WPI-CS-TR-03-25 July 2003

Addressing Order Challenges in XML Query Processing

by

Maged El-Sayed
Katica Dimitrova
Elke A. Rundensteiner

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Addressing Order Challenges in XML Query Processing

Maged El-Sayed, Katica Dimitrova, Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute, Worcester, MA 01609
(maged|katica|rundenst)@cs.wpi.edu

Abstract

Query processing over XML data sources has emerged as a popular topic. 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 challenges related to handling order in the XML context, namely challenges imposed by the XML data model, by the variety of distinct XML operators and by incremental view maintenance. We have proposed an efficient solution that addresses these issues. We use a key encoding for XML nodes that supports both node identity and node order. We have designed order encoding rules based on the XML algebraic query execution data model and on node encodings that does not require any actual sorting for intermediate results during execution. Our approach supports more efficient incremental view maintenance as it makes most XML operators distributive with respect to bag union. Our approach is implemented in the context of Rainbow [26], an XML data management system developed at WPI. We prove the correctness of our order encoding approach, namely that it ensures order handling for query processing and for view maintenance. We also show, through experiments, that the overhead of maintaining order in our approach is indeed neglectible.

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

1 Introduction

XML has been widely accepted as data format for modelling and exchanging data for internet applications. Since interest has recently surged in the development of advanced data management systems for huge repositories of XML documents, many solutions started to emerge [1]. Researchers and industry alike have began to tackle the data management challenges presented by the XML model. At the core, efficient solutions for both storage of the XML documents and query processing [16].

XML query performance depends to a large degree on how the XML data is physically broken down, stored and how it is accessed. One factor here is the access structure of XML nodes at the physical storage level. XQuery [25], a W3C working draft of an XML query language, has been proposed as standard for querying XML. Hence it is important for a general XML data management solution to support this proposed query language. Many current solutions are now attempting to support some XQuery functionality. Along with query support, an integrated XML data management solution must also be able to handle updates processing efficiently.

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 domain is content management where document data is intrinsically ordered and where queries often need to rely on this order [19]. For example, if Shakespeare's plays are stored as an XML document, the order among acts in plays is relevant, and queries asking for a certain act in a play given its order must be supported. By W3C specifications [25], XQuery expressions return results that have a well-defined order, unless otherwise specified. The result of a path expression is always returned in document order [23]. The result of a FLWR expression is in addition determined by any $OrderBy$ clause, if present, as will as the expressions in its FOR clauses [25]. Hence, the result of an XQuery reflects in an interrelated manner both the implicit XML document order and the explicitly imposed order in the XQuery definition by the $OrderBy$ clauses and/or by any nested subclauses. Support for such order when processing XQueries can result in a major performance hit by severely limiting the possibilities of query optimization [25]. For this reason XQuery provides a function, named $unordered()$, that can be used for those expressions when the order of the result is not significant [25]. This allows for turning sequences processed during query execution into sets which then offers opportunities for optimization. Given that order can not be always ignored efficient techniques for handling order in XML query processing must be devised. That is, we need to have the ability to support order in query execution and to maintain view extents under updates and at the same time to minimize the overhead that comes with it. We now provide a general solution to this open problem.

The work presented here has been conducted as part of the Rainbow system, an integrated XML data management system, that supports XQuery. The Rainbow storage manager supports efficient retrieval and updates for both normal XML data as will as intermediate XML data fragments results. Rainbow uses a unique node id encoding scheme for efficient reference-based query execution and efficient view maintenance. Order handling for queries in Rainbow is efficient, note in particular that no actual sorting is required for intermediate results nor maintaining them in an order representation. Our order strategy migrates the ordered bag semantics of intermediate query results into a non-ordered bag semantics. Our approach removes the overhead of maintaining order at the level of individual algebra operators, while at the same time order of intermediate results is preserved implicitly using our proposed order-encoding scheme. This provides more query optimization opportunities. Sorting in our approach is necessary only when de-referencing the XML document at the end of the algebra tree execution. Even then only minimum partial sorting might be needed depending on the query. Our contributions in 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 a node encoding scheme that supports both node identity and order. This encoding allows for updates without the need to renumber nodes.

3) We propose an efficient overall order encoding strategy that preserves order of intermediate data in XML algebraic query processing. This strategy removes the overhead of maintaining order at the level of individual algebra operators and removes the need for sorting intermediate data. In other words it migrates the ordered bag semantics of intermediate data into non-ordered bag semantics.

4) We prove the correctness of our approach in that it ensures ordered semantics with order preservation.

5) We have implemented and integrated our order strategy into the Rainbow XML data management system.

6) We show, via an experimental study, that our order preserving approach comes with a relatively small overhead on query execution and view maintenance.

This paper is organized as follows. Section 2 describes related work. Section 3 provides background on the XML algebra. In Section 4 we introduce challenges of maintaining order in XML query processing, while Section 5 describes our order-preserving solution strategy. Section 6 discusses the implications of our proposed order solution. Section 7 describes the Rainbow implementation framework. Section 8 gives the results of our experiments and we finally conclude in Section 9.

2 Related Work

Many solutions for XML data management use relational database technology [6, 18] as the underlying storage medium. One main drawback of such an approach is that to capture the rich structure of XML data typically a large number of relational tables may need to be generated. When querying the XML model, simple XQuery expressions may thus be translated into expensive SQL statements requiring multiple joins between tables. One issue that arises when using this approach is how to support the order nature of the XML data in relational model context since order information is lost while converting from XML to relational [13].

Object oriented databases have also been used as the storage medium for XML database systems [5]. However similar to the relational model, this approach is not efficient when dealing with schema-less XML documents [5]. Researchers have observed a similarity between semi-structured data and XML. Many solutions for semi-structured data have thus been extended to support XML data [10, 7]. These solutions tend not to support XQuery, and more important, not order sensitive.

Concurrently with the above efforts to exploit existing database technologies, XML storage manager systems [2, 9, 3] have also been proposed. An advantage of storing XML data in an XML-specific storage manager is that XML documents may potentially be stored in physical XML document order, thus facilitating efficient children/decendant access. Such tree navigation are very frequent operations in XML query processing [8]. [20] shows that customized XML storage solutions perform better than other storage solutions when dealing with XML documents without DTD or when the DTD has cycles. One problem with using some of these storage managers for XML query processing is the overhead associated with using their object-level interfaces [20].

