# Incremental Maintenance Of Materialized XQuery Views

by

Maged F. El-Sayed

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

August 9, 2005

**APPROVED:**

Prof. Elke A. Rundensteiner
Advisor

Prof. Murali Mani
Co-Advisor

Prof. Carolina Ruiz
Committee Member

Prof. Jayavel Shanmugasundaram
Cornell University
External Committee Member

Prof. Mike Gennert
Head of Department

# Abstract

Keeping views fresh by maintaining the consistency between materialized views and their base data in the presence of base updates is a critical problem for many applications, including data warehousing and data integration. While heavily studied for traditional databases, the maintenance of XML views remains largely unexplored. Maintaining XML views is complex due to the richness of the XML data model and the powerful capabilities of XML query languages, such as XQuery.

This dissertation proposes a comprehensive solution for the general problem of maintaining materialized XQuery views. Our solution is the first to enable the maintenance of a large class of XQuery views including XPath expressions, FLWOR expressions, and Element Constructors. These views may contain arbitrary result construction and arbitrary grouping and join operations. Our solution also supports the unique order requirements of XQuery including source document order and query order.

The contributions of this dissertation include: (i) an efficient solution for supporting order in XML query processing and view maintenance, (ii) an identifier-based technique for enabling incremental construction of XML views, (iii) a mechanism for modeling and validating source XML updates, (iv) a counting algorithm for supporting view maintenance on delete and modify updates, (v) an algebraic solution for propagating bulk XML updates, and (vi) an efficient mechanism for refreshing materialized XML views on propagated updates. We provide proofs of correctness of our proposed techniques for materialized XQuery maintenance.

We have implemented a prototype of our view maintenance solution on top of the Rainbow XML query engine, developed at WPI. Our experiments confirm that our solution provides a practical and efficient solution for maintaining materialized XQuery views even when handling heterogeneous batches of possibly large source updates.

Our solution follows the widely adopted propagate-apply framework for view maintenance common to all mainstream query engines. That is, our solution produces incremental maintenance plans in the same algebraic language used to define the views. These plans can thus be optimized and executed by standard query processing techniques. Being compatible with standard frameworks paves the way for our XML view maintenance solution to be easily adopted by existing database engines.

# Acknowledgments

I would like to express my great gratitude to my advisor, Prof. Elke A. Rundensteiner for her guidance, encouragement, and support since my first day here at WPI. She has always inspired me with her knowledge and dedication and has helped me to become the researcher I am today.

Many thanks go to my other PhD committee members, Prof. Murali Mani, Prof. Carolina Ruiz, and Jayavel Shanmugasundaram for their valuable feedback and suggestions on my dissertation. In particular I would like to thank my co-advisor Prof. Murali Mani for his time and valuable directions. I would like also to thank Prof. Carolina Ruiz who was always supportive and helpful.

Many thanks go to Prof. Michael Gennert for his thoughtfulness and support during critical times of my dissertation. Many thanks also go to Prof. Micha Hofri who gave me the chance to teach at WPI and to gain such a valuable experience. I would also like to thank other faculty and staff members from the CS department at WPI, in particular Prof. Stanley Selkow, Glynis Hamel, Sharon Demaine, Jessica Pollock, Michael Voorhis, and Jesse Banning.

I would like to thank Katica Dimitrova for the joint work related to XML order and to thank members of the Rainbow team, in particular Xin Zhang, Song Wang, Ling Wang, and Luping Ding for the joint work related to the Rainbow XML query engine.

The long time I spent in my office would have been harder without the company of Bin Liu, Mariana Jbantova, Venkatesh Raghavan, and Rimma Nehme who made the office such a warm and nice place. I was also fortunate to meet many other current and pervious DSRG members.

I would like to acknowledge the support given to me by my university back home (Arab Academy for Science and Technology, Alexandria, Egypt). I would like in particular to thank Dr. Gamal Mokhtar, Mr. Mahmoud Ghoneim, Mr. Nabil Fahmy, Prof. Said El-Neshokaty, Prof. Asaad Elnidani, Prof. Ismail Hussien, and Mrs Fayza Boghdady. I would like also to thank my friend and colleague Mahmoud Youssef and many other friends who always supported me.

Lastly, I would like to express my deepest gratitude to my parents, who have been always behind every advance in my life, and also to my lovely wife Hanan, my lovely sons Youssef and Sief for all the love, support, and understanding.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

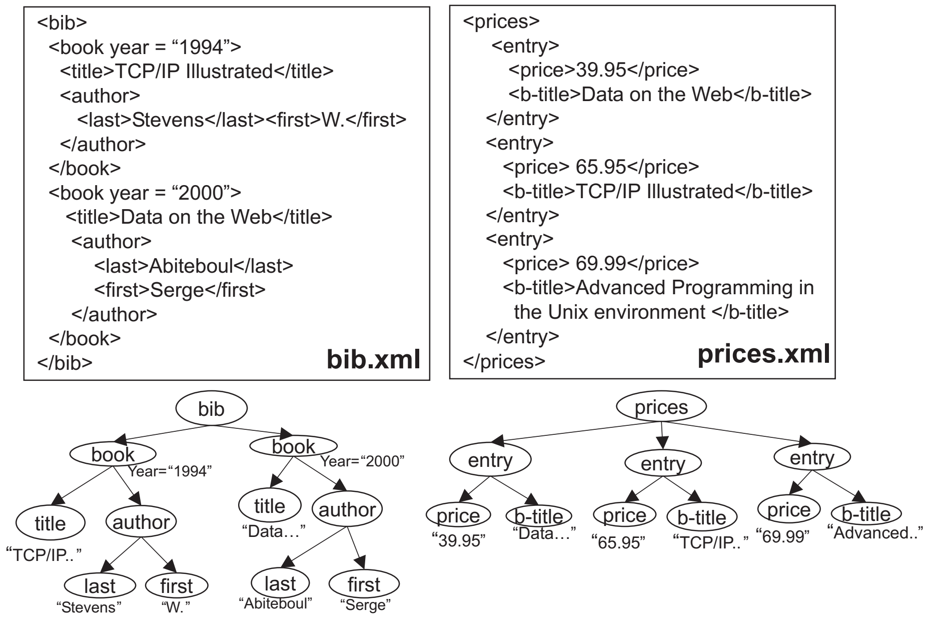
A view is a function defined on top of data sources. Each time the view is executed, it returns a collection of data called view extent. Materialization of view extents has important applications including providing fast access to derived database repositories, optimizing query processing based on cached results, and increasing availability. Materialized views facilitate advanced database applications like data warehousing, data integration, and online analytical processing.

