Incremental Fusion of XML Fragments through Semantic Identifiers

Maged El-Sayed, Elke A. Rundensteiner, and Murali Mani
Department of Computer Science
Worcester Polytechnic Institute Worcester, MA 01609
(maged | rundenst | mmani)@cs.wpi.edu

Abstract

Many applications, like materialized view maintenance and stream query processing, construct views incrementally over data sources. This results in computed pieces of objects that need to be merged by fusing corresponding objects together. This problem is challenging when dealing with XML data for many reasons including the hierarchical and semi-structured nature of XML data. Also XML query languages (e.g., XQuery) are capable of performing complex operations and transformations such as arbitrary nesting and result reconstruction. Moreover, since XML is an ordered data model, XML order has to be taken into consideration when constructing XML results incrementally. In this paper we study the problem of how to fuse XML pieces (fragments) generated by incrementally processing XML data into XML results. We consider an expressive subset of XQuery language transformations and propose an id-based solution for this problem that supports XML order. We prove the correctness of our approach, in particular that using our mechanism we can correctly yet incrementally merge XML result fragments. We have implemented our proposed semantic identifiers solution. Our experimental results show that it comes with a very small overhead to the query execution time.

1 Introduction

Object fusion is a core operation in information integration where mediators collect and integrate data objects from different sources [14]. Such integration needs to support the process of merging the corresponding objects resulting from processing the source data objects into the result. In some applications the data to be processed may not arrive all at once. One example is materialized view maintenance applications where source updates are propagated to materialized views in an immediate or even a deferred mode to maintain them incrementally. Another example is stream query processing where data arrives as streaming units at different times. In such cases we may have an initial materialized view extent (a partial result) as well as newly computed pieces of data that result over time from processing source updates (or stream units). These newly computed pieces of data need to be correctly merged (fused) with the initial result. This merging issue is relatively easy when considering relational views because of their flat nature and known schema. While for XML data this problem is more challenging due to many factors including the hierarchical nature of the data, the possibility of no known schema for the data, and the powerful capabilities of the XML query languages. XQuery views for example can restructure the XML view to take on a structure and hierarchical organization that is completely different from that of the base data, possibly turning children nodes into ancestors or generating multiple copies of the same node. Order is another factor that adds to the complexity of object fusion in XML views. Unlike other data models, XML is an ordered data model. Moreover, XQuery expressions return by default results that have a well-defined order based on document order unless otherwise defined. The result of an XQuery path expression is always returned in document order and the order in the result of a FLWOR expression can in addition be imposed by the expression itself in many ways, including the use of *order by* clauses, the nesting of for clauses, and the order defined by the $return$ clauses. See [7] for more details.

Motivating Example. We use a materialized view maintenance example to motivate the problem. Consider the two XML document shown in Figure 1. The source document "bib.xml" stores book information and the source document "prices.xml" stores prices of books. Consider the simple XQuery view in Figure 2(a) defined over those two sources that creates a new XML document with a root node "result" and a totally new structure. The result of executing this XQuery expression over the source documents is shown in Figure 2(b)[1]. Now consider that the source document "bib.xml" is updated by appending the new book element shown in Figure 3(a) and (b) to the end of that XML document (reflecting a desired document order). A view maintenance solution would need to propagate such an update into one update that is applicable to the materialized view in Figure 2(b). The propagated update, shown in Figure 3(c), needs to be applied correctly to the materialized view to refresh it. Correctly applying the update to the materialized view means that the refreshed materialized view should be equivalent to the materialized view we would obtain if we were to recompute the query over the updated sources. This

[1] Highlighted nodes represent newly constructed nodes.

includes maintaining the correct view order.

The question that we raise now is how to merge (fuse) the propagated update in Figure 3(c) with the original materialized view shown in Figure 2(b). This involves deciding for each incrementally propagated node if it should be merged with any existing node (or even nodes) in the view extent into possibly one combined node, or if it should be added as a new node, separate from existing ones, to the view extent. It also involves deciding how the order of the materialized XML view is maintained as a result of such an update to the view extent.

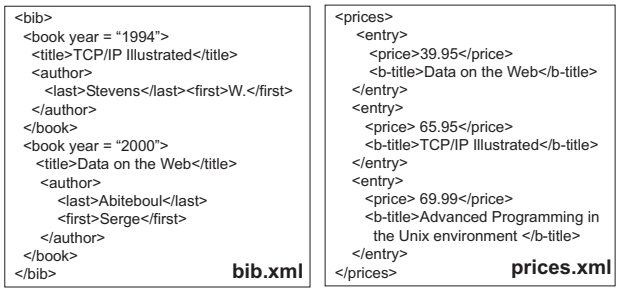

Figure 1. Two input XML documents "bib.xml" and "prices.xml".

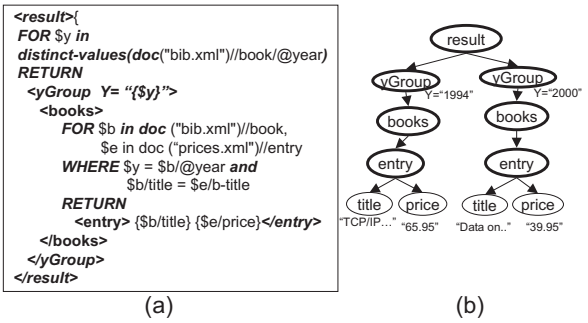

Figure 2. (A) An XQuery expression defined over the two XML documents in Figure 1 and (b) the resulting XML view extent.

In previous view maintenance solutions, this problem has been addressed in a variety of ways. Some solutions [1, 2, 22] have materialized large auxiliary data beyond the actual view contents. Other solutions [11, 14, 17] have used Skolem functions (or variations of them). We will discuss these solutions in more details in Section 2.

In this paper we study the problem of object fusion for XML views that are constructed incrementally. We consider views defined using the XML query language, XQuery [19], over XML data sources. These views are capable of performing a large class of complex operations and transformations including navigation, grouping, aggregation, nesting, unnesting, and element construction. We propose a

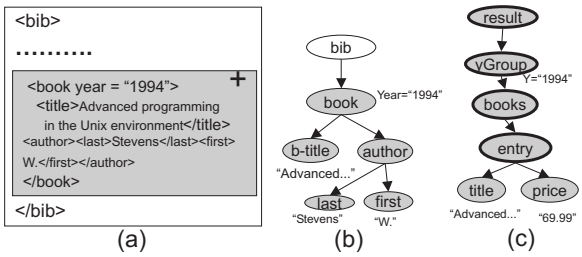

Figure 3. (a) A new "book" element to be inserted into the source document "bib.xml" shown in Figure 1, (b) the corresponding XML tree, and (c) the expected result of propagating the update through the view in Figure 2(a).

mechanism for generating semantically meaningful identifiers for processed XML nodes. These semantic identifiers are reproducible for corresponding objects[2], hence enable identifier-based fusion. They also encode lineage and order information for view nodes in a compact way.

Our solution works at the algebraic query representation level. In the first phase it takes the query algebra tree as input and automatically defines rules of how lineage and order specifications can be computed for processed nodes. We call such lineage and order specification the *Context Schema*. This phase takes place during the query plan generation and optimization phase. In the second phase, semantic identifiers are generated for processed nodes based on the *Context Schema* previously defined in the first phase. This step takes place during query execution time. We summarize the contributions of this paper as follows: (I) we propose a mechanism for generating "semantically meaningful" node identifiers for XML views. Such identifiers can be used to incrementally fuse XML fragments to construct results. To the best of our knowledge our solution is the first solution that provides all the following advantages: (i) it supports an expressive class of XQuery views (ii) it is fully automated, it does not, for example require the definition of Skolem functions on the query syntax level by the end user (iii) the semantic identifiers that we generate compactly encode lineage information for XML nodes in the result, hence enabling tracing back to the source nodes, (iv) The semantic identifiers that we generate also encode order information of nodes, hence enabling order-aware incremental result construction. (II) We define a mechanism for the id-based fusion for XML fragments using the *Deep Union* operation. (III) We prove the correctness of our approach. In particular we prove that the result of merging the incrementally processed data is equivalent to the result we

[2]This means that the semantic id generated for a newly processed node (resulting for example from a source update) is guaranteed to be equivalent to the id of an existing node in the result if there exists such a node that semantically corresponds to the newly processed node.

would obtain if we were to execute the query over the entire source. (IV) We have implemented and integrated our proposed solution within the Rainbow XML query engine [20] and have tested its performance. The results of our experiments show that our solution comes with very small overhead to the query execution time.

The rest of this paper is organized as follows. In Section 2 we discuss related work. Section 3 gives the necessary background. Section 4 describes how we encode derivation and order specifications through the *Context Schema*. Section 5 discusses how we generate the semantic identifiers from the *Context Schema*. Section 6 shows how we use the semantic identifiers for fusing processed XML nodes. Section 7 gives the results of our experimental evaluation. Lastly, Section 8 provides conclusions.

2 Related Work

A core operation to materialized view maintenance is the apply phase, namely the application of propagated updates to materialized views. This involves determining how to correctly merge (fuse) propagated updates with the materialized views. This problem is more challenging in the context of object-oriented and semi-structural data models than in the context of the flat relational data model. Many view maintenance solutions (e.g., [1], [2], and [22]) require materializing and maintaining large auxiliary data in order to be able to perform this task. For example, for maintaining the simple graph structured views (Select-Where views) used in [22] each view object needs to be annotated with identifiers of all the source objects from which the object is derived from. Unlike the approaches above we do not require the use of auxiliary data to relate view extent objects to their sources. Some view maintenance solutions (e.g., [4] and [11]) have avoided such need to materialize auxiliary data through the use of mechanisms for generating reproducible identifiers for the view objects. For example, the work proposed in [11] for maintaining semi-structured views annotates edges in processed trees with special keys that can be used in the fusion process. The proposed key system may generate keys with a deeply nested structure. It also comes with some limitations to the view maintenance solution itself including limitations on updating source values used in constructing the keys. Other solutions (e.g., [4] and [14]) have used Skolem functions to generate identifiers that can be used for fusing propagated updates with materialized views. Skolem functions were first used in the context of object-oriented systems [12] to produce object identifiers and were used later in many integration and mediation systems [14, 15]. The use of Skolem functions typically requires specifying these functions at the query syntax level by indicating what input is to be used by them to generate the identifiers. Papakonstantinou et al. [14] have proposed a technique for generating semantic object identifiers based on a special use of Skolem functions to fuse semi-structured data specified using the MSL mediator specification language. This work [14] supports only simple views and requires semantic identifiers to be defined as part of the mediator specification process. To the best of our knowledge no Skolem function solution supports incremental fusion of the class of XML views that we consider including order-aware views. For example, no Skolem function solution supports the unique identification and order semantics of views that allow multiple copies of the same source node (or constructed nodes bound to the same source nodes) to appear as siblings in the result. This is important for incremental view maintenance since certain updates to the source node might, for example, insert (or delete) only one of the node copies and not the others. This would also affect the local order among the node siblings in the view extent. Unlike the approaches that use Skolem functions or similar mechanisms to generate identifiers, our solution does not require manual specification of what input values they should take to generate ids, when writing the query.

In the context of their data integration work, Ives et al. [10] have proposed a solution for combining and restructuring XML views over streaming XML data by adding special extra attributes to the intermediate tuples. Their solution does not support the case of 1:n parent-child relationships in the returned output in which an element can occur more than once in different combinations of input bindings. This restricts the solution from handling query expressions with correlated nested sub-queries, which are very common in XQuery. Fegaras et al. [9] have proposed a mechanism for assembling streamed XML fragments to construct the XML result on the client side. Their solution is based on a special annotation called the fillers-holes annotation. The work in [9] requires fillers and holes to be defined before streaming the XML fragments. Once an XML fragment is streamed only fillers to previously defined holes into it can be processed. New inserts to other locations in the XML fragments afterwards are not allowed.

Our work relates to the problem of handling order in XML query processing since the generated identifiers also encode order information. In [7] we have studied this problem of XML order. We now extend and optimize our solution proposed there in support of handling integrated semantic identifiers.

3 Background

XQuery. We consider an expressive subset of the XQuery language [19]. This subset includes XPath expressions, nested FLWOR expressions, element constructors[3].

Source Node Identifiers. By $node$ we mean an XML element, attribute, or text. We use Fast Lexico-graphical keys ($FlexKey$s) encoding [6] for encoding source node identifiers and order in XML trees. The $FlexKey$ encoding

[3]The grammar of this subset is described in [8].

is similar to the $Dewey$ encoding [18], it encodes hierarchy and order information for each source node. Yet, the $FlexKey$ uses variable length byte strings instead of numbers. Figure 4(a) shows the $FlexKey$ encoding of nodes in the "bib.xml" XML document in Figure 1, Figure 4(b) shows the $FlexKey$ encoding for the source update shown in Figure 3(b), and Figure 4(c) shows the $FlexKey$ encoding of nodes in the "prices.xml" XML document in Figure 1.