Object identity is widely used in semi-structured data [11, 10] and in object-oriented databases [5]. In XML, W3C recommends that each node should have a node identifer [25]. Some XML algebra operators might be able to perform functionalities like duplicate elimination using only the node identifers without the need to access the actual data [25]. An example for this is the $groupby$ operator [14]. The relational database model has the equivalent notion of a tuple id. 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 [19, 8]. Several encoding techniques have been proposed for handling XML order. [19] describes three order encoding methods: global, local and dewey order. The dewey order encodes the full path from the root node to the current node and is shown to outperform the other two on a workload composed of both queries and updates. The main disadvantage of these order encoding methods is that in the presence of updates we need to perform renumbering for certain portions of the XML tree. [3] proposed an order encoding scheme (called FlexKey) as a generalization of dewey. This method avoids the problem of renumbering in the case of updates by using variable length byte strings instead of numbers.

The Agora system [12] stores XML in relational tables and provides support for handling order-sensitive XQuery. XQueries are first normalized then translated and rewritten into SQL queries to be executed over the relational tables. However, this solution is limited to XQueries that semantically match SQL and can successfully translated and rewritten into SQL. And even for supported order-sensitive XQueries handling order is an expensive process where an XQuery is translated into many SQL queries requiring several passes and materializing intermediate XML results. [17] introduces mechanisms to publish relational data as XML documents using an extension to SQL. The use of sorted outer union approach is proposed to retrieve the relational data needed for constructing XML documents when the resulting XML document does not fit in main memory. However, this approach requires total ordering of the relational result even when only a partial ordering is needed.

Timber [8], proposed at the University of Michigan, is an example of a native XML data management system. It uses Shore [2] as storage manager. Timber has the ability to deal with order in query processing. However to preserve order, sorting for some of the intermediate results appears to be required during execution [8]. The order handling strategy in Timber is built on top of a special label identifer for nodes, and thus re-labelling might be required if a large number of insertions takes place within the same small label range. Timber provides access methods that support updates, however view maintenance functionality has not yet been considered.

In this paper we introduce our order handling method that does not require any sorting of intermediate results. A key point in our solution is that the order is implicitly encoded in the node identifier and in intermediate result schema in a way that allows the migration of intermediate results from ordered-bag semantics into non-order bag semantics. Unlike in [8] our operators no longer need to be aware of the order of data they process and we do not need to incorporate any sorting operations for intermediate results. Our operators are distributive with respect to bag union. This allows for efficient incremental view maintenance. As a recent parallel effort, order-sensitive view maintenance functionality has indeed been added to Rainbow, more details can be found in [4].

3 Background: XML Algebra

In this section we introduce the XML algebra of Rainbow. While our order handling solution is illustrated using this algebra, its principles are generally applicable. We first provide some basic notations, and later we explore the semantics of some algebra operators.

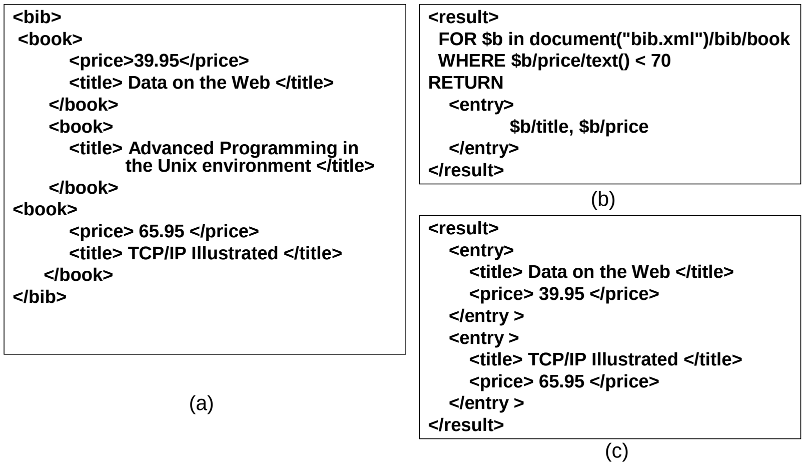
3.1 Basic Notations

In this paper, we adopt the XML standard defined by W3C [21]. 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 $n1 == n2$ [24].

Definition 3.1 *Given m sequences of XML nodes, let $seq_j = (n_{1j}, n_{2j}, ..n_{k_j j})$, $1 \leq j \leq m$, $k_j \geq 0$, n_{ij} is an XML node, $1 \leq i \leq k_j$.* ***Order sensitive bag union*** *of such sequences is defined as:* $\overset{\circ}{\biguplus}{}_{j=1}^{m} seq_j \overset{def}{=} (n_{11}, n_{21}, ...n_{k_1 1}, n_{12}, ...n_{k_2 2}, ..., n_{1m}, ...n_{k_m m})$. ***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 $\biguplus$ 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 $\overset{\circ}{\biguplus}$, $\bigcup$, $\biguplus$ or $\dot{-}$, it is treated as a singleton sequence [22].

We use the term **path** to refer to a path expression [25] 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 bib from Figure 1 (a), and $path$ = “$//price$”, then $\overset{\circ}{\phi}(path : n) = (< price > 39.95 < /price >, < price > 65.95 < /price >)$.



```
<bib>
 <book>
         <price>39.95</price>
         <title> Data on the Web </title>
      </book>
      <book>
         <title> Advanced Programming in
                 the Unix environment </title>
      </book>
<book>
         <price> 65.95 </price>
         <title> TCP/IP Illustrated </title>
     </book>
</bib>
```

(a)

```
<result>
  FOR $b in document("bib.xml")/bib/book
  WHERE $b/price/text() < 70
RETURN
   <entry>
            $b/title, $b/price
   </entry>
</result>
```

(b)

```
<result>
   <entry>
      <title> Data on the Web </title>
      <price> 39.95 </price>
   </entry >
   <entry >
      <title> TCP/IP Illustrated </title>
      <price> 65.95 </price>
   </entry >
</result>
```

(c)