Maintaining the consistency between materialized views and their base data in the presence of source updates is important to ensure that the materialized views are up-to-date. The straightforward solution for this problem is to recompute the view from scratch over the updated sources. This process is expensive and might take a long time (hours or even days for large data sources). Hence, it is not practical to do this each time the sources are updated, considering that source updates are typically small relative to the original source size. To address this problem, incremental maintenance has been advocated as a cheaper solution over full recomputation [GM95]. Such view maintenance work has however been largely done in the context of traditional databases. This does no longer fulfill current needs with XML emerging as important medium for representing and exchanging data over the Internet.

XQuery [W3C05] has been proposed as standard for querying XML. XQuery is a powerful query language for specifying queries over the XML data model that incorporates the capabilities of many languages including *XPath* [W3C99], *SQL* [Int99], and *XML-QL* [DFF+99] languages. Solutions proposed thus far in the literature for maintaining XML or semi-structured views have fallen short in supporting many of the core features of XML languages, especially the XQuery language, as we will discuss in Section 1.3.

## 1.2 Research Challenges in Maintaining Materialized XQuery Views

**Motivating Example.** Consider the two XML documents shown in Figure 1.1. The source document "bib.xml" stores book information and the source document "prices.xml" stores prices of books. Assume that the relative order among elements in each source document, as shown in Figure 1.1, is a desired source document order that is important for the domain. Consider the XQuery view shown in Figure 1.2(a) defined over the two source documents in Figure 1.1. The view extent resulting from executing the XQuery view is shown in Figure 1.2(b), with highlighted nodes representing newly constructed nodes. Now assume that the updates shown in Figure 1.3[1] are applied to the source XML documents. The view maintenance solution must refresh the materialized view to reflect the effect of these source updates on the materialized view in Figure 1.2(b). A correctly refreshed materialized view should be equal to the materialized view we would get if we were to recompute the view over the updated sources. The expected result of refreshing the materialized view extent (by recomputing it) in Figure 1.2(b) in response to the three source updates shown in Figure 1.3 is shown in Figure 1.4.



```
<bib>
  <book year = “1994”>
    <title>TCP/IP Illustrated</title>
    <author>
      <last>Stevens</last><first>W.</first>
    </author>
  </book>
  <book year = “2000”>
    <title>Data on the Web</title>
    <author>
      <last>Abiteboul</last>
      <first>Serge</first>
    </author>
  </book>
</bib>
```

bib.xml

```
<prices>
    <entry>
      <price>39.95</price>
      <b-title>Data on the Web</b-title>
  </entry>
  <entry>
    <price> 65.95</price>
    <b-title>TCP/IP Illustrated</b-title>
  </entry>
  <entry>
    <price> 69.99</price>
    <b-title>Advanced Programming in
     the Unix environment </b-title>
  </entry>
</prices>
```

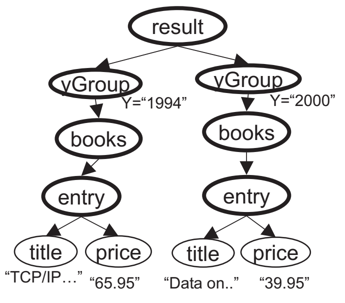
prices.xml

Figure 1.1: Two input XML documents.

[1]Source updates are defined using the XQuery update language [TIHW01].

```
<result>{
 FOR $y in
 distinct-values(doc("bib.xml")/bib/book/@year)
ORDER BY $y
 RETURN
   <yGroup  Y= "{$y}">
     <books>
         FOR $b in doc ("bib.xml")/bib/book,
               $e in doc ("prices.xml")/prices/entry
         WHERE $y = $b/@year and
                    $b/title = $e/b-title
         RETURN
            <entry> {$b/title} {$e/price}</entry>
     </books>
   </yGroup>
</result>
```

(a)



(b)

Figure 1.2: (a) XQuery expression defined on top of the two XML documents in Figure 1.1 and (b) resulting XML view extent.

Below we highlight issues that must be considered when maintaining XQuery views, using the above as running example.

**Modeling and Validating Source XML Updates.** In the relational context source updates simply correspond to flat tuples that conform to the same schema as that of the table to be updated. In the XML context, source XML documents might not have a schema. Also an update to an XML document might be provided in different shapes. For example, the update might be provided as an entire XML fragment as in Figure 1.3(a), a root to an XML fragment as in Figure 1.3(b), or a leaf node value as in Figure 1.3(c). We need to decide how to:

- *Model XML source updates.* This includes what data structures to use, how to encode hierarchial and order information of the update, and how to represent different update operations. For example, the update shown in Figure 1.3(a) inserts a new "book" fragment in a par-

```
for  $book in document("bib.xml") /bib/book[2]
update  $book
insert  <book year = "1994"><title>Advanced programming in
the Unix environment</title><author><last>Stevens</last>
<first>W. </first></author></book> after $book
```

(a)

```
for $book in document("bib.xml") /bib/book
where $book/title ="Data on the Web"
update $book
delete $book
```

(b)

```
for $entry in document("prices.xml") /prices/entry
where $entry/b-title ="TCP/IP Illustrated"
update $entry
replace $entry/price/text() with "70"
```

(c)

Figure 1.3: Three XQuery updates.

ticular location in the XML document (after the second "book" in "bib.xml"). This affects the relative order of all elements that come after that "book" (if any). In addition, the order in the materialized view extent might be affected by that update. Such update should be modeled in a way that specifies the exact location and order of the insertion.

- *Check of relevancy of source updates.* It might be more efficient to filter out irrelevant updates before propagating them as the propagation algorithm will then deal with a smaller number of updates. Hence, we wish to propagate only relevant updates. In the relational context this mainly involves filtering out irrelevant updates by predicates [BLT86]. In XML, checking the relevancy of updates involves more than just checking predicates. An XML update is not relevant if

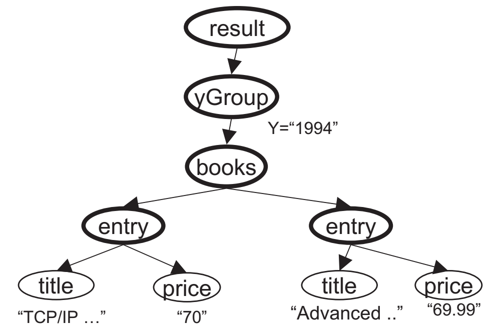


Figure 1.4: The expected result of refreshing the materialized view extent shown in Figure 1.2(b) in response to the source updates shown in Figure 1.3.

it does not have certain paths relevant to the query even if there are no predicates defined on these paths. We need a mechanism to define if an XML update is relevant or not to a view.

- *Check of sufficiency of source updates.* The update should have sufficient information so that it is even possible to propagate the update. This issue intersects with the relevancy of the update. For example, the update shown in Figure 1.3(b), that deletes a book node given its book title appears to be relevant to the view in Figure 1.2(a), yet it might have insufficient information to be propagated since the update provides only the "title" sub-element of the deleted book. One solution for this problem here is to provide the entire subtree of this deleted "book" node as part of the update. This solution is not practical when we consider updating nodes with huge subtrees. It might make the view maintenance inefficient due to the propagation of much unnecessary information.

- *Batching of source updates.* We typically wish to model updates in batches of possibly different types where such batch encodes only relevant updates using minimum yet sufficient information for propagation. In the relational context [GL95], each update in a batch of updates is independent from other updates. While in the XML context, an update may share the same prefix path with other updates possibly of different types.

**Supporting Update Propagation for Complex View Expressions.** Typically, even a simple XQuery expression contains fairly complex algebraic operations. For example, the XQuery expression shown in Figure 1.2(a) performs operations like navigation, unnesting, join, grouping, distinct, sorting, and result construction. We need a view maintenance solution that is general enough to support an expressive subset of the XQuery language. This includes supporting:

- *Simple Relational-like operations.* Even this class of operators is more powerful than its traditional relational counterpart as it deals with full fledged XML data rather than only flat tuples. In contrast to the traditional relational operators, where an update typically represents a flat tuple insert or delete, an update to an input source of a relational-like operator in the XML context might be inserting, deleting, or modifying a node anywhere in the hierarchy of XML data. The updated node itself might be an entire XML tree. For example, the view query shown in Figure 1.2(a) performs a join on book titles ($\$b/title$ = $\$e/b-title$) between "book" fragments (bound to variable

$\$b$) from the "bib.xml" document and "entry" fragments (bound to variable $\$e$) from "prices.xml" document. Propagating the modify update shown in Figure 1.3(c) involves updating a leaf node that is hidden deep in the processed "entry" fragments. One other complexity related to propagating updates through relational-like operations in the XML context is that such operators are order sensitive. We will discuss this issue in more detail later.

- *General queries involving arbitrary grouping and join operations.* Such class of queries is hard to maintain even in the relational context [GM05]. For example, propagating an update to an input source of a join operation, where such source had prior been grouped, might result in propagation of duplicate results. We will discuss this problem in more detail in Chapter 7. This problem is even harder in the XML context because of the nature of XML updates, as discussed above, and because the grouping operation in XQuery supports both cases when grouping is coupled with aggregation, like in the traditional relational context, and when the grouping is done without aggregation hence creating nested results. Propagating updates in groups of non-aggregated nodes is more complex than propagating updates through groups where each group has one flat aggregate value. Views with outer joins pose another challenge when maintaining them on updates to their input sources. See Chapter 7 for more detail.

- *XML-specific operations.* This type of operators exploit and manipulate the flexible structure of XML data. Restructuring and order are two main XML-specific capabilities that contribute to the difficulty of XML view maintenance. Propagating source updates through views that perform arbitrary result construction may require that we identify parts of previously constructed results. Palpanas et al. [PSCP02] classified XML constructor functions as non-distributive that may require recomputation. In other words, they place them in a class of opetrtors that may not be incrementally computable. The view query in Figure 1.2(a) shows an example of result restructuring. Propagating the modify update shown in Figure 1.3(c) to update the corresponding "price" node in the materialized view extent shown in Figure 1.2(b) for example requires that we identify ancestor nodes in the path from the root of the materialized view extent to that node. Such path contains newly constructed nodes.

**Supporting XML Order.** XML is an ordered data model, which is one important feature that sets XML apart from other data models. A source XML document has an implicit order among its nodes, called *document order*. For example, the relative order among nodes in each of the XML documents shown in Figure 1.1 shows a source document order. XQuery expressions return results that have a well-defined order. This order can be the implicit document order, a new order imposed by the XQuery expression, or a mixture of the two. XQuery can impose order in a variety of ways including (i) the use of *order by* clauses, (ii) the nesting of variable binding in the query $for$ and $let$ clauses, (iii) the order defined by XQuery $return$ clauses, and (iv) the order defined by new result constructions.

For example the XQuery view defined in Figure 1.2(a) returns a result that has a mixture of order semantics. (1) The query returns the constructed “yGroup” nodes ordered based on the book year value. (2) No specific order semantics is given to the child node “books” of “yGroup” nodes since there is only one “books” node for each “yGroup” node. (3) The children nodes of each “books” node should be returned in document order (with a major order that follows the order of the source “book” nodes and a minor order that follows the source “entry” nodes). (4) The query imposes an explicit order among the “title” and the “price” children nodes of each “entry” node. (5) Lastly, the order among nodes in each subtree of “title” and the “price” nodes (if there is any) should again follow document order.

XQuery operations (including relational-like algebra operators) thus must be order-aware. Hence, sorting operations might be needed to maintain the order of processed data at different stages of query execution. For example, in the example above, sorting operations might be needed to determine the order among “Ygroup” nodes or among “entry” nodes during query execution time. We wish to avoid such sorting of intermediate data during query processing time and to only perform partial sorting on the result when the final result is indeed required in an ordered manner. From the view maintenance point of view we wish to allow the relative order of the propagated update to be computed in a distributive manner, without having to be aware of order information with previously processed data. This implies that we wish to avoid materialization (or access to the source documents) for computing the relative order of propagated updates with respect to the order of previously processed data. This will allow propagated updates to be processed in a distributive manner with respect to order, hence achieving efficient maintenance of order-sensitive views.

**Incrementally Refreshing Materialized XML Views.** When refreshing the materialized XML view extent using propagated updates that represent the net effect of source updates, we encounter the following issues:

- Due to the powerful restructuring capabilities of XQuery views the XML result may have a totally different structure than the underlying source(s) that were used to construct it. Applying a propagated update to the materialized XML view requires a mechanism to identify the correct location where the update should be applied. Consider for example that we wish to refresh the view extent in Figure 1.2(b) on the source update in Figure 1.3(a), that inserts a new "book" to "bib.xml". The propagation of this source update should create a delta update that inserts a new "entry" element into the view extent. This "entry" element should be inserted into a particular location (as a child of the "books" element that has a parent "yGroup" element with attribute "Y" equal to "1994") and also in a particular order (after the existing "entry" element). These issues are not encountered in the relational context since applying propagated updates to relational materialized views simply involves inserting or deleting tuples from the materialized flat tuple set (a table) where neither order nor particular location is an issue.

