WPI-CS-TR-03-17 May 2003

Order-sensitive View Maintenance of Materialized XQuery Views

by

Katica Dimitrova
Maged El-Sayed
Elke A. Rundensteiner

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

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

Order-sensitive View Maintenance of Materialized XQuery Views

Katica Dimitrova, Maged El-Sayed and Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609
Tel.: (508) 831–5857, Fax: (508) 831–5776
{katica, maged, rundenst}@cs.wpi.edu

May 8, 2003

Abstract

Materialized XML views are a popular technique for integrating data from possibly distributed and heterogeneous data sources. However, the problem of the incremental maintenance of such XML views poses new challenges which to date remain unaddressed. One, XML views not only filter the data, but may radically restructure it to construct new XML nested document structures. Moreover, order is inherent in the XML model, and XML views reflect both the implicit document order of the underlying sources and the order explicitly imposed in the view definition. Therefore, order also has to be preserved at view maintenance time.

In this paper we present an algebraic approach for the incremental maintenance of XQuery views, called VOX (View maintenance for Ordered XML). To the best of our knowledge, this is the first solution to order-preserving XML view maintenance. Our strategy correctly transforms an update to source XML data into sequences of updates that refresh the view. Our technique is based on an algebraic representation of the XQuery view expression using an XML algebra. The XML algebra has ordered bag semantics; hence most of the operators logically are order preserving. We propose an order-encoding mechanism that migrates the XML algebra to (non-ordered) bag semantics, no longer requiring most of the operators to be order-aware. Furthermore, this now allows most of the algebra operators to become distributive over update operations. This transformation brings the problem of maintaining XML views one step closer to the problem of maintaining views in other (unordered) data models. We are thus now able to adopt some of the existing (relational) maintenance techniques towards our goal of efficient order-sensitive XQuery view maintenance. In addition we develop a full set of rules for propagating updates through XML specific operations. We have proven the correctness of the VOX view maintenance approach. A full implementation of VOX on top of RAINBOW, the XML data management system developed at WPI, has been completed. Our experimental results, performed using the data and queries provided by the XMark benchmark, confirm that incremental XML view maintenance indeed is significantly faster than complete recomputation in most cases. Incremental maintenance is shown to outperform recomputation even for large updates.

Keywords: XML, View Maintenance, Order, Rainbow.

1 Introduction

1.1 Problem Description

XML views are a popular technique for integrating data from distributed and heterogeneous data sources. Many systems employing XML views, often specified by the XML query language XQuery [27], have been developed in recent years [4, 15, 30, 31]. Materialization of the view content has many important applications including providing fast access to complex views, optimizing query processing based on cashed results, and increasing availability. Materialization however raises the issue of how to efficiently refresh the content of views in this new context of XML in response to base source changes. It has been shown for relational views that it is often cheaper to apply incremental view maintenance strategies instead of full recomputation [9]. However the problem of incremental maintenance of XQuery views has not yet been addressed in the literature.

The problem of incremental XML view maintenance poses unique challenges compared to the incremental maintenance of relational or even object-oriented views. The work in [17] classifies XML result construction as being a non-distributive function which in general is not incrementally computable. Also, unlike relational or even unlike object-oriented data, XML data is ordered. Supporting XML's ordered data model is crucial for applications like content management, where document data is intrinsically ordered and where queries may need to rely on this order [22]. In general, XQuery expressions return sequences that have a well-defined order [27]. The resulting order is determined both by the implicit XML document order possibly overwritten by other orders explicitly imposed in the XQuery definition by the Order By clauses or by nested subclauses [27]. As a consequence, a view has to be refreshed correctly not only concerning the view content but also concerning the order of the view result document.

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

(a)

```
<result>
    for $b in document("bib.xml")/bib/book
    where $b/price/text() < 60
    return
        <cheap_book>
            $b/title
        </cheap_book>
</result>
```

(b)

```
<result>
    <cheap_book>
        <title>Data on the Web</title>
    </cheap_book>
</result>
```

(c)

Figure 1: Example (a) XML data, (b) XQuery view definition and (c) initial extent of view

Incremental view maintenance strategies for data models that preserve order remain an open problem to date. 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. Such approach does not apply to the XML context, where all operations have to be order sensitive. Even if explicit reordering occurs (due to an Order By clause in the view definition) it does not necessary completely reorder the XML view result, as the elements deeper than the element(s) on which the ordering was performed still have to be returned in document order.

1.2 Motivating Example

In this paper, we use the XML document *bib.xml* shown in Figure 1.a as running example. It contains a list of book titles and optionally their prices. The XQuery definition of the example view, which lists the titles of all books that cost less than \$60, is shown in Figure 1.b and the initial content of that view in Figure 1.c. Suppose that the price has been left out of the second book by mistake. Hence, the update as in Figure 2.a is specified to insert a price element with value \$55.48. As to date there is no one standard Update XQuery syntax, we express this update using the update XQuery syntax introduced in [22]. The affected book now passes the selection condition and should be inserted into the view extent, resulting in the content in Figure 2.b. Even though the view definition XQuery does not explicitly refer to the document order in this example, this new book has to be inserted before the one already in the view, to preserve document order.

```
FOR $book IN document("bib.xml")//book[position()=2],
      $title IN $book/title
UPDATE $book
{
      INSERT <price>55.48</price> BEFORE $title
}
```

(a)

```
<result>
      <cheap_book>
            <title>TCP/IP Illustrated</title>
      </cheap_book>
      <cheap_book>
            <title>Data on the Web</title>
      </cheap_book>
</result>
```

(b)

Figure 2: (a) Update XQuery and (b) extent of the view defined in Figure 1.b after the update in (a)

1.3 State-of-the-art on View Maintenance

Early work on relational view maintenance [3, 10, 5] when considering rather simple views took an algorithmic approach, that is, they propose a fixed procedure to compute the changes to the view given the changes to the base relations. Later efforts on more complex view definitions including duplicates [8] or aggregations [19, 16] and also object-oriented views [2] often have instead taken an algebraic approach. Unnesting and restructuring of data is core even in the simplest XQuery view definitions due to the nested structure of XML data. Thus any practical solution for XQuery views should support a rather large set of complex operations including unnesting, aggregation and tagging. The algebraic approach, illustrated in Figure 3, is therefore the appropriate foundation for tackling incremental view maintenance in the XML context.

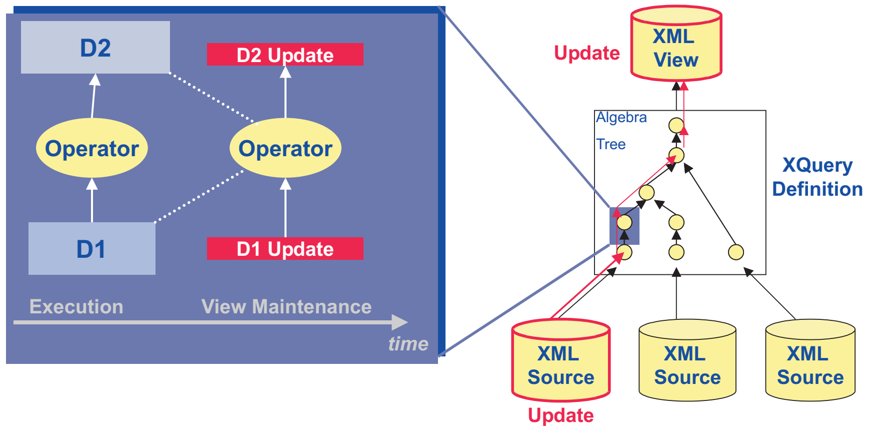

Figure 3: Illustration of algebraic approach to XML view maintenance

As pointed out in [8], the main advantages of an algebraic approach to view maintenance include:

- It is independent from the view definition language syntax. This is critical for XML given that XQuery is still a working draft, and changes to its syntax are likely to occur. Experience with SQL also has shown that even for standardized query languages, commercial database management systems introduce proprietary modifications. The same may happen for XQuery as well. Hence we favor a syntax independent solution.
- The modularity of the algebraic approach enables us with ease to extend our algebra with more operators. Also, if the semantics of any one of the existing XML algebra operators should change, the approach can easily be adapted to incorporate the change by locally adjusting some propagation rules.
- As the update rules are defined independently for each operator, existing propagation rules for operators in other data models that now also are present in the XML algebra can be reused here. This could for example include most relational algebra operators.
- The algebraic approach naturally leads itself towards establishing a proof of correctness. If all individual rules for the different operators lead to correct output of the corresponding operator, then the final output in terms of the maintained view can easily be shown to be correct as well.

1.4 VOX Approach

In this work, we propose VOX (View maintenance for Ordered XML), an algebraic XML view maintenance strategy that is order sensitive. VOX covers the core subset of the XQuery language. Our approach is based on the XML algebra called XAT [32]. For each operator in the algebra and for each type of update, we define update propagation rules that specify the modification of the operator's output as a response to the modification of its input. We provide a scalable order-preserving strategy to minimize the overhead of maintaining order during view maintenance. Also, VOX significantly reduces the amount of intermediate data to be kept. By using node identity in intermediate results and storing the actual data in a shared storage, it minimizes the auxiliary maintenance information requirements and decreases the computational effort for maintaining such auxiliary views. Our solution is flexible providing both an order-preserving and a non-order-preserving mode. Even though order is inherit to XML, there are XML applications where the ordering is not important and our solution also serves these applications.

Contributions of this work include:

- We identify and analyze new challenges imposed on incremental view maintenance by the ordered hierarchical nature of the XML data model.
- We propose an order-encoding mechanism that migrates the XML algebra from ordered bag semantics to (non-ordered) bag semantics, thus making most of the operators distributive with respect to the bag union and bag set difference.
- We give the first order-sensitive algebra-based solution for incremental view maintenance of XML views defined with the XQuery language.
- We prove the correctness of the approach.
- We have successfully implemented our proposed solution in the XML data management system Rainbow.
- We describe the experiments we have conducted to gain insight into the performance of our strategy. In the experiments the cost of view maintenance is compared to the cost of recomputation.

1.5 Outline

In the next section we briefly review related research. Section 3 introduces the XML algebra XAT. In Section 4 we describe the VOX strategy for maintaining order in the presence of updates using a scalable order encoding mechanism. In Section 5 we present the order-sensitive incremental view maintenance strategy for XQuery views. The correctness of our approach is proven in Section 6. Section 7 gives an overview of the system implementation of VOX. Section 8 describes our experimental evaluation while Section 9 concludes the document.

2 Related Work

The incremental maintenance of materialized views has been extensively studied for relational databases [3, 9, 10, 33, 14, 5, 8, 16, 19]. In [8], an algebraic approach for maintaining relational views with duplicates, i.e., for bag semantics, has been proposed. This work emphasizes the advantages of an algebraic over an algorithmic solution. These advantages also equally hold for the XML context as we have emphasized in Section 1.3. The work in [19] extends [8] for views with aggregation. Being algebraic, our approach is closely related to [8, 19]. However, our work targets the richer XML setting. In [16] the problem of making aggregate views self maintainable by also maintaining additional relations, called auxiliary views, is investigated. Palpanas and others [17] propose an incremental maintenance algorithm that maintains views whose definition includes aggregate functions that are not distributive over all operations. They perform selective recomputation to maintain such views.

To a lesser degree, view maintenance has also been studied for object-oriented views. In the Multi-View system [12, 11], incremental maintenance of OQL views exploits object-oriented properties such as inheritance, class hierarchy and path indexes. [2] proposes a solution for maintaining materialized OQL views that yields incremental maintenance plans on an algebraic level. Alike our technique of storing only node identity encodings rather than actual data, they store OID-s with the same aim of avoiding access to base data.

[34] proposes methods for the maintenance of select-project graph structured views defined as collections of objects. Maintenance for such materialized views over semi-structured data based on the graph-based data model OEM and the query language Lorel is studied in [1]. Unlike our work, they consider only atomic update operations: insertion or deletion of an edge between existing objects, or the change of the value of an atomic object. Also, more importantly, they do not consider order. In [18], an efficient maintenance technique for materialized views over dynamic web data was proposed, but based on XPath, thus excluding result restructuring. They have developed a path structure to index the view, tracking the data items that meet path branch conditions of the view query. They also do not consider order.

An architecture for defining and maintaining views over hierarchical semistructured data is proposed in [13]. Their work is on maintaining views defined with their query language called WHAX-QL which is based on XML-QL. Similar to the concept of distributiveness with regard to the bag union that we exploit, they base their work on the distributiveness with respect to a deep tree union operation that they define (they call that multi-linearity). They pose restrictions to the expressiveness of the view definition language, considering only multi-linear views and not considering order.

An algebraic approach for incremental maintenance of XQuery views has recently been proposed here at WPI [7]. The ideas as well as the shortcomings from that project have motivated this current work as follow-on effort. Unlike VOX, that work does not address the problem of maintaining order. Rather, it assumes that all intermediate data is physically stored in order, and that insertions can be done at specified positions. Also, it requires maintenance of large auxiliary data for the purpose of the next propagation. Unlike VOX, the work in [7] has no notion of node identity. Thus it may potentially need to keep and maintain same source or constructed XML nodes multiple times as intermediate results.

The problem of encoding XML structure as well as XML order has lately been studied for the purpose of storing XML documents (either in relational databases, or in a proprietary XML storage systems). Several explicit order encoding techniques for such XML documents once shredded into pieces have been proposed [23, 6] and experimentally compared. The technique from [6] is used in this work. However, the focus of our work is different from that of [6] (and [23]), as we target views and consider constructed XML nodes in addition to base data XML nodes.

3 Background: XML Query Model

3.1 Notation

We adopt standard XML [26] as data model. In this paper, an XML node refers to either an element, attribute, or text node in a 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 $seq_j = (n_{1j}, n_{2j}, ..n_{k_j j})$, $1 \leq j \leq m$, $k_j \geq 0$, 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 basically creates a set 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, $\dot{-}$ to denote monus (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 [28].

We use the term **path** to refer to a path expression [27] consisting of any combination of forward steps, including // and $*$. **Position** refers to a path that uniquely locates a single node in an XML tree, containing the element names and the ordering positions of all elements from the root to that node, e.g., $bib[1]/book[3]/title[1]$.

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, and $path =$ "$//price$", then $\overset{\circ}{\phi}(path : n) = (<price>65.95</price>, <price>39.95</price>)$.

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, $\overset{\circ}{\phi}(path : seq) \overset{def}{=} \overset{\circ}{\biguplus}{}_{i=1}^{k} \overset{\circ}{\phi}(path : r_i)$. 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.

For a position pos and a path $path$, we use the notation $pos \trianglerighteq path$ to denote that pos is "contained" in the node set implied by $path$. More precisely, an ancestor of the node n located by pos or the node n itself must be among the nodes located by $path$, if both pos and $path$ are applied on the same XML data. For example, we have $/book[1]/author[2]/name[1] \trianglerighteq /book/author$ and $/book[2]/author[2]/phone[1] \trianglerighteq //author$. When $pos \trianglerighteq path$, we define $pos - path$ as the remainder position that starts from n's ancestor located by $path$. For example, $/book[1]/author[2]/phone[1] - /book/author = /phone[1]$ and $/book[2]/author[2]/phone[1] - //author = /phone[1]$. Similarly, if some (but not necessarily all) descendants of the node located by pos may be located by $path$ we note this as $pos \lhd path$, e.g., $/book[1] \lhd /book/author$. Then $path - pos$ gives the path that starting from the node located by pos would locate all the nodes located by $path$ that have the node located by pos as an ancestor, e.g., $/book/author - /book[1] = /author$.