Figure 1: An XML example of (a) XML document, (b) XQuery and (3) XML result

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, $\overset{\circ}{\phi}(path : seq) \overset{def}{=} \overset{\circ}{\biguplus}{}_{i=1}^{k} \overset{\circ}{\phi}(path : r_i)$. The notation $\overset{\circ}{\phi}(path : seq)[i]$ stands for the i^{th} element of that sequence, and $|\overset{\circ}{\phi}(path : seq)| = \sum_{i=1}^{k} |\overset{\circ}{\phi}(path : r_i)|$. 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 [25], a World Wide Web Consortium working draft for an XML query language, as query language. Figure 1 shows (a) an XML document “bib.xml”, (b) an XQuery expression defined over this document and (c) the result generated by executing the XQuery over the document. The XQuery expression is translated into an XML algebraic representation. Given that to date no one standard XML algebra for query processing purposes has emerged that has been widely accepted, we now introduce the XML algebra called XAT [28]. The Rainbow XML data management system [26] developed at WPI is based on this algebra. The XAT algebra defines a set of operators used to explicitly represent the semantics of XQuery. 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 $R = (t_1, t_2, .., t_p)$[1]. 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 $\leq$ j $\leq$ p) is a sequence of k cells c_{ij} (1 $\leq$ i $\leq$ k), that is $t_j = (c_{1j}, c_{2j}, ..., c_{kj})$, where k is the number of columns. Each cell c_{ij} (1 $\leq$ i $\leq$ k, 1 $\leq$ j $\leq$ 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. Note that 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) (except for the $Source$ operator it represents an XML document). Some of the XAT operators along side with their XAT tables are shown in Figure 2 that depicts the XAT algebra tree generated for the query in Figure 1(a). 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* $\delta(R)$, *Group By* $\gamma_{col[1..n]}(R, func)$ and *Order By* $\tau_{col[1..n]}(R)$, where R and P denote XAT tables. Those operators are equivalent to their relational counterparts[2], 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. For example, the Join family of operators (*Cartesian Product, Theta Join, Left Outer Join*) outputs the tuples sorted by the left input table as major order and the right input table as minor order. $Distinct$ and *Group By* are the only operators in the XAT algebra that output an unordered XAT table, following the specification in [25]. *Order By*, like its relational counterpart, orders the tuples by the values in the columns given as arguments.

The XML operators, used to represent the XML specific operations, are defined below.

Source $S_{xmlDoc}^{col'}$ 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] More precisely, an XAT table supports order preservation of the tuples. That is, when there is meaning of the order the XAT tables preserve it. Otherwise, when the order is undefined, then it is not guaranteed to be preserved.

[2] The operator Group By may take any arbitrary subquery or function, but we only consider the MIN, MAX, COUNT, AVERAGE and POS(), the last being used for outputting for each tuple its absolute order in its group.

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 $tout^{(l)}_j$, where $1 \leq l \leq m$, $m = |\phi(path : tin_j[col])|$, $tout^{(l)}_j[col'] = \overset{\circ}{\phi}(path : tin_j[col])[l]$. The tuples $tout^{(l)}_j$ are ordered by major order on j and minor order on l. The $\phi^{\$col1}_{\$b,price}$ operator in Figure 2 generates on tuple for each "price" element for each of the books in the input XAT table tuples, resulting only in two tuples in the output tables since only two of books have a "price" element for each of them.

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 $tout[col] = c = \overset{\circ\, m}{\biguplus}_{j=1} tin_j[col]$. Note that $Combine$ has only column col in its output XAT table. The $C_{\$col4}$ operator in Figure 2 grouped all the "entry" elements in $col4 tuples into one cell.

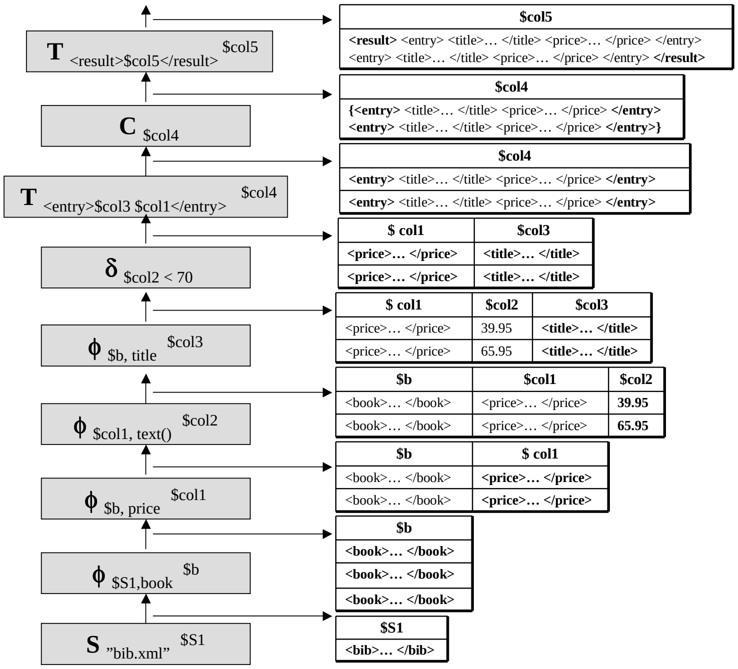
Tagger $T^{col}_p(R)$ constructs new XML nodes by applying the tagging pattern p to each input tuple. A pattern p is a template of a valid XML fragment [21] with parameters being column names, e.g., *<result>$col5</result>*. For each tuple tin_j from R, it creates one output tuple $tout_j$, where $tout_j[col]$ contains the constructed XML node obtained by evaluating the pattern p for the values in tin_j. For example, the $T^{\$col4}_{<entry>\$col3\$col1</entry>}$ in Figure 2 constructs a new "entry" node from the "title" and "price" nodes for each input tuple.

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 $tin_j[col1] \cup tin_j[col2]$ arranged in document order (unless that set contains constructed nodes, then the ordering is not defined). The other two XML set 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 and also arrange the resulting set in document order. 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. Its purpose is to output the content of column col into XML data in textual format.

By definition, all columns from the input table are retained in the output table of an operator, plus an additional one may be added, 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 in a query tree is defined as the subsequence of all columns, retaining only the columns needed later by the ancestors of that operator [27]. The process of determining the Minimum Schema for the output XAT table of each operator in the algebra tree, called **Schema Cleanup**, is described in [27].

In the XAT algebra tree shown in Figure 2 we $navigate$ to price elements and title elements of books and $select$ based on certain price. We then $construct$ new "entry" nodes from title and price elements for the selected books and tag this collection of "entry" nodes using the "result" tag. Finally we would use the $expose$ operator to extract the result from the output XAT table as an XML document (not shown in the figure due to space limitations). In Figure 2 we only show the columns that are in the Minimum Schema for each table, as the other columns would be removed due to Schema Cleanup [27].



$col5
<result> <entry> <title>… </title> <price>… </price> </entry> <entry> <title>… </title> <price>… </price> </entry> **</result>**

$col4
{<entry> <title>… </title> <price>… </price> **</entry>** **<entry>** <title>… </title> <price>… </price> **</entry>}**

$col4
<entry> <title>… </title> <price>… </price> **</entry>**
<entry> <title>… </title> <price>… </price> **</entry>**

$ col1	$col3
<price>… </price>	**<title>… </title>**
<price>… </price>	**<title>… </title>**

$ col1	$col2	$col3
<price>… </price>	39.95	**<title>… </title>**
<price>… </price>	65.95	**<title>… </title>**

$b	$col1	$col2
<book>… </book>	<price>… </price>	**39.95**
<book>… </book>	<price>… </price>	**65.95**

$b	$ col1
<book>… </book>	**<price>… </price>**
<book>… </book>	**<price>… </price>**

$b
<book>… </book>
<book>… </book>
<book>… </book>

$S1
<bib>… </bib>

Figure 2: The algebra tree for the XQuery in Figure1(b)

4 Challenges of Order Handling for XML Query Processing

4.1 Challenges Posed by the Data Model

The data model of data processed by ordered-sensitive XML query processing can be viewed 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). The XAT table corresponds to the container sequence and the tuples (or cells) in that table are the sequences inside the container sequence. Each cell can store a single node or a sequence of nodes.

Given such a data model three main issues related to order arise in this XML Query data model:

1) Order between nodes in an ordered XML tree. These nodes can be either original nodes from the source document or nodes constructed during query execution.

2) Order between items in a cell in an XAT table, which can be full trees, simple XML nodes or atomic values. This order can follow the source document order and/or can be a new order imposed by the query. In some cases order might not be of importance.

3) Order between sequences (tuples in an XAT table). This order can also follow document order and/or new order imposed by the query order. And in some cases order might not be of importance.

4.2 Challenges Posed by the Different Nature of XML Operators

Different operators in XML algebra deal with order in a different way. We here categorize XAT operators in terms of the way they deal with order into six categories. We discuss order issues related to these different categories in connection to issues posed by the data model below:

Category I. The operators Tagger $T_p^{col}(R)$, Navigate Collection $\Phi_{col,path}^{col'}(R)$, Union $\overset{x\,col}{\cup}_{col1,col2}(R)$, Intersection $\overset{x\,col}{\cap}_{col1,col2}(R)$, Difference $\overset{x\,col}{-}_{col1,col2}(R)$ and Select $\sigma_c(R)$. 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. For any operator in this category relative order between tuples in its output XAT table should be the same as the relative order between tuples in its input XAT table assuming that order was imposed there. As an example, let us consider the Select operator in Figure 2 ($\sigma_{\$col2<70}$). The input table for this operator has tuples each of them represent a the price and the tilted of one of the books in the source XML document. And since that no order where specified before this point nor the order was destroyed by any operator we expect the order between the two book tuples to be the same as the document order which basically book order then price order then title order. The output table of this operator should have the same order as the order of the input table just described.

Category II. The operator $Combine$ 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. Hence the problem of order among tuples is not applicable here. On the other hand, the issue or maintaining order between those nodes arises here. The $Combine$ operator $C_{\$col4}$ in Figure 2 groups all "entry" nodes in $col4 from its input XAT table into one cell. At this particular case the order between nodes should follow source document order (which was represented by the order between tuples in the input XAT table). This is not always the case since the input of the source might not follow the order of the source document. In general we can say that the order between nodes in the cell sequence generated by the $Combine$ should follow the order of tuples in the input XAT table of that operator.

Category III. The Join family of operators behaves in the same way with respect to order. Their output XAT table order follows the order of the left input table R as major order and the right table P as minor order (see Section 3). The order of tuples in the output XAT table of a "Join" operator depends on the order of tuples in its input XAT tables.

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 the operator $\phi_{\$b,price}^{\$col1}$ in Figure 2. The input XAT table for this operator has three tuples each representing a book. The order between these tuples should follow document order. Now the operator extracts "price" element from those books and since only two of the books have price element, the output table includes only two tuples. The order between the tuples in the output XAT table has to follow the order of the books (like the input table) then augmented by the order between the extracted "price" elements. This new order will be particularly important as the Navigate Unnest operator can extract more than one such node from the same element (for example more than one price element for the same book).

Category V. The $OrderBy$ operator $\tau_{col[1..n]}(R)$ destroys the order of the input XAT table and imposes a new order based on a certain criteria, typically some column values. Hence the output table will have a new computed order based on the order of some other columns.

Category VI. The operator $Expose\ \epsilon_{col}(R)$ outputs an XML document rather than an XAT table. This document is extracted as a tree from a certain col in the input XAT table. The extracted tree node has order among its elements that reflects all order decisions taken at previous stages among this algebra-based determination process.

The issues of order between XAT table tuples and order between nodes in a cell in an XAT table are explicit in the discussion above, order between nodes in any XML tree that is stored in cells of the XAT tables is an implicit issue for all operators. In general both challenges of the data model and of the operators are interrelated. Hence we will address them together when we introduce our order solution.

4.3 Challenges Posed by Order-sensitive View Maintenance

As we mentioned earlier, we use FlexKey encoding for encoding node identifier and order in XML trees. This encoding allows for XML updates to be applied to source documents efficiently without the need for reordering nodes. For view maintenance, this encoding by itself does not insure the support for order-sensitive propagation of the updates, as the same challenges of the query data model, described above, are still applicable not only for query processing but also for update processing. Without a special efficient solution, this may lead to the need for performing expensive scans of intermediate XAT tables, when propagating updates. For example, to determine the order of an inserted tuple in an XAT table we might need to scan the entire 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 query processing but also update processing.

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 physical sequential storage medium to be always ordered, or (2) consecutively numbering XAT tuples in each table and the members in each cell sequence. Both solutions are not efficient especially in the presence of incremental updates. Our solution for handling order relies on three main assumptions: (1) order is ignored when storing XML data, (2) order is ignored when processing XML intermediate result, and (3) after the last operator a partial sorting is performed (if needed) to return the result in the desired order. An explicit order encoding technique suitable for both expressing the order among the XAT tuples and among XML nodes within one cell in the presence of updates is needed. Such order encoding technique should allow for deriving updates to the output given the updates to the input while minimizing the requirement for accessing other information such as input and output tables. Our Storage manager relies on the MASS system [3], also developed at WPI, for providing scalable storage and indexing for XML data with guaranteed update performance. The Storage Manager provides interfaces for storing and retrieving XML nodes (both original nodes and constructed nodes). MASS generates Fast Lexicographical keys (*FlexKey*s) for all nodes in the XML tree. Original XML nodes are guaranteed to be returned in document order, eliminating the need for sorting to order results. Sorting is only needed for children of constructed nodes (if any). Mass provides scalable I/O performance for all XPath axes, moreover, it provides an integrated indexing support for XPath Node tests, position predicates and counts aggregations.

5.1 Techniques for Encoding XML Node Order

In most cases the order among the tuples in an XAT table (and among nodes in a cell) depends on the document order of the XML nodes present in these tuples (cell). Hence, the concept of node identity can serve the dual purpose of node identifier and encoding order, given that it encodes 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 [19, 3]. [19] proposes three encoding methods: (1) global order encoding, where each node is assigned a globally unique number that represents the node's absolute position in the document, (2) local (sibling) ordering, where each node is assigned a locally unique number that represents its relative position among its siblings and (3) Dewey ordering, where each node is assigned a vector of numbers that represents the path from the document's root to that node. From these three techniques, only the Dewey ordering captures the hierarchical structure among the nodes, but like the other two ordering encodings, it also requires partial renumbering in the presence of inserts in the XML document. Such renumbering is clearly undesirable if we consider support for view maintenance.

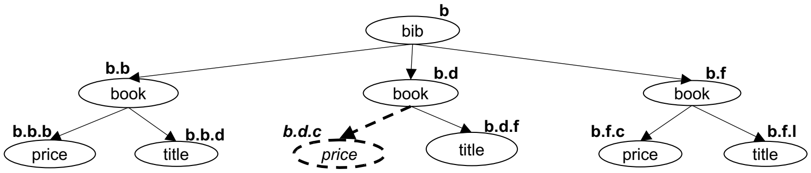


Figure 3: Lexicographical ordering of the XML document presented in Figure 1(a)

In [3] a lexicographical order encoding technique that does not require reordering on updates is proposed. It is analogous to the Dewey ordering, 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 then equal to the concatenation of all keys of its ancestor nodes and of that node's own key (see Figure 3 for example).

This order-reflecting node identity encoding is called **FlexKey**. This 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. This relationship is very frequent 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 FlexKeys lexicographically.
- It does not require reordering on updates.

We use the notation $k_1 \prec k_2$ to note that FlexKey k_1 lexicographically precedes FlexKey k_2. The FlexKeys node identity encoding for nodes in an XML document has the following properties. If k_1 and k_2 are the FlexKeys of nodes n_1 and n_2 respectively, then:

- $k_1 \prec k_2$ if and only if n_1 appears before n_2 in the document.
- k_1 is a prefix of k_2 if and only if n_1 is an ancestor of n_2 in the XML document.

For insertion and deletion of nodes the following properties hold:

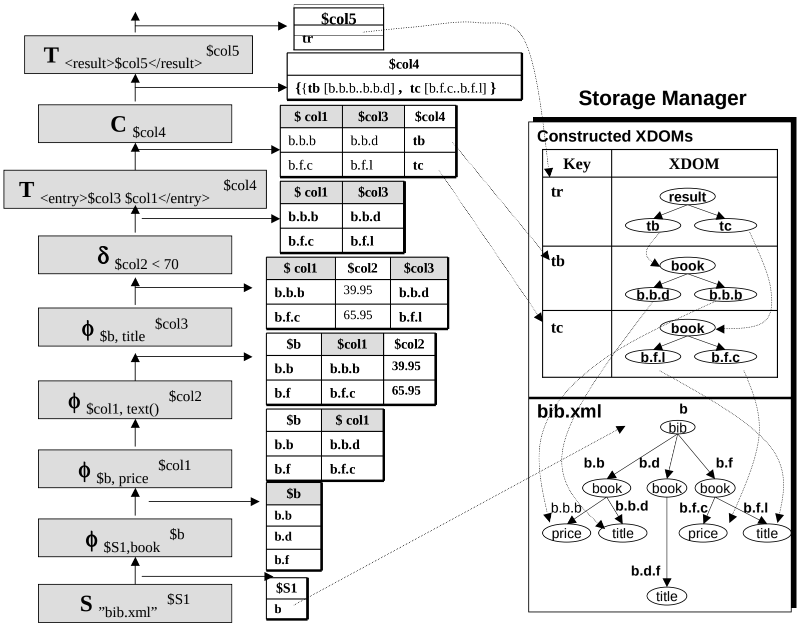
- It is always possible to generate a FlexKey for newly inserted nodes at any position in the document without updating existing keys.

- The deletion of any node does not require modification of the FlexKeys of other existing nodes.

Our order encoding scheme using FlexKeys as explained above allows for transforming the XAT algebra from *ordered bag* to *(unordered) bag semantics*, as we will show below.

5.2 Maintaining node order in XML trees using FlexKeys

We use FlexKeys 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 FlexKeys assigned to all of its nodes. For reducing redundant updates and avoiding duplicated storage we store references (that is FlexKeys) in the XAT tables rather than actual XML data. This is sufficient as the FlexKeys serve as node identifiers and also capture the order. From here on, when we mention cell in a tuple we mean the FlexKeys or the collection of FlexKeys stored in that cell. The actual XML data is stored only once in the Storage Manager. Given a FlexKey, the Storage Manger supports access to its value and to its children nodes. Figure 4 illustrates the usage of FlexKeys as references to source XML nodes instead of the real values as shown in Figure 2.



$col5
tr

$col4
{{tb [b.b.b..b.b.d] , tc [b.f.c..b.f.l] }

$ col1	$col3	$col4
b.b.b	b.b.d	tb
b.f.c	b.f.l	tc

$ col1	$col3
b.b.b	b.b.d
b.f.c	b.f.l

$ col1	$col2	$col3
b.b.b	39.95	b.b.d
b.f.c	65.95	b.f.l

$b	$col1	$col2
b.b	b.b.b	39.95
b.f	b.f.c	65.95

$b	$ col1
b.b	b.b.d
b.f	b.f.c

$b
b.b
b.d
b.f

$S1
b

Key	XDOM
tr	result tb tc
tb	book b.b.d b.b.b
tc	book b.f.l b.f.c

Figure 4: Reference-based execution using FlexKeys for XQuery in Figure1(b). Output view extracted is extracted by exposing the node with id = *tr*.

As FlexKeys are references to the base data, they can be used for accessing that data when needed by some operator. For example, the *Navigate* operator, that navigates to book elements, processes the input tuple, retrieves the children of the root node bib which are all book elements from the Storage Manager, and places their FlexKeys in the output XAT table.

Constructed Nodes. We also use FlexKeys to encode the node identity of any constructed nodes either in intermediate states of the algebra tree or in the final extent. The FlexKeys assigned to constructed nodes are algebra-tree-wide unique. They can be reproduced by the operator ($Tagger$) that created them initially based on information about the input tuple they were derived from. Rather than instantiating the actual XML fragments in our system, we only store a skeleton representing their structure in the Storage Manager. Instead references (FlexKeys) to the other source data or to constructed nodes that are included in the newly constructed node are kept. E.g., in Figure 4, although the constructed node tr is representing the whole output view extent, it is only stored as *<result>tb tc</result>* .

Composed Keys. In addition to the FlexKeys described above, we also use FlexKeys created as a composition of such keys. The purpose of this is for maintaining any order that is different than the document order in sequences of XML nodes, as in more detail is explained in Section 5.4. This follows the logic of treating keys as symbols and composing them into higher-level keys. For example, the FlexKey k = "$b.b.b..b.b.d$" is a composition of the FlexKeys k_1 = "$b.b.b$" and k_2 = "$b.b.d$" and ".." is used as delimiter. We denote this by $k = compose(k_1, k_2)$. Note that the way FlexKeys are composed guarantees that given two composed FlexKeys, $k_1 = (k_{11}, ..k_{1n})$ and $k_2 = (k_{21}, ..k_{2m})$, it holds that: $k_1 \prec k_2 \Leftrightarrow ((\exists j, 1 \leq j \leq min(n,m))(\forall i, 1 \leq i < j)(k_{1i} == k_{2i}) \wedge (k_{1j} \prec k_{2j})) \vee ((n < m) \wedge (\forall i, 1 \leq i \leq n)(k_{1i} == k_{2i}))$. The composed FlexKey k_1 precedes the composed FlexKey k_2 in two cases: (1) if the first $j-1$ FlexKeys from which both k_1 and k_2 are composed are equal and the $j-th$ FlexKey from which k_1 is composed precedes the $j-th$ FlexKey from which k_2 is composed, or (2) if k_1 is composed of smaller FlexKeys than k_2 and k_1 is prefix of k_2.

5.3 Maintaining Order Among XAT Tuples

The order among the tuples in an XAT table can now be determined by comparing the FlexKeys 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 semantically should 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 $t_1 = (b.b.b, b.b.d)$ and $t_2 = (b.f.c, b.f.l)$ in the input XAT table of the operator $T^{\$col4}_{<entry>\$col3\$col1</entry>}$ 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 FlexKeys in $t_1[\$b]$ and $t_2[\$b]$ 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 FlexKeys. Such columns are said to compose the *Order Schema* of the table. For any two tuples in the output XAT table of the *Distinct* 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.*

We now formally define how two tuples are compared lexicographically.

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 FlexKeys.

For example, let us consider the rule for computing the Order Schema of the operator *Navigate Unnest* $\phi^{\$col3}_{\$b,title}$ in Figure 4. By the semantics of this operator presented in Section 3, it processes one tuple at time. However, it may produce zero or more tuples in its output XAT table Q for each tuple in its input table R. The order of any two tuples in Q derived from two different tuples in R should be same as of those they 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 $col1$ and $col3$. The corresponding rule from Table 1 specifies that the Order Schema of the output XAT table Q should be composed of all the columns that compose the Order Schema of R except for col and of the newly produced column col'. The column col' should be added as last column into the Order Schema of Q. That is, col' subsumes col in terms of ordering capabilities. That means that given

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

Cat.	Operator *op*	OS^{*}_{Q}
I	$T^{col}_{p}(R)$ $\Phi^{col'}_{col,path}(R)$ ${}^{x}\cup^{col}_{col1,col2}(R)$ ${}^{x}\cap^{col}_{col1,col2}(R)$ ${}^{x}-^{col}_{col1,col2}(R)$ $\sigma_c(R)$	OS_R
II	$S^{col'}_{xmlDoc}$ $C_{col}(R)$ $\delta_{col}(R)$ $\gamma_{col[1..n]}(R, fun)$	ø
III	$\times(R, P)$ $\bowtie_c (R, P)$ $\overset{\circ}{\bowtie}_{L\,c}(R, P)$	$(on_1^{(R)}, on_2^{(R)}, ...on_{mr}^{(R)}, on_1^{(P)}, on_2^{(P)}, ...on_{mp}^{(P)})$ $mr = \lvert OS_R \rvert, mp = \lvert OS_P \rvert$
IV	$\phi^{col'}_{col,path}(R)$	$(on_1^{(R)}, on_2^{(R)}, ...on_p^{(R)}, col')$ if $on_m^{(R)} = col$ then $p = m - 1$, else $p = m$.
V	$\tau_{col[1..n]}(R)$	(col''), col'' is new column[3]
VI	$\epsilon_{col}(R)$	N/A
* $Q = op^{out}_{in}(R)$, $OS_R = (on_1^R, on_2^R, ...on_m^R)$		

Table 1: Rules for computing Order Schema

that Order Schema of the input R is $OS_R = (\$col1)$, the Order Schema of the output Q should be $OS_Q = (\$col1, \$col3)$. The column $\$col3$ is added to capture the order among title elements, as by the properties of the FlexKeys, the FlexKeys present in that column reflect the document order of the nodes they reference. Also, by the properties of the FlexKeys, all the FlexKeys contained in $\$col1$ and $\$col3$ have the FlexKeys from $\$b$ (books) as prefixes. Thus columns $\$col1$ and $\$col3$ automatically capture the order references of the column $\$b$ (books), and thus column $\$b$ needs no longer be retained in the Order Schema of Q.

Some of the rules presented in Table 1 can be further optimized, that is, they do not necessarily produce the minimal Order Schema. In particular, 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 Q, and are last columns in sequence of columns composing the Minimum Schema of the input XAT table R, they can be dropped from the Order Schema of Q even if they are present in the Order Schema of R.

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 FlexKeys present in the tuple and we explicitly encode it in a new column created for the purpose of encoding the order.

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 $t_1 \prec t_2$. And vice versa, 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.

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*.

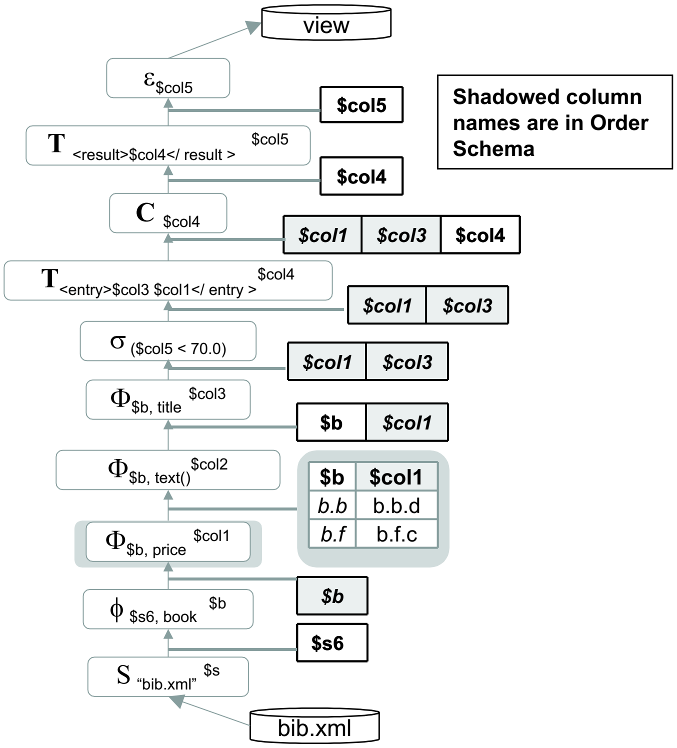


Figure 5: Order Schema computation example

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 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* $\phi^{col'}_{col,path}(R)$ 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 LexKeys 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 LexKey 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 $Expose$, the theorem has been proven. The $Expose$ outputs an XAT document rather then 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, and thus we get a contradiction. □

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. This enables correct order-sensitive query execution even when individually processing tuples one at a time. It also allows efficient order-sensitive view maintenance because for most operators insertions and deletions of tuples in their output XAT table can be performed without accessing other tuples, nor performing any reordering.

5.4 Maintaining Order in Sequences of XML Nodes

For sequences of XML nodes contained in a single cell that have to be in document order, namely those created by the *XML Union*, *XML Difference*, *XML Intersection* and *Navigate Collection*, the FlexKeys representing the nodes accurately reflect their order. This is due to the fact that the FlexKeys capture the correct document order among the base data XML nodes and the semantics of these operators does not specify the order among constructed nodes. 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. For example, let us consider any two XML nodes in the output XAT table of the *Combine* algebra operator that are derived from two different tuples in the input XAT table, when the $Combine$ operator takes as input the output of the $Distinct$ operator. The order among tuples in the output XAT table of the $Distinct$ operator is irrelevant, and the order among the nodes in the output XAT table of the $Combine$ operator reflects the order of the input tuples they derived from.

Thus the relative order among any two XML nodes derived from different input tuples is undefined. 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 it may be different from the order captured by the node identity FlexKeys of these XML nodes. We thus must provide a different scheme of maintaining this order.

```
function $combine$ (Sequence $in$, Tuple $t$, ColumnName $col$)
    Sequence $out \leftarrow copy(in)$
    if ($col = OS_R[i]$ [4], $1 < i \le |OS_R|$)
        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 $out$
```

Figure 6: The function *combine*

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 *Overriding 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 FlexKeys 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 to the FlexKeys that it places in its output XAT table, as described in Figure 6. Thus, assuming that the input R contains p tuples tin_j, $1 \le j \le p$, 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 example, let us consider the case when the column col is not part of the Order Schema of R. Then the overriding order should capture the complete tuple order encoded in all the cells corresponding to the Order Schema. Thus the overriding order of the FlexKeys in the output XAT table is composed of the order references present in all columns in the

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

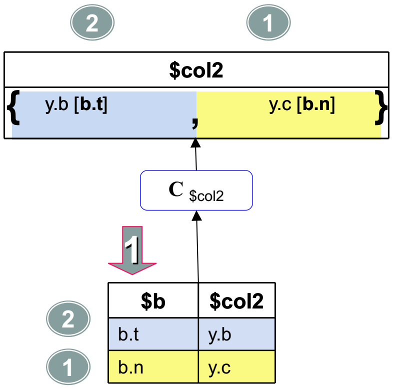


Figure 7: Example of setting overriding order by Combine

Order Schema of the input. For the combine operator $C_{\$col4}$ in Figure 4, $\$col1$ and $\$col3$ are columns 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 equal to the order represented by the FlexKey present in column $\$col1$ and $\$col3$ in the tuple it is derived from (which will be composed of both column keys), that is $b.b.b..b.b.d$. Thus tb after being processed by $Combine$ becomes $tb[b.b.b..b.b.d]$.

The XML set operators *XML Union*, *XML Difference*, *XML Intersection* remove the overriding order (if present) of the node identity FlexKeys 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.

Theorem 5.2 *Let $kout_1$ and $kout_2$ be two FlexKeys in a same cell in an XAT table R in an XAT algebra tree. Let these FlexKeys 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 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 LexKeys 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 6, 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 illustrated in Figure 7. 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. □

6 Implications of Proposed Order Solution

6.1 Migration of XML Algebra to Non-ordered Bag Semantics

This technique of encoding order with FlexKeys 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. Algebra operators are thus not responsible for maintaining order of intermediate results. Hence no sorting is needed at any intermediate execution step even while achieving nested ordered XML restructuring. The only sorting associated with the order execution is the sorting needed when we de-reference the final result at the root of the algebra tree. This cost of such sorting is typically small because this sorting is done only for children of constructed nodes that were created as a result for tagging a collection of nodes created by the Combine operator. We will show, through experiments, that such cost is indeed very small.

Figure 4 shows the full intermediate result for the execution of our running example XQuery. 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. The result of the XQuery is obtained by de-referencing the FlexKey tr. First, the skeleton of the constructed node identified by tr is retrieved and then the FlexKeys contained in that skeleton are de-referenced. The children of tr need to be returned in order. Sorting done here returns tb then tc. Now we take each of these nodes and de-reference it recursively, and the resulting XML document is obtained. Generally, the de-referencing may require partial reordering of sibling FlexKeys that are children of the same constructed node based on their order. In this example the only reordering required was for children of tr.

6.2 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, so they have the flexibility to reshuffle data in any order they wish for efficiency. This way, a $Join$ operator could perform a hash join producing the output in any order without requiring any intermediate sort.

6.3 Efficient Order-sensitive View Maintenance

These non-ordered bag semantics also facilitate efficient XML incremental view maintenance. This 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. And since the $Selecet$ operator becomes distributive, the inserted tuple can be processed independently of other tuples in input table, by the $Select$ operator and if it satisfies the operator predicate it is directly propagated to the output table. Note that without the distributiveness feature, the operator would have to compare the new tuple against all other tuples in the input table to determine the relative order for it among the output table tuples. See [4] for our view maintenance solution built on top of Rainbow that exploits this order encoding schema.

7 System Implementation

Figure 8 shows the overall architecture of the Rainbow system [26]. The system has three main subsystems: Storage manager, Query Engine and View Manager.

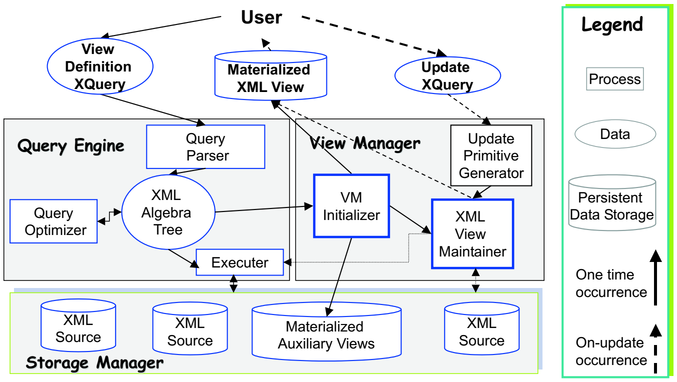


Figure 8: Architecture of Rainbow II.

The **Storage Manager** heavily relies on MASS system [3]. The Mass system provides scalable storage and indexing for XML data with guaranteed update performance. MASS generates Fast Lexicographical keys (*FlexKeys*) for all nodes in the XML tree. The Storage Manager provides interfaces for storing and retrieving XML nodes (both original nodes and constructed nodes). The Storage Manager is also responsible for providing the means to store and retrieve such materialized intermediate results needed by the View Manager.

The **Query Engine** is the core of the Rainbow system, it uses the XML algebra XAT [28] for execution and optimization of queries. Query Engine uses the *Query Parser* [15] to parse XQuery expressions. The parsed tree is analyzed and an *XML Algebra Tree* (XAT) is generated. The generated XATs are then optimized by the query optimizer using heuristic driven rewrite rules. The *XAT tree* is executed against the underlying XML documents by the *Executor* generating an XML document.

The **View Manager** is responsible for managing and maintaining the materialized views that are generated by the system. At update time the system accepts updates to the specified XML documents in the form of XQueries. It generates sets of update primitives that are passed to the Storage Manager to be applied to the stored document. The View Maintainer then propagates the updates through the XAT algebra tree and refreshes the view extents exploiting our order encoding schema.

8 Experiments

We have implemented our order approach in Java within the Rainbow system framework [26]. We performed preliminary evaluation for our system using the XMark benchmark [16]. Figure 9 shows the cost of handling order relative to the total execution time for two queries that test two different order cases. The first query evaluates the cost of maintaining order for a query that preserves document order in the resulting view extent. The query selects “person” elements (based on the person id attribute) and returns their names. We vary the selectivity of the query to test the effect of final sorting (at dereferencing time) on the performance. The first chart in Figure 9 shows that the cost of maintaining order is very small relative to the total cost of query execution for such query. We also evaluated another query that destroys document order and enforces new order, using the Order By operation, in which the result is ordered by the persons’ income. The second chart in Figure 9 shows that the cost of maintaining order in this query is slightly bigger than the first query due to the enforced new order, but yet small comparing to the total query execution time.

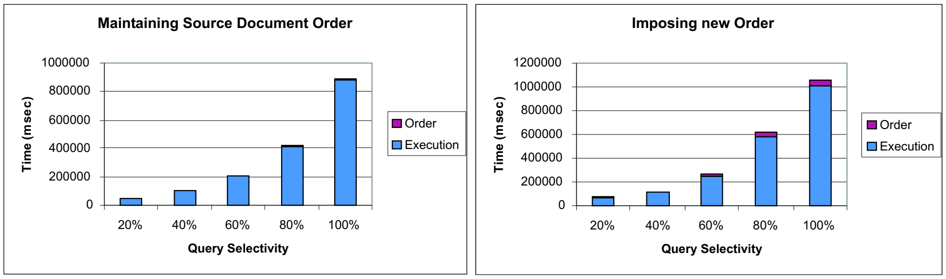


Figure 9: Experimental evaluation

9 Conclusion

In this paper we have presented our novel order handling approach for XML queries. This approach is introduced within the framework of our Rainbow system that supports reference-based execution for XQueries. We introduce a special node encoding and a schema encoding for intermediate query results that migrate the XML algebra to non-ordered bag semantics. Hence algebra operator do not have to care about order. One advantage of our approach is that it allows query optimization to be performed without the restrictions typically imposed by the need to support order. No sorting is required for intermediate result data during query execution. Our approach provides the basis for efficient incremental view maintenance [4]. We prove the correctness of our approach and we show, through experiments, that the overhead of supporting XML ordered query execution is very small.

References

[1] R. Bourett. XML Database Products. http://www.rpbourret.com/xml/XMLDatabaseProds.htm, 2000.

[2] M. J. Carey, D. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up Persistent Applications. In *SIGMOD*, pages 383–394, 1994.

[3] K. W. Deschler and E. A. Rundensteiner. Mass: A multi-axis storage structure for large xml documents. Technical Report WPI-CS-TR-03-23, Worcester Polytechnic Institute, 2003.

[4] K. Dimitrova, M. El-Sayed, and E. A. Rundensteiner. Order-sensitive view maintenance of materialized xquery views. In *ER*, 2003. to appear.

[5] L. Fegaras and R. Elmasri. Query Engine for Web-Accessible XML data. In *The VLDB Journal*, pages 251–260, 2001.

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

[7] R. Goldman, J. McHugh, and J. Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *WebDB (Informal Proceedings)*, pages 25–30, 1999.

[8] 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. In *VLDB Journal Volume 11 Issue 4*, pages 274–291, 2002.

[9] C.-C. Kanne and G. Moerkotte. Efficient storage of XML data. In *ICDE*, page 198, 2000.

[10] H. Liefke. Horizontal query optimization on ordered semistructured data. In *WebDB (Informal Proceedings)*, pages 61–66, 1999.

[11] H. Liefke and S. B. Davidson. View maintenance for hierarchical semistructured data. In *Data Warehousing and Knowledge Discovery*, pages 114–125, 2000.

[12] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *VLDB, Roma, Italy*, pages 241–250, Sept. 2001.

[13] U. Nambiar, Z. Lacroix, S. Bressan, M. L. Lee, and Y. G. Li. Xml benchmarks put to the test. In *IIWAS*, September 2001.

[14] S. Paparizos, S. Al-Khalifa, H. Jagadish, L. Lakshmanan, A. Nierman, D. Srivastava, and Y. Wu. Grouping in xml. *XMLDM'02*, vol.2490, 2002.

[15] A. Sahuguet. Kweelt: More than just "yet another framework to query xml!". In *Demo Session Proceedings of SIGMOD'01*, page 602, 2001.

[16] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, D. Florescu, and R. Busse. XMARK: A benchmark for XML Data Management . In *VLDB*, pages 974–985, August 2002.

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

[18] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB*, pages 302–314, 1999.

[19] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *SIGMOD*, 2002.

[20] F. Tian, D. J. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of alternative xml storage strategies. In *SIGMOD Record special issue on Data Management Issues in E-commerce*, March 2002.

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

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

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

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

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

[26] 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 *SIGMOD Demo*, page 671, 2003.

[27] X. Zhang, B. Pielech, and E. A. Rundensteiner. Honey, I Shrunk the XQuery! — An XML Algebra Optimization Approach. In *WIDM*, pages 15–22, Nov. 2002.

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