- Delete updates are generally harder to handle than insert updates [GM95]. Deleting a source tuple may not necessarily translate into a deletion of the tuple(s) derived from it in the view extent, because a tuple in the view extent may have multiple (and thus alternative) derivations from source tuples. Such derivation issue appears even in simple relational $SPJ$ views [BLT86, GMS93]. This problem becomes harder when we consider XML views that typically have more expressive power as we have discussed earlier. For example, a constructed "yGroup" node in the view shown in Figure 1.2(a) might be representing more than one source "book" node. Hence, a delete of a source "book" should not delete the "yGroup" it falls under unless there are no other "book" nodes grouped under that "yGroup" node.

- Deleting a source node might cause the deletion of an entire subtree from the view extent. For example, the delete update shown in Figure 1.3(b) deletes the only "book" node grouped under the second "yGroup" node in the view extent in Figure 1.2(b). Hence, the correct propagation of such update requires the deletion of not only the "entry" node it maps to in the view extent but also the deletion of the entire XML fragment rooted at the "yGroup" node with attribute "Y=2000". An efficient treatment of such update should delete this entire fragment directly from its root rather than deleting all descendant nodes of that root node one by one before figuring out at the end that the root node "yGroup" should be deleted, like what is done for example in [LD00].

## 1.3 State-of-The-Art in View Maintenance

The incremental maintenance of materialized views has been extensively studied for relational databases [BLT86, GM95, GL95, CW91, GMS93, GL95, MQM97, MK00, PSCP02, GM05]. Blakeley et al. [BLT86] proposed a differential solution that was focused on the view maintenance problem of SPJ views. Griffin and Libkin [GL95] extended the solution in [BLT86] considering more algebraic operations and providing support for views with duplicates. Ceri and Widom [CW91] proposed a solution for maintaining a subset of SQL views. Some solutions [GMS93, MQM97] focused on views with aggregation. Yet, these solutions are limited to views with one aggregation operation that comes as the last operator in the expression tree. Quass [Qua96] has considered general views with aggregation. Griffin and Kumar [GK98] proposed a solution for maintaining outer join operations. Gupta and Mumick [GM05] have proposed an efficient solution for maintaining general views with aggregation and outer join operations.

To a lesser degree, view maintenance has also been studied for object-relational and object-oriented views [KR98, LVM00, AFP03]. Most of the solution for maintaining object-oriented views considered non-standard models or supported only limited views. Ali et al. [AFP03] proposed a solution for maintaining a large class of the standard object query language OQL.

Little work has been done on the incremental maintenance of XML and semi-structured views. Early solutions for maintaining semi-structured views [AMR$^+$98, ZGM98] have considered only a limited class of views.

These solutions require materialization of large auxiliary data structures. [LD00] proposed a solution for maintaining views defined using a restricted subset of the XML-QL language. It places additional limitations on the supported language. For example, it does not support explicit union operations and complex nested queries and places restrictions on updating some source values. It also provides an inefficient treatment for delete updates.

In [ESWDR02] we have proposed the first solution for supporting incremental maintenance of a subset of XQuery using a set of well-defined update primitives. This solution requires materialization of some intermediate results. Quan et al. [QCR00] proposed a solution for maintaining XQL views. Their solution uses auxiliary data of size that depends on the source data size. Sawires et al. [STP$^+$05] proposed a solution for maintaining a subset of the XPath expressions. Their solution uses auxiliary data that depends on the expression size and the answer size and does not depend on the source data size. Bohannon et al. [BCF04] proposed two solutions for the incremental evaluation of ATGs, a formalism for schema-directed XML publishing. Their solution considers only XML views defined over flat relational tables and does not support full fledged XML sources.

In all the work above XML order was not considered. In [DESR03], we proposed an extension to [ESWDR02] that supports order-sensitive view maintenance of a subset of XQuery views. Both solutions [ESWDR02] and [DESR03] were based on propagation rules that require special-purpose processing for propagating updates. This does not follow the mainstream framework for view maintenance that uses the query engine to propagate updates.

## 1.4 Our Approach

We now propose a comprehensive framework for solving the problem of maintaining XML views defined in XQuery. Our solution supports an expressive subset of XQuery views including $XPath$ expressions, $FLWOR$ expressions, and *Element Constructors*. We support XML order including source document order and query imposed order. Our solution avoids intermediate result materialization and avoids accessing source documents during view maintenance time for most of the views. Hence, the majority of our views becomes self-maintainable. We take an algebraic approach for propagating updates. Our proposed approach generates incremental maintenance plans in the same language used to define the view, just like view maintenance strategies used in mainstream database systems, like DB2 [LSPC00]. This makes implementing and integrating the view maintenance solution with the current XQuery processing engine an easy task. In fact this should facilitate the adaptation of our XML view maintenance solution within future commercial XML engines. We also provide support for bulk update processing of heterogeneous mixtures of update types.

### 1.4.1 View Maintenance Framework

We adapt a framework similar to that used in mainstream commercial database systems that support view maintenance where the view maintenance is done in two phases called the *Propagate Phase* and *Apply Phase*. We add one phase that we call *Validate Phase* that we find essential for XML view maintenance. We call our framework the $VPA$ view maintenance frame-

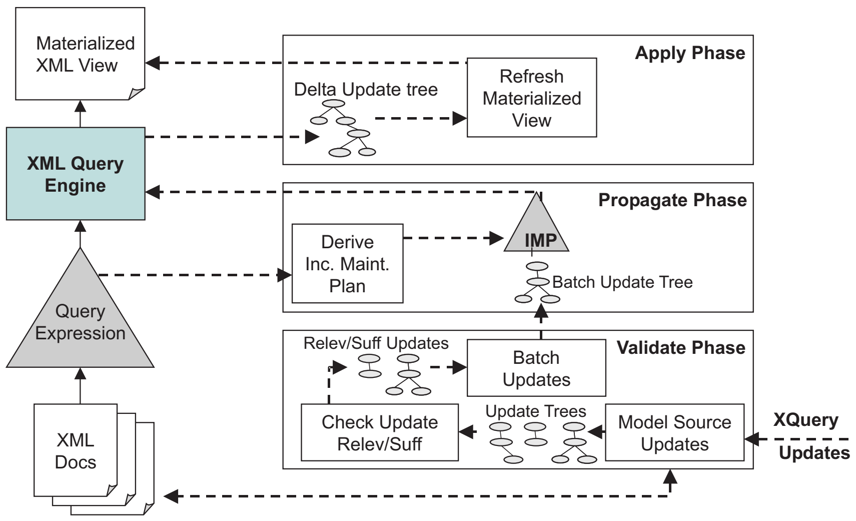


Figure 1.5: Our $VPA$ view Maintenance Framework.

work. Figure 1.5 shows these three phases. We now briefly present these three phases in our solution.

1). The *Validate Phase*. We define how a source XML update is modeled, namely using a structure called *update tree*. An *update tree* encodes hierarchy and order information of the source updates and also their type. The relevancy of each update tree with respect to its potential effect on the view is verified. We also determine if the source update has sufficient information for propagation. Relevant updates with sufficient information are then batched in a structure called *batch update tree* and are made available for update propagation.

2). The *Propagate phase*. The most important task in this phase is to derive *Incremental Maintenance Plan*s (**IMP**s) from the view query. Batch update trees are processed using the *Incremental Maintenance Plan*s to generate propagated updates, called *delta update trees*. Delta update trees are to be used in the next phase to incrementally refresh the view extent. IMPs are expressed in the same algebraic language used in computing the materialized view extents. Hence, they are processed using the XML query engine used to generate the view extent.

3). The *Apply Phase*. In this final phase, delta update trees that had been computed in the *Propagate Phase* are applied to the materialized view to refresh it. This involves merging nodes in the delta update tree with nodes in the materialized view and performing any necessary insertion, deletion, or modification to the materialized view.

### 1.4.2 Proposed Solutions

Our work relies on enabling a basic property in views, called the *distributive property*. The distributive property of a view in the relational context is typically defined over the union operator ($\cup$) [BLT86]. For example a select view defined over a source $R$ is distributive because for a source update $\triangle R$ the equation $\sigma_p(R \cup \triangle R) = \sigma_p(R) \cup \sigma_p(\triangle R)$ holds. That is, by processing the view over the update and combining the result $\sigma_p(\triangle R)$ with the initial materialized view $\sigma_p(R)$ we get the same final result that we would get if we were to fully recompute the view over the updated source $\sigma_p(R \cup \triangle R)$.

To enable the *distributive property* for the class of XQuery views that we support, we need to provide mechanisms for (i) supporting distributive processing of XML data and incremental construction of XML results, (ii) supporting the distributive property of views in XQuery order-aware environment, and (iii) supporting the distributive property of views on delete updates when the view extents contain nodes that have multiple derivations from the underlying sources.

To support the XML view maintenance framework presented above, this dissertation proposes the following solutions:

**An Efficient Solution for Supporting Order in XML Query Processing and View Maintenance**

We propose a solution [ESDR03, ESDR05] that addresses issues related to handling order in the XML context. This includes the variety of order requirements of the XQuery language and the need to maintain order in the presence of updates to the XML data. Our solution is based on a special order encoding for XML nodes. One important effect of this technique is that it removes the overhead for each individual algebra operator to have to maintain order. It also removes the need for unnecessary sorting of intermediate data. In other words it migrates the ordered bag semantics of intermediate data into non-ordered bag semantics. This opens up opportunities for optimization, since operators become free to manipulate the data they process in any efficient way they wish with no regards to the order of that data. Our approach enables efficient order-sensitive query processing and incremental view maintenance. See Chapter 3 for more detail.

### A Technique for Enabling Incremental Fusion of XML Fragments through Semantic Identifiers

We have studied the problem of how to fuse XML pieces (fragments) generated by incrementally processing XML data into XML results. We propose an identifier-based solution for this problem [ESRM05a]. This solution assigns semantic node identifiers to nodes in XML results. A semantic identifier for an XML node encodes both lineage and order information in a compact manner. A semantic identifier of a node in the XML view extent is reproducible. This means that for any node in the XML view extent, any propagated updates that would affect that node would be assigned the same semantic identifier. Semantic identifiers enable many XQuery views to be distributive. Hence, they provide a base for our view maintenance solution. See Chapter 4 for more detail.

### A Mechanism for Validating Source XML Updates

We model XQuery source updates [TIHW01] as a set of well defined update primitives, called *update trees*. An update tree specifies the hierarchy and order information of the update. We define a mechanism for checking relevancy of source updates through the use of a special pattern tree, called *Source Access Pattern Tree* ($SAPT$). We also use $SAPT$ to determine if the source update is relevant or not to the view. Irrelevant updates are discarded. Hence, we prevent unnecessary update propagations. Updates that are potentially relevant to the query are annotated with any missing information that may be required to enable successful propagation. This includes any other nodes needed by the query. This additional information allows the update to contain sufficient information for propagation. Such sufficient update should be ideally be minimum to achieve efficiency as realized through a small number of nodes in the update being propagated and a faster application of the propagated updates to the view extent being achieved. Lastly, different update trees are batched together into a structure called the *batch update tree*. See Chapter 5 for more detail.

**A Counting Solution for Supporting XML Delete Updates**

Views may not be distributive on delete updates due to possibility of nodes in the view extent having multiple derivations from source node. We propose a counting solution for solving this problem. Our counting solution is an extension to the counting algorithm in [BLT86, GMS93]. Our solution annotates every XML node with a count representing the number of derivations of that node from source data. We define rules on how this count annotation is computed for different query operations. Our counting solution allows efficient deletion of large XML fragments from the XML view. See Chapter 6 for more detail.

**An Algebraic Solution for Propagating Source XML Updates**

We propose an algebraic solution for propagating updates. In contrast to our previous work [ESWDR02, DESR03] we now generate incremental maintenance plans that can be executed using the XML query engine. Our solution defines algebraic propagation equations for propagating updates through different algebra operators. We use these equations to derive the incremental maintenance plans from the view query definition. Executing the incremental maintenance plans produces *delta update tree*s that are to be used to refresh the XML view extents. Our propagation solution supports a large class of views including *XPath* expressions, *FLWOR* expressions, and *Element Constructors*. It supports many complex queries including queries with nested sub-queries and general queries with arbitrary grouping and join operations and queries with left outer joins. See Chapter 7 for more detail.

### An Efficient Solution for Refreshing XML View Extents

We address the issue of how to refresh materialized XML views using delta update trees resulting from the propagation phase. We utilize a special operator, the *Deep Union*, as the refresh operator. Our solution offers an efficient apply phase where nodes in the materialized XML view are updated in a top-down fashion. In fact, an entire fragment can be deleted from the XML materialized view by directly disconnecting its root from the XML materialized view rather than having to first delete descendant nodes of that root one-by-one.

We prove the correctness of our proposed view maintenance approach, in particular that using our mechanism we can correctly refresh materialized view extents. We also provide the results of extensive experimental evaluation of our solution that we have obtained using a prototype implementation of our system. The results of our experiments confirm that our solution provides a practical and efficient framework for maintaining materialized XML views. See Chapter 8 for more detail.

## 1.5 Outline

The rest of this dissertation is organized as follows. Chapter 2 defines the XML query model that we use. Chapter 3 discusses our order solution. Chapter 4 discusses our semantic identifier solution. Chapter 5 discusses how we model and validate source XML updates. Chapter 6 introduces our counting solution for supporting XML delete updates. Chapter 7 discusses our algebraic solution for propagating XML updates. Chapter 8 discusses how to apply propagated XML updates to the materialized XML view extents to refresh it. Chapter 9 presents and analyzes the result of experimental evaluation. Chapter 10 discusses related work. Lastly, Chapter 11 provides a summary of the contributions of this dissertation and discusses future work.

# Chapter 2

# Background

## 2.1 XQuery

We consider the subset of the XQuery language [W3C05] specified by the grammar shown in Figure 2.1. This subset includes XPath expressions, nested FLWOR expressions, and element constructors. In addition we support some functions like the distinct-value function and some aggregate functions. We do not support queries that evaluate predicates on collections (like for example comparison operations between sequences) or on results of functions (like for example predicates over the result of aggregate functions or the position function). We support XPath expressions involving only the most used axes in practice; the child "/" and the descendant "//" axes.

```
FLWORExpr          ::= (ForClause | LetClause)+WhereClause?
                       OrderByClause? ReturnClause
ForClause          ::= "for" $VarName in ExprSingle
                       (, $VarName in ExprSingle)*
LetClause          ::= "let" $VarName := ExprSingle
                       (, $VarName := ExprSingle)*
WhereClause        ::= "where" ComparisonExpr
OrderByClause      ::= "order" "by" OrderList
ReturnClause       ::= "return" PrimaryExpr
PrimaryExpr        ::= Literal | $VarName | Expr |
                       DirElemConstructor
Expr               ::= ExprSingle (,  ExprSingle)*
ExprSingle         ::= FLWORExpr | XPathExpr
DirElemConstructor::= "<" QName AttributeList ("/>" |
                       (">" PrimaryExpr* "</" QName">"))
```

Figure 2.1: Syntax of XQuery Subset

## 2.2 The XML Algebra XAT

Given that to date no one standard XML algebra for query processing purposes has emerged that has been widely accepted, we use the XML algebra called *XAT* [ZPR02][1] implemented in the Rainbow engine [Zea03]. Figure 2.2 shows an algebraic representation for the XQuery expression in Figure 1.2(a) using the XAT algebra. We will discuss next the data model of this algebra and give an overview of its operators.

### 2.2.1 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 $T$ is an order-sensitive table

[1]This algebra is similar to NAL [MHM04] xatTreeand SAL [BT99] algebras.

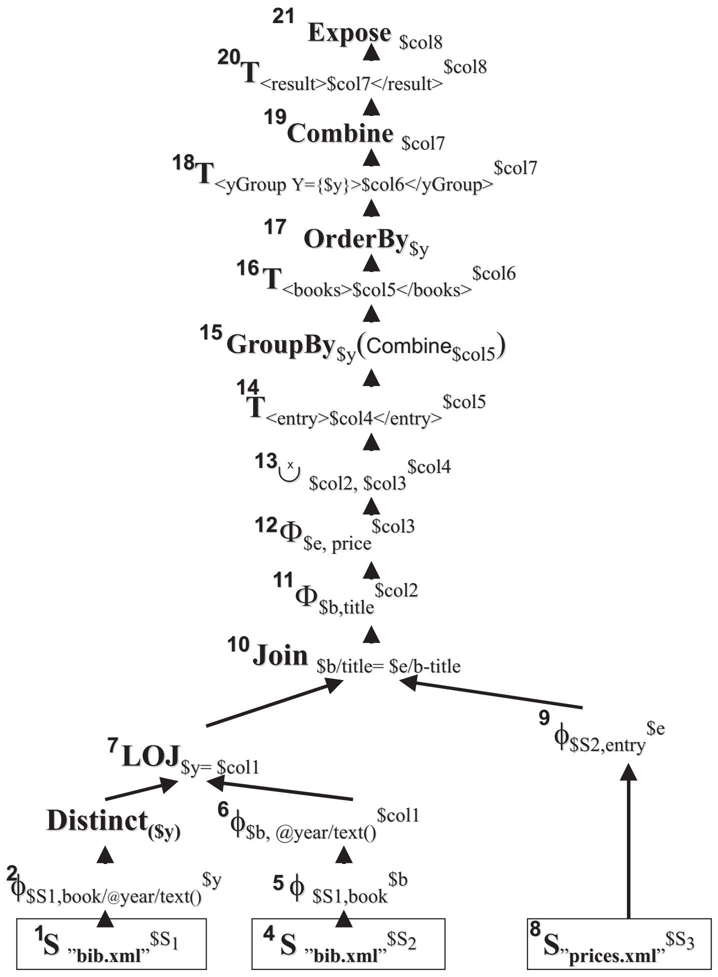

21 Expose $col8
20 T <result>$col7</result> $col8
19 Combine $col7
18 T <yGroup Y={$y}>$col6</yGroup> $col7
17 OrderBy $y
16 T <books>$col5</books> $col6
15 GroupBy $y (Combine $col5)
14 T <entry>$col4</entry> $col5
13 $col2, $col3 $col4
12 Φ $e, price $col3
11 Φ $b,title $col2
10 Join $b/title= $e/b-title
9 φ $S2,entry $e
7 LOJ $y= $col1
Distinct ($y)
6 φ $b, @year/text() $col1
2 φ $S1,book/@year/text() $y
5 φ $S1,book $b
1 S "bib.xml" $S1
4 S "bib.xml" $S2
8 S "prices.xml" $S3

Figure 2.2: An Algebra Tree for the XQuery expression in Figure1.2(a)

of $p$ tuples $t_i$, $1 \leq i \leq p$, $p \geq 0$ that is $T = \{t_1, t_2, .., t_p\}$. The column names in an XAT table schema of $T$ 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 $t_i$ (1 ≤ i ≤ p) is a sequence of $m$ cells $c_{i,j}$ (1 ≤ j ≤ m) that is $t_i = [[c_{i,1}, c_{i,2}, ..., c_{i,m}]]$, where $m$ corresponds to the number of columns. Each cell $c_{i,j}$ (1 ≤ i ≤ p, 1 ≤ j ≤ m) with $col_j$ in a tuple $t_i$, denoted by $t_i[col_j]$, can store an XML node or a sequence of nodes bound to column $col_j$. Atomic values are treated as text nodes. During XML query evaluation algebra operators process input XML nodes stored in cells of XAT tables.

### 2.2.2 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 2.2[2]. Below we introduce the core subset of the XAT algebra [ZPR02].

The relational subset of the XAT algebra includes *Select* $\sigma_c(T)$, *Cartesian Product* $\times(T_1, T_2)$, *Theta Join* $\bowtie_c (T_1, T_2)$, *Left Outer Join* $⟕_c(T_1, T_2)$, *Distinct* $\delta_{col}(T)$, *Group By* $\gamma_{col[1..n]}(T, func)$ *Order By* $\tau_{col[1..n]}(T)$, and the column renaming operator *Name* $\rho_{col1,col2}(T)$, where $T$, $T_1$, and $T_2$ denote XAT tables.

These operators are equivalent to their relational counterparts with the added responsibility of maintaining order. All operators above, except the

[2]We discuss the details of algebra tree execution later in this document.

*Distinct*, *Group By*, and *Order By* operators reflect the relative order of their input XAT tables to their output XAT tables. We discuss order semantics of different operators in more detail in Chapter 3.

Note that the XAT *Group By* operator is more powerful than its relational counterpart. The *Group by* operator in relational algebra only allows aggregation (on the non-grouping columns). While the XAT *Group By* operator allows other operations and functions as well as aggregation, such as *Combine* operator. In this work, we mainly consider the parameter $func$ to be a the *Combine* operator or an aggregate function. The XAT *Group By* my also perform grouping by values or by node identifiers.

**Source** $S_{xmlDoc}^{col'}$ is 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 $tout_1 = (c_{1,1})$, where $c_{1,1}$ is the XAT table cell that contains a reference to the entire XML document.

**Navigate Unnest** $\phi_{col,path}^{col'}(T)$ unnests the element-subelement relationship through a navigation followed by an unnest. For each tuple from the input XAT table $T$, it creates a sequence of output tuples in which $path$ navigates to. The $\phi_{\$S1,book}^{\$b}$ operator in Figure 2.2 (node $\#5$) generates one tuple for each "book" element extracted from the "bib" element in the input. Tuples in the output XAT table of the *Navigate Unnest* operator are generally ordered by major order on entry point nodes and a minor order on the destination nodes.

**Navigate Collection** $\Phi_{col,path}^{col'}(T)$ is similar to *Navigate Unnest*, except it only performs the navigation functionality without the unnesting functionality. It extracts a collection from each node in column $col$. For each tuple from $T$, it creates one output tuple containing the result of navigating through $path$. The $\Phi^{\$col2}_{\$b,title}$ operator in Figure 2.2 (node $\#11$) generates one tuple for each input XAT table tuple. This results in extracting two collections of "title" elements, one for each "book" in column $\$b$. Tuples in the output XAT table of the *Navigate Collection* operator are ordered based on only the order of the entry point nodes.

**Combine** $C_{col}(T)$ groups the content of all cells in column $col$ into one sequence (with duplicates). Given the input $T$ with $m$ tuples $tin_i$, $1 \leq i \leq m$, *Combine* outputs one tuple $tout = (c)$, where $tout[col] = c = \overset{\circ}{\biguplus}{}^{m}_{i=1} tin_i[col]$ [3]. $Combine$ has only column $col$ in its output XAT table. The $C_{\$col7}$ operator in Figure 2.2 (node $\#19$) groups all the "yGroup" elements in \$col7 tuples into one cell. Hence, there is only one row in its output XAT table.

**Tagger** $T^{col}_{p}(T)$ creates a new column $col$ in which it 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 [W3C98] with a parameter being a column name, e.g., *<result>\$col7</result>*. For each tuple $tin_i$ from $T$, 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 $tin_i$. For example, the $T^{\$col5}_{<entry>\$col4</entry>}$ in Figure 2.2 (node $\#14$) constructs a new "entry" node from the contents of column $\$col4$ for each input tuple (containing "title" and "price" nodes previously unioned). The *Tagger* does not build the result hierarchy; instead the result structure is built by a sequence of grouping/nesting, union, and tagging operations.

[3] $\overset{\circ}{\biguplus}{}^{m}_{i=1}$ is the order sensitive bag union.

**XML Unique** $\upsilon_{col}^{col'}(T)$ removes duplicate from sequences of XML nodes by node identifier. For each tuple $tin_i$ from $T$, it creates one output tuple $tout_i$, where $tout_i[col']$ is a sequence containing the unique members in $tin_i[col]$ after removing duplicates by node identifier.

**XML Union** $\overset{x\,col}{\cup}_{col1,col2}(T)$ is used to union multiple sequences into one sequence (duplicates are not eliminated). For each tuple $tin_i$ from $T$, it creates one output tuple $tout_i$, where $tout_i[col]$ is a sequence containing the members of the set $tin_i[col1] \cup tin_i[col2]$. Note that the operator *XML Union* performs set operations on columns in a single XAT table, not on multiple XAT tables.

**Expose** $\epsilon_{col}(T)$ exposes the specific column $col$ into XML fragments or XML documents in text format. It appears as the root node of an algebra tree[4].

The **Map** operator $Map_{a:e(Attr)}(T)$ is used to simplify the translation from XQuery FLWOR expressions to the XAT algebra. It directly represents the nesting of XQuery expressions. The *Map* operator is a binary operator with a left hand side (LHS) input defining one for-variable and a right hand side (RHS) input defining an algebra expression $e$ (a tree or a directed acyclic graph). $Attr$ represents the for-variable in the FLWOR expression and $a$ is the new attribute name whose value is calculated from expression $e(Attr)$. Intuitively the *Map* operator forces a nested loop evaluation strategy. At the algebraic level optimization all *Map* operators are rewritten; hence they are removed.

[4]For simplicity of presentation, we do not show the *Expose* operator in the later algebra tree figures.

The **Merge** operator $M(T_1, T_2)$ merges two XAT tables vertically into one XAT table by concatenating columns. This operator merges results of two independent sub-expressions in the query into one XAT table, hence a combined result can be created from the merged XAT table (using *XML Union* operator). Each input XAT table of the *Merge* operator is typically generated using a *Combine* operator, hence it has one tuple that stores a sequence of nodes.

## 2.3 XAT Generation

The XAT generation is also called the "Query Decomposition" phase of query processing. This phase includes the following steps:

- *Parsing.* The query is lexically and syntactically analyzed using the parser. We use here the Kweelt engine parser [Sah01].
- *Normalization.* The query is converted into a normalized format that can be easily manipulated. We will discuss the normalization rules used in Rainbow later this section.
- *XPath.* Appropriate operators are generated for each XPath expression.
- *FLWOR Expressions.* By using the $Map$ operator, all operators generated for XPath expressions are connected to the query plan to form an XAT algebra tree. Through the use of Directed Acyclic Graph (DAG), common XPath expressions can be eliminated.

### 2.3.1 XQuery Normalization

Prior to translating any XQuery expression into the XAT algebra expression, we use source-level normalization similar to that used in [MFK01b]. We apply the following normalization rules:

- *Normalization Rule 1:* The *Let* clause in an XQuery expression defines let-variables that can be used later in the query block. The let-variables are treated as temporary variables. During normalization, they can be eliminated: the expression binding the let-variable is substituted for all occurrences of the let-variable. Note that in the implementation, the let-variable is calculated only once and is shared among all the occurrences. This would turn the algebraic query plan into a DAG.

- *Normalization Rule 2:* Each *for* clause is represented as a *Map* operator[5]. Since the *Map* operator is binary, the *for* clause defining more than one for-variable would first be split into a sequence of nested *for* clauses. Each clause defines one for-variable only. For example, the following For clause:

  for $\$b$ in doc("bib.xml")/bib/book, $\$e$ in doc("prices.xml")/prices/entry

  can be split into:

  for $\$b$ in doc("bib.xml")/bib/book,

  for $\$e$ in doc("prices.xml")/prices/entry

[5]This operator is later removed during the query optimization phase and before query execution.

- *Normalization Rule 3:* To simplify the translation of XPath expressions into the *Navigation* operator in XAT, we substitute the predicate in an XPath which refers to outer variables by a *where* clause in a FLWOR query block. Since such predicates have existential semantics in both cases, the original XPath semantics are not changed. After normalization, the XPath expression must have a variable or a document as its entry point. Also, it will not refer to any other variables in its navigation path and node tests.

### 2.3.2 Converting XPath Expressions into XAT

An XPath expression is composed of location steps (with axis and node test), predicates, and optional parenthesis for grouping. An XPath is represented by a combination of navigation, select and grouping operations.

#### XPath Expression without Predicates

A path expression with location steps that have no predicates can be represented using an navigation operation. In the XQuery example in Figure 2.1(a), the XPath expression $/bib/book$ that navigates from "bib.xml" and binds result to variable $\$b$ is represented as $\phi^{\$b}_{\text{``}bib.xml\text{''},/bib/book}$.

#### XPath Expression with Predicates

A predicate in an XPath expression is represented as a selection operator after the navigation operator. In general, an expression $E1[E2]$, it is translated into the XAT expression $E2(E1)$. For example, the XPath expression

$/book[title ='' \ Data \ on \ the \ Web'']$, where *book* is navigated from an entry point $S$, is translated into:

$$\sigma_{y=''Data \ on \ the \ Web''}\left(\phi^{y}_{x,title}\left(\phi^{x}_{S,book}()\right)\right).$$

### 2.3.3 Translating Normalized XQuery Expressions to XAT Algebra

Normalized XQuery expressions are translated into their corresponding XAT algebra representations in two steps: (1) translating XPath expressions and (2) translating the FWOR query expressions (now without the Let clause).

The translation pattern of a flat FWOR query block to the XAT algebraic expression is illustrated in Fig. 2.3. A nested XQuery block can be translated recursively using this pattern. In this translation pattern, the *Map* operator introduces one for-variable for the *for* clause in the LHS expression. This for-variable can be referred to in the nested query blocks in the RHS. The *Combine* operator on top of the $Map$ is used to construct a sequence of all intermediate results. The *where* clause is also placed in the LHS of the $Map$ operator, just like the $orderby$ clause.

Algebraic operators are generated during the translation from an XAT algebra tree. Sharing of common subexpressions (e.g., the let-variable expression) is allowed among multiple operators. This turns the XAT tree into a DAG. For simplicity, we do not emphasize the difference between them and just generally call them XAT trees.

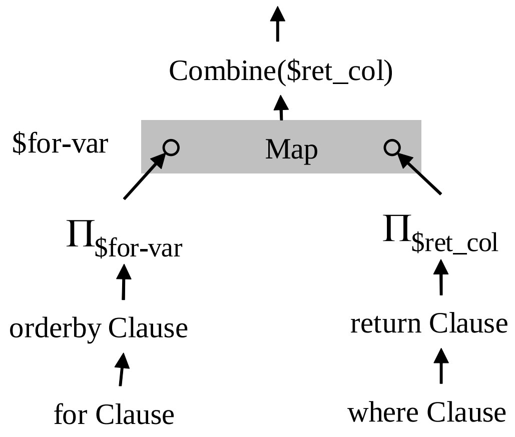


Figure 2.3: Building the Algebra Tree for an XQuery FWOR Expression.

## 2.4 XAT Optimization

After XQuery normalization and translation, the correlation in an XQuery expression is represented in the XAT tree by the *Map* operator and *linking* operators (operators in the inner query blocks referring to variables in the outer FLWOR query block). The $Map$ operator introduces the for-variable from the LHS *for* clause and the linking operator refers to it in the RHS. Intuitively the *Map* operator forces a nested loop evaluation strategy. We use an XAT decorrelation algorithm that removes the $Map$ operator in the XAT tree. This is done by pushing the $Map$ operator along the RHS until the linking operator is reached. At this point the $Map$ operator is rewritten as a $Join$. In our system, we use an optimizer that is based on the work done in [ZPR02] and [WRM05].

# Chapter 3

# Efficiently Supporting Order In XML Query Processing and View Maintenance

## 3.1 XML and Order

Unlike most common data models including semi-structured, relational and object-oriented data models, XML data is order-sensitive. Supporting XML's ordered data model is crucial for many domains. An example is content management where document data is intrinsically ordered and where queries often need to rely on this order [TVB$^+$02]. For example, if Shakespeare's plays are modeled as XML documents, the order among acts in plays is relevant. Then queries asking for a certain act in a play given its order must be supported.

XQuery expressions return results that have a well-defined order, unless specified otherwise. The result of a path expression is always returned in document order [W3C99]. The order in the result of a $FLWOR$ expression can in addition be imposed by the expression itself in many ways, as we will describe next. Hence, the result of an XQuery expression reflects in an interrelated manner both the implicit XML document order and the explicitly imposed order by the XQuery expression.

Support for XML order when processing XQuery queries can severely affect query optimization opportunities. Thus, a major performance hit may result [W3C05]. For this reason the XQuery language provides a function, named $unordered()$, that can be used for those expressions where the order of the result is not significant [W3C05]. This allows us to turn sequences processed during query execution into sets. Set-oriented processing is known to offer potential opportunities for optimization.

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

The problem of handling order poses unique challenges to incremental XML view maintenance. XML views have to be refreshed correