(R)}$, *i.e.,* $tin_1^{(R)} = tin_2^{(R)} = tin^{(R)}$.

For case (1) it holds that $before(tout_1, tout_2) \Leftrightarrow before(tin_1^{(R)}, tin_2^{(R)})$. *Hence, this case can be easily reduced to that for the operators in Category I.*

For case (2), when $tin_1^{(R)} = tin_2^{(R)} = tin^{(R)}$, *as* $before(tout_1, tout_2) \Leftrightarrow before(tin_1^{(P)}, tin_2^{(P)})$ *and by the induction hypothesis* $before(tin_1^{(P)}, tin_2^{(P)}) \Rightarrow (tin_1^{(P)} \prec tin_2^{(P)})$, *in order to prove* $before(tout_1, tout_2) \Rightarrow (tout_1 \prec tout_2)$, *it is sufficient to show* $(tin_1^{(P)} \prec tin_2^{(P)}) \Rightarrow (tout_1 \prec tout_2)$. *By the rules in Table 1, the* Order Schema *of* Q *contains all the columns from the* Order Schema *of* R, *followed by all the columns from the* Order Schema *of* P. *As the operators considered do not modify any values in the columns retained from the input tuples, it holds that* $(\forall i, 1 \leq i \leq |OS_R|)((tout_1[on_i^{(R)}] == tin^{(R)}[on_i^{(R)}]) \wedge (tout_2[on_i^{(R)}] ==$

$tin^{(R)}[on_i^{(R)}]))$ *and* $(\forall j, 1 \leq j \leq |OS_P|)((tout_1[on_i^{(P)}] == tin_1^{(P)}[on_i^{(P)}]) \wedge (tout_2[on_i^{(P)}] == tin_2^{(P)}[on_i^{(P)}]))$. *Thus,* $(\forall i, 1 \leq i \leq |OS_R|)(tout_1[on_i^{(R)}] == tout_2[on_i^{(R)}])$ *and then by Definition 5.2* $(tin_1^{(P)} \prec tin_2^{(P)}) \Rightarrow (tout_1 \prec tout_2)$.

Category IV. *The operator* Navigate Unnest $\phi_{col,path}^{col'}(R)$ *by its definition presented in Section 3 processes one tuple at time. However, it may produce zero or more tuples in its output XAT table* Q *for each tuple in* R*. Consider any two tuples* $tout_1$ *and* $tout_2$ *from* Q*. There are two cases: (1) Both* $tout_1$ *and* $tout_2$ *are derived from the same tuple* tin*, or (2)* $tout_1$ *is derived from* tin_1 *and* $tout_2$ *is derived from* tin_2*,* $tin_1 \neq tin_2$*.*

For case (1), let l_1 *and* l_2 *be indexes such that* $tout_1[col'] = \phi(path : tin[col])[l_1]$ *and* $tout_2[col'] = \phi(path : tin[col])[l_2]$*. As* $(l_1 < l_2) \Leftrightarrow before(tout_1, tout_2)$*, in order to prove* $before(tout_1, tout_2) \Rightarrow (tout_1 \prec tout_2)$*, it is sufficient to show* $(l_1 < l_2) \Rightarrow (tout_1 \prec tout_2)$*. Suppose* $l_1 < l_2$*. Then, due to the properties of the* FlexKeys *we have* $tout_1[col'] \prec tout_2[col']$*. By the rule in Table 1,* col' *is now part of the* Order Schema *for the output table* Q*. The fact that* $tout_1$ *and* $tout_2$ *are derived from the same tuple* tin *implies that* $(\forall i, i \leq p)(tout_1[on_i] == tout_2[on_i])$*, with* p *the maximum index of the* Order Schema *(basically the new column) as defined in Table 1. Thus, by Definition 5.2,* $on_j = col$ *and* $tout_1 \prec tout_2$*.*

For case (2), because $before(tin_1, tin_2) \Leftrightarrow before(tout_1, tout_2)$ *and by the induction hypothesis* $before(tin_1, tin_2) \Rightarrow (tin_1 \prec tin_2)$*, in order to prove* $before(tout_1, tout_2) \Rightarrow (tout_1 \prec tout_2)$*, it is sufficient to show* $(tin_1 \prec tin_2) \Rightarrow (tout_1 \prec tout_2)$*. Suppose* $tin_1 \prec tin_2$*. Thus a* j *as specified in Definition 5.2 must exist. There are two sub-cases: (2.a)* $j \leq p$*, and (2.b)* $j > p$*, with* p *as in Table 1. Case (2.a) can be easily reduced to that for the operators in Category I, as the cells corresponding to all the* j *columns belonging to the* Order Schema *from* tin_1 *(*tin_2*) are present in an unmodified format in* $tout_1$ *(*$tout_2$*).*

For (2.b), when $(j > p)$*, it must be that* $p = m - 1$ *(which also implies* $on_m = col$*) and* $j = m$ *by the rules in Table 1. This is because* $tin_1 \prec tin_2$*, and thus they must differ on cells corresponding to columns that are in the* Order Schema *of the input XAT table, but are not retained in the output XAT table. Thus,* $tin_1[col] \prec tin_2[col]$*. The two output tuples* $tout_1$ *and* $tout_2$ *on the other hand differ only in the keys in their cells corresponding to* col'*. By the definition of the* Navigate Unnest *(see Section 3):* $(\exists l_1, l_1 > 0)|(tout_1[col'] = \phi(path : tin_1[col])[l_1])$*, and* $(\exists l_2, l_2 > 0)|(tout_2[col'] = \phi(path : tin_2[col])[l_2])$*. As the* FlexKey *assigned to a node always has the keys of all its ancestors as prefixes,* $tout_1[col']$ *has the key in* $tin_1[col]$ *as prefix and* $tout_2[col']$ *has the key in* $tin_2[col]$ *as prefix. Therefore* $tin_1[col] \prec tin_2[col] \Rightarrow tout_1[col'] \prec tout_2[col']$ *and consequentially* $(tin_1 \prec tin_2) \Rightarrow (tout_1 \prec tout_2)$*.*

Category V. *The theorem holds by definition.*

Category VI. *If op is the operator $Expose$, it outputs an XAT document rather than an XAT table. Thus all the XAT tables in the algebra tree have already been covered.*

We have shown that (I) holds for the output XAT table of the operator op, when op is any operator and thus completed the proof for (I). Using that result, we can easily prove (II), that when $(t_1 \prec t_2)$ either $before(t_1, t_2)$ is $true$ or the order between the tuples is irrelevant. Suppose the opposite holds, that there exist two tuples t_1 and t_2 in an XAT table in the algebra tree such that $(t_1 \prec t_2) \wedge before(t_2, t_1)$. By (I), which has been proven, $before(t_2, t_1) \Rightarrow t_2 \prec t_1$. But $t_2 \prec t_1$ and $t_1 \prec t_2$ cannot be true simultaneously. Thus we get a contradiction. □

5.3 Maintaining Order Among XML Nodes in Sequences

For sequences of XML nodes in a single cell that have to be in document order, namely those created by the *XML Difference*, *XML Intersection* and *Navigate Collection*, the *FlexKey*s of the nodes reflect their order. This is due to the fact that the *FlexKey*s capture the correct document order among the base XML nodes and the semantics of these operators do not specify the order among constructed nodes. However, the $Combine$ algebra operator creates a sequence of XML nodes that are not necessarily in document order and whose relative position depends on the relative position of the tuples in the input XAT table that they originated from. Thus the order among the XML nodes in the created sequence may be different from the order captured by the node identity *FlexKey*s of these XML nodes. We thus must provide a different scheme for maintaining this order.

function $combine$ (Sequence in, Tuple t, ColumnName col)
  Sequence $out \leftarrow copy(in)$
  if ($col = OS_R[i]$ [3], $1 < i \leq |OS_R|$)
    for all k in out
      $k.overridingOrder \leftarrow compose(\Pi_{OS_R[1]}t, .., \Pi_{OS_R[i]}t)$
  else if ($col \notin OS_R$)
    for all k in out
      $k.overridingOrder \leftarrow (\Pi_{OS_R[1]}t, .., \Pi_{OS_R[m]}t, order(k)), m = |OS_R|$
  return out

Fig. 5. The function *combine*

For two XML nodes n_1 and n_2 in the same cell in a tuple in an XAT table, we define the expression $before(n_1, n_2)$ to be $true$ if the node n_1 should semantically be ordered before the node n_2, $false$ if n_2 is before n_1 and $undefined$ if the order between the two nodes is irrelevant.

To represent an order that is different than the one encoded in the *FlexKey* k serving as the node identity of the node, we attach an additional *FlexKey* to k (called *Over-*

[3] OS_R, the *Order Schema* of the input XAT table R, is known to the $Combine$ operator performing the $combine$ function.

riding Order) which reflects the node's proper order. We denote that as $k.overridingOrder$ and we use $order(k)$ to refer to the order represented by k. When the *FlexKey* k has overriding order k_o it is denoted as $k[k_o]$. If the overriding order of k is set, then $order(k) = k.overridingOrder$, otherwise $order(k) = k$. When comparing lexicographically two *FlexKey*s k_1 and k_2, $order(k_1)$ and $order(k_2)$ are really being compared. Thus $k_1 \prec k_2$ is equivalent to $order(k_1) \prec order(k_2)$.

The $Combine$ operator sets the overriding order for the *FlexKey*s in its output XAT table, as described in Figure 5. Thus, assuming that the input R contains p tuples tin_j, $1 \leq j \leq p$, then the output of $Combine\ C_{col}(R)$ can now be denoted as $C_{col}(R) = tout = (\biguplus_{j=1}^{p} combine(tin_j[col], tin_j, col))$. How $Combine\ C_{col}(R)$ sets the overriding order depends on the presence of the column col in the *Order Schema* OS_R of the input XAT table R. For the combine operator $C_{\$col5}$ in Figure 4, $\$col1$ and $\$col2$ are in the *Order Schema* of the input. Thus, when the input XML node referenced by tb is placed in the output XAT table it is assigned an *Overriding Order* composed of the order represented by the *FlexKey*s present in columns $\$col1$ and $\$col2$ in the tuple it is derived from, that is $b.b.b..e.f.f$. Thus tb after being processed by $Combine$ becomes $tb[b.b.b..e.f.f]$ [4].

The XML collection operator *XML Union* $\overset{x\ col}{\cup}_{col1,col2}(R)$ creates a new collection, for each tuple it process, from the contents of two input columns *col1* and *col2* . A new order is imposed by this process among the nodes originating from each of the input columns. We define this order by assigning an *Overriding Order* for each node that reflects its input column order in the union operation, if no *Overriding Order* keys are already defined. For example, if *col1* contains (b.f, b.l) and *col2* contains (f.b) the output column *col3* will contain (b.f[b], b.l[b], f.b[f]). If nodes in the input columns already have *Overriding Order* keys we extend these keys by adding a prefix to it that reflects the input column order. For example, if *col1* contains (b.f[b], b.l[f]) and *col2* contains (f.b) the output column *col3* will contain (b.f[b.b], b.l[b.f], f.b[f]). This order encoding ensures that we maintain order among nodes from different input columns and at the same time maintain the original order among nodes from the same input source. Other XML collection operators (*XML Unique*, *XML Difference*, and *XML Intersection*) remove the overriding order (if present) of the node identity *FlexKey*s that they place in their output XAT tables. By definition (see Section 3) they produce a column in which the nodes are in document order.

The *Group By* operator $\gamma_{col[1..n]}(R, func)$ does not define or maintain order among

[4] Note that if we have schema information about the source XML documents, it is possible to optimize *Order Schema* in a way that reduces the size of *Overriding Order*, as discussed earlier. For example, knowing that there is only one possible "title" child for each input "book" node and only one possible "b-title" child for each input "entry" node allows us to use columns $\$b$ and $\$e$ as the *Order Schema* instead of columns $\$col1$ and $\$col2$. This result in generating an *Overriding Order* key $b.b..e.f$ instead of $b.b.b..e.f.f$ for the node tb

the created groups. The *Group By* in the XAT algebra might create collections. This is mainly when the *Group By* performs nesting operations (when its $func$ argument is composed of a *Combine* operator). In such a case nodes are grouped creating collections based on the grouping columns. Order among nodes of each collection is of importance. This order is already maintained through the $FlexKey$s of the nodes (the id key or the nodes' *Overriding Order* key if it was set in a previous step). The *Group By* operator does not have to perform any further order operations.

Theorem 5.2 *Let $kout_1$ and $kout_2$ be two* FlexKey*s in a same cell in an XAT table R in an XAT algebra tree. Let these* FlexKey*s serve as node identities of the XML nodes n_1 and n_2 respectively. Then with $before(n_1, n_2)$ defined as in Section 3: (I) $before(n_1, n_2) \Rightarrow (kout_1 \prec kout_2)$, and (II) $(kout_1 \prec kout_2) \Rightarrow (before(n_1, n_2) \vee (before(n_1, n_2) = undefined))$.*

Proof: *For proving (I), we inspect the different cases depending on the type of the operator op that outputs the XAT table R. The operators of interest are those that output columns that may contain collections of* FlexKey*s. Such operators are* Navigate Collection*,* XML Union*,* XML Difference*,* XML Intersection*m* Group By*, and* Combine*. All the other operators do not create collections of* FlexKey*s, but may only retain in their output the collections present in their input in unmodified format.*

The case when the operator op is Navigate Collection *is trivial. For any two* FlexKey*s $kout_1$ and $kout_2$ in the output XAT table of* Navigate Collection*, $before(n_1, n_2)$ holds only when n_1 is ordered before n_2 regarding document order. In such case, $(kout_1 \prec kout_2)$ also holds, and thus (I) holds. Note that the* FlexKey*s $kout_1$ and $kout_2$ can not have an overriding order set, as they are retrieved from the Storage Manger by op.*

The case when the operator op is any of XML Unique*,* XML Difference*, or* XML Intersection *is similar. Again, $before(n_1, n_2)$ holds only when n_1 is ordered before n_2 based on document order. These operators remove the overriding order of the* FlexKey*s $kout_1$ and $kout_2$ if present, thus, $(kout_1 \prec kout_2)$ must also hold. The* XML Union *assigns (or maintain) the* Overriding Order *for nodes hence $before(n_1, n_2)$ holds only when n_1 is ordered before n_2 based on the* Overriding Order *keys order. For any two nodes in a collection created by the* Group By *operator $before(n_1, n_2)$ holds only when n_1 is ordered before n_2 based on the* Overriding Order *keys order or on document order if* Overriding Order *keys are not assigned.*

For proving (I) when op is the operator $Combine\ C_{col}(R)$, we inspect the possible cases depending on the presence of the column col in the Order Schema OS_R *of the input XAT table R: (1) $col = OS_R[1]$, (2) $col = OS_R[l]$, $1 < l \leq |OS_R|$, or (3) $col \notin OS_R$.*

Let kin_1 and kin_2 be the FlexKey*s from which $kout_1$ and $kout_2$ are derived. Thus*

both kin_1 and $kout_1$ (kin_2 and $kout_2$) are node identities for n_1 (n_2), but may have different overriding order. Let t_1 and t_2 be the tuples in R such that $kin_1 \in t_1[col]$ and $kin_2 \in t_2[col]$.

For both case (1) and case (2), when the column col is part of the Order Schema *of R, it must be that $kin_1 = t_1[col]$ and $kin_2 = t_2[col]$, as cells corresponding to the* Order Schema *never contain sequences, only single keys.*

For case (1), we observe that $before(n_1, n_2)$ can only hold if $t_1[col] \prec t_2[col]$. The function $combine$ does not modify the overriding order in this case, thus $kout_1 \prec kout_2$. Note that if $t_1 \prec t_2$ but $t_1[col] \prec t_2[col]$ does not hold, then by Definition 5.2 it must be that $t_1[col] == t_2[col]$. In such case $kin_1 == kin_2$ implying $kout_1 == kout_2$, which in turn yields $n_1 == n_2$. Hence, in such case the order between n_1 and n_2 is irrelevant.

Similarly, for case (2), given that the Order Schema *of R is $OS_R = (on_1, on_2, ...on_m)$, $before(n_1, n_2)$ can only hold if $(\exists j, 1 \leq j \leq l)(((\forall i, 1 \leq i < j)(t_1[on_i] == t_2[on_i])) \wedge (t_1[on_j] \prec t_2[on_j]))$. As shown in Figure 5, the function $combine$ sets the overriding order of $kout_1$ and $kout_2$ as a concatenation of all $t_1[on_j]$ and $t_2[on_j]$ respectively, $1 \leq j \leq l$. Thus, $before(n_1, n_2) \Rightarrow (kout_1 \prec kout_2)$. Again, if $t_1 \prec t_2$ but $(\forall i, 1 \leq i \leq l)(t_1[on_i] == t_2[on_i])$, then as $kin_1 == kin_2$, and $(kin_1 == kin_2) \Rightarrow (kout_1 == kout_2) \Rightarrow (n_1 == n_2)$, the order between n_1 and n_2 is irrelevant.*

For case (3), the column col may also hold sequences of XML nodes. Therefore, there are two subcases: (3.a) kin_1 and kin_2 are in the same tuple t, i.e., $t_1 = t_2 = t$, or (3.b) t_1 and t_2 are two different tuples. For case (3.a), $order(kout_1)$ and $order(kout_2)$ are composed of the same keys except for the last key that represents the order of kin_1 and kin_2 within the collection contained in $t[col]$. As in this case $before(n_1, n_2)$ for n_1 and n_2 in the output XAT table may only hold when it holds for n_1 and n_2 in the input XAT table, the overriding order is correctly set. For case (3.b), $before(t_1, t_2) \Leftrightarrow before(n_1, n_2)$. As the overriding order of $kout_1$ and $kout_2$ is composed of all the keys corresponding to the Order Schema *in t_1 and t_2 respectively, $before(t_1, t_2) \Rightarrow (kout_1 \prec kout_2)$. By transitivity, $before(t_1, t_2) \Leftrightarrow before(n_1, n_2)$ and $before(t_1, t_2) \Rightarrow (kout_1 \prec kout_2)$ imply $before(n_1, n_2) \Rightarrow (kout_1 \prec kout_2)$.*

We have proven (I) for all the cases. Using that result, (II) can be proven by contradiction, using the same arguments used for proving (II) in Theorem 5.1. □

5.4 Maintaining Order of Internal Nodes of Processed XML Nodes

Order of Internal Nodes of Base XML Nodes. Some base XML nodes (fragments) might be processed and exposed in the result as whole pieces without in-

serting, deleting or changing any of their contents. The relative local order among internal (children/descendant) nodes of a base XML fragment does not change during execution time even if the order of the whole fragment is changed. Hence the *FlexKey*s of those internal nodes remain to reflect the relative order among them. For example, in Figure 4 the "author" node with the *FlexKey* $b.b.f$ is processed as one XML fragment without any changes to its components. Hence the *FlexKey*s of its children remain to reflect their local order.

Order of Internal Nodes of Constructed XML Nodes. Order among internal (children/descendant) nodes of a constructed node is determined by the $Tagger$ pattern. Such order might be different than the order of the underlying XML document. Moreover, children nodes of a constructed node might be themselves constructed nodes and/or originating from different source XML documents. Hence, there is no relationship between their *FlexKey*s. For example, the constructed node tb in Figure 4 has three children nodes with *FlexKey*s $b.b.b$, $e.f.b$, and $b.b.f$ (as shown in the *Storage Manger*) corresponding to the "title", the "price", and the "author" nodes respectively. These three nodes are originating from two different source XML document. The local order among them is defined by the $Tagger$ pattern.

We encode the local order among internal nodes of constructed nodes by assigning *Overriding Order* keys. This applies to any type of internal nodes (base or constructed). We assign the *Overriding Order key*s b, f, and l to the "title", "price", and "author" nodes respectively. Note that using such order encoding leaves a space for updates. For example, if a source update inserts a new "author" node with *FlexKey* $b.b.d$ before the existing "author" node with *FlexKey* $b.b.f$, the new "author" node should be inserted as a child of the constructed node tb (shown in Figure 4). The correct order treatment for that new "author" node is to insert it between the "price" node (with id $e.f.b$ and local order f) and the "author" node (with id $b.b.f$ and local order l). This is easily archived by assigning an *Overriding Order Key* that is lexicographically between f and l (i for example) to the new "author" node. Note that we never run of out of keys between any two *FlexKey* as we will discuss in Section 6.

5.5 De-referencing the Final Result.

Generally, when de-referencing the final result we may require partial reordering as we will discussed later. For the example in Figure 4, the result of the XQuery expression is obtained by de-referencing the *FlexKey* tr. First, the skeleton of the constructed node identified by tr is retrieved and the *FlexKey*s contained in that skeleton are de-referenced. The children of tr need to be returned in the correct order. We sort these nodes based on their *Overriding Order* and return node tb

first then tc[5]. Now we take these two nodes one by one and de-reference them recursively so that the resulting XML document is obtained. When de-referencing the node tb we obtain the "title" node $b.b.b$ first, the "price" $e.f.b$ second, and finally the "author" node $b.b.f$. This nodes are returned in their tagging order as was maintained by the *Storage Managed*. If any of these three returned terms were a collection a local sort among its collection content might be required. Recursively, we de-reference each of the nodes we obtained so far. Since they are all base nodes, their descendants (if any) are returned in document order without any sorting. Note that because the collections returned in the result are de-referenced one collection at a time and they are often small sets of nodes, sorting can often be done in main memory thus becoming very efficient.

5.6 Discussion On Different Types of Order

Document Order. Given the order encoding schema discussed above we can now maintain document order. This provides support for XQuery queries that return the result (or part of it) in document order. It also provides support for XQuery functions and predicates that exploit document order like *before*, *after*, *range*, and *position*. Figure 4 shows the full intermediate result for the execution of our running example XQuery in Figure 1. The order schema columns of intermediate result tables are shaded. The figure also shows, on the right hand side, the storage manager and how the source XML document and the constructed nodes are stored there. We note that for the query shown in Figure 4 columns *$col1* and *$col2* serve as the order determining columns (*Order Schema*) for all intermediate XAT tables below the $Combine$ operator. Such *Order Schema* is composed of the *Order Schema*s of the input tables of the $Join$ operator. The $Combine$ operator creates a collection out of all its input tuples. At this point order between tuples disappears. The $Combine$ operator instead defines order between nodes in the collection it creates. This is done by assigning an *Overriding Order* key for each node in the created collection. This *Overriding Order* key is composed of the keys in the corresponding order determining columns in the input XAT table. Note that in this example the way these *Overriding Order* keys are assigned ensures that the order between the two newly constructed "entry" nodes still follows the underlying documents order.

Query Order Imposed By the Query *order by* Clauses. The *order by* clauses in XQuery expressions are translated into $OrderBy$ operators in XAT query plans. Maintaining order in such queries is also done using order determining columns (*Order Schema*). The main difference is that we discard the order that is based on

[5] The order of these nodes follows the underlying document order of the "book" and the "entry" elements. Although we do not require the physical order of processed nodes to be retained during execution time (to open up query optimization opportunities), if such physical order happens not to be destroyed we might avoid sorting of some of the returned nodes. For example, we may get nodes tb and tc in the right order without any sorting.

document order (Only at the node level manipulated by the *order by* clauses) when the $OrderBy$ operator is encountered during execution. Instead we use a new *Order Schema* that is generated by the $OrderBy$ operator.

Figure 6 shows an execution plan similar to the one in Figure 4 with an added $OrderBy$ operator that sorts based on book prices in a descending order. In this query plan, below the $OrderBy$ operator, order still follows the document order represented by column $\$b$. Hence it is similar to the order in the other query plan in Figure 4. When the $OrderBy$ operator is processed, it removes columns $\$col1$ and $\$col2$ from the order schema and adds a new column $\$Ord1$ to it. This new column has new order keys that are assigned based on the ascending values of price elements (values not shown in intermediate result) in column $\$col3$. Starting from this point this new order is to be used instead of the order of $\$col1$ and $\$col2$. The $Combine$ operator will use the new order keys in column $\$Ord1$ to override order of nodes in the collection it creates. In this case the order between the newly constructed "entry" nodes will follow the order specified by these keys (ascending order based on book prices).

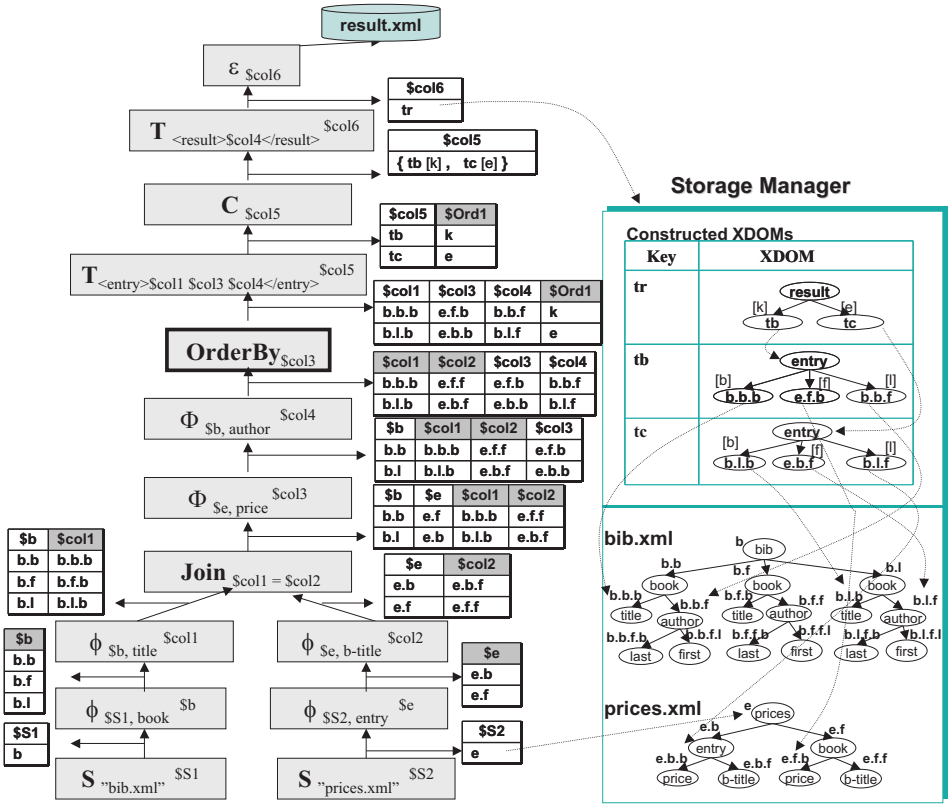


Fig. 6. A query plan similar to the one in Figure 1 extended with an *OrderBy* operator. Shaded columns represent order schema

Query Order Imposed by the Nesting of Variable Binding in the Query for **and** let **Clauses.** Such variable nesting is translated into $Join$ operations on the algebra level. Hence the order treatment follows the rules described in Section 5.2.

These rules gives a major influence on the order to the data bound to the outside variable of the for clause and a minor influence on the order to data bound to the inside variable of the for clause. Such order is encoded only at the *Order Schema* level and no extra keys are needed to reflect it at this point.

Query Order Imposed by the Query $return$ clauses and by the New Result Construction. On the algebra level this type of order is handled by the $Combine$, the *XML Union*, the $Tagger$, and the *Group By* operators. The $Combine$, the *XML Union*, and the $Tagger$ operators all set the *Overriding Order* keys for the nodes they process. Such *Overriding Order* now reflects the relative order between the processed nodes. The *Group By* operator preserves the original order between the nodes in each created group. Such order is reflected by the nodes' *FlexKey*s (the id key or the *Overriding Order* key if assigned at earlier stage).

6 Discussion on our Proposed Order Solution

6.1 The Cost of our Solution

Cost Components. The cost of handling order in our approach is composed of three main cost elements:

1) The cost of computing the *Order Schema*. This cost depends on the number of operators in the query plan and does not depend on the size of processed data. It involves traversing the algebra tree and assigning an *Order Schema* for each XAT table in the algebra tree. This step can be integrated with the query plan generation and optimization phases to avoid a separate traversal for the tree.

2) The cost of assigning *Overriding Order* keys for processed XML nodes. Only three operators out of all the seventeen operators shown in Table 1 need to assign *Overriding Order* for the nodes they process. These three operators are the $Combine$, the *XML Union*, and the $Tagger$. Integrating the process of assigning the *Overriding Order* keys with the actual query execution of these operators would result in a lot of time saving.

3) The cost of sorting when we de-reference the final result. Such sorting is a key-based sorting (on the *FlexKey*s), and is typically a partial sorting. Sorting might be required mainly for collections created by the query during execution (using the *Combine*, the *XML Union*, or the *Group By* operators). In many cases such sorting might involve only one scan over the nodes, if they are already sorted. This may occur when the correct order of the processed nodes has not been destroyed by the query execution. All internal nodes (children/descendants) of returned base nodes are directly de-referenced from the *Storage Manger* in document order [3], thus no

sorting is required. For internal nodes of constructed nodes sorting depends on what is included under the constructed node. A constructed node may tag single nodes, collections of nodes, or combinations of them. The skeleton representation of a constructed node created and stored during execution time reflects the structure and the relative order among its internal nodes and/or collections. Our system ensures that all internal nodes and collections of constructed nodes are returned in the result directly in their tagging order, hence not sorting is required. Sorting might only be required for contents of collections as discussed above. In the worst case, total sorting for all nodes in the result of an XQuery might be required only if the result returns one collection of base leaf nodes or of constructed nodes each of which is a parent of one base leaf node.

Proposed Optimizations. Here are some ideas on how to optimize our proposed order solution:

- Some of the rules presented in Table 1 can be further optimized by removing/replacing certain columns in the *Order Schema*. This would reduce the number of columns in the *Order Schema* or replace them with columns with smaller *FlexKey*s. Hence when producing order keys based on the *Order Schema* we get smaller keys. For example, for the operators *Select* and *Theta Join* if any of the columns present in the selection or joining condition are not in the *Minimum Schema* of the output XAT table, they can be dropped from the *Order Schema* of output XAT table (if it has other columns) or replaced by the column(s) in which they originate from (if the *Order Schema* has no other columns). This is because such columns are created to be used in the *Select* or *Join* predicates and are not part of any later processing operations. Hence their specific order is not of interest to the query. For example, if an input XAT table for a *Select* operator has an *Order Schema* that is composed of columns $\$a$ and $\$p$, and assuming that column $\$p$ is used in the *Select* condition and it is not in the *Minimum schema* of the output XAT table of that operator. This signifies that column $\$p$ is not needed for any next operation and its order is not of importance to the result of the query. Hence, we can drop column $\$p$ from the *Order Schema* of that operator output XAT table. This makes the order among tuples in that output XAT table determined only by the contents of column $\$a$. Another example is column $\$col1$ in Figure 4. This column is used in the *Join* operator and is not part of the *Minimum Schema* of the output table of that operator (since it is not needed for any next operation). Since this is the only column in the *Order Schema*, of the input XAT table of the operator, we replace it by the column it originally came from (column $\$b$). Column $\$b$ is hence used to reflect the order instead of column $\$col1$. In this case if we are to extract the order of that table, at a later stage, we get the smaller *FlexKey*s based on column $\$b$ instead of the larger *FlexKey*s in column $\$col1$.
- It is also possible to optimize the *Order Schema* using schema information of the source XML documents if available. Such optimization may again result in generating smaller order keys. For example, the *Navigate Unnest* operator

$\phi^{\$col1}_{\$b,tite}(R)$, shown in Figure 4, has column $\$b$ (that represents the "book" nodes) as the *Order Schema* of its input XAT table. Based on the rules presented in Table 1 the *Order Schema* of the output XAT table of that operator becomes column $\$col1$. If schema information exists that specifies that there is only one possible "title" child for each "book" node, we may keep using column $\$b$ as the *Order Schema* of the output XAT table instead of column $\$col1$. Again, the size of order keys extracted from column $\$b$ is smaller than that order keys extracted from column $\$col1$. Such reduction in order keys size is more significant when the navigation operation involves many navigation steps.

- In many cases the query may not destroy the desired order of the returned result. But we may still need to perform one scan over the returned collections to conclude that it is in the desired order. One possible optimization to eliminate such unnecessary scan is to maintain a flag for processed collection (might be annotated at the XAT table schema level). This flag specifies if the order of processed collection(s) is preserved or not. The value of this flag is set by different operators in the algebra tree. When returning a collection in the final query result, if its flag reflects that the collection order is not destroyed we can directly return the nodes in the collection without checking if it is in the desired order or not.
- It might be also possible to tune the query optimization and execution itself to achieve better overall performance in terms of the total cost of execution and order. For example, if savings form certain optimization or execution strategy is wiped out by an added final sorting cost we might choose another strategy, possibly of higher cost, that results in less overall cost for the execution and order together. For example, if a hash-based *Join* hashes the smallest table and scans the biggest table and joins tuples from the biggest table with the hashed tuples, the result will be sorted based on the order of the biggest table. Hence, in some cases (for example, if the two tables are close in size) we may choose to hash the right input table in particular so we generate a result that reflects the major order of the left input table. Since the order of the *Join* output follows the order of its left input table as a major order and then the order of its right input table as a minor order, this treatment reduces the final sorting time (or eliminate it if the minor order of the right table is not of importance).
- In some cases it might be possible to avoid assigning *Overriding Order* keys for nodes. For example, if a *Tagger* operator constructs a new node and assigns some base nodes as children for it. If the tagging pattern places these nodes in a relative order similar to that of their source XML document, there is no need to assign *Overriding Order* keys for these nodes.

6.2 Implications of our Proposed Solution

Migration to Non-ordered Bag Semantics. Our technique of encoding order with *FlexKey*s and intermediate *Order Schema* enables migration of the XAT algebra semantics from ordered bag semantics to non-ordered bag semantics. That is, (1)

the physical order among the tuples is no longer of significance and (2) the physical order among the nodes in a cell is not of significance. This implies that we separate out the reasoning about order into a separate abstraction independent of each operator's logic. In general, algebra operators are thus not responsible for maintaining order of intermediate results. One exception is the *Order By* operator. The *Order By* operator has to define a new order among the data it processes. This cost is encountered anyways regardless of the order solution used. The only added cost in our approach for maintaining order while processing the *Order By* operator is the cost of assigning new order keys to the data. Also, while the $Combine$, the *XML Union*, and the $Tagger$ operators do not perform any sorting, they assign new order keys for nodes while they process them. All other operators process the data while they are unaware of its order. In general, our solution does not require sorting of any intermediate results. This is true for all algebra operators (the *Order By* operator is the only exception) even while achieving nested ordered XML restructuring.

Efficient Order-sensitive Query Processing. This transformation from ordered to non-ordered bag semantics is the key ingredient to facilitate XML query optimization. It removes the restrictions of manipulating sequences of XML data in a strict order. Order is encoded at the XML node level and at intermediate result schema level. Operators do not need to be aware of the order associated with data they manipulate. For that reason operators have the flexibility to reshuffle data in any order they wish for efficiency. This way, a $Join$ operator could use any efficient join algorithm (e.g., hash-based, index-based, or sort-merge join) producing the output in any order dictated by the join implementation strategy without requiring any intermediate sort. For example, the $Join$ operator in Figure 7 joins its two input tables on the values in columns $col2$ and $col4$. Order-determining columns (*Order Schema*) for each XAT table is shaded. The number in a circle that appears besides each tuple illustrates the implicit order of each tuple implied from the *Order Schema*. Now assume that the join implementation outputs the resulting tuples in any arbitrary physical order as in Figure 7. We are still capable of deriving the right order of tuples in the output table (major order from left input table and minor order from right input table) by comparing the keys in the columns representing the *Order Schema* (columns $col1$ and $col3$) of the resulting table. The numbers in circles that appear next to tuples in the output table in Figure 7 show the order of tuples as we can derive it using the *Order Schema*. Note that this order is only an implicit order. That is, the tuples are not actually sorted based on this order at this point of query execution.

Efficient Order-sensitive View Maintenance. The migration to the non-ordered bag semantics also facilitate efficient XML incremental view maintenance. This is because it ensures that most XAT XML operators become distributive with respect to bag union, leading to more efficient view maintenance. As an example, consider that the input table of a $Select$ operator has received an update in the form of a tuple insertion. Since the $Select$ operator becomes distributive, the inserted tuple can be processed independently of other input tuples. If the inserted tuple satisfies the

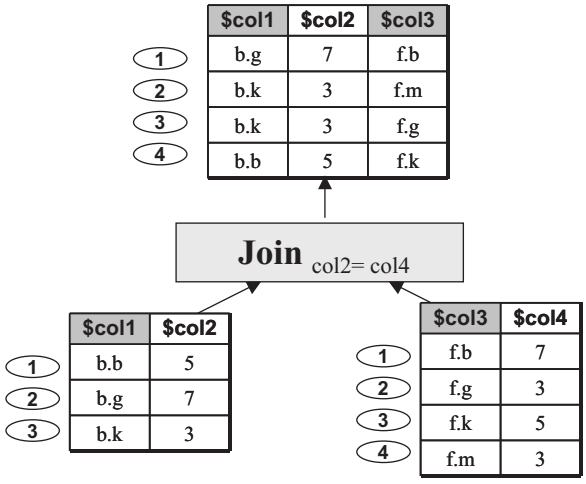


Fig. 7. An example for order handling in the $Join$ operator. Shaded columns determine the *Order Schema* of each table and numbers appearing in circles beside tuples determine the tuple induced order.

operator predicate it is directly propagated to the output table. Without the distributive feature, the operator would have to determine the relative order of the inserted tuple among the output table. They may require storing and accessing auxiliary information to determine that order. See [4] for our work in view maintenance that exploits this order-encoding schema.

6.3 Other Discussions

The Generality of our Solution. Our solution requires defining rules for maintaining order (the *Order Schema*) on the query execution model level (the XAT tables). There are two main XML query execution models: (1) the tuple-oriented model, like the one we use and that is also used in [9], and (2) the pattern tree model, like the one used in [10]. The tree-oriented model uses pattern trees to match trees from the input documents. It is easy to generalize our order solution to the tree-oriented model by understating how the tree-oriented model maps to our tuple-oriented model. There is a direct mapping between the two models as each attribute in the XAT table maps to a variable binding in the pattern tree. Hence a tuple in the XAT table is a labeled container that holds all the bindings as well as binding relationships that exists in pattern trees. A similar mapping is also identified in [9]. So for the pattern tree execution models we simply need to define the *Order Schema* for on the node level of the pattern trees binding variable nodes. This corresponds to defining them on the column level in the XAT tables.

Re-labeling (Reordering keys) on Updates. Unlike other order approaches [6,10,20] our order encoding schema guarantees that we do not run out of keys even for a large batch of skewed insertions focused on possibly one small region within the underlying XML document. The reason is two-fold: (1) we leave gaps between keys

when we first assign them (as in Figure 3), and (2) we are capable of producing a key between any two keys at all times even if there is no gap between them. This is because our key is composed of variable length byte strings as described earlier. Thus, even if we run out of keys due to a large number of inserts that fill the gap between two keys we can opens up new gaps by adding one more character to the encoding. For example, if we need to insert a new node between two nodes with keys $b.c$ and $b.d$ we may simply give the new node the key value $b.ck$. This will open up new gaps between $b.c$ and $b.ck$ and between $b.ck$ and $b.d$ and so on. This prevents the need to re-lablel keys not only for the source document node keys but also for the order encoding of the processed data since we also use $FlexKey$s to encode new order imposed by the query. Please, refer to the example of inserting a new "author" with a key $b.b.d$ that we have presented in Section 5.4.

Order Among Multiple Documents. Our order approach supports order also for queries over multiple XML documents. There are two issues to consider here: (1) base node key and order encoding and (2) query order encoding. (1) On the base node level, each XML document has order among its nodes encoded separately using the keys of its nodes as we have shown earlier. The *Storage Manager* [3] ensures that each document will have a unique key for the root node. Hence all nodes will have a unique key among all documents. For example, although the two nodes $b.b.f$ and $e.b.f$ share the suffix $b.f$, but because they are from two different documents (with root keys b and e), the key for each one of them is unique. For any base XML node (fragment), originating from any document, the local order of its internal nodes is reflected by the nodes' *FlexKey*s, as discussed in Section 5.4. (2) On the query level, the order among data from different source documents is determined by the query itself. This is typically handled by the order imposed by the nesting of variable binding in the for and let clauses, and the order imposed by the query $return$ clause and the new result construction. Hence, the treatment of order among multiple documents follow the same guidelines we gave for handling these types of query imposed order.

7 Experiments

We have tested the efficiency of our solution and have found that our order solution provides support for different types of XQuery order with little overhead for the query engine. Our evaluation focuses on two main dimensions. (1) What is the overhead added to the query processing cost when we support different types of order-sensitive queries. (2) Where does the cost of handling order come from and what are the cost elements of order in different types of queries.

We have implemented our order approach in Java and integrated it with the Rainbow system [27]. We have run the experiments on a Windows PC with 733 MHz Pentium processor and 512MB of memory. We have used the XMark benchmark

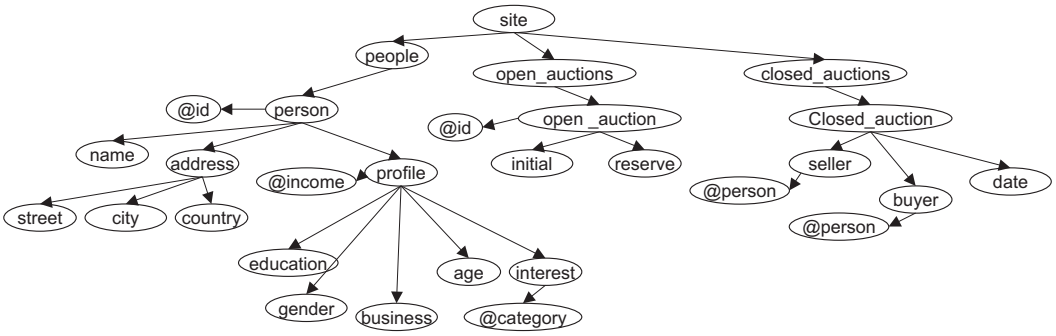


Fig. 8. Part of the structure of the “site.xml” file used in the experiments.

data [17] in our experimental evaluation. Figure 8 shows part of the structure of the XMark “site.xml” data set that is relevant to the queries we use. We use XML files of different sizes in our experiments, varying from 5MB to 25MB. We use four queries (shown in Figure 9) that come with different order requirements. We have designed each of the four queries to reflect mainly one form of the four order types that we have discussed earlier. This ensures that we measure the cost of each type of order in isolation of the other types. For each of the four queries we show the overhead of handling order relative to the total query execution time. We also break down the order cost in each query to its cost elements. We now analyze the results we have obtained using these queries.

```
<result>{
  for $p in doc(“site.xml")
                  /people/person/profile
  return
     $p
}</result>                                   Query 1
```

(a)

```
<result>{
  for $c in distinct-values doc(“site.xml")
                  /people/person/address/city
  order by $c/text()
  return $c
}</result>                                   Query 2
```

(b)

```
<result>
for  $p in doc(“site.xml")/people/person
   for $c in  doc(“site.xml")/closed_auctions/closed_auction
  where $p/@id = $c/seller/@person
return
   $c/date
</result>                                    Query 3
```

(c)

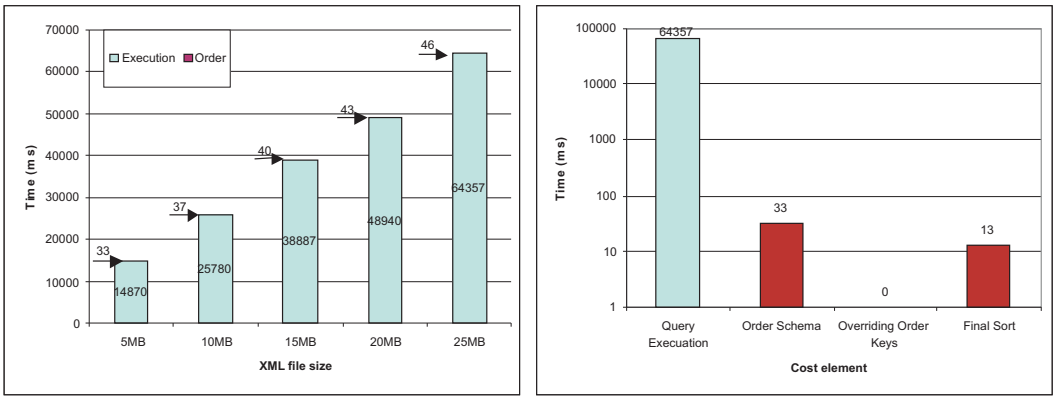
```
<result>
 {<customers>
   for $p IN doc(“site.xml")/ people/person
   return
      <customer>{<location>$p/address/city/text()</location>} {$p/name}</customer>
 </customers>}
{ <open_bids>
    for $oa IN doc(“site.xml")/ open_auctions/open_auction
    return
          <bid> {$oa/reserve} {$oa/intial} </bid>
</open_bids>}
</result>                                    Query 4
```

(d)

Fig. 9. Different XQuery expressions that are used in the experiments.

Query 1. This query navigates to all the “profile” nodes (fragments) reachable from the root of the XML document “site.xml” through the path “/people/person”. The extracted XML fragments form a collection that is tagged using the “result” tag. This query reflects only document order in which order among all nodes in the result follows the order of the input document. This applies to the order among the returned XML fragments and also to the order among their internal nodes.

Figure 10(a) shows that the total cost of handling order in this query is very small (negligible) compared to the query execution time. The break down of this order cost is shown in Figure 10(b), measured using the input XML file of size 25MB. The cost of maintaining order in a query that processes only document order is mainly composed of two cost elements: (1) the *Order Schema* computation cost and (2) final result sorting cost. The *Order Schema* computation cost is fixed regardless of the size of the processed data (it only depends on the number of operators in the query plan). The cost of the final sorting depends on the processed data size. It also depends on how the query manipulates the order among processed nodes. For *Query 1* only partial sorting might be needed on the level of the returned fragments (“profile” elements) if the correct order among those fragments was destroyed during query time. Internal nodes of those nodes are returned in document order without any sorting, as discussed earlier. Figure 10(b) shows that the cost of the final (partial) sorting for *Query 1* is very small. *Query 1* did not perform any operation that destroys the order among nodes in the returned collection. Hence a very small cost is needed to conclude that the returned result is in the correct order and no sorting is needed.



(a) (b)

Fig. 10. Results obtained for Query 1: (a) the order cost to the execution cost on different input XML file sizes, and (b) the break down of order cost on 25MB XML input file size.

Query 2. This query navigates to the “city” nodes reachable through the path “/people/person/address”. A collection of distinct cities is created using the *distinct-values* operator. This collection of distinct “city” elements is sorted alphabetically on the “city” name by the *order by* clause. Finally the collection is tagged using the “result” tag. This query reflects a query order imposed only by the *order by* clause.

No document order or any other type of query order is affecting the result.

Figure 11(a) shows that the total cost of handling order in this query is also very small (negligible) compared to the query execution time [6] . The break down of this order cost is shown in Figure 11(b). The cost of maintaining order in a query that imposes order through the *order by* clause is mainly composed of three cost elements, (1) the *Order Schema* computation cost, (2) the cost of assigning *Overriding Order* keys, and (3) the final result sorting cost. The *Order Schema* computation cost is fixed regardless of the size of the processed data. The cost of assigning *Overriding Order* keys and the cost of the final sorting depend on the processed data size. For *Query 2* the cost of assigning the *Overriding Order* keys is the highest among the other order cost elements. This is mainly because all the returned nodes in this query are affected by the *order by* clause and hence are assigned *Overriding Order* keys [7] . The *order by* operation in this query performs a sort for the processed nodes generating an ordered collection at the intermediate result. This order is not destroyed by any other operations in the query. Hence the final sorting cost shown in Figure 11 involves mainly verifying that the returned collection of "city" nodes is already in the desired order.

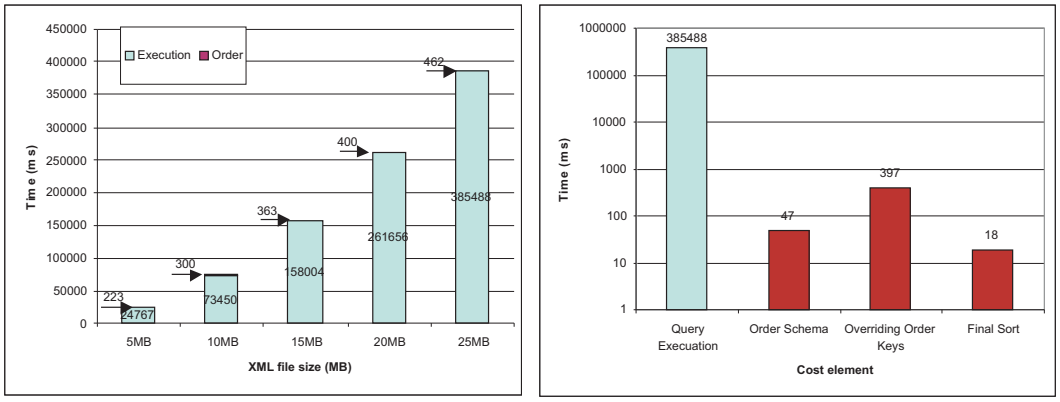


Fig. 11. Results obtained for Query 2: (a) the order cost to the execution cost on different input XML file sizes, and (b) the break down of order cost on 25MB XML input file size.

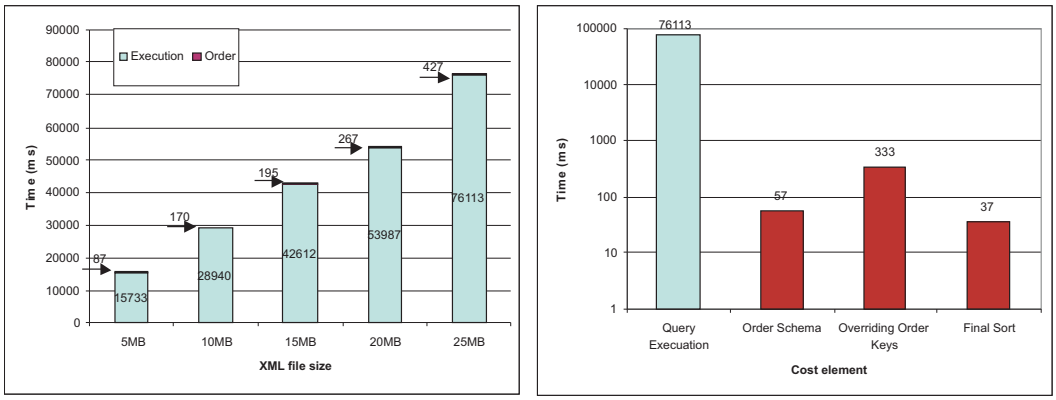
Query 3. This query navigates to two different collections. It navigates to "/people/person" and navigates to "/closed_auctions/closed_auction". For all the "person" elements, the query returns a collection of "date" elements (of "closed_auction" elements) in which the person is a seller in a closed auction. This query involves a

[6] Note that the cost of the processing (sorting) done by the *order by* operator is considered as part of the query execution cost and not as part of our order solution since such cost is encountered anyways regardless of the order solution. Only cost elements that are introduced by our order solution itself are measured as part of the order cost.

[7] Note that we are considering this cost as being entirely part of the overhead of our order solution cost although it might be considered (or part of it) as part of the cost of executing the *order by* operator.

join operation on “/person/@id” and “closed_auction/seller/@person”. Finally the collection is tagged using the “result” tag. This query reflects a query order imposed only by the nesting of variable binding in the *for* clauses. The order of the returned “date” elements does not follow their document order. It follows the order of the “person” elements as a major order and the order of the “closed_auction” element as a minor order. In other words, the “date” elements are not returned in their document order but in the order the “person” elements (that join with the “seller” elements) appear. If there are multiple “date” elements under different “closed_auction” for the same “person” element, the minor order takes place here and determines the order among those elements.

Figure 12(a) shows that the total cost of handling order in *Query 3* is also very small compared to the query execution cost. The break down of this order cost is shown in Figure 12(b). The *Order Schema* computation cost is slightly higher than the last two queries because the query plan of *Query 3* has more operators. The cost of assigning *Overriding Order* keys here involves assigning *Overriding Order* keys to all the returned “date” elements. Such keys reflect the major and the minor order imposed by the *for* clause. The cost of the final sort is affected by the implementation of the *join* operator. The implementation of the *join* operation here is performed using a hash-based join. The XAT table representing the closed auctions is the one that gets hashed because of its size. This caused only the minor order of the processed data is destroyed. Hence returning the result in the correct order requires minor sorting for the returned result. The cost of this sort is shown in Figure 12(b).



(a) (b)

Fig. 12. Results obtained for Query 3: (a) the order cost to the execution cost on different input XML file sizes, and (b) the break down of order cost on 25MB XML input file size.

Query 4. This query creates a result with a new structure by performing many node construction operations as shown in Figure 9(d). This query reflects mainly a query order that is imposed by new node construction and the order specified in the *return* clauses [8] . Figure 13(a) shows that the total cost of handling order in

[8] Some implicit document order is also present in this query, in which the con-

this query is very small compared to the query execution cost. The break down of this order cost is shown in Figure 13(b). The *Order Schema* computation cost is higher than that for the last three queries because the query plan of *Query 4* has more operators. The cost of assigning *Overriding Order* keys is also high because it involves assigning *Overriding Order* keys to all the nodes in the returned result (except for the "result"). These *Overriding Order* keys reflect the query imposed order (and document order for nodes "customer" and "bid"). A small final sort cost is encountered while deriving the right order among the returned "customer" and among the returned "bid" elements.

Although all results reported here have been run on the basic Rainbow system, i.e., without employing any of the order-oriented optimization strategies pointed out earlier in Section 6, the cost of handling order has still been shown to be negligible. We expect that the cost of handling order can be even further significantly minimized by incorporating these optimization techniques into the system.

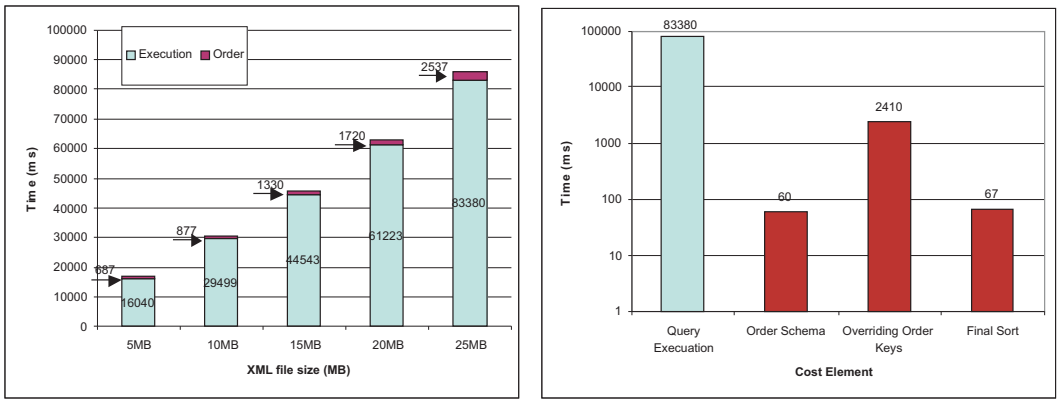


Fig. 13. Results obtained for Query 4: (a) the order cost to the execution cost on different input XML file sizes, and (b) the break down of order cost on 25MB XML input file size.

8 Conclusion

In this paper we have presented a novel approach for handling order of XML queries. Our approach supports different types of order possibly presented by XQuery expressions. This includes document order and different types of query imposed order. We have proposed a special encoding mechanism for encoding order of processed XML nodes. For most of the XML algebraic operators we encode order at the query schema level of the execution model using the *Order Schema*. Hence, these operators need not to be aware of the order of the XML nodes they process.

structed nodes "customer" and "bid" follow the document order of the "person" and the "open_auction" elements respectively. Note that the order among descendants of each of these constructed nodes is different from that of their source elements.

Only few operators need to handle the order at the nodes level. This is done by assigning special order keys, called *Overriding Order* keys, for the nodes. We do not require any special sorting operations for processed intermediate nodes. The only sorting required in our solution is when we de-reference the final XML result. Even then, typically only partial sorting is required. Our solution migrates the XML algebra to non-ordered bag semantics. Now query optimization can be performed without the restrictions typically imposed by the need to support order. Our approach provides the basis for efficient incremental view maintenance [4].

In this paper we have proven the correctness of our approach. In addition we have implemented our proposed solution and integrated it with the Rainbow XML query engine [27]. For testing our proposed solution we have used different queries that come with different order requirements. The results of our experiments shows that the overhead of maintaining order of XQuery expressions vary depending on the type of order supported by the query. There are three main cost elements in our solution. (1) The cost of computing the *Order Schema*. This cost is encountered in all queries. Such cost is very small and is fixed for the same query regardless of the size of processed data. (2) The cost of assigning *Overriding Order* keys. This cost is only encountered in queries that involve imposing new order and it varies with the input data size. (3) The cost of final sorting. This cost depends on the size and nature of collections created in the result. This cost also varies with the input data size. In general, for all the different types of order, the total overhead of maintaining order in our solution is very small compared to the query execution time.

References

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of the International Conference on Data Engineering (ICDE'02), San Jose, California, USA*, pages 141–153, Feb. 2002.

[2] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Publishing object-relational data as XML. In *Workshop on the Web and Databases (WebDB'00), Dallas, Texas, USA*, pages 105–110, 2000.

[3] K. Deschler and E. Rundensteiner. Mass: A multi-axis storage structure for large XML documents. In *Proceedings of the International Conference on Information and Knowledge Managemen (CIKM'03), New Orleans, Louisiana, USA*, pages 520–523, Nov 2003.

[4] K. Dimitrova, M. El-Sayed, and E. A. Rundensteiner. Order-sensitive view maintenance of materialized XQuery views. In *Proceedings of the International Conference on Conceptual Modeling (ER'02), Chicago, Illinois, USA*, pages 144–157, Oct. 2003.

[5] L. Fegaras and R. Elmasri. Query engine for web-accessible xml data. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'01), Roma, Italy*, pages 251–260, 2001.

[6] D. K. Fisher, F. Lam, W. M. Shui, and R. K. Wong. Efficient ordering for XML data. In *Proceedings of the International Conference on Information and Knowledge Managemen (CIKM'03), New Orleans, Louisiana, USA*, pages 350–357, Nov 2003.

[7] D. Florescu and D. Kossman. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 11(3):27–34, 1999.

[8] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *Proceedings of the Workshop on the Web and Databases (WebDB'99), Philadelphia, Pennsylvania, USA*, pages 25–30, June 1999.

[9] Z. G. Ives, A. Halevy, and D. Weld. An XML query engine for network-bound data. *The VLDB Journal*, 11(4):402–402, 2002.

[10] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native XML database. *VLDB Journal*, 11(3):274–291, 2002.

[11] C.-C. Kanne and G. Moerkotte. Efficient storage of XML data. In *Proceedings of the International Conference on Data Engineering (ICDE'00), San Diego, California, USA*, page 198, 2000.

[12] H. Liefke. Horizontal query optimization on ordered semistructured data. In *Proceedings of the Workshop on the Web and Databases (WebDB'99), Philadelphia, Pennsylvania, USA*, pages 61–66, 1999.

[13] H. Liefke and S. B. Davidson. View maintenance for hierarchical semistructured data. In *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery (DAWAK'00), Greenwich, UK*, pages 114–125, 2000.

[14] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML queries on heterogeneous data sources. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'01), Roma, Italy*, pages 241–250, Sept. 2001.

[15] U. Nambiar, Z. Lacroix, S. Bressan, M. L. Lee, and Y. G. Li. XML benchmarks put to the test. In *Proceedings of the International Conference on Information Integration and Web-Based Applications and Services (IIWAS'01), Linz, Austria*, September 2001.

[16] L. P. Quan, L. Chen, and E. A. Rundensteiner. Argos: Efficient refresh in an xql-based web caching system. In *Proceedings of the Workshop on the Web and Databases (WebDB'00), Dallas, Taxas, USA*, pages 23–28, May 2000.

[17] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, D. Florescu, and R. Busse. XMARK: A benchmark for XML data management. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'02), Hong Kong, China*, pages 974–985, August 2002.

[18] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. *VLDB Journal*, 10(2–3):133–154, 2001.

[19] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *of the International Conference on Very Large Data Bases (VLDB'99), Edinburgh, Scotland, UK*, pages 302–314, 1999.

[20] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proceedings of the International Conference on Management of Data (SIGMOD'02), Madison, Wisconsin, USA*, pages 204–215, 2002.

[21] F. Tian, D. J. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of alternative XML storage strategies. *SIGMOD Record Special Issue on Data Management Issues in E-commerce*, 31(1):5–10, March 2002.

[22] W3C. XML^{TM} . http://www.w3.org/XML, 1998.

[23] W3C. XQuery 1.0 Formal Semantics. http://www.w3.org/TR/query-semantics/, June 2001.

[24] W3C. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath, November 2003.

[25] W3C. XML Query Data Model. W3C Working Draft. http://www.w3.org/TR/xpath-datamodel/, May 2003.

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

[27] 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 *Proceedings of the International Conference on Management of Data (SIGMOD'03), San Diego, California, USA*, page 671, 2003.

[28] X. Zhang, B. Pielech, and E. A. Rundensteiner. Honey, I shrunk the XQuery! — An XML algebra optimization approach. In *Proceedings of the Workshop on Web Information and Data Management (WIDM'02), McLean, Virginia, USA*, pages 15–22, Nov. 2002.

[29] Y. Zhuge and H. G. Molina. Graph structured views and their incremental maintenance. In *Proceedings of the International Conference on Data Engineering (ICDE'98), Orlando, Florida, USA*, pages 116–125, February 1998.