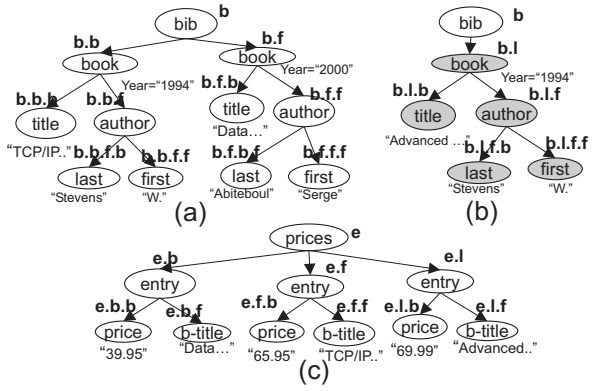

Figure 4. The Lexicographical key encoding for (a) the XML document "bib.xml" in Figure 1, (b) the newly inserted "book" element to that document (as shown in Figure 3), and (c) the XML document "prices.xml" in Figure 1.

The XML Algebra XAT. We use the XML algebra called *XAT* [21][4] implemented in the Rainbow engine [20][5]. Figure 5 shows an algebraic representation for the XQuery in Figure 2(a) using the XAT algebra.

Data Model. 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 is an order-sensitive table of tuples. The column names in an XAT table represent either a variable binding from the user-specified XQuery or an internally generated variable name.

XAT Operators. An XAT operator is denoted as $op_{in}^{out}(s)$, where op is the operator type symbol, in represents the input parameters, out the newly produced output column that is to be appended to the output table generated by the operator and s the input XAT table(s). Some XAT operators and their XAT tables are shown in Figure 5[6]. The relational complete subset of the XAT algebra includes *Select* $\sigma_c(R)$, *Cartesian Product* $\times(R,P)$, *Theta Join* $\bowtie_c (R,P)$, *Left Outer Join* $\overset{\circ}{\bowtie}_{Lc}(R,P)$, *Distinct* $\delta_{col}(R)$, *Group By* $\gamma_{col[1..n]}(R, func)$ *Order By* $\tau_{col[1..n]}(R)$, *Union* $\cup(R,P)$, *Intersection* $\bigcap(R,P)$, *Difference* $-(R,P)$, and the column renaming operator *Name* $\rho_{col1,col2}(R)$, where R and P denote XAT tables. Those operators are equivalent to their relational counterparts[7] with added responsibility of maintaining order.

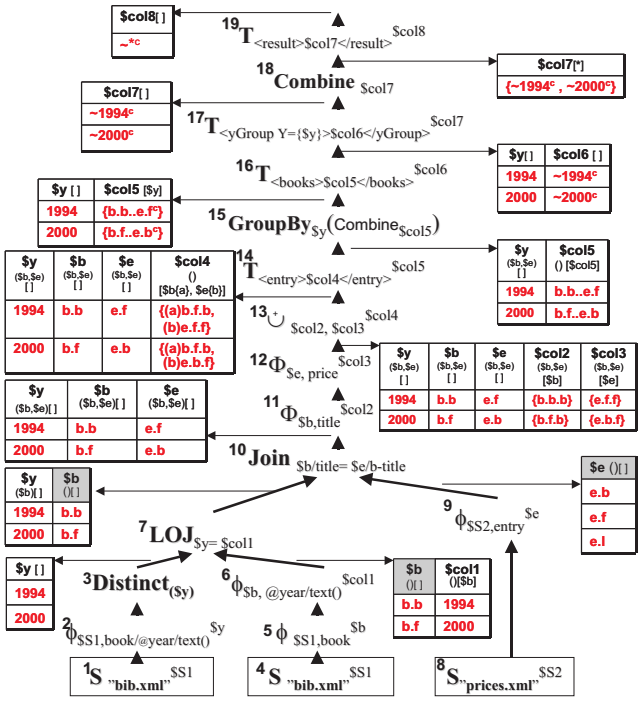

Figure 5. An algebra tree for the XQuery in Figure2(a). Annotations appearing in subscript font to the left (or below) column names represent the *Context Schema*. Shaded column names represent the *Order Schema*. Both schemas are presented in Section 4.

We now describe some of the XAT XML-specific operators. **Source** $S_{xmlDoc}^{col'}$ is a leaf node in an algebra tree that takes the XML document $xmlDoc$ and outputs an XAT table with a single column col' and a single tuple $tout_1 = (c_{1,1})$, where $c_{1,1}$ is the XAT table cell that contains a reference to the entire XML document. **Navigate Unnest** $\phi_{col,path}^{col'}(R)$ unnests the element-subelement relationship through a navigation followed by an unnest. **Navigate Collection** $\Phi_{col,path}^{col'}(R)$ is similar to *Navigate Unnest*, except it only performs the navigation functionality without unnesting. It extracts a collection from each node in column col. **Combine** $C_{col}(R)$ groups the content of all cells in column col into one sequence. **XML Union** $\overset{x\,col}{\cup}_{col1,col2}(R)$ is used to union multiple sequences into one sequence. For each tuple tin_i from R, it creates one output tuple $tout_i$, where $tout_i[col]$ is a sequence containing the members of

[4]This algebra is similar to NAL [13] and SAL [3] algebras.

[5]Translating XQuery expressions to XAT algebra can be found in [8].

[6]We discuss the details of algebra tree execution later in this paper.

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

the set $tin_i[col1] \cup tin_i[col2]$. **Tagger** $T_p^{col}(R)$ creates a new column col in which it constructs new XML nodes by applying the tagging pattern p to each input tuple.

Other XAT XML-specific operators include: **XML Unique** $\upsilon_{col}^{col'}(R)$, **XML Intersection** $\overset{x\,col}{\cap}_{col1,col2}(R)$, **XML Difference** $\overset{x\,col}{-}_{col1,col2}(R)$, and **Expose** $\epsilon_{col}(R)$. See [8] for details on those operators.

4 The Context Schema: Encoding Node Lineage and Order Information

In this section we show how we encode lineage and order specification for processed XML nodes, referred to as the *Context* of the nodes. We will use this encoding later to generate semantic identifiers for nodes in the XML result. We require that a *Context* specification is defined for each node and collection of nodes processed by the query. While at first sight this may seem expensive to maintain, it is not. We only define the *Context* specifications *schematically*, at the schema level of the query execution model (the column names of intermediate XAT tables in our case). We call such schema-level method of defining the *Context* the *Context Schema*. The *Context Schema*, defined for each column in the intermediate XAT tables, is generated during query translation and optimization time. During query execution time, we might need to obtain the *Context* itself for a given node when we generate or manipulate semantic identifiers. But such actual access of nodes is only limited to a few query operations, as we will show later in Section 5.

Definition 4.1 *We define the* ***Context*** cxt *of a node (or a collection of nodes) as a tuple (*$lngCxt$, $ordCxt$*), where* $lngCxt$ *the Lineage Context of the node is composed of a sequence of lineage values (*$lngVal_1$, $lngVal_2$,.., $lngVal_v$*), and* $ordCxt$ *the Order Context of the node is either (I) a sequence of order values (*$odrVal_1$, $odrVal_2$,.., $odrVal_y$*) or (II) a null value. A lineage value* $lngVal_i$, $1 \geq i \geq v$, *can be (1) a source node identifier, (2) a source data value, or (3) a special constant "*" . An order value* $odrVal_j$, $1 \geq j \geq y$, *can be (1) a source node identifier or (2) a newly generated order key by the query.*

The *Lineage Context* ($lngCxt$) of a processed node (or a collection) can be (1) driven directly from a specific source node, (2) driven from a certain data value from the domain of values of the source XML document, (3) not related to a specific source node or value (this case applies only to collections of nodes), (4) a composition of one or more of (1), (2), and (3). For (1) we use the relevant source node identifier to describe the lineage. For (2) we use the value that the node is bound to, to describe the lineage. For (3) we use a special constant "*" to describe the lineage, indicating that the collection itself is not bound to any specific source node. This last case occurs if at a point of the query execution the entire result is composed of one big collection of nodes (this occurs when using a $Combine$ operator), hence the lineage for the entire collection depends on all ("*") the lineage of the nodes it is composed of. To understand the *Order Context* ($ordCxt$) for processed XML node (or a collection) we need to consider three possible scenarios for order among nodes during certain point of query execution. (I) The order among the processed nodes, or even between processed collections of nodes, follows document order[8]. For this case the *Order Context* $ordCxt$ assigned to a processed node is a sequence of order values, where an order value is an identifier of a source node that reflects the document order of the processed node[9]. (II) The order is imposed by the query and is different than the document order (e.g., as a result of some *order by* clauses). In this case the *Order Context* $ordCxt$ assigned to a processed node is a sequence of order values. (III) There is no order among processed nodes (or processed collections). In this case the *Order Context* assigned to a processed node is $null$, signifying that there is no order defined. This case happens when the order is destroyed as a result of certain query operation (e.g., $Distinct$).

We will now discuss how the *Lineage Context* and the *Order Context* are encoded using the *Context Schema*. We first define the *Order Schema* (Shortly OS) to represent the order between tuples in an XAT table. An *Order Schema* is a sequence of column names from an XAT table where the order between tuples of the table can be determined solely by comparing the $FlexKey$s in those columns.

Definition 4.2 *The* ***Order Schema*** OS_R *of an XAT table* R *in an algebra tree is a sequence of column names, where the order among tuples in that table can be found by comparing the values projected from these columns.*

For example, in the output XAT table of operator #6 in Figure 5, the *Order Schema* is column $\$b$ (we use shaded column name to represent that). The order among any two tuples t_1 and t_2 in that XAT table can be derived by lexicographically comparing $t_1[\$b]$ to $t_2[\$b]$. In general, the order among cells in each column is reflected by the *Context Schema* of the column, as we will see next. Yet, the order among tuples in the XAT table as whole, may be of importance in some cases. In particular when the query involves join operations. Hence, the *Order Schema* is only maintained for queries with joins. See [8] for rules to compute the *Order Schema*.

Definition 4.3 *The* ***Context Schema*** *(*CxtSma*) for a column* col *in the XAT table (corresponding to an implicit or an explicit query variable binding) is a rule that defines how the Context (lineage and order specifications) of nodes (or collections of nodes) in that column can be extracted.*

```
CxtSma    ::= (Order)? + Lineage
Order     ::= "()" | "("+OrdCols+")"
OrdCols   ::= colName + ("," + OrdCols)*
```

[8] Note that this does not only apply to source nodes but may also apply to constructed nodes constructed over source nodes.

[9] If the *Lineage Context* itself reflects the order, we define the *Order Context* as an empty sequence of order values.

```
Lineage   ::= "[]" | ("[" + LngCols + "]")
LngCols   ::= (colName | colsUnion) +
              (","+ (colName| colsUnion))*
colsUnion ::= (colName + "{" + ColID+ "}") +
              (colName + "{" + ColID + "}")
ColID     :: FlexKey
```

The *Context Schema* ($CxtSma$) for a column col is a composition of an optional order prefix phrase ($Order$) and a lineage phrase ($Lineage$). The order phrase can be an empty list "()" indicating that the order information of nodes in col can be derived from the lineage phrase. In that case there is no need to have an extra encoding for order. If the lineage phrase does not reflect the correct order, the order phrase will contain a list of column names ($colName$) that determine how the order of nodes in col can be derived. The absence of the order prefix phrase (equals to *null*) indicates that no order is defined for the column. In other words the *Order Context* for any node in that column is $null$. In general, the order encoding in the *Context Schema* of a column enables us to derive the order among cells in that column.

The lineage phrase is a list of XAT table column names from which the lineage of nodes in col can be derived. The list can be empty "[]" indicating that the lineage of the column is related to itself. A non-empty list may contain regular column names ($colName$) and/or annotated column names ($colsUnion$). An annotated column name ($colsUnion$) is a column name annotated with an identifier ($ColID$), namely a $FlexKey$ identifier that is assigned by the *XML Union* operator and is unique for each unioned column. It is used to distinguish each column used as input to the union operation. This is used later when we generate the semantic identifiers, to ensure the uniqueness of nodes originating from different input columns when unioned. $ColID$ also helps in maintaining the order among nodes originating from different unioned columns.

Computing the Context Schema. The *Context Schema* is first created for the $Source$ operator since it is the leaf operator in any XAT algebra tree. Other operators may create *Context Schema*s for newly created columns or manipulate the *Context Schema* for existing columns. Table 1 shows rules for generating and manipulating the *Context Schema*. Table 1 uses the following conventions: (1) $col.ord$ to refers to the *Order Context* of a column col, (2) $col.lng$ to refers to the *Lineage Context* of a column col, (3) $p.col$ to refers to the column in a tagger pattern p, (4) $R[col_i]$ to refers to the column with index i in the XAT table R, (5) $R[col_i].cxtSma$ to refers to the *Context Schema* for a column col_i in the XAT table R, and (6) $R.OS$ to refers to the *Order Schema* of the XAT table R.

We now discuss the *XML Union* as an example of computing the *Context Schema*[10]. The *XML Union* operator ${}_{x}\cup^{col}_{col1,col2}(R)$ creates new collections in column col from the contents of columns $col1$ and $col2$. The *Lineage Context* of the new column col is derived from the *Lineage Context* of both columns $col1$ and $col2$. Hence the *Lineage Context* for col will be $[col_1.lng\{fk_1\}, col_2.lng\{fk_2\}]$ where fk_1 and fk_2 are identifying $FlexKeys$ to distinguish between the two columns[11]. For example if an *XML Union* operator is used for creating a collection from two columns col_1 and col_2 the *Context Schema* might be $[col_1.lng\{a\}$,$col_2.lng\{b\}]$. Here a and b reflect the order in which the columns are unioned. If the *Order Context* of each of the source columns ($col1$ and $col2$) is equivalent to its *Lineage Context*, then the order context of col is assigned to its *Lineage Context*. Otherwise, the *Order Context* of col is set to union of the *Order Context*s of the source columns.

Example. Figure 5 shows how the *Context Schema* is defined for columns in the intermediate XAT tables based on the rules shown in Table 1. The *Context Schema* is shown in a subscript font to the right of column names (or below them). The output XAT table of operator # 3, for example, has one column ($\$y$) representing the distinct values of years. Based on the rules in Table 1 it is assigned a *Lineage* phrase that references itself [] and a null *Order* phrase (indicating that there is no order semantics for that column). The output XAT table of operator # 6 has two columns ($\$b$ and $\$col1$). Column ($\$b$) contains the node identifier of extracted books. The *Lineage* phrase assigned to that column is derived from itself [], as this column is obtained through a *Navigate Unnest* operation. The *Order* phrase of that column is set to $()$, signifying that it is equal to the *Lineage* phrase ($\$b$). Hence, if we wish to derive the order between nodes in column $\$b$ we compare the $FlexKey$s in that column lexicographically. Column $\$col1$ gets a *Context Schema* $()[\$b]$ (based on the second case in rule category III in Table 1). The LOJ operator (# 7) does not affect the *Lineage* phrase. It only changes the *Order* phrase. It uses the *Order Schema* of the input tables (highlighted columns) and the input column's *Order* phrase to determine the new *Order* phrase. Based on the rules in Table 1, the *Order* phrase of column $\$y$ is set to $(\$b)$. The *Order* phrase of column $\$b$ is not affected because the left input table has no *Order Schema*. The $Join$ operator (# 10) also sets only the *Order Schema* of its output columns $\$y$, $\$b$, and $\$e$ as shown in Figure 5. Note that for the operators on top of the operator # 10, the *Order Schema* is no longer needed to compute the *Context Schema* rules, since there is no other $Join$ operations. Hence, the *Order Schema* is not defined for those other operators.

As a result of the *Navigate Collection* operators # 11 and # 12, columns $\$col2$ and $\$col3$ are created and each of them is assigned a *Context Schema* that is derived from that of the column it was extracted from. The *XML Union*

[10] See [8] for discussion on all operators.

[11] This identifying extension is used later when we generate the semantic identifiers to ensure uniqueness of the identifiers and to reflect order.

Cat.	Operator op	Affected column	Assigned Context Schema	Node Level operation
I	S^{col}_{xmlDoc}	col	$()[col]$ or shortly $()[]$	$None$
II	$\Phi^{col'}_{col,path}(R)$	col'	$if(col.ord == empty)$, $()[col.lng]$ $elseif(col.ord == null)$, $[col.lng]$ $else$, $(col.ord)[col.lng]$	$None$
III	$\phi^{col'}_{col,path}(R)$	col'	$if((col.ord == empty)\|\|(col.ord == null))$, $()[]$ $else\ if$ (path navigate to text node), $(col.ord)[col.lng]$ $else$, $(col.ord + col')[]$	$None$
IV	$C_{col}(R)$	col	$[*]$	for each tuple t in R apply $assignOverRidOrd(t, col)$
V	$T^{col}_{p}(R)$	col	$if(p.col.ord == empty)$, $()[col]$ $elseif(p.col.ord == null)$, $[col]$ $else$, $(p.col.ord)[col]$	for each tuple t in R apply $composeNodeIds(t, col, p)$
VI	$\gamma_{col[1..n]}(R, C_{col})$	$All\ columns$	if (grouping by id), $(col_1.ord, .., col_n.ord)[col_1.lng, .., col_n.lng]$ else if (grouping by value), $[col_1.lng, .., col_n.lng]$	The Combine operator uses the function $assignOverRidOrd$
VII	$\overset{x\ col}{\cup}_{col1,col2}(R)$	col	$if((col_1.ord == empty)\&\&(col_2.ord == empty))$, $()[col_1.lng\{fk_1\}, col_2.lng\{fk_2\}]$ $else$, $(col_1.ord, col_2.ord)[col_1.lng\{fk_1\}, col_2.lng\{fk_2\}]$ (where fk_1 and fk_2 are Flexkeys reflecting order)	for each tuple t in R apply $assignColIdPrfx(t, col_1, col_2)$
VIII	$v^{col'}_{col1}(R)$ $\overset{x\ col'}{\cap}_{col1,col2}(R)$ $\overset{x\ col'}{-}_{col1,col2}(R)$ $\rho_{col1,col'}(R)$	col'	$if(col_1.ord == empty)$, $()[col_1.lng]$ $elseif(col_1.ord == null)$, $[col_1.lng]$ $else$, $(col_1.ord)[col_1.lng]$	$None$
IX	$\delta_{col}(R)$	$All\ columns$	$[col.lng]$	$None$
X	$\times(R, P)$ $\bowtie_c(R, P)$ $\overset{\circ}{\bowtie}_{Lc}(R, P)$	$R[col_1..col_m]$ $P[col_1..col_n]$	$for(i = 1; i <= m; i++)$ $R[col_i].CxtSma = (R[col_i].ord + P.OS)[R[col_i].lng]$ $for(i = 1; i <= n; i++)$ $P[col_i].CxtSma = (R.OS + P[col_i].ord)[P[col_i].lng]$	$None$
XI	$\sigma_c(R)$	$None$	N/A	$None$
XII	$\tau_{col[1..n]}(R)$	$R[col_1..col_m]$	$for(i = 1; i <= m; i++)$ $R[col_i].CxtSma = (col[1..n])[R[col_i].lng]$	$None$

Table 1. Rules for computing the *Context Schema* for different XAT operators.

operator (operator # 13) creates new collections in column $\$col4$ from the contents of columns $\$col2$ and $\$col3$. Hence, the *Lineage* phrase of column $\$col4$ becomes a composition of the *Lineage* phrases of columns $\$col2$ and $\$col3$ which is $[\$b\{a\}, \$e\{b\}]$ after assigning the special column source identifiers a and b. The *Order* phrase of the new column $\$col4$ is derived from the *Order* phrases of both input columns. Hence, it becomes $(\$b, \$e, \$b)$, or simply $(\$b, \$e)$, since removing the redundant $\$b$ will not affect the order semantics. And since this *Order* phrase is equivilant to the *Lineage* phrase, we simply set the *Order* phrase to $()$. The *Tagger* operator (operator # 14) constructs new nodes in column $\$col5$ assigning a "self" *Lineage* phrase to it $[]$ and an *Order* phrase equal to $()$[12]. The *Group By* operator (operator # 15) changes the *Lineage* phrases of all the output columns to be equivalent to the *Lineage* phrase of the grouping column $\$y$. It also sets the *Order* phrase of the output columns to $null$ since the *Group By* destroys the order among tuples (created groups)[13]. The remaining operators in the algebra tree are easy to follow.

[12] Note that only columns $\$y$ and $\$col5$ remain in the output at this point. Other columns are pruned out through an optimization process that discards columns that are not used by later operators or that are not referenced by the *Context Schema* of any column.

[13] Since this is a value-based *Group By*. An id-based *Group By* (representing a nesting operation) would define certain order among the create groups, as shown in Table 1.

5 Generating Semantic Identifiers from the Context Schema

We now describe how we utilize the *Context Schema* to generate the semantic ids for processed XML nodes.

Definition 5.1 *The **Semantic Identifier** (*SemID*) is an identifier assigned to a node in the XML result. Such identifier is locally unique (among sibling nodes) and carries lineage information that references the source from which the node is derived. It also encodes local order of the nodes among sibling nodes.* SemID *is a composition of an optional order id prefix term (*OrdPrefix*) and a body part that can be a base node id (*BaseNodeID*) or a constructed node id (*ConstNodeID*). The body part carries lineage information and determines the node type (source node or constructed node).*

```
SemID       ::= (OrdPrefix)? + (BaseNodeID | ConstNodeID)
OrdPrefix   ::= "~" |  OverRideOrd
OverRideOrd ::= "(" + FlexKey + ")"
BaseNodeID  ::=  FlexKey
ConstNodeID ::= LngCxt + "c"
LngCxt      ::= (FlexKey | "*" |StringLiteral) +
                ("." + LngCxt)*
```

In many cases, the lineage information encoded in the semantic id body can reflect the node order as well. If this is not the case then a special order prefix (OrdPrefix) is added to the semantic id body. The prefix order id can be either a $FlexKey$ representing a new order that overrides the order implied by the lineage information encoded in the semantic id body or a special constant "$\sim$" indicating that there

is no order defined locally for the node. The body of the semantic id depends on the type of the node. A node in the view extent can be of two types: (I) a base node originating from a source document that is exposed without any modifications[14], or (II) a newly constructed node. The body of the semantic id *SemID* for a base node that is exposed in the view is simply the same as its id (a $FlexKey$). The body of *SemID* for a constructed node is composed of a *Lineage Context* value ($LngCxt$) and a constant suffix (^{c}) indicating that the id reflects a constructed node. The *Lineage Context*, as we discussed earlier can be a reference to a $FlexKey$, a reference to string value from the domain of values of the source XML document, a constant "*", or a composition of one or more of them separated by a delimiter "..". This *Lineage Context* can be derived from the *Lineage* phrase of the *Context schema* during query execution.

The last column in Table 1 shows the node-level operations required for actually generating and maintaining semantic ids. As shown in Table 1, we require node-level access for only four algebra operations. Namely, the *Combine*, the *Tagger*, the *XML Union*, and the *Group By* operators. We define two functions $getLngCxt()$ and $getOrdCxt()$ that when invoked for a node (or collection) return the *Lineage Context* and the *Order Context* of that node (or collection), respectively.

We now discuss, briefly, the logic of the functions in Table 1[15] (1) The function $generateNodeId$ is used by the $Tagger$ operator to generate semantic id for the newly constructed nodes. This includes setting the order prefix, if necessary. (2) The function $assignColIdPrfx$ is used by the *XML Union* operator to assign the order prefix part of the semantic id for nodes originating from different columns. (3) The function $assignOverRidOrd$ is used by the $Combine$ operator to set the order prefix part of the semantic id for combined nodes. (4) The function $assignOverRidOrd$ is also used by the *Group By* to set the order prefix part of the semantic id for nodes in the created groups[16].

Example. In the algebra tree in Figure 5 we note that before the *XML Union* (operator # 13) query processing is performed normally without the need to perform any additional id-specific operations. The *XML Union* (operator # 13) assigns source column prefix order ids (a and b) to the nodes in the new column $\$col4$. The *Tagger* (operator # 14) constructs new nodes "entry" from the collections in $\$col4$ using the *Context Schema* of $\$col4$. The *Lineage* phrase of column $\$col4$ consists of columns $\$b$ and $\$e$. Hence, we derive the body of the semantic id from corresponding projected nodes in those columns. Since the *Order* phrase also refers to the same columns "()", we conclude that the semantic id body can represent the order. For example, for the first tuple in the XAT table we generate the semantic id $b.b..e.f^c$ for the newly constructed node in column $\$col5$[17]. This new node becomes a parent to the collection containing the two nodes $(a)b.f.b$ and $(b)e.f.f$.

The *Group By* (operator # 15) groups the constructed nodes in column $\$col5$ by the year (column $\$y$). The *Order* phrase of the grouped column ($\$col5$) indicates that the ids in that column already reflect the order. Hence we do not assign any prefix node ids. The *Tagger* (operator # 16) constructs new nodes "books" for the collections in column $\$col5$. The created nodes are assigned semantic ids that are derived from the "year" values in column $\$y$. And since the *order* phrase of column $\$col5$ is $null$, the *Tagger* operator assigns a prefix order constant "$\sim$" to each new node, indicating that no-order is defined for those nodes. For example, the first constructed node is given a semantic node id $\sim 1994^c$ and is becoming the parent for the collection with one "entry" node with id $b.b..e.f^c$. The *Tagger* (operator # 17) constructs new nodes "yGroup". with semantic ids $\sim 1994^c$ (on top of the "books" node with id $\sim 1994^c$) and $\sim 2000^c$ (on top of the "books" node with id $\sim 2000^c$). Next the *Combine* (operator # 18) creates a collection out of those nodes. The input column ($\$col7$) for the *Combine* operator has a $null$ *Order Context*. Hence, nodes in the created collection keep their no-order prefix ($\sim$). Finally the *Tagger* (operator # 19) constructs a root node for the result on top of the collection in column $\$col7$. The semantic id assigned to this root node is $\sim *^c$ as derived from the *Context Schema* of column $\$col7$.

The final result of executing the query is shown in Figure 6(a). Note that the generated semantic ids serve as local unique ids for nodes and at the same time encode the nodes local order (semantic ids that start with $\sim$ reflects no-order semantics).

6 XML Fusion Using Semantic Identifiers

We first define a mechanism for merging XML fragments processed incrementally with the existing XML result. For this we use the *Deep Union* operator. The *Deep Union* operator was introduced in the context of the semi-structured data model in [5]. We here adapt the *Deep Union* operation to the general XML tree model.

Definition 6.1 *The **Deep Union** ($\bigsqcup$) of two XML trees*[18] *$t_1 = (r_1 : ch_1)$ and $t_2 = (r_2 : ch_2)$ unions their root nodes r_1 and r_2 by node identifier and recursively performs deep union on their respective lists of child nodes ch_1 and ch_2. The resulting XML tree includes all nodes in the two XML trees with only one occurrence of any matching nodes (by node ids) from the two trees.*

[14] Such node is an exact copy of the source node including its subtree.

[15] The algorithms can be found in [8].

[16] Although that the value-based *Group By* does not define order between created groups, there might be order among nodes in each group.

[17] The semantic id itself reflects the desired order (document order of the source "book" node as major order and document order of the source "entry" node as minor order.

[18] Each XML tree is annotated with semantic node identifiers.

$$r_1 \bigsqcup r_2 = \begin{cases} r_1 \cup r_2 & \text{if } r_1.id \neq r_2.id \\ r : (ch_1 \bigsqcup ch_2) & \text{if } r_1.id = r_2.id \end{cases}$$

where $r = r_1 = r_2$.

Our solution enables views to be distributive over the *Deep Union* operator. This means that we can process insert source updates incrementally without recomputing the view. For example, if the a view $V(S1, S2)$ = $S1 \bowtie S2$ is distributive over the *Deep Union* operator, we should be able to maintain the view as follows: $V(S1 \bigsqcup \triangle S1, S2)$ = $(S1 \bowtie S2) \bigsqcup (\triangle S1 \bowtie S2)$ where $\triangle S1$ is an update to $S1$. This means that we can propagate the update by simply processing $\triangle S1 \bowtie S2$ and merging the result with the existing view extent ($S1 \bowtie S2$). Applying this to our running example, and since the example involves a self-join we treat each access to the same source as a separate source. Hence, for the view $v(S1, S1, S2)$ shown in Figure 2(a), where $S1$ ="bib.xml" and $S2$= "prices.xml", and as a result of the update $\triangle S1$ (shown in Figure 3(a)), we wish to show that $V(S1 \bigsqcup \triangle S1, S1 \bigsqcup \triangle S1, S2)$ = $V(S1, S1, S2) \bigsqcup V(\triangle S1, S1, S2) \bigsqcup V(S1, \triangle S1, S2) \bigsqcup V(\triangle S1, \triangle S1, S2)$. This is also equal to $V(S1, S1, S2) \bigsqcup V(\triangle S1, S1, S2) \bigsqcup V(S1', \triangle S1, S2)$ by merging the third and the fourth terms and given that $S1'$ = $(S1 \bigsqcup \triangle S1)$.

This is possible because when processing the source updates, our solution reproduces old node identifiers and generate new ones, as appropriate, in a way that enables fusing the processed updates with the result. We establish the correctness of our solution using the following theorem.

Theorem 6.1 *Given a view V= (S_1, S_2, ..., S_n) defined over input XML data sources S_1, S_2, ..., S_n by an XAT algebraic expression T. Let $\triangle S_i$ be an update to one of T's data sources S_i, $1 \leq i \leq n$. Let V^{rec} = $V(S_1, .., S_i \bigsqcup \triangle S_i, .., S_n)$ be the view extent after recomputation. Let V' = $V(S_1, .., S_i, .., S_n) \bigsqcup V(S_1, .., \triangle S_i, .., S_n)$ be the view after propagating and applying the update using the* Deep Union *operator. We find that V^{rec} = V'.* □

We prove Theorem 6.1 by first proving the distributivity of XAT operators. We then prove the distributivity of V composed of any number of algebraic operators by induction on the height of T. The proof can be found in [8].

Example. First we assign appropriate $Flexkey$s to the new nodes inserted into the source document as shown in Figure 4(b). Next we process the incremental parts of the propagation formula above. Namely, $V(\triangle S1, S1, S2)$ and $V(S1', \triangle S1, S2)$. Lastly we fuse the results of propagating the updates with the original view extent ($V(S1, S1, S2)$). Figure 6(a) shows the original view extent ($V(S1, S1, S2)$). Figures 6(b) and (c) show the results of executing $V(\triangle S1, S1, S2)$ and $V(S1', \triangle S1, S2)$ respectively[19]. Merging the incremental results (propagated updates in 6(b) and (c)) with the original view extent (in Figure 6(a)) using the *Deep Union* operator results in the refreshed view extent shown in Figure 6(d). Note that as a result of that, only the XML fragment with root node $b.l..e.l^c$ is added to the view extent. Other nodes that appear in the propagated updates are fused with the corresponding nodes from the original view extent as due to equivalent ids. Also note that the order is maintained in the refreshed view extent, as the order encoded in the node identifier $b.l..e.l^c$ indicates that it should come second when compared with the other sibling node with id $b.b..e.f^c$. In general, the final result we get in Figure 6(d) is equivalent to the result we would get if we process the view query over the entire source document after applying the source update to it.

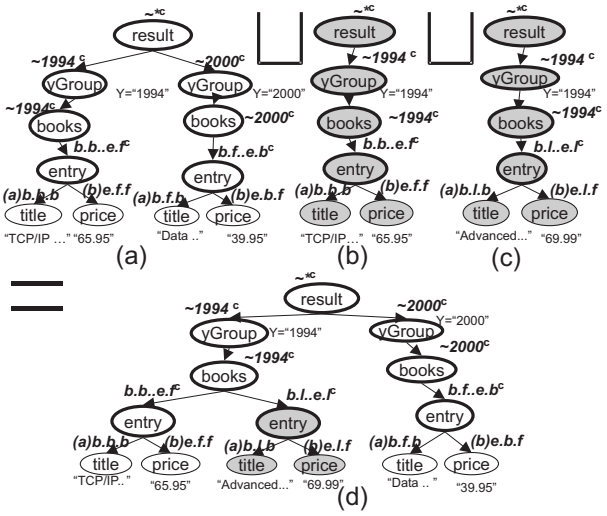

Figure 6. (a) The original view extent $V(S1, S1, S2)$, (b) the incremental computation $V(\triangle S1, S1, S2)$, (c) the incremental computation $V(S1', \triangle S1, S2)$, and (d) the refreshed materialized view computed as (a) $\bigsqcup$ (b) $\bigsqcup$(c).

Implications of Our Solution. Our semantic id solution enables two important features. (1) Distributive XML query processing even for queries involving operations like group by, nesting, distinct, and sorting, typically known to be non-distributive. This provides a base for enabling applications like efficient XML incremental view maintenance[20] and efficient XML stream processing. (2) Efficient support for XML order, where the need to perform intermediate sorting is removed and new query optimization opportunities might open.

[19]Due to space limitations we do not show the detailed execution of each of these two plans. Generating semantic ids during those executions follows the same logic used in the initial execution shown in Figure 5. The detailed execution of those incremental queries can be found in [8].

[20]We wish to point out that in this paper we do not propose a full XML view maintenance solution. Our solution can be a key component in an XML view maintenance solution. Such VM solution should address other issues like handling other updates types in addition to insert updates.

7 Experimental Evaluation for the Cost of Generating Semantic Identifiers

We have implemented our semantic id solution in Java within the Rainbow system framework [20]. We have performed preliminary evaluation using the XMark benchmark data [16]. Figure 7(a) and (b) show the results obtained when using a query that exploits our semantic id system intensively. In that query, most of the returned nodes are constructed ones. Hence, a lot of node construction and new semantic id generation is required. The query also involves a mixture of order decisions, where some nodes are returned in document order and some are returned in an order imposed by the query. Figure 7(a) shows the cost of generating semantic ids relative to the total query execution time on different input XML document sizes. The figure shows that this cost is negligible compared to the total cost of query execution. Figure 7(b) shows the breakdown of the cost of our approach and compare it to the cost of execution (using 500MB input document size). The cost of our solution is mainly composed of three elements. (1) The cost of computing the *Order* and *Context Schema*s. This cost depends in the number of operators in the query plan and does not depend on the size of data. (2) The cost of generating semantic ids for constructed nodes. This cost depends on the size of processed data and on the amount of node construction the query performs. (3) The cost of assigning the order prefix for the semantic ids. Figure 7(b) shows that the cost of generating new semantic ids on node constructions is higher than the other two cost element. The cost of generating the *Order* and *Context Schema*s is very small.

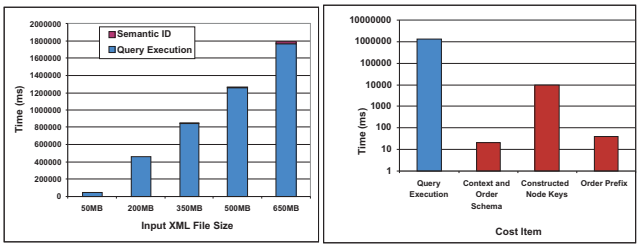

Figure 7. (a) The overhead of generating semantic identifiers to query execution time and (b) the break down of the cost of generating semantic identifiers.

8 Conclusions

We have proposed a solution to the problem of incrementally constructing XML views. Our solution utilizes semantic identifiers to perform id-based fusion of XML fragments. Our solution is performed in three phases. First, we define how lineage and order information of processed XML data is encoded using the *Context Schema*. Second, we use the *Context Schema* to generate reproducible semantic ids for nodes in the XML result and for incrementally processed nodes. Third, the id-based fusion is performed for the processed XML fragments through a special operation called *Deep Union*. Our solution supports an expressive class of XML transformations and does not require any manual specification of how identifiers are to be generated or materialization of large auxiliary data. The solution can be easily integrated with the XML query processing framework and comes with a very small processing overhead to the query execution time, as shown by our experiments.

References

[1] S. Abiteboul and et al. 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. MOVIE: An incremental maintenance system for materialized object views. *DKE Journal*, 47(2):131–166, 2003.

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

[4] P. Bohannon, B. Choi, and W. Fan. Incremental evaluation of schema-directed XML publishing. In *SIGMOD*, pages 503–514, 2004.

[5] P. Buneman, A. Deutsch, and W. C. Tan. A deterministic model for semi-structured data. In *Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jan 1999.

[6] K. Deschler and E. Rundensteiner. Mass: A multi-axis storage structure for large xml documents. In *CIKM*, pages 520–523, Nov 2003.

[7] M. El-Sayed and et al. Efficiently Supporting Order in XML Query Processing. *DKE Journal*, 54(3), September 2005. in press.

[8] M. El-Sayed, E. A. Rundensteiner, and M. Mani. An Id-based Solution for Incremental Fusion of XML Fragments. Technical Report WPI-CS-TR-04-17, Worcester Polytechnic Institute, June 2004.

[9] L. Fegaras and et al. Query processing of streamed xml data. In *CIKM*, pages 126 – 133, 2002.

[10] Z. G. Ives and et al. An xml query engine for network-bound data. *The VLDB Journal*, 11 (4):402–402, December 2002.

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

[12] D. Maier. A logic for objects. In *Workshop on Foundations of Deductive Database and Logic Programming*, pages 6–26, 1986.

[13] N. May and et al. Nested queries and quantifiers in an ordered context. In *ICDE*, pages 239–250, 2004.

[14] Y. Papakonstantinou and et al. Object fusion in mediator systems. In *VLDB*, pages 413–424, 1996.

[15] L. Popa and et al. Translating web data. In *VLDB*, pages 598–609, 2002.

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

[17] D. Suciu. An overview of semistructured data. *SIGACTN: SIGACT News*, 29(4):28–38, 1998.

[18] I. Tatarinov and et al. Storing and Querying Ordered XML Using a Relational Database System. In *SIGMOD*, pages 204–215, 2002.

[19] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, February 2005.

[20] X. Zhang and et al. Rainbow: Multi-XQuery Optimization Using Materialized XML Views. In *SIGMOD Demo*, page 671, 2003.

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

[22] Y. Zhuge and H. Garcia-Molina. Graph Structured Views and Their Incremental Maintenance. In *ICDE*, pages 116–125, 1998.