3.2 View Definition Language and the XML Algebra XAT

We use XQuery [27], a World Wide Web Consortium working draft for an XML query language, as the view definition language. The XQuery expression defining the XML view is translated into an XML algebraic representation that is used for both the initial computation of the view extent and for the incremental maintenance. Given that to date no standard XML algebra for query processing purposes has emerged, for the purpose of describing and evaluating our approach, we select the XML algebra called XAT [32]. 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 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 ($1 \leq \text{j} \leq \text{p}$) is a sequence of k cells c_{ij} ($1 \leq \text{i} \leq \text{k}$), that is $t_j = (c_{1j}, c_{2j}, ..., c_{kj})$, where k is the number of columns. Each cell c_{ij} ($1 \leq \text{i} \leq \text{k}$, $1 \leq \text{j} \leq \text{p}$) in a tuple t_j can store an XML node or a sequence of nodes. Note that atomic values are treated as text nodes. To refer to the cell c_{ij} in a tuple t_j that corresponds to the column col_i we use the notation $t_j[col_i]$.

The XAT algebra tree for the XQuery view definition for the running example (Figure 1.b) is presented in Figure 4.

The XAT algebra has order sensitive bag semantics: (1) The order among the tuples t_j may be of significance, (2) The order among the XML nodes contained in a single cell may be of significance, and

[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.

(3) Duplicate tuples in a table or nodes in a single cell are allowed.

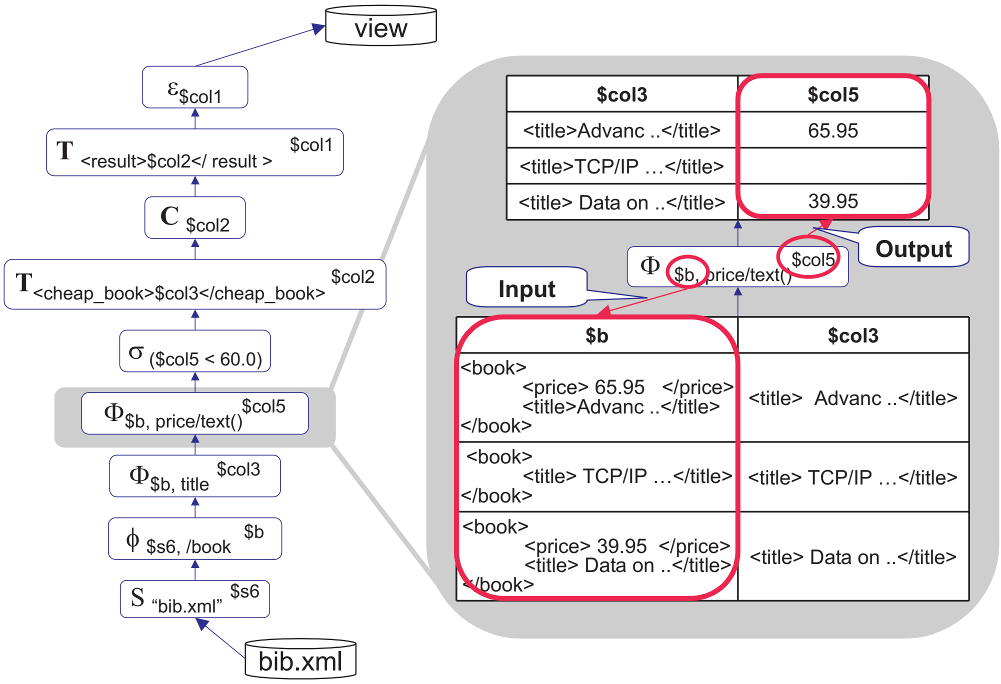

Figure 4: The XAT algebra tree for the running example

In general, an XAT operator is denoted as $op_{in}^{out}(s)$, where op is the operator type's symbol, in represents the input parameters, out the newly produced output column and s the input source(s) for that operator, which for all operators except for *Source* are XAT tables. We restrict ourselves to the core subset of the XAT algebra operators [32]. We omit operators only used temporarily during XQuery optimization, such as before decorrelation. The XAT operators are classified into two general categories: XML operators and XAT SQL operators.

XAT SQL operators correspond to the relational complete subset of the XAT algebra and include *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. In the output XAT table of *Select*, the relative order between each pair of tuples corresponds to the relative order between those two tuples in its input XAT table, as illustrated in Figure 5. 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 always output an unordered XAT table, following the

[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.

specification in [27]. *Order By*, alike its relational counterpart, orders the tuples by the values in the columns given as arguments.

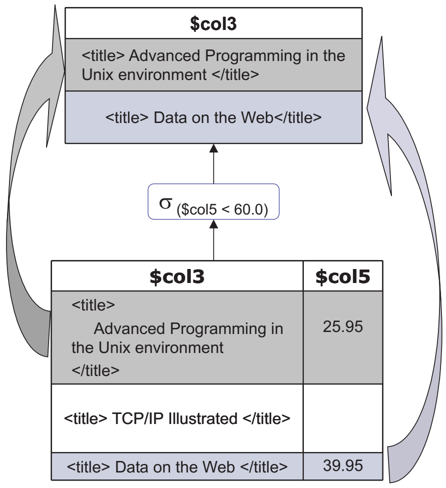

Figure 5: Example of XAT Select operator

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

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.

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.

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 $tout_j[col'] = \overset{\circ}{\phi}(path : tin_j[col])$. For an example see Figure 4.

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 $tout[col] = c = \overset{\circ}{\biguplus}_{j=1}^{m} tin_j[col]$. Note that *Combine* has only column col in its output XAT table.

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 [26] with parameters being column names, e.g., *<result>$col2</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.

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 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.

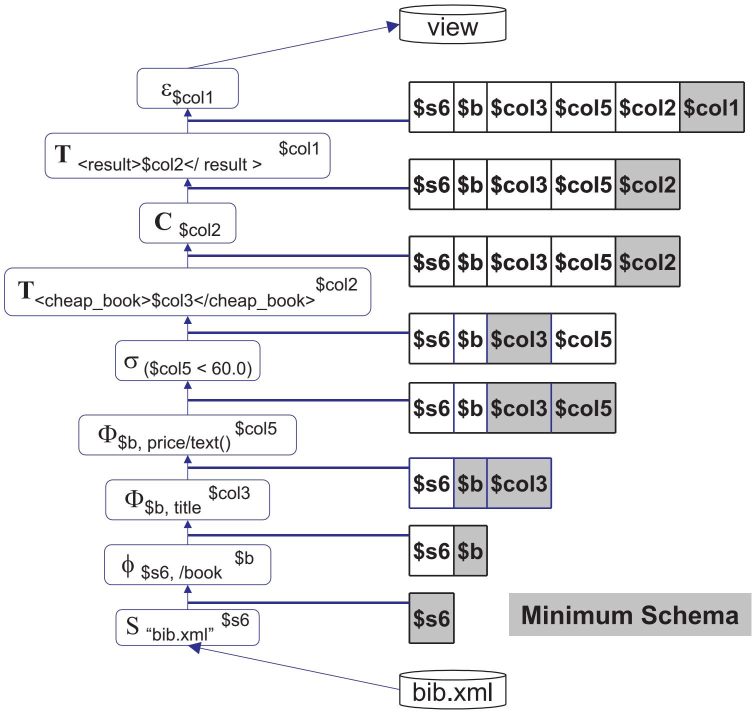

Figure 6: Full and Minimum Schema for running example

By definition, all columns from the input table are retained in the output table of an operator (except for the *Combine* operator), plus an additional one may be added. 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 [31]. 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 [31].

The Full and the Minimum Schema for the running example view definition XQuery are shown in Figure 6.

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, for any two tuples in the output XAT table of the *Distinct* the relative order is undefined.

Similarly, 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 the 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 the any two XML nodes derived from different input tuples is undefined.

4 The VOX Approach for Maintaining Order

4.1 Preserving Order in the Context of the XML Algebra

The requirement of preserving document order makes the maintenance of XML views significantly different from the maintenance of relational views. We note that the basic notion enabling efficient incremental maintenance of relational select-project-join views is that such views are distributive with regard to the union. For example, for any two relations R and Q, any joining condition c and any delta set ΔQ of inserted tuples into Q, the equation $R \bowtie_c (Q \cup \Delta Q) = (R \bowtie_c Q) \cup (R \bowtie_c \Delta Q)$ holds. Thus, when the relation Q is updated by inserting the delta set ΔQ, only the newly inserted tuples need to be joined with the tuples in R, that is $R \bowtie_c \Delta Q$ needs to be calculated. The updated view extent can be obtained as union of the the view extent before the update $R \bowtie_c Q$, and the newly computed $R \bowtie_c \Delta Q$. More generally, the distributiveness of the operators over different operations is often exploited. Relational views that contain non-distributive operators are maintained by performing selective recomputation [17], for example by recomputing only the set of groups affected by an update, or by maintaining auxiliary views derived from the intermediate results of the view computation [16].

It is important to note here that with the requirement of maintaining the order among the tuples, none of the XAT operators is distributive over any update operation, as due to an update tuples may be inserted at arbitrary positions. For example, assume a new *j-th* tuple tin_j is inserted in the input XAT table R of the operator *Navigate Unnest*. As a result, a sequence of new zero or more XAT tuples $tout_j^{(l)}$ may have to be inserted into the output XAT table. However, these tuples must be placed after the tuples derived from all tin_i, $i < j$ and before the tuples derived from all tin_k, $k > j$.

A similar issue arises due to the requirement of maintaining order among XML nodes contained in a single cell. When insertions or deletions of XML nodes from a cell occur as a result of an update, then they have to be done at specific positions. The essence of this problem is the same as that for tuples in an XAT table, as again the new sequence cannot be obtained as union (or difference) of the old sequence and the new member.

The two obvious solutions are: (1) relying on physical sequential storage medium that allows for insertions or deletions at specified positions and that is always kept sorted, or (2) consecutively numbering the XAT tuples and the members of sequences. For (1), the tuples in a table and the nodes in a cell would be stored sequentially in correct order. However, in most cases iterations over the tuples in the input or the output XAT tables would have to be done for determining the correct position where the update should be done. Also, such storage system that supports insertions and deletions at specific positions would have to be provided. For (2), insertions and deletions would lead to frequent renumbering. Hence, these obvious solutions would not be practical, as both would require extra processing and distributiveness over update operations would again not be achieved.

Thus 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.

4.2 Techniques for Encoding XML Order

We observe that in most cases the order among the tuples in an XAT table (and among nodes in a sequence) is dependent 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 encoding order, if the node identity encodes the unique path of that node in the 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 in the presence of updates [23, 6]. The work in [23] 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. Such renumbering is clearly undesirable for view maintenance.

In [6] 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,

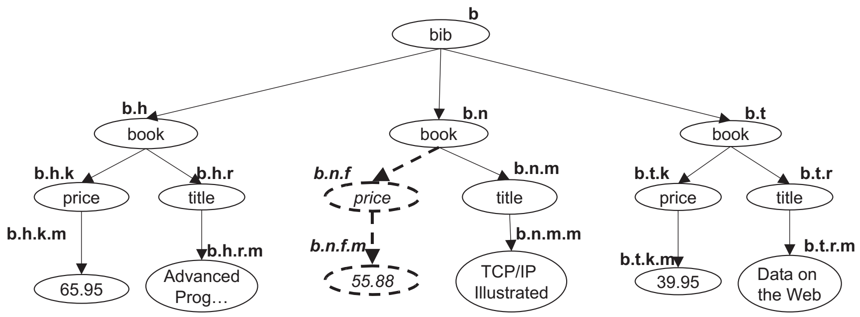

Figure 7: Lexicographical ordering of the XML document presented in Figure 1

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 7 for example).

This encoding is well suited for our purpose of view maintenance for the following reasons. It does not require reordering on updates, identifies a unique path from the root to the node and embeds the relative order on each level. These order-reflecting node identity encodings are called **LexKey**-s. We use the notation $k_1 \prec k_2$ to note that LexKey k_1 lexicographically precedes LexKey k_2.

The LexKeys node identity encoding for nodes in an XML document has the following properties: If k_1 and k_2 are the LexKeys of nodes n_1 and n_2 respectively, then:

- $k_1 \prec k_2$ if and only if n_1 is 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.

For insertion and deletion of nodes the following properties hold:

- It is always possible to generate a LexKey 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 LexKeys of other existing nodes.

4.3 Using LexKeys in the Context of XML Algebra

We use LexKeys 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 LexKeys assigned to all of its nodes. For reducing redundant updates and avoiding duplicated storage we only store references (that is LexKeys) in the XAT tables rather than actual XML data. This is sufficient as the LexKeys serve as node identifiers and capture the order. From here on, when saying a cell in a tuple we mean the LexKeys or the collection of LexKeys stored in that cell. The actual XML data is stored only once in a shared storage, called Storage Manager. Given a LexKey, the Storage Manger supports access to its value and to its children nodes. Figure 8 illustrates the usage of LexKeys as references to source XML nodes.

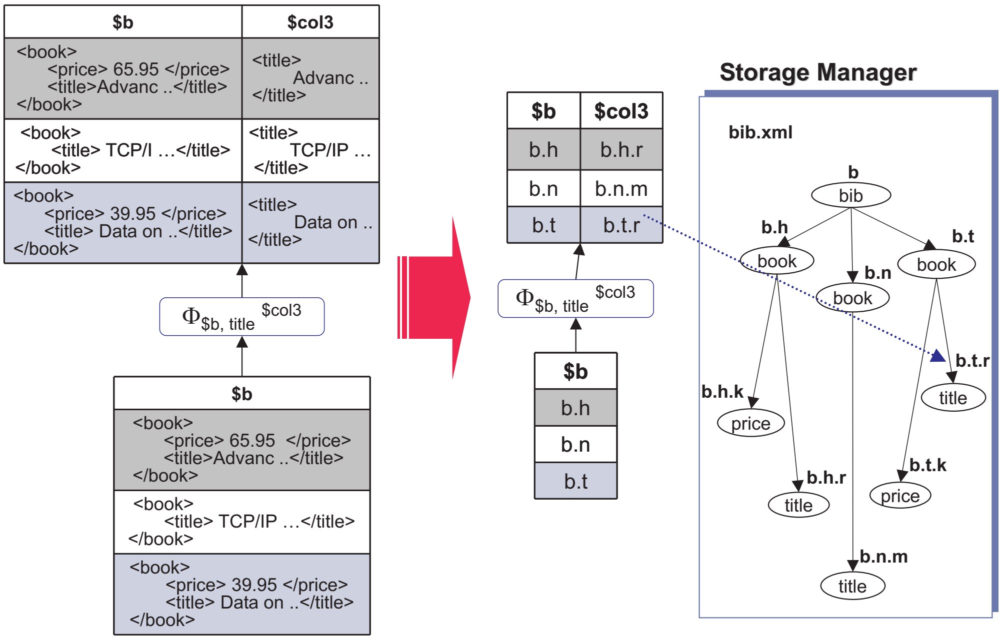

Figure 8: LexKeys as references to source XML nodes

As LexKeys are references to the base data, they can be used for accessing that data when needed by ceratin operator. For example, the *Select* operator needs to access the XML node values in order to evaluate a condition, and it does so by retrieving the needed nodes referenced by the LexKeys in the tuple it is evaluating. Similarly, the *Navigate Collection* operator shown in Figure 8, for processing the first tuple from the input, retrieves the children of $b.h$ which are of type title from the Storage Manger, and places their LexKeys in the output XAT table.

We also use LexKeys to encode the node identity of any constructed nodes either in intermediate states of the view algebra tree or in the final view extent. The LexKeys 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, and instead reference through LexKeys the other source data or constructed nodes that are included in the newly constructed node, e.g., *<cheap_book>b.t.r</cheap_book>* as shown in Figure 9.

In addition to the LexKeys described above, we also use LexKeys 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 4.4.2. This follows the logic of treating keys as symbols and composing them into higher-level keys. For example, the LexKey k = "*b.c..c.d*" is a composition of the LexKeys k_1 = "*b.c*" and k_2 = "*c.d*" and ".." is used as delimiter. We denote this by $k = compose(k_1, k_2)$. Note that the way LexKeys are composed guarantees that given two

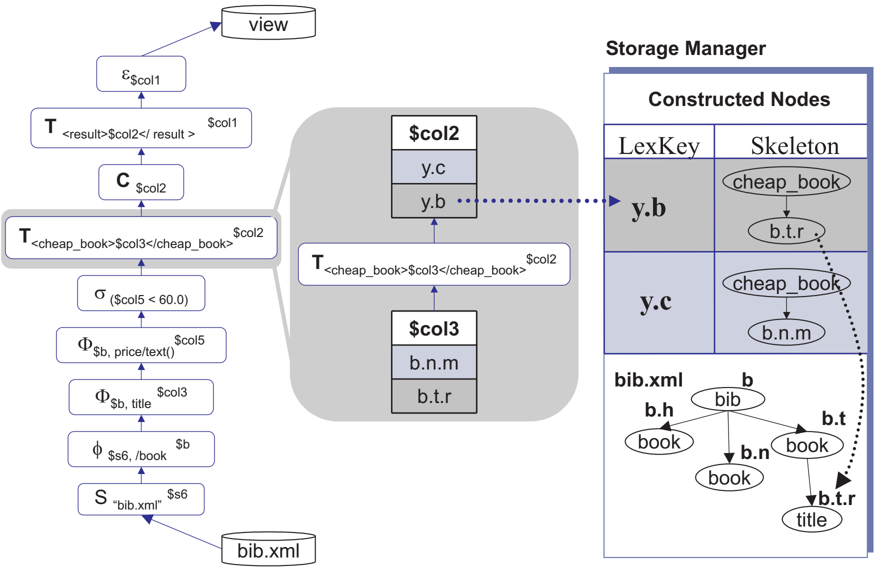

Figure 9: LexKeys as references to constructed XML nodes

composed LexKeys, $k_1 = (k_{11}, ..k_{1n})$ and $k_2 = (k_{21}, ..k_{2m})$, it holds that: $k_1 \prec k_2 \Leftrightarrow ((\exists j, 1 \leq j \leq min(n, m))(\forall i, 1 \leq i < j)(k_{1i} == k_{2i}) \wedge (k_{1j} \prec k_{2j})) \vee ((n < m) \wedge (\forall i, 1 \leq i \leq n)(k_{1i} == k_{2i}))$. Basically the composed LexKey k_1 precedes the composed LexKey k_2 in two cases: (1) if the first $j-1$ LexKeys from which both k_1 and k_2 are composed are equal and the $j-th$ LexKey from which k_1 is composed precedes the $j-th$ LexKey from which k_2 is composed, or (2) if k_1 is composed of less LexKeys than k_2 and k_1 is prefix of k_2.

4.4 Maintaining Order Using LexKeys

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

4.4.1 Maintaining Order Among XAT Tuples

The order among the tuples in an XAT table can now be determined by comparing the LexKeys stored in cells corresponding to some of the columns. For example, consider the tuples $t_1 = (b.h, b.h.r)$ and $t_2 = (b.n, b.n.m)$ in the input XAT table of the operator $\Phi^{\$col3}_{\$b,/title}$ in Figure 11. Here t_1 should be before t_2, that is $before(t_1, t_2)$ is true. This can be deduced by comparing the LexKeys 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 and can be determined by comparing those LexKeys. Such columns are said to compose the *Order Schema* of the table.

Definition 4.1 *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 4.2 *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 presented 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 LexKeys.

Cat.	Operator op	OS^*_Q
I	$T^{col}_p(R)$ $\Phi^{col'}_{col,path}(R)$ $\bigcup^{x\,col}_{col1,col2}(R)$ $\bigcap^{x\,col}_{col1,col2}(R)$ $-^{x\,col}_{col1,col2}(R)$ $\sigma_c(R)$	OS_R
II	$S^{col'}_{xmlDoc}$ $C_{col}(R)$ $\delta_{col}(R)$ $\gamma_{col[1..n]}(R, fun)$	$\emptyset$
III	$\times(R, P)$ $\bowtie_c (R, P)$ $\overset{\circ}{\bowtie}_{Lc}(R, P)$	$(on^{(R)}_1, on^{(R)}_2, ...on^{(R)}_{mr}, on^{(P)}_1, on^{(P)}_2, ...on^{(P)}_{mp})$ $mr = \lvert OS_R \rvert$, $mp = \lvert OS_P \rvert$
IV	$\phi^{col'}_{col,path}(R)$	$(on^{(R)}_1, on^{(R)}_2, ...on^{(R)}_p, col')$ if $on^{(R)}_m = col$ 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
* $Q = op^{out}_{in}(R)$, $OS_R = (on^R_1, on^R_2, ...on^R_m)$		

Table 1: Rules for computing Order Schema

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 Order Schema of the input XAT table R. An example of such a case is presented in Figure 10. By the semantics of this operator presented in Section 3.2, it processes one tuple at 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 derived from in R. For example, the order among the tuples marked as 1 and 2 in the output XAT table in Figure 10 should correspond to the order of the tuples marked as 1 and 2 in the input XAT table in that figure, as the output tuples marked as 1 and 2 are derived form the input tuples 1 and 2 correspondingly. 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'. In Figure 10, for example, the output tuples marked as 2 and 3 are both derived from the input tuple marked as

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

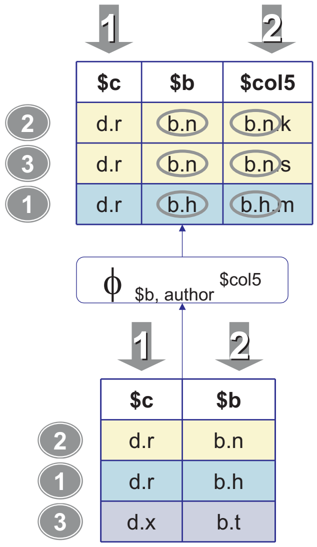

Figure 10: Example of Order Schema computation for *Navigate Unnest*

2. Thus, the order between them should correspond to the relative document order between the XML nodes referenced by $b.n.k$ and $b.n.s$.

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.

For the example in Figure 10, that means that given that Order Schema of the input R is $OS_R = (\$c, \$b)$, the Order Schema of the output Q should be $OS_Q = (\$c, \$col5)$. The column $\$col5$ is added to capture the order among tuples derived from the same tuple in R, as by the properties of the LexKeys, the LexKeys present in that column reflect the document order of the nodes they reference. Also, by the properties of the LexKeys, all the LexKeys contained in $\$col5$ have the LexKeys from $\$b$ as prefixes. Thus column $\$col5$ automatically captures the order references of the column $\$b$, and thus column $\$b$ need 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.

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 4.2 of the tuples always yields $t_1 \prec t_2$. And vice versa, if $t_1 \prec t_2$, then either t_1 should really 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 and we prove its correctness.

Theorem 4.1 *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 3.2, (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

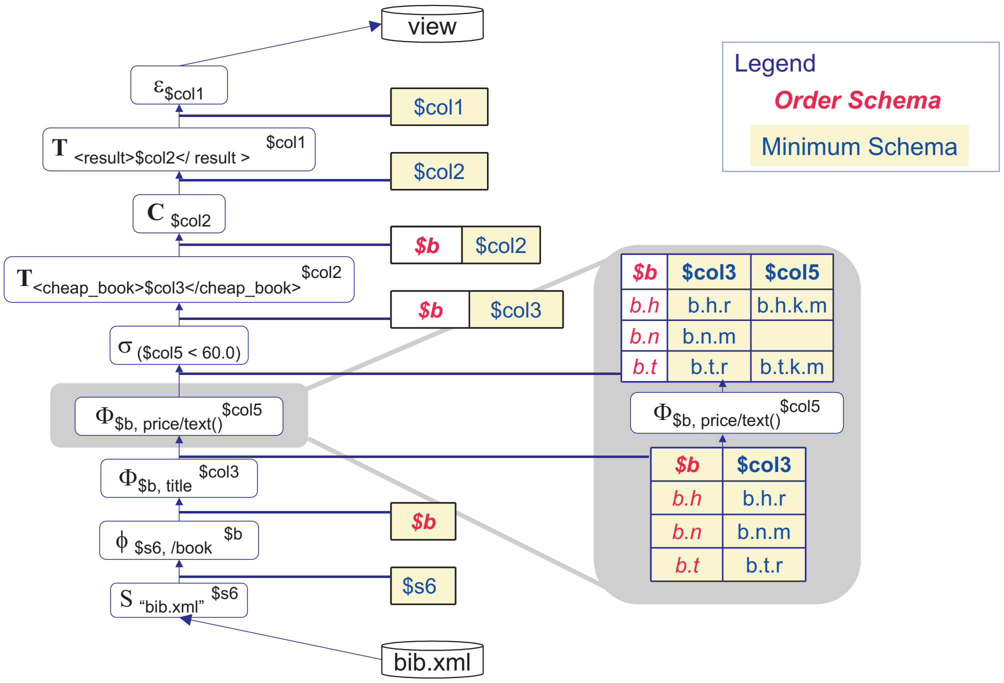

Figure 11: Order Schema computation example

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 4.2, 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 3.2). 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 4.2 $(tin_1^{(P)} \prec tin_2^{(P)}) \Rightarrow (tout_1 \prec tout_2)$.

Category IV. The operator *Navigate Unnest* $\phi_{col,path}^{col'}(R)$ by its definition presented in Section 3.2 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 4.2, $on_j = col$ 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 4.2 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 3.2): $(\exists l_1, l_1 > 0)|(tout_1[col'] = \phi(path : tin_1[col])[l_1])$, 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 4.1 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 more 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.

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

The schema composed of all the columns present in the Minimum Schema (as defined in Section 3.2) or in the Order Schema (as per Definition 4.1) for an XAT table R is called the **Real Schema** (RS) of R. The Real Schema of an XAT table is the schema assumed for that table for view maintenance. The Real Schema for each XAT table present in the running example is composed of all the columns shown for each table in Figure 11.

4.4.2 Maintaining Order in Sequences of XML Nodes

For sequences of XML nodes contained in a single cell that have to be in document order, as those created by the *XML Union*, *XML Difference*, *XML Intersection* and *Navigate Collection*, the LexKeys representing the nodes accurately reflect their order. This is due to the fact that the LexKeys 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. 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 LexKeys 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], $1 < i \leq |OS_R|$)
        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

Figure 12: The function *combine*

To represent an order that is different than the one encoded in the LexKey k serving as the node identity of the node, we attach an additional LexKey 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 LexKey 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 LexKeys 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 $order(k_1) \prec order(k_2)$.

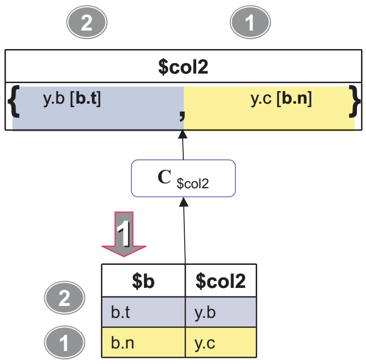

Figure 13: Example of setting overriding order by Combine

The *Combine* operator sets the overriding order to the LexKeys that it places in its output XAT table, as described in Figure 12. 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 $C_{col}(R) = tout = (\biguplus_{j=1}^{p} combine(tin_j[col], tin_j, col))$.

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

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. Such a case is presented in Figure 13. 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 LexKeys in the output XAT table is composed of the order references present in all columns in the Order Schema of the input. In the example in Figure 13, $\$b$ is the only column in the Order Schema of the input. Thus, when the input XML node referenced by $y.b$ is placed in the output XAT table it gets overriding order equal to the order represented by the LexKey present in column $\$b$ in the tuple it derived from, that is $b.t$. Thus $y.b$ after being processed by *Combine* becomes $y.b[b.t]$.

The XML set operators *XML Union*, *XML Difference*, *XML Intersection* remove the overriding order (if present) of the node identity LexKeys that they place in their output XAT tables, as by definition (see Section 3.2) they produce a column in which the nodes are in document order.

Theorem 4.2 *Let $kout_1$ and $kout_2$ be two LexKeys in a same cell in an XAT table R in an XAT algebra tree. Let these LexKeys 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 3.2: (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 collection of LexKeys. Such operators are: *Navigate Collection*, *XML Union*, *XML Difference*, *XML Intersection* and *Combine*. All the other operators do not create collections of LexKeys, 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 LexKeys $kout_1$ 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 LexKeys $kout_1$ 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 *Navigate Collection*, *XML Union* or *XML Difference* is similar. Again, $before(n_1, n_2)$ holds only when n_1 is ordered before n_2 regarding document order. These operators remove the overriding order of the LexKeys $kout_1$ and $kout_2$ if present, thus, $(kout_1 \prec kout_2)$ must also hold.

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 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 4.2 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 12, 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 13. 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 4.1. □

4.5 Migration of XML Algebra to (Non-Ordered) Bag Semantics

We can thus conclude that the technique of encoding order with LexKeys 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.

Figure 14 illustrates the execution of the running example XQuery using LexKeys as references and for encoding order. It shows all the intermediate results and the content of the Storage Manager.

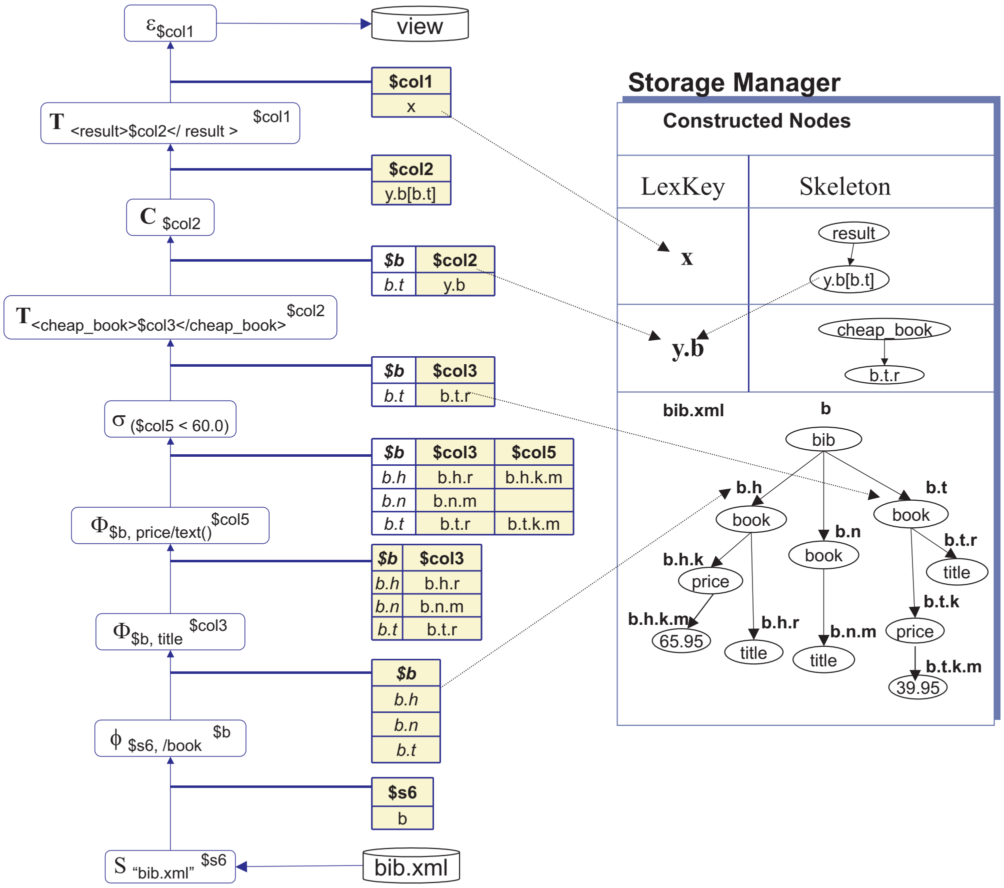

Figure 14: Reference-based execution for running example

The execution starts at the *Source* operator at the bottom of the algebra tree, which gets the LexKey b representing the XML document *bib.xml* from the Storage Manager and places it in its output XAT table. This table now becomes input for the operator $\phi^{\$b}_{\$s6,/book}$, which extracts the book elements that are children of the nodes in column $\$s6$. The node identified by b is the only such node, thus the LexKeys of its children of type *book* are retrieved from the Storage Manager and placed in the output XAT table of $\phi^{\$b}_{\$s6,/book}$. Next, the $\Phi^{\$col3}_{\$b,title}$ processes the input tuples extracting the title elements. Note that the order in which the input tuples are processed and the output is generated is irrelevant, as the order is preserved in the Order Schema of that table, that is the LexKeys in column $\$b$ in this case. The next operator $\Phi^{\$col5}_{\$b,/price/text()}$ extracts the prices from the nodes present in $\$b$. The second book does not have a price, thus the content of the cell corresponding to $\$col5$ of the respective output tuple is empty. In order to evaluate the selection condition, the *Select* operator retrieves the values of the nodes identified by the LexKeys present in $\$col5$, over which the selection condition is specified. Only the book with price \$39.95 passes the selection condition, thus only one tuple is output. Next, the title of this book is tagged and the newly constructed node is passed to the Storage Manager along with the LexKey $y.b$

assigned to it by the *Tagger* operator. This LexKeys is tree-wide unique, serves as reference to the node, but does not encode order. The *Combine* operator sets the overriding order of $y.b$ to reflect the order of the tuple it derived from. The next *Tagger* tags this node, creating the node identified by x, which is also passed to the Storage Manager.

The result of the XQuery is obtained by dereferencing the LexKey x. First, the skeleton of the constructed node identified by x is retrieved and then the LexKeys contained in that skeleton are dereferenced. The process continues recursively, and finally the resulting XML document is obtained. Generally, the dereferencing may require partial reordering of sibling LexKeys that are children of the same constructed node based on their order. However, in this example that is not the case, thus no reordering is needed.

5 Rules for Incremental Maintenance of XML Views

5.1 Update Operations and Format of the Delta

When an update XQuery is being applied to one of the input sources, a sequence of XML updates as presented in Table 2 is produced by the XQuery processor. Each such update is then applied to the document. Note that an insertion or deletion of a complex element is specified as a single XML update. For the position parameter *pos* to which an XML update refersto, rather then including integer ordering positions (as specified in the definition of position in Section 3.1), the LexKeys of the corresponding nodes are given. The LexKey k represents the root element of the document affected by the update. As illustration, the update presented in Figure 2 is specified as $Insert(b.n.f, book[b.n]/price[b.n.f], b)$ which corresponds to $\delta^{+}_{b.n.f,book[b.n]/price[b.n.f]}b$ when using the shorter notation introduced in Table 2.

Update Operation	*Description*	*Notation*
Insert *(n, pos, k)*	Insert node with LexKey n at position *pos* starting at k into node with LexKey k	$\delta^{+}_{n,pos}k$
Delete *(n, pos, k)*	Delete node with LexKey n at position *pos* starting at k from node with LexKey k	$\delta^{-}_{n,pos}k$
Replace*(new, pos, k)*	Replace value at position *pos* starting at k with *new* from node with LexKey k	$\delta^{r}_{pos,new}k$

Table 2: XML update operations (δk)

We also define a set of update operations over XAT tables, referred to as *delta*. The format of the *delta*, that is, the set of possible intermediate updates on XAT tables is described in Table 3. The intermediate XAT updates specify modifications of an XAT table, whereas the intermediate XML updates only carry information that a node referenced by the specified LexKey has been modified.

All the intermediate updates except for $\triangle R$ need to specify the tuple(s) to which the update applies, that is which tuple(s) have to be deleted or modified. A popular approach in relational view maintenance work is for the update to specify the full tuple to which the update applies. In the context of the XML algebra this is not necessarily the best choice, as often recomputation would have to be performed for that purpose. For example, for the *Combine* operator to meet such a requirement of being able to completely specify a tuple, the *Combine* would either have to perform full recomputation for each update it produces, or have its output XAT table materialized, because it always outputs only one tuple derived from all the tuples in the input. In both cases it would have to propagate the entire content of its output XAT table each time.

Intermediate XAT Updates		
$u(\triangle R)$	Insertion of tuples $\triangle R$ into XAT table R	$R^{new} \leftarrow R \uplus \triangle R$
$u(\bigtriangledown R)$	Deletion of tuples $\bigtriangledown R$ from XAT table R	$R^{new} \leftarrow R \dot{-} \bigtriangledown R$
$u(\triangle\!\!\!\!\circ\, c, col, tid)$	Insertion of LexKeys into the cell $t[col]$, tid identifies the tuple t [5]	$t[col]^{new} \leftarrow t[col] \uplus \triangle\!\!\!\!\circ\, c$
$u(\bigtriangledown\!\!\!\!\circ\, c, col, tid)$	Deletion of LexKeys from the cell $t[col]$, tid identifies the tuple t	$t[col]^{new} \leftarrow t[col] \dot{-} \bigtriangledown\!\!\!\!\circ c$
Intermediate XML Updates		
$u(\delta k, col, tid)$	Modification of LexKey k in cell $t[col]$ by δk, where δk is any updates from Table 2, tid identifies tuple t	

Table 3: The format of the intermediate updates

We thus instead choose to assign to each tuple t in each XAT table R an integer identifier, *tid*, unique within that table. Thus, even though here we use $\bigtriangledown R$ to represent the deleted tuples, the actual update in fact only carries the *tid*-s of the deleted tuples.

Given an XAT table R, the function $R.getTuple(tid)$ returns the tuple t identified by tuple id *tid*.

5.2 Update Propagation Algorithm

We augment each algebra operator with incremental propagation functionality in addition to its primary computation functionality needed for query execution. The view maintenance process is triggered by an XML update as in specified in Table 2. Our update propagation algorithm performs a bottom-up postorder traversal of the tree, invoking each operator with a sequence of zero or more updates. The *Source* operator accessing the updated XML document is invoked first with the sequence of XML updates resulting from the update XQuery[6]. The *Source* operator then translates the update into an intermediate update (Table 3). From there onwards, each operator in the algebra tree, processes one intermediate update at a time and translates it into a sequence of zero or more intermediate output updates. After the node has processed the entire sequence of its input updates, it outputs the sequence of updates it has generated. Due to the post-order traversal, each node processes the updates only after all of its children have processed their updates first. After all nodes have been visited at most once the view is refreshed.

Figure 15 gives an overall illustration of the update propagation process.

Below we define update propagation rules for pairs of each algebra operator and each type of update.

[5]In this section, we consistently use t for the tuple identified by *tid*.

[6]If the algebra tree contains more *Source* operators accessing the updated XML document, then they are invoked in a postorder manner.

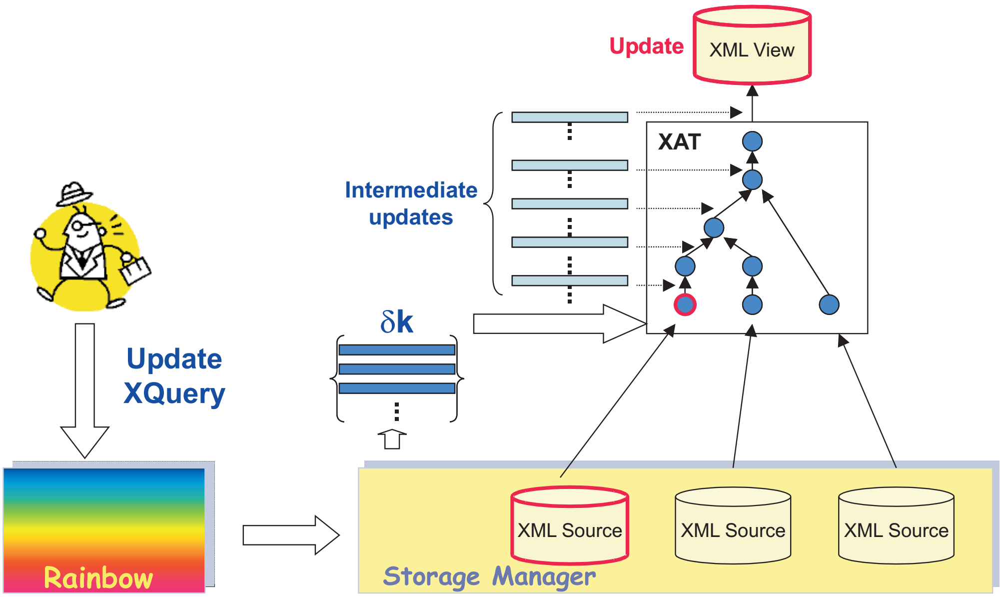

Figure 15: Update propagation illustration

Some operators can process any update without requiring any auxiliary information beyond the input update notification (the *delta*). But for certain operators, the output *delta* cannot always be calculated using only the input *delta*, but additional auxiliary information is required, in particular subsets of either the respective input or output XAT tables. In this case our system stores the needed columns of the input or the output XAT tables extracted from the intermediate results of the initial computation of the view extent as auxiliary views. That is, these extra (partial) XAT tables kept as auxiliary views now must be also incrementally maintained. These auxiliary views only store LexKeys, thus are compact. Each column in an XAT table is materialized if and only if its materialization is required by at least one of the operators having that XAT table as input or output.

While propagating an update, the operators may also access XML nodes that are part of an inserted (deleted) complex element. In such case, the operators request the children (or the value) of an inserted (deleted) node.

The operators need to be able to identify the respective output tuples affected by an update or a delete of an input tuple. By associating parent-child relationships between the tuples in the input table R and the tuples in the output XAT table Q we are always able to identify the tid of the output tuple(s) given the tid of the input tuple that they are derived from. We denote that as $Q.getDerived(tid)$. For the operators that for each tuple in R produce at most one tuple in Q, such as *Select* for example, the tid of any output tuple always corresponds to the tid of the input tuple it derived from, that is $tid = Q.getDerived(tid)$. For such operators, the parent-child relationship is implicit and is not explicitly kept. For the operators which may produce several output tuples corresponding to one input tuple (the Join family of operators and *Navigate Unnest*) we maintain an index of tuple parent-child relationships. This index is created at the same time as when the auxiliary views are created, that is while initially evaluating the view extent, and is stored as part of the corresponding XAT table. The tuple parent-child relationship is not meaningful for the operators *Combine*, *Source* and *Expose*. *Combine* produces only one output tuple derived from all the tuples in the input XAT table. Since tuple identifiers are only unique within an XAT table, we always assign it $tid = 1$. The *Source* operator does not have an input XAT table. The *Expose* operator does not have an output XAT table.

5.3 Propagation Rules for Individual Operators

Our mechanism of encoding order using LexKeys empowers most XAT operators to become distributive over update operations. For example, for operators with only one input that means that (1) the result after the modification can be obtained by performing bag union or difference of the old result and the computed delta, and (2) the output delta can be computed using only the update, without requiring extra information beyond the input update.

When the view maintenance process is triggered by an XML update δk over the XML document *xmlDoc*, the respective *Source* operator $S^{col'}_{xmlDoc}$ translates it into $u(\delta k, col', 1)$. From there on, only intermediate (XAT and XML) updates are propagated, thus the rules below are defined for such updates.

5.3.1 Propagation of Updates through XAT SQL Operators

The migration from ordered bag semantics to bag semantics makes our XAT SQL operators equivalent to their relational counterparts, i.e., relational bag algebra [8, 19]. We can now adopt the update propagation rules for those SQL-like operators from the respective relational view maintenance work ([8], [19]). Thus we do not discuss them further.

5.3.2 Propagation of Updates through XAT XML Operators

Propagating Insertions and Deletions of Tuples

All XAT XML operators become distributive over insertions and deletions of tuples. In particular, if *op* is any of the operators: *Tagger*, *Navigate Collection*, *Navigate Unnest*, *XML Union*, *XML Intersect* or *XML Difference*, the following propagation equations hold:

$op^{out}_{in}(R \uplus \triangle R) = op^{out}_{in}(R) \uplus op^{out}_{in}(\triangle R)$, and

$op^{out}_{in}(R \dot{-} \bigtriangledown R) = op^{out}_{in}(R) \dot{-} op^{out}_{in}(\bigtriangledown R)$.

The *Combine* operator has the equivalent property, but at the cell level. Let $tout^{old} = (c^{old})$ and $tout^{new} = (c^{new})$ denote the results of $C_{col}(R)$ and $C_{col}(R^{new})$ correspondingly. Then:

$C_{col}(R \uplus \triangle R) = tout^{new} = (c^{new}) = (\Pi_{col} C_{col}(R) \uplus \Pi_{col} C_{col}(\triangle R)) =$
$(c^{old} \uplus \Pi_{col} C_{col}(\triangle R))$, and

$C_{col}(R \dot{-} \bigtriangledown R) = tout^{new} = (c^{new}) = (\Pi_{col} C_{col}(R) \dot{-} \Pi_{col} C_{col}(\bigtriangledown R)) =$
$(c^{old} \dot{-} \Pi_{col} C_{col}(\bigtriangledown R))$.

These propagation equations are derived directly from the semantics of the operators defined in Section 3.2, when the order among tuples and nodes is explicitly maintained. The propagation rules for the XML operators on insertions and deletions of tuples can be directly deduced from these maintenance equations. For example, if the output of an $\phi^{col'}_{col,path}(R)$ is denoted as $Q = \phi^{col'}_{col,path}(R)$, then for an input update $u(\triangle R)$, the operator $\phi^{col'}_{col,path}(R)$ propagates $u(\triangle Q)$, where $\triangle Q = \phi^{col'}_{col,path}(\triangle R)$.

Propagating Insertions and Deletions of LexKeys in a Cell

Operator	*Propagate*	*Info Accessed*
$\phi^{col'}_{col,path}(R)$	$u(\triangle Q) \mid (\forall t' \in \triangle Q)$ $((\forall cn \in RS_Q \mid cn \neq col')(t'[cn] = t[cn]) \wedge$ $(\forall n \in \phi(path : \triangle\!\!\!\circ\, c))(\exists! t' \in \triangle Q)(t'[col'] = n)$	t
$\Phi^{col'}_{col,path}(R)$	$u(\triangle\!\!\!\circ\, c', col', tid) \mid \triangle\!\!\!\circ c' = \phi(path : \triangle\!\!\!\circ\, c)$	none
$T^{col'}_{p}(R)$	$(\forall ppath \mid \phi(ppath : p) = col)(\forall n \in \triangle\!\!\!\circ\, c)$ $u(\delta^{+}_{n,ppath} k', col', tid)$ k' is the key reproduced from tid	none
$C_{col}(R)$	$u(\triangle\!\!\!\circ\, c', col, 1) \mid \triangle\!\!\!\circ c' = combine(\triangle\!\!\!\circ\, c, t, col)$	$(\forall on \in OS_R)t[on]$
$\overset{x\,col'}{\cup}_{col1,col2}(R)$[7]	$u(\triangle\!\!\!\circ\, c', col, tid) \mid \triangle\!\!\!\circ c' = \{n \mid (n \in \triangle\!\!\!\circ\, c) \wedge$ $(n \notin t[col1]) \wedge (n \notin t[col2])\}$	$t[col1], t[col2]$
$\overset{x\,col'}{\cap}_{col1,col2}(R)$	$u(\triangle\!\!\!\circ\, c', col, tid) \mid \triangle\!\!\!\circ c' = \{n \mid (n \in \triangle\!\!\!\circ\, c) \wedge$ $((col = col1) \wedge (n \notin t[col1]) \wedge (n \in t[col2])) \vee$ $((col = col2) \wedge (n \in t[col1]) \wedge (n \notin t[col2]))\}$	$t[col1], t[col2]$

$\overset{x\,col'}{-}_{col1,col2}(R)$	if (col=col1)
	$u(\triangle\!\!\!\circ\, c', col, tid) \mid \triangle\!\!\!\circ c' = \{n \mid (n \in \triangle\!\!\!\circ\, c) \wedge$
	$(n \notin t[col1]) \wedge (n \notin t[col2])\}$
	if (col=col2)
	$u(\bigtriangledown\!\!\!\circ\, c', col, tid) \mid \bigtriangledown\!\!\!\circ c' = \{n \mid (n \in \triangle\!\!\!\circ\, c) \wedge$
	$(n \in t[col1]) \wedge (n \notin t[col2])\}$
		$t[col1], t[col2]$

Table 4: Propagation rules for $u(\triangle\!\!\!\circ\, c, col, tid)$ for XAT XML operators

The rules for propagating $u(\triangle\!\!\!\circ c, col, tid)$ when col is among the input columns of the corresponding operator are shown in Table 4. In that table, when a rule needs to access the tuple t identified by the tuple identifier tid, it is assumed that it can read the state of that tuple before the incoming update $u(\triangle\!\!\!\circ\, c, col, tid)$ has been applied. This is achieved by applying the updates only after they have been propagated. The rules presented in Table 4 are directly derived from the corresponding maintenance equations and they can easily be proven correct.

For example, consider the rule for $\Phi^{col'}_{col,path}(R)$ when col in $u(\triangle\!\!\!\circ c, col, tid)$ matches column col, which is the input column col for $\Phi^{col'}_{col,path}(R)$. Let $t = (c_1, c_2, ...c, ...c_n)$ and $t^{new} = (c_1, c_2, ...c \uplus \triangle\!\!\!\circ c, ...c_n)$ be the state of the tuple t before and after the update respectively. Let $tout$ and $tout^{new}$ be the corresponding derived tuples in the output table, where $tout^{new}$ is obtained by recomputation over t^{new}. Let the last cells of $tout$ and $tout^{new}$ correspond to col'. Then:

[7]The rule assumes that either $col = col1$ or $col = col2$ holds.

Operator	*Propagate*	*Info Accessed*
$\phi^{col'}_{col,path}(R)$	$u(\bigtriangledown Q)\vert(\forall tid' \in \bigtriangledown Q)(tid' \in Q.getDerived(tid))\wedge$ $(\forall n \in \phi(path : \triangledown c))(\exists! tid' \in \bigtriangledown Q)$ $(t'[col'] = n, t' = Q.getTuple(tid'))$	$t'[col']$ tid index
$\Phi^{col'}_{col,path}(R)$	$u(\triangledown c', col', tid)\vert\ \triangledown c' = \phi(path : \triangledown c)$	none
$T^{col'}_{p}(R)$	$(\forall ppath\vert\phi(ppath : p) = col)(\forall n \in \triangledown c)$ $u(\delta^{-}_{n,ppath} k', col', tid)$ k' is the key reproduced from tid	none
$C_{col}(R)$	$u(\triangledown c', col, 1)\vert\ \triangledown c' = combine(\triangledown c, t, col)$	$(\forall on \in OS_R)t[on]$
${\overset{x}{\cup}}^{col'}_{col1,col2}(R)$	$u(\triangledown c', col, tid)\vert\ \triangledown c' = \{n\vert(n \in \triangledown c)\wedge$ $((col = col1) \wedge (n \notin (t[col1] \mathbin{\dot{-}} \triangledown c)) \wedge (n \notin t[col2]))\vee$ $((col = col2) \wedge (n \notin t[col1])) \wedge (n \notin (t[col2] \mathbin{\dot{-}} \triangledown c))\}$	$t[col1]$, $t[col2]$
${\overset{x}{\cap}}^{col'}_{col1,col2}(R)$	$u(\triangledown c', col, tid)\vert\ \triangledown c' = \{n\vert(n \in \triangledown c)\wedge$ $((col = col1) \wedge (n \notin (t[col1] \mathbin{\dot{-}} \triangledown c)) \wedge (n \in t[col2]))\vee$ $((col = col2) \wedge (n \in t[col1]) \wedge (n \notin (t[col2] \mathbin{\dot{-}} \triangledown c)))\}$	$t[col1], t[col2]$

${\overset{x}{-}}^{col'}_{col1,col2}(R)$	if (col=col1)
	$u(\triangledown c', col, tid)\vert\ \triangledown c' = \{n\vert(n \in \triangledown c)\wedge$
	$(n \notin t[col1] \mathbin{\dot{-}} \triangledown c)) \wedge (n \notin t[col2])\}$
	if (col=col2)
	$u(\triangle c', col, tid)\vert\ \triangle c' = \{n\vert(n \in \triangledown c)\wedge$
	$(n \in t[col1]) \wedge (n \notin t[col2] \mathbin{\dot{-}} \triangledown c))\}$
		$t[col1], t[col2]$

Table 5: Propagation rules for $u(\triangledown c, col, tid)$ for XAT XML operators

$tout = \Phi^{col'}_{col,path}(t) = \Phi^{col'}_{col,path}(c_1, c_2, ...c, ...c_n) = \Pi_{RS}(c_1, c_2, ...c, ...c_n, \phi(path : c))$

$tout^{new} = \Phi^{col'}_{col,path}(t^{new}) = \Phi^{col'}_{col,path}(c_1, c_2, ...c \uplus \triangle c, ...c_n) = \Pi_{RS}(c_1, c_2, ...c \uplus \triangle c, ...c_n,$

$\phi(path : c \uplus \triangle c)) = \Pi_{RS}(c_1, c_2, ...c \uplus \triangle c, ...c_n, \phi(path : c) \uplus \phi(path : \triangle c))$

By comparing $tout$ and $tout^{new}$, we can conclude that on $u(\triangle c, col, tid)$, $\Phi^{col'}_{col,path}(R)$ should propagate $u(\triangle c', col', tid)$, where $\triangle c' = \phi(path : \triangle c)$. In addition (as explained below), the original update $u(\triangle c, col, tid)$ should be propagated if col is in the Minimum Schema of the output XAT table. In this case, the update propagation can be done without any additional information, i.e., the output updates are directly derived from the LexKeys contained in the original update and the XML nodes that they identify.

The rules for $u(\triangledown c, col, tid)$ are similar and are presented in Table 5. Again, when a rule needs to access the tuple t identified by the tuple identifier tid, the state of that tuple before the incoming update $u(\triangledown c, col, tid)$ has been applied is assumed to be still accessible by the maintainer.

Propagating Intermediate XML Updates

The intermediate XML update operations only affect the XAT XML operators *Navigate Collection* and *Navigate Unnest* because they require accessing keys at a level deeper than the updated node k. The other XAT XML operators do not require accessing the children nor the values of the nodes identified by the LexKeys in their input XAT table. Thus, if the structure or the value of a certain input LexKey has changed, their output is not modified. Also, if the affected LexKey is present in their output XAT table, the node it identifies has already been updated, as the XML nodes are stored only once in the Storage Manager, and two equal LexKeys even in different XAT tables always identify the same XML node. This is a major gain from having only LexKeys in XAT tables and storing the nodes only once in the Storage Manager, as when a certain XML node is updated, the update is done only once.

Operator	***Cases***	***Propagate***	***Info Accessed***
$\phi^{col'}_{col,path}(R)$	$pos \lhd path$	$u(\triangle Q) \mid (\forall t' \in \triangle Q)(\forall cn \in RS_Q \mid cn \neq col')$ $(t'[cn] = t[cn]) \wedge (\forall n' \in \phi(path - pos : n))$ $(\exists! t' \in \triangle Q)(t'[col'] = n')$	t
	$pos \trianglerighteq path$	$u(\delta^{+}_{n,pos-path}k', col', tid') \mid$ $(tid' \in Q.getDerived(tid)) \wedge$ $(t' = Q.getTuple(tid')) \wedge (t'[col'] = k')$	tid index $t'[col']$
$\Phi^{col'}_{col,path}(R)$	$pos \lhd path$	$u(\triangle c', col', tid) \mid \triangle c' = \phi(path - pos : n)$	none
	$pos \trianglerighteq path$	$u(\delta^{+}_{n,pos-path}k', col', tid)$	none
When $pos \trianglerighteq path$, k' is name of first forward step in pos that is not in $pos - path$.			

Table 6: Propagation rules for $u(\delta^{+}_{n,pos}k, col, tid)$ for XAT XML operators

Operator	***Cases***	***Propagate***	***Info Accessed***
$\phi^{col'}_{col,path}(R)$	$pos \lhd path$	$u(\bigtriangledown Q) \mid (\forall tid' \in \bigtriangledown Q)(tid' \in Q.getDerived(tid)) \wedge$ $(\forall n' \in \phi(path - pos : n))(\exists! tid' \in \bigtriangledown Q)$ $(t'[col'] = n', t' = Q.getTuple(tid'))$	$t'[col']$ tid index
	$pos \trianglerighteq path$	$u(\delta^{-}_{n,pos-path}k', col', tid') \mid$ $(tid' \in Q.getDerived(tid)) \wedge$ $(t' = Q.getTuple(tid')) \wedge (t'[col'] = k')$	tid index $t'[col']$
$\Phi^{col'}_{col,path}(R)$	$pos \lhd path$	$u(\bigtriangledown c', col', tid) \mid \bigtriangledown c' = \phi(path - pos : n)$	none
	$pos \trianglerighteq path$	$u(\delta^{-}_{n,pos-path}k', col', tid)$	none
When $pos \trianglerighteq path$, k' is name of the first forward step in pos that is not in $pos - path$.			

Table 7: Propagation rules for $u(\delta^{-}_{n,pos}k, col, tid)$ for XAT XML operators

Operator	***Cases***	***Propagate***	***Info Accessed***
$\phi^{col'}_{col,path}(R)$	$pos \trianglerighteq path$	$u(\delta^{r}_{pos-path,new}k', col', tid') \mid$ $(tid' \in Q.getDerived(tid)) \wedge$ $(t' = Q.getTuple(tid')) \wedge (t'[col'] = k')$	tid index $t'[col']$
$\Phi^{col'}_{col,path}(R)$	$pos \trianglerighteq path$	$u(\delta^{r}_{pos-path,new}k', col', tid)$	none
k' is name of the first forward step in pos that is not in $pos - path$.			

Table 8: Propagation rules for $u(\delta^{r}_{pos,new}k, col, tid)$ for XAT XML operators

The propagation rules for *Navigate Collection* and *Navigate Unnest* on $u(\delta^{+}_{n,pos}k, col, tid)$, $u(\delta^{-}_{n,pos}k, col, tid)$ and $u(\delta^{c}_{pos,new}k, col, tid)$ are given in Tablew 6, 7 and 8 correspondingly.

How an intermediate XML update is propagated depends on the mutual containment of the position pos to which the update refers and the navigation path $path$ of the considered *Navigate Unnest* or *Navigate Collection* operator. The two cases of interest are $pos \trianglerighteq path$ and $pos \lhd path$.

The case $pos \trianglerighteq path$ arises when a node that has already been located by $path$ is being updated, that is, either its descendant is added or deleted or that node's value is changed. The last may only occur if the located node is an attribute or a text node. Therefore, when $pos \trianglerighteq path$, the transformed update is of the same type as the original update, only the position of the update is rewritten. For example, consider the update $u(\delta^{-}_{n,pos}k, col, tid)$ for the operator $\phi^{col'}_{col,path}(R)$. The LexKey k being updated must have previously been present in $t[col]$, where $t = R.getTuple(tid)$. Thus, if $pos \trianglerighteq path$, the node k' (as in Table 7) that is a ascendent of the node located by pos must have already been located by $path$ either during the initial view extent computation or during a prior update propagation and is present in a cell corresponding to column col' of a tuple t' derived from t. Thus an update specifying the deletion of the descendant n of k', located at position $pos - path$ is generated, as shown in Table 7.

The case $pos \triangleleft path$ can only occur when the intermediate XML update is an insertion or a deletion, but not for a replacement. This is due to the fact that a replacement may only be specified on leaf XML nodes, i.e., text nodes or attribute nodes. Leaf nodes do not have descendants and $pos \triangleleft path$ can only hold if pos locates a node that has one or more descendants. In this case, the inserted (deleted) node could also be located by $path$. Therefore, such an insert (delete) may cause insertion or deletion of LexKeys or even full tuples from the output XAT table.

Note that the remaining case when neither $pos \trianglerighteq path$ nor $pos \triangleleft path$ holds is not of interest. This last case arises when an update on an irrelevant position occurs. As a concrete example, consider the update $u(\delta^{+}_{b.n.f,price[b.n.f]}b.n, \$b, 2)$ for the operator $\Phi^{\$col3}_{\$b,/title}$. The navigation path specifies the extraction of children of type $title$ from the nodes in the column $\$b$. The update position specifies that a node element of type $price$ is inserted as a child of a node in $\$b$. Thus, this insertion is irrelevant for $\Phi^{\$col3}_{\$b,/title}$.

Additional Rules Common for all XAT XML Operators

Several propagation rules are common to all operators, and have thus not been repeated in the propagation tables. Most importantly, the operators are optimized to never propagate updates on cells corresponding to columns that are not in the Minimum Schema of their output. The reasons for this are the following. First, the operators should not propagate any updates on cells that are not in the Real Schema of their output, as such updates would always be irrelevant for the operators later in the algebra tree. Second, if the column on which the update is specified is in the Real Schema, but not in the Minimum Schema, then that column is only in the Order Schema of the output. As the columns that make up the Order Schema never contain collections, but only single LexKeys, only intermediate XML updates can be specified on such columns, but not intermediate XAT updates. As none of the operators later in the algebra tree have such column as their input column (otherwise they would have been in the Minimum Schema)), intermediate XML updates on columns that are only in the Order Schema of the output are irrelevant and thus are never propagated.

Also, each received $u(\blacktriangle c, col, tid)$, $u(\triangledown c, col, tid)$ and $u(\delta k, col, tid)$ is always propagated in addition to the transformed update, if the column col is in the Minimum Schema of that operator.

Updates on columns that are not in the input parameters of the operator do not trigger the operator to propagate a transformed update.

The propagation rules show that for columns containing a single LexKey rather than a collection of LexKeys, the LexKey cannot be modified, nor deleted, nor replaced. As the Order Schema always includes only columns in which single LexKeys are stored, the LexKeys that contribute to the order preserving process are never modified. Therefore, it is never needed to modify the overriding order of the LexKeys.

Auxiliary Information

Table 9 summarizes the auxiliary information that is required by the XAT XML operators for the purpose of the update propagation. The information in this table is derived from the requirements of accessing auxiliary information of the individual update propagation rules.

Operator	***Input Columns***	***Output Columns***	***tid Index***
$\phi_{col,path}^{col'}(R)$	all	col'	yes
$\Phi_{col,path}^{col'}(R)$	none	none	no
$T_{p}^{col'}(R)$	none	none	no
$C_{col}(R)$	$OrderSchema$	none	no
${}^{x}\cup_{col1,col2}^{col'}(R)$	$col1$, $col2$	none	no
${}^{x}\cap_{col1,col2}^{col'}(R)$	$col1$, $col2$	none	no
${}^{x}-_{col1,col2}^{col'}(R)$	$col1$, $col2$	none	no

Table 9: Auxiliary Information for XAT XML Operators

5.3.3 Exposing the Updated View

When a sequence of update operations reaches the root *Expose* operator of the algebra tree, a partial reordering is performed to determine the absolute positions of the updates. The reordering is done only for correctly placing the nodes that have been added (or whose order has been modified) among their siblings. Thus the overhead of preserving order is greatly minimized.

5.4 Propagation Example

Figure 16 shows the update propagation for our running example. The XAT tables shown in the figure are the needed auxiliary views have been materialized when the view extent was initially computed. Not all of the materialized auxiliary views are necessarily needed in this particular update propagation process. However, they may be needed when a different update is propagated, thus must be maintained by each update propagation process including this one.

While the update XQuery presented in Figure 2 is being applied to the input XML document *bib.xml* presented in Figure 2, the XML update $\delta^{+}_{b.n.f,book[b.n]/price[b.n.f]}b$ specifying the position from the root XML node to the updated element is produced and passed to the Source operator $S^{s6}_{\text{“}bib.xml\text{”}}$. $S^{s6}_{\text{“}bib.xml\text{”}}$

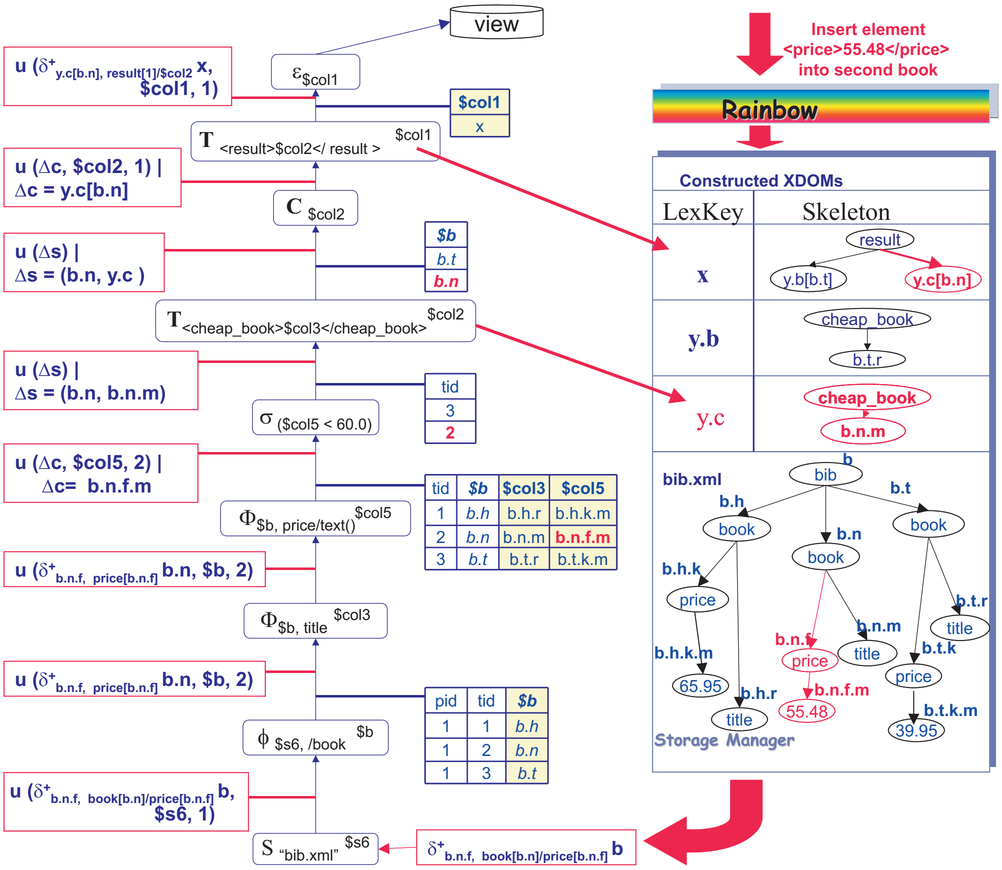

Figure 16: Update propagation for running example

transforms the incoming update into the intermediate update $u(\delta^{+}_{b.n.f,book[b.n]/price[b.n.f]}b, \$s6, 1)$ as described in Section 5.3.

$\phi^{\$b}_{\$s6,/book}$ compares the position $book[b.n]/price[b.n.f]$ of the update to its path (*/book*), and as $book[b.n]/price[b.n.f] \trianglerighteq /book$, it rewrites the position for the output update to $book[b.n]/price[b.n.f] - /book = price[b.n.f]$. It then generates a transformed update which is now over the output column $\$b$. In order to determine the tuple identifier for the transformed update, it accesses the *tid* index and the output column $\$b$ and by comparing the LexKey $b.n$ from the input update to the LexKeys in the column $\$b$, it figures out that the tuple identifier for the transformed update should be 2. Thereafter, the transformed update $u(\delta^{+}_{b.n.f,price[b.n.f]}b.n, \$b, 2)$ is propagated up. However, the original input update is not propagated, as the column $\$s6$ on which it was originally specified is not in the Minimum Schema (and not even in the Real Schema) of the output.

Next, the propagated update is received by $\Phi^{\$col3}_{\$b,/title}$. The navigation path specifies the extraction of children of type *title* from the nodes in the column $\$b$. The update position specifies that a node element of type *price* is inserted as a child of a node in $\$b$. Thus, this insertion is irrelevant for $\Phi^{\$col3}_{\$b,/title}$. However, this time the column $\$b$ on which the input update is specified is in the Minimum Schema of the output XAT table. Thus the input update is propagated.

When the update reaches $\Phi^{\$col5}_{\$b,/price/text()}$, which extracts the prices of the books, the position of the update $price[b.n.f]$ and the navigation path $/price/text()$ are compared. As $price[b.n.f] \lhd /price/text()$ the operator requests the children of $b.n.f$ that are of type $text$ from the Storage Manager and gets the LexKey $b.n.f.m$. The update is translated into an intermediate XAT update specifying insertion of the LexKey $b.n.f.m$ into the cell corresponding to tuple with tuple identifier 2 (equal to the tuple identifier of the input update) and column $\$col5$. The output XAT table of $\Phi^{\$col5}_{\$b,/price/text()}$ is materialized, thus the update is applied to that XAT table. The column $\$b$ over which the input update is specified is not in the Minimum Schema of the output. Thus it is not further propagated. The intuition is that $\$b$ is not an input column for any of the operators following $\Phi^{\$col5}_{\$b,/price/text()}$, thus such update is irrelevant for all subsequent operators.

The newly inserted price now makes the corresponding book pass the selection condition. The $\sigma_{(\$col5<60)}$ operator recognizes that by first noting that the column $\$col5$ on which the update is specified is in the selection condition, and then reevaluating the selection condition over the updated tuple. As the affected tuple now passed the selection condition, but its tuple identifier is not not among the tuple identifiers that were previously passing the selection condition, a tuple insertion is generated by $\sigma_{(\$col5<60)}$. The tuple identifier 2 is inserted in the output tid index, which is also materialized.

The new title is tagged by the $Tagger$ $T^{\$col2}_{<cheap\_book>\$col3</cheap\_book>}$ operator, and the constructed node is assigned the LexKey $y.c$, as y is the LexKey identifying this $Tagger$. The skeleton of the newly constructed node is passed to the Storage Manager and an intermediate XAT update $u(\Delta S)$, where $\Delta S = (b.n, y.c)$ specifying the insertion of the new tuple is output. The overriding order of this key is set by C_{col2} to reflect the order of the tuple from which has been derived. An update specifying the insertion of $y.c[b.n]$ into the cell corresponding to column $\$col2$ and the tuple identified by 1 is generated, as the only tuple in the output XAT table of the $Combine$ operator always has tuple identifier 1. When the final $Tagger$ operator receives this update, it recognizes that the input intermediate XAT update is on the column $\$col2$ that is present in its tagging pattern. Thus, the constructed XML node derived from the input tuple identified by $tid = 1$ needs to be correspondingly updated. It therefore generates the XML update $\delta^{+}_{y.c[b.n],result[1]/\$col2}x$ where the position from the root of the constructed node to the location of $\$col2$ is specified. This update is passed to the Storage Manager, which applies it to the node identified by the LexKey x. It is also propagated upwards in the form of an intermediate update, that is, $u(\delta^{+}_{y.c[b.n],result[1]/\$col2}x, \$col1, 1)$.

When this final update is passed to ϵ_{col1}, it refreshes the view as shown in Figure 17. The process of exposing the updated view is performed in a sequence of steps, as shown in Figure 17. First, the LexKey x representing the result is dereferenced, that is, the skeleton of the constructed node it identifies is retrieved (step 1 in the figure). Next, as this constructed node has been updated, a reordering is

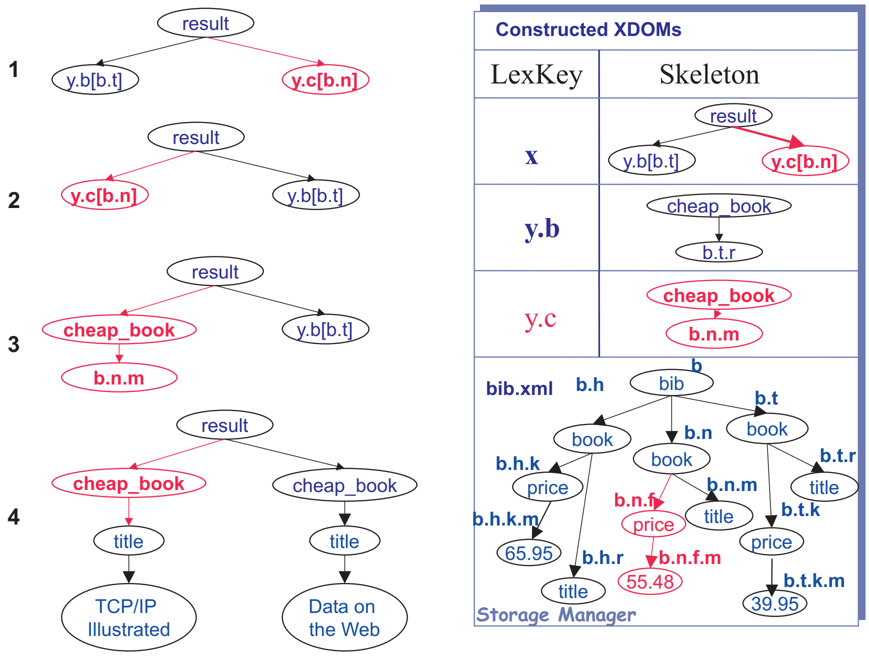

Figure 17: Example of exposing updated view

performed over the LexKeys $y.c[b.n]$ and $y.b[b.t]$. $order(y.c[b.n]) = b.n$, $order(y.b[b.t]) = b.t$ and $b.n \prec b.t$ imply that $y.c[b.n] \prec y.b[b.t]$. Thus the new constructed node identified by $y.c[b.n]$ is put before the one already in the view (step 2 in the figure). After the reordering is performed, the LexKeys $y.c$ and $y.b$ are dereferenced, as shown in steps 3 and 4 in the figure correspondingly and the updated view is obtained.

6 Correctness

The VOX approach is composed two key mechanisms, one, the order preservation mechanism and two, the update propagation strategy. Thus its correctness derives from the correctness of these two parts. The correctness of the order preservation has been proven in Section 4. Thus we now focus on the correctness of the propagation strategy.

Theorem 6.1 *Let $op_{in}^{out}(s)$ be any operator excluding the Source operator in an XAT algebra tree outputting the output Q. Let an intermediate update operation as defined in Table 3 be applied to one of its input XAT tables. Let Q^{req} be the output of op after recomputation. Let the propagation rules as defined in Section 5.3 generate a sequence of intermediate updates over Q that would transform Q into view Q^{inc}. Then $Q^{inc} == Q^{req}$ (Q^{inc} and Q^{req} are equal based on equality by node identity), $Q^{inc} = Q^{req}$ (Q^{inc} and Q^{req} are equal based on equality by value), and the sequence of generated intermediate updates carries the information about all the modifications of nodes referenced by the keys present in the columns of the Minimum Schema of Q, that are specified by the input update.*

The correctness of some propagation rules, i.e., that they yield $Q^{inc} == Q^{req}$ and produce a correct output sequence of update primitives, has been shown in Section 5.3 by deriving them from the respective propagation equations. The propagation equations are derived algebraically from the definitions of the operators, when the order among the tuples and the nodes in a collection is explicitly maintained and thus are correct. Following the same procedure, the remaining rules can also be shown correct. As the actual XML nodes are stored only once in the shared storage, and the same LexKey present even in different tables always references the same XML node, $Q^{inc} == Q^{req}$ always implies $Q^{inc} = Q^{req}$. The completeness of the output update sequence can also be derived from the respective propagation equations.

Corollary 6.1 *The propagation of any intermediate update defined on an input XAT table R through any operator $op_{in}^{out}(s)$ produces an update sequence containing exactly all and no other modifications caused over the output XAT table Q as a consequence of the update on R.*

This corollary states that not only the update sequence is complete, but it does not contain any updates of Q that are not consequence from the input update on R.

Theorem 6.2 *Let V be a view defined over input XML data sources in_1, in_2, ..., in_n. Let an XML update operation δk as defined in Table 2 be applied to one source in_i, $1 \leq i \leq n$. Let V^{req} be the view extent after recomputation. Let the VOX algorithm as defined in Section 5 transform the view V into view V^{inc}. Then $V^{inc} = V^{req}$.*

Proof: The proof is over the height h of the XAT algebra tree representing the view V, i.e., the maximum of the numbers of ancestors of any leaf node. To simplify the proof, we prove a generalization of the theorem that covers not only algebra trees that represent an XML view, that is, have an *Expose* operator as a root, but also algebra trees that have other operators as a root. Thus, the generalized statement is the same as that in the theorem, only the view V now refers to the output of the root operator of an XAT algebra tree, which may be an XAT table or XML data. The statement in this theorem is a corollary of such a generalized theorem.

Base Case: The base case is for $h = 0$. The algebra tree has a single operator node, which must be a *Source* operator $S_{in_i}^{col'}$, and whose output XAT table is the view of interest V. Before the update, V has a single tuple t consisting of a single cell $t[col']$, which contains the LexKey k of the root node of the XML document. After the update δk, if recomputation is performed, again $t[col']$ would contain k, as the update cannot modify the LexKey identifying the root node of the document. Any single XML update is propagated through a *Source* operator by simply rewriting it into an equivalent intermediate XML update (see Section 5.3). Thus no modification in terms of inserting or deleting LexKeys from the output V is specified. Hence, for $h = 0$ we have $V^{inc} == V^{req}$ and consequentially $V^{inc} = V^{req}$.

Induction Hypothesis: Let V be the output of the root node of an algebra tree with height l, $1 \leq l \leq h$, defined over the set of input XML data sources in_1, in_2,..., in_n. Let an XML update operation δk as defined in Table 2 be applied to one source in_i, $1 \leq i \leq n$. Let V^{req} be the view extent after recomputation. Let the VOX algorithm as defined in Section 5 transform the view V into view V^{inc}. Then $V^{inc} = V^{req}$.

Induction Step: It is to show that $V^{inc} = V^{req}$ for the output V of an XAT algebra tree of height $h+1$.

Let op be the operator at the root node of such algebra tree. op can be any operator, excluding $Source$, as $Source$ can only appear as a leaf node. All the children nodes of the root must themselves be roots of algebra trees each of height not exceeding h. Let k of these algebra trees have the updated source as a leaf. Noting their output as R_i, $1 \leq i \leq k$, by the induction hypothesis $R_i^{inc} = R_i^{req}$, $1 \leq i \leq k$. The sequences of updates on all R_i will propagate a sequence of intermediate updates to the node representing op. Let the number of updates in that sequence be m. That sequence of updates will cause op to transform its output V into a sequence of m intermediate (temporary) XAT tables V_1^{inc}, V_2^{inc}, ..., V_m^{inc}, each of which is equivalent to the corresponding state V_1^{req}, V_2^{req}, ..., V_m^{req} that would be reached by recomputing the output of op after each update in the sequence. Note that $V^{req} = V_m^{req}$. After the application of all m updates to V we have $V^{inc} = V_m^{inc} = V_m^{req}$. If any update gets propagated correctly (valid by Corollary 6.1), the sequence of updates propagated to op in particular must also be correctly propagated, and thus $V^{inc} = V^{req}$. $\square$

7 Implementation

The VOX system (see Figure 18) has been implemented in Java on top of the XQuery engine Rainbow [30, 29] also developed at WPI.

The Storage Manager is a repository for storing the XML data over which the view is defined, the constructed nodes and the auxiliary views. It assigns the LexKeys and supports efficient value-based and reference-based access to the data it stores. Two different implementations of the Storage Manager have been carried out: (1) Main-memory-based, and (2) Storage Manager relying on the native XML storage system called MASS [6] also developed at WPI.

The initial main-memory-based Storage Manager has been implemented as part of this thesis for the purpose of evaluating VOX. This main-memory Storage Manager accurately provides the basic functionality expected from a secondary-storage-based Storage Manager, as it supports efficient access to any XML node and its direct children given the LexKey identifier of that node. That is achieved by storing both the base data as well as the constructed nodes in a tree-structure, and also maintaining hashtables having LexKeys as keys and the XML nodes as values.

The Storage Manager relying on the native XML storage system called MASS [6] is being implemented as a separate project by an undergraduate Major Qualifying Project team. The MASS system is a storage

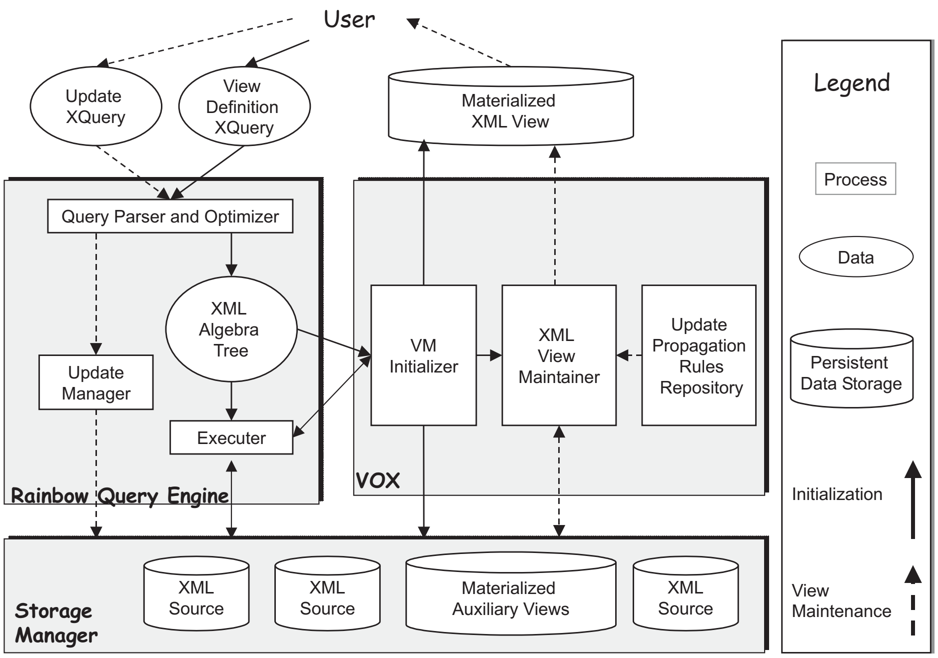

Figure 18: System architecture

and indexing solution for large XML documents based also on the XML node identity encoding using LexKeys.

Due to time constraints (the implementation of the MASS-based Storage Manager was not completed at the time this thesis was completed) the evaluation presented in this document is based on the main-memory-based Storage Manager.

As shown in Figure 18, the tasks performed by the VOX system can be divided into two categories: tasks that are carried out once per each XML view at initialization time, and tasks that are carried out for maintaining the view when source updates occur.

Initially, Rainbow's Query Engine translates the XQuery view definition into an XML algebra tree and optimizes it [31]. The optimization is done by applying rewrite rules, which swap and combine the operators, while still preserving the validity of the algebra tree. Then, the Minimum Schema for all the XAT tables is determined [31].

Next, the Order Schema is computed for each intermediate XAT table (see Section 4.4.1). The VM Initializer determines the needed auxiliary views by doing a postorder traversal of the algebra tree (see Section 5.2). Each column in an XAT table or *tid* index is materialized if and only if its materialization is required by at least one of the operators having that particular column of that XAT table or that *tid* index as input or output.

The initial view extent is evaluated by the Executer. Prior to this project, Rainbow's Executer supported only value-based execution. That is, it did not account for node identity. Thus, implementing reference-based execution was inevitable for the purpose of evaluating the VOX approach. However, we note that reference-based execution relying on the order encoding technique presented in this thesis is not only suitable for view maintenance, but also preferable compared to value-based execution for several reasons. First, it creates smaller intermediate results. Second, it allows for the evaluation of queries over large data by sharing references over base data instead of copying that data repeatedly. Finally, the concept of Order Schema allows for different physical operators. For example, without explicit order encoding the *Theta Join* operator was implemented as a nested-loop join, as only such implementation outputs the result in the order specified by the semantics of the *Theta Join* operator. Now, however, different algorithms for the implementation of the *Theta Join* operator are feasible, like Hash Join [20] or Sort-Merge Join [20], for example.

While evaluating the initial view extent, the Executer communicates with the VM Initializer which takes the intermediate results from the Executer and materializes the content of the needed auxiliary views.

When an update XQuery is issued by the user, it gets processed by the Rainbow's Update Manager, applied to the sources, and in the form of XML updates passed to the XML View Maintainer. The later, using the update propagation rules, incrementally refreshes the XML view, as described in this document (see Section 5). The update propagation rules are implemented as methods, one for each type of operator and each type of intermediate update. These methods implement the logic of the update propagation rules and create the output update sequence. In addition, they are responsible for invoking the module that applies the updates to the materialized auxiliary views. That way, each rule can access the state of the auxiliary view both before and after the update if needed. Upon invoking an operator, the sequences of updates output by its children, that are now input for this operator are traversed in an arbitrary order and the corresponding method is invoked for each update in these sequences. Only when all the input updates for a certain operator have been processed and the complete output update sequence has been produced, the next operator is invoked.

Once the update propagation process finished, the LexKey(s) identifying the result are dereferenced. The partial reordering currently relies on the merge-sort algorithm.

8 Evaluation

8.1 Experimental Set-up

We have performed a performance evaluation of VOX on a Pentium III PC with 512MB of RAM running Windows 2000. For the experiments we use data and queries provided by the XMark benchmark [21]. The relationship among the elements of interest is shown in Figure 19. The queries we use extract data from "person" elements. Hence we show the number of such elements in the charts shown in this section.

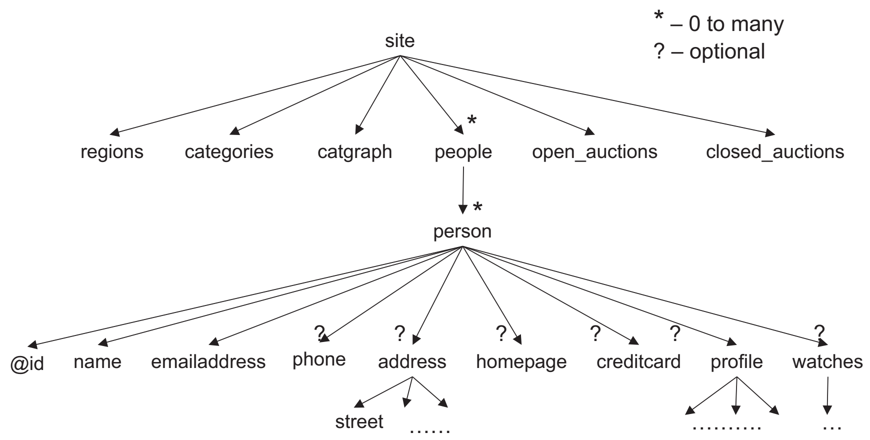

Figure 19: Relationship between queried elements

In the experiments we vary the size of the data source, the type of the update, the size of the update relative to the original size of the base data, the selectivity of the view definition query and the location of the update. The performance of incremental view maintenance is compared to the performance of full recomputation. We also compare the costs of the different update operations.

```
<Result>
    FOR    $b IN document("auction.xml")/people/person
    WHERE  $b/@id > 317
    RETURN $b/name
</Result>
```

Figure 20: Example view definition

In most of the experiments we use the query shown in Figure 20. This query is rather simple, but it is suitable for varying a single parameter at a time while guaranteeing that the other parameters stay constant. Thus it allows us to isolate effects of individual experimental variables. Also, it does contain the typical XML operations, including in particular XML structure navigation (the operators *Navigate Unnest* and *Navigate Collection*) and node construction (the operator *Tagger*). The XAT algebra tree for this query is given in Figure 21.

8.2 Cost of Different Update Operations

Figure 22 shows that incremental maintenance significantly outperforms recomputation for all three types of updates. In this experiment, each type of the update targets a single "person" element, that is, one "person" element with @*id* less than 317 is inserted or deleted or the "id" of one person element is replaced from a value greater than 317 to value less than 317, thus making it to pass the selection condition. The cost of recomputation shown in Figure 22 is averaged over 5 runs for each type of update. Clearly, the cost of recomputation for the different types of updates in this experiment differs very slightly, as the

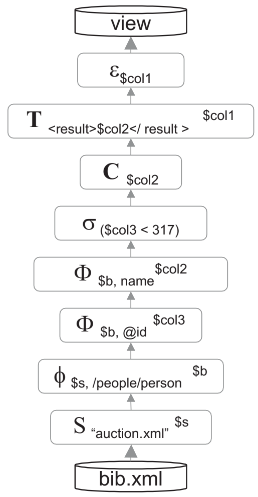

Figure 21: The XAT algebra tree for the view in Figure 20

update is small compared to the size of the base data, which originally consists of 637 "person" elements.

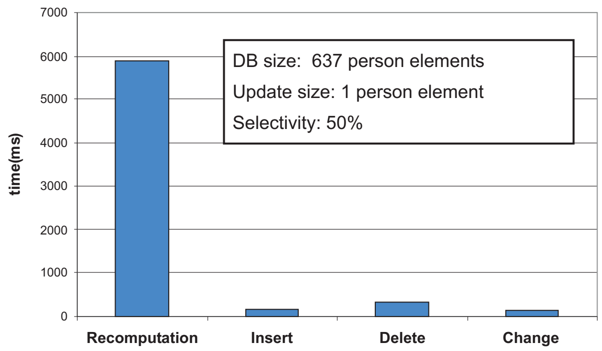

Figure 22: Cost of different update operations

Figure 22 also shows that that different update operations have different costs. In particular, deletions are more expensive to propagate than insertions or replacement. The cost of propagating replacement and insertion is similar. As a result of a deletion and replacement often intermediate updates that require locating the tuple that has to be updated are produced. Such an overhead is less present when an insertion is propagated. Replacement on the other hand usually leads to less updates on the materialized auxiliary views compared to insertion and deletion. In this example, it leads to the same effect on the view as the insert, but still to less auxiliary view updates than the insert. Therefore, the deletion is most expensive to propagate, and replacement and insertion have a similar cost.

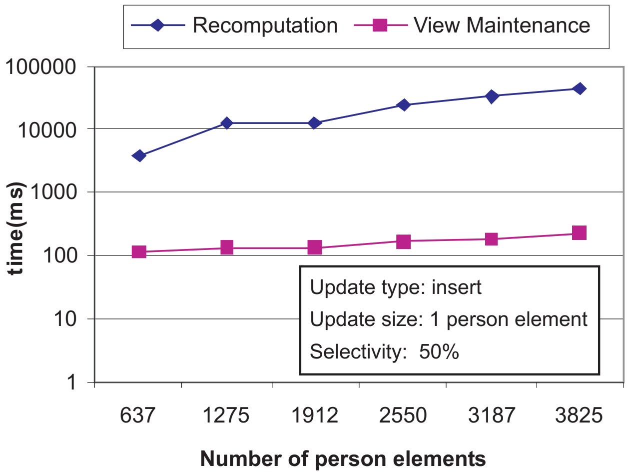

Figure 23: Varying database size

8.3 Varying Database Size

Figure 23 compares the performance of our solution to recomputation when a new "person" element is inserted for different base XML data sizes. The cost is presented on a logarithmic scale. The chart shows that the cost of recomputation follows the growth of the data size. While the cost of incremental maintenance also increases, it does so at a lower rate. For example, the cost of recomputation when there are 3825 "person" elements in the base data is 3 times greater than when there are 1275 "person" elements. That is expected, as the size of the data is 3 times larger in the first case. However, the cost of incremental maintenance when there are 3825 "persons" elements is approximately 1.5 times larger than when there are 1275 such elements. The reason the cost of incremental maintenance increases even though always the insertion of a single "person" element is propagated, is because the size of the auxiliary views is larger for larger base data, and thus locating the position of the update is more expensive.

8.4 Varying Update Size

Figures 24, 25, 26 and 27 show that incremental maintenance is much faster even for large updates. In this experiment we vary the size of the inserted (deleted) element. We insert or delete a "people" element that contains a different number of "person" elements. Figure 24 shows the results for variable size insertions on a logarithmic scale. The results from the same experiment are shown in Figure 25 on a linear scale for a clearer view of when the lines showing the cost of view maintenance and recomputation cross. For the same reason, the experiments for variable size deletions are shown in two charts (Figures 26 and 27). For the experiment of varying the size of the inserted element, the original size of the base data is constant. However, for the experiments of varying the size of the delete, we start from different initial sizes, and after the delete is performed, we get the same size of the updated base data for different sizes of the delete. Therefore, the recomputation line is constant.

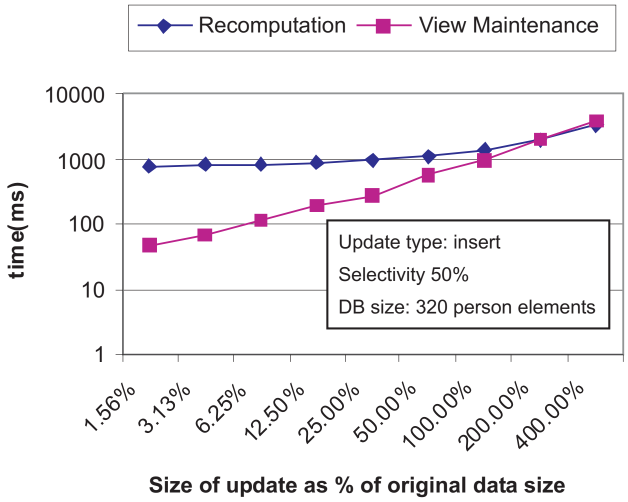

Figure 24: Varying size of insert (logarithmic scale)

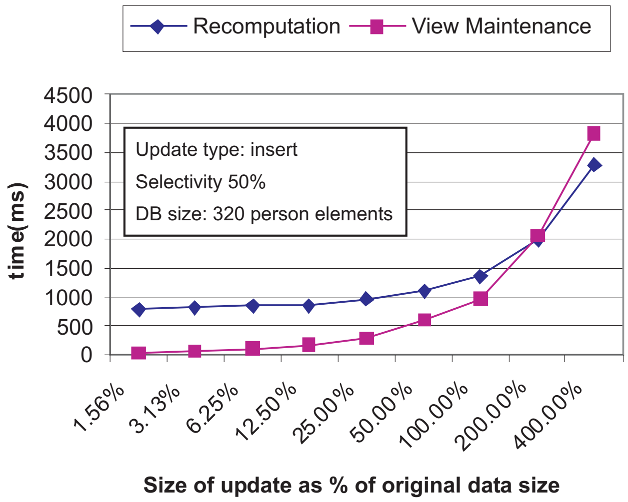

Figure 25: Varying size of insert (linear scale)

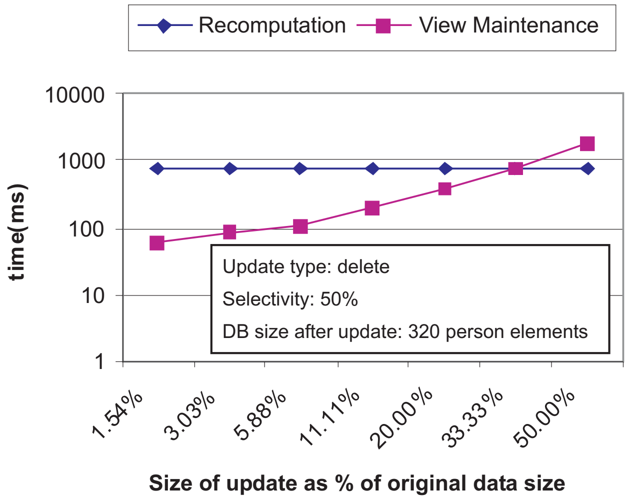

Figure 26: Varying size of delete (logarithmic scale)

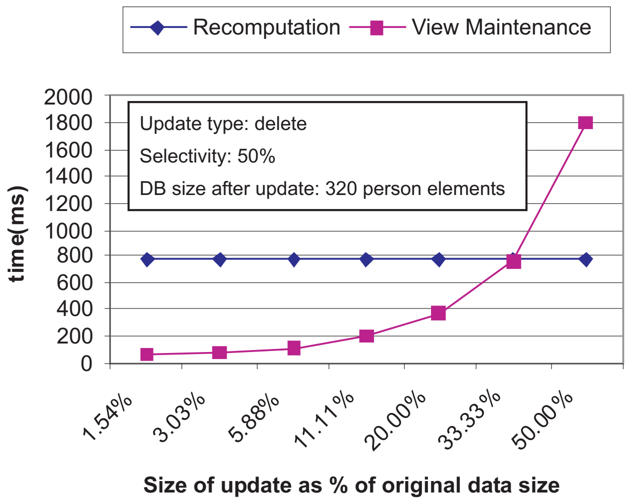

Figure 27: Varying size of delete (linear scale)

In Figures 24 and 25 the number of inserted "person" elements ranges from 1% to being 400% of the number of "person" nodes already in the database. As the size of the inserted nodes increases, the cost of view maintenance approaches the cost of recomputation. In particular, when the size of the update is 200% of the original size of the data, view maintenance becomes more expensive than recomputation. This means that for this setting, the cost of incremental view maintenance per "person" element is 1.5 times higher than the cost of recomputation, as recompuation evaluates base data with size 3 times larger the original base data, and view maintenance is performed for an update that is twice the original base data.

Figures 26 and 27 show that view maintenance outperforms recomputation for deletions of size up to 33% of the original data. This shows that propagating deletions is more expensive than propagating insertions. The reason is that propagating deletions often requires accessing additional information, as deletions trigger more intermediate updates for which the tuples (LexKeys) that have to be updated have to be located. In particular, for this setting the cost of view maintenance per "person" element is 2 times higher then the cost of recomputation per "person" element, as the size the deleted element is 33% of the original database size and the size of the date over which the recomputation is performed is 67% of the original database size.

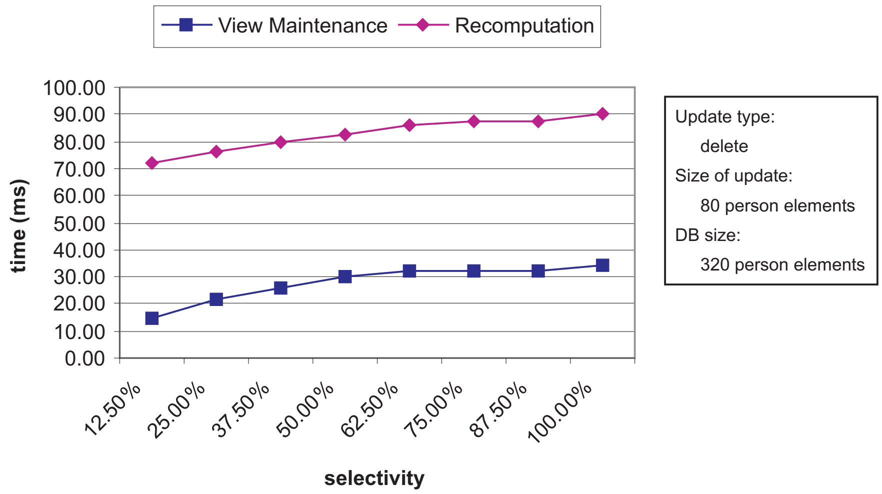

Figure 28: Varying selectivity

8.5 Varying Selectivity

Figure 28 shows that view maintenance performs well regardless of the selectivity of the view query. For this experiment the size of the delete is 20% of the size of the base data. In this experiment the selectivity of 50% is not only over the base data, but also over the deleted elements. This means that only half of the deleted "person" elements pass the selection condition and were initially present in the view. When the selectivity of the view query grows both the times for performing recomputation and view maintenance increase with a similar rate.

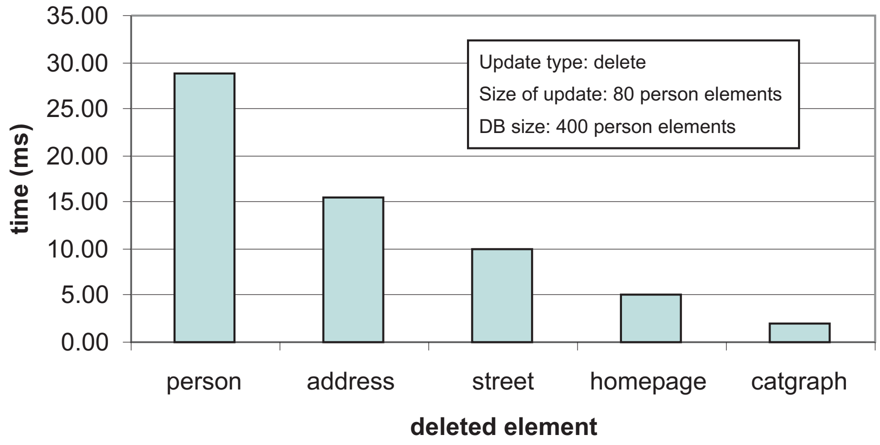

Figure 29: Varying location of update

8.6 Varying Location of Update

Figure 29 compares the cost of view maintenance for different locations of the update. The view query used for this experiment is similar to that shown in Figure 20, however it only extracts addresses rather than names. Thus the deletion of both "homepage" and "catgraph" does not affect the result. The experiments confirm the cost of propagating such irrelevant updates is very small. The deletion of "homepage" is slightly more expensive to propagate than the deletion of "catgraph". This is due to the fact that "homepage" is child of "person", whereas "catgraph" is a sibling of "person", and thus the irrelevance of the deletion of "catgraph" is detected faster than the irrelevance of the deletion of "homepage".

The deletion of the element "street" affects the content of the view, as it is a descendant of a node present in the view. However, it does not affect any of the intermediate results. Therefore such update is cheaper to propagate than deletion of elements that are "exposed" by the query, such as "person" and "address".

8.7 Overhead of Maintaining Order

We have measured the overhead of maintaining order by also running the system in a non-order-sensitive mode, that is, without performing partial reordering upon refreshing the view. The cost of maintaining order was not noticeable compared to the overall cost of incremental view maintenance. Intuitively, that is expected, as the partial sorting is only done on the LexKeys when exposing the updated view.

9 Conclusion

9.1 Summary and Conclusions

We have presented VOX, the first solution for order-preserving maintenance of XQuery views. Our solution takes an algebraic approach, and shows how by using an lexicographical order encoding technique, the XML algebra can be transformed from ordered bag to (non-ordered) bag semantics, thus enabling efficient view maintenance. Our solution handles complex updates, that is insertions and deletions of complete subtrees as well as replacement of an atomic value. The techniques proposed here are general and are not not bound only to our particular system.

Our experiments have confirmed that VOX outperforms recomputation in most cases. When the size of the update is relatively small compared to the size of the base data, which is a common case in the reality, VOX is significantly faster than recomputation for all three types of updates. Our solution has also proven to be feasible for large updates. In particular, incremental view maintenance is faster then recomptuation for inserts of size up to 200% of the base data and deletes for up to 33% of the base data. This means that deletions are slightly more expensive to propagate then insertions. Our solutions performs well regardless of the database size and the selectivity of the view query. Also, it is able to efficiently detect irrelevant updates, thus the cost of propagating such updates is very low. Most importantly, the overhead of maintaining order has shown to be unnoticeable.

9.2 Contributions

Contributions of this work include:

- We have identified and analyzed new challenges imposed on incremental view maintenance by the ordered hierarchical nature of the XML data model.
- We have proposed an order-encoding mechanism that migrates the XML algebra from ordered bag semantics to (non-ordered) bag semantics, thus making most of the operators distributive with respect to the bag union and bag set difference.
- Using LexKeys as references to XML nodes and for explicit order-encoding not only enables efficient view maintenance, but also leads to more scalable XQuery execution compared to naive value-based execution. First, it creates smaller intermediate results. Second, it allows for the evaluation of queries over large data by sharing references over base data instead of copying that data repeatedly. Finally, the concept of Order Schema allows for different physical operators to be incorporated into a physical plan, without concern of the order preservation.
- We have given the first order-sensitive algebra-based solution for incremental view maintenance of XML views defined with the XQuery language. We have proposed an overall update propagation

strategy and have developed a full set of rules for propagating updates through XML specific operations.

- We have proven the correctness of the approach.
- We have successfully implemented our proposed solution in the XML data management system Rainbow.
- We have experimentally evaluated our approach. In the experiments the cost of view maintenance is compared to the cost of recomputation, and in most cases view maintenance is cheaper than recomputation.

9.3 Future Work

We believe that this work forms a solid basis for enabling view maintenance for XML views. The work carried out in this project can be extended in two directions: optimizing the incremental view maintenance for materialized XQuery views and optimizing the reference-based XQuery execution.

Regarding the incremental view maintenance, VOX can be optimized to perform batching of concurrent updates coming from possibly different sources and by considering algebra tree rewrites, as we now explain below.

The current approach can further be optimized to take into account the interrelationships between concurrent updates derived from base updates on different data sources. In particular, currently the system takes one XML update at a time, and assumes that the sources remain in the same state while that update is being propagated. Thus, while a single XML update is propagated, another update cannot be propagated. However, it is expected that propagating simultaneously concurrent updates would lead to cheaper propagation, especially in a large scale distributed environment. The updates may cancel each other out or may share reads of auxiliary data.

Also, VOX does not address the issue of reshaping the algebra tree for optimizing the update propagation process. The algebra tree is generated by the Rainbow system, which optimizes it for one-time query execution [31]. However, such shape of the tree may not be optimal for performing the view maintenance. By rewriting the algebra tree it may be possible for example to decrease the auxiliary information requirements and reduce the computation costs for update propagation. The goal would be to explore algebra tree rewrite algorithms that lead to decreased intermediate result materialization and/or to faster update propagation. As enumerating all the alternative query plans is prohibitively expensive [20], heuristics would be applied for narrowing the set of considered alternatives. An optimal query plan would then be chosen based on cost estimations.

The reference-based XQuery execution with explicit order encoding allows for flexibility in choosing different physical implementations of the algebra operators. Thus, different strategies for evaluating individual operators can be developed. Also, currently Rainbow does not support all the access axes specified by the World Wide Web Consortium [24]. However, the MASS system [6], on the top of which an implementation of our Storage Manager is build, supports efficient access to all access axs. This functionality can thus be exploited by the Rainbow system to expand the set of covered XQueries.

References

[1] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental maintenance for materialized views over semistructured data. In *VLDB*, pages 38–49, 1998.

[2] M. A. Ali, A. A. A. Fernandes, and N. W. Paton. Incremental maintenance of materialized OQL views. In *International Workshop on Data Warehousing and OLAP*, pages 41–48, 2000.

[3] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD*, pages 61–71, 1986.

[4] M. J. Carey, J. Kiernan, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for publishing object-relational data as XML documents. In *The VLDB Journal*, pages 646–648, 2000.

[5] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *VLDB*, pages 557–589, 1991.

[6] K. Deschler and E. A. Rundensteiner. XML navigation indexing. Technical report, Worcester Polytechnic Institute, 2003. In progress.

[7] M. El-Sayed, L. Wang, L. Ding, and E. A. Rundensteiner. An algebraic approach for incremental maintenance of materialized XQuery views. In *WIDM*, pages 88–91, 2002.

[8] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD*, pages 328–339, 1995.

[9] A. Gupta and I. S. Mumick. Maintenance of materialized views: problems, techniques, and applications. In *Bulletin of the Tech. Com. on Data Eng., 18(2)*, pages 3–18, 1995.

[10] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, 1993.

[11] H. A. Kuno and E. A. Rundensteiner. Using object-oriented principles to optimize update propagation to materialized views. In *ICDE*, pages 310–317, 1996.

[12] H. A. Kuno and E. A. Rundensteiner. Incremental maintenance of materialized object-oriented views in MultiView: strategies and performance evaluation. In *IEEE Transaction on Data and Knowledge Engineering*, volume 10(5), pages 768–792, 1998.

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

[14] J. Liu, M. W. Vincent, and M. K. Mohania. Maintaining views in object-relational databases. In *Int. Conference on Information and Knowledge Management*, pages 102–109, 2000.

[15] M. Fernandez and A. Morishima and D. Suciu and W. Tan. Publishing relational data in XML: the SilkRoute approach. *IEEE Transactions on Computers*, 44(4):1–9, 2001.

[16] M. K. Mohania and Y. Kambayashi. Making aggregate views self-maintainable. *Data Knowledge Engineering*, 32(1):87–109, 2000.

[17] T. Palpanus, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental maintenance for non-distributive aggregate functions. In *VLDB*, pages 802–813, 2002.

[18] L. P. Quan, L. Chen, and E. A. Rundensteiner. Argos: Efficient refresh in an XQL-based web caching system. In *WebDB*, pages 23–28, 2000.

[19] D. Quass. Maintenance expressions for views with aggregation. In *SIGMOD*, pages 110–118, 1996.

[20] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2000.

[21] A. R. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *VLDB*, pages 974–985, 2002.

[22] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD*, pages 413–424, 2001.

[23] 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*, pages 204–215, 2002.

[24] W3C. XML Path Language (XPath). http://www.w3.org/TR/xpath.

[25] W3C. XML Query Data Model. http://www.w3.org/TR/query-datamodel.

[26] W3C. XML^{TM} . http://www.w3.org/XML.

[27] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/.

[28] W3C. XQuery 1.0 Formal Semantics. http://www.w3.org/TR/query-semantics/.

[29] X. Zhang, K. Dimitrova, L. Wang, M. El-Sayed, B. Murphy, B. Pielech, M. Mulchandani, L. Ding, and E. A. Rundensteiner. Rainbow: Mapping-Driven XQuery Processing System. In *SIGMOD Demonstration*, 2003. To appear.

[30] X. Zhang, M. Mulchandani, S. Christ, B. Murphy, and E. A. Rundensteiner. Rainbow: Mapping-Driven XQuery Processing System. In *SIGMOD Demonstration*, page 614, 2002.

[31] X. Zhang, B. Pielech, and E. A. Rundensteiner. Honey, I shrunk the XQuery! — An XML algebra optimization approach. In *WIDM*, pages 15–22, 2002.

[32] X. Zhang and E. Rundensteiner. XAT: XML Algebra for Rainbow system. Technical Report WPI-CS-TR-02-24, Computer Science Department, WPI, 2002.

[33] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, pages 316–327, 1995.

[34] Y. Zhuge and H. G. Molina. Graph structured views and their incremental maintenance. In *14th Int. Conf. on Data Eng.*, pages 116–125, 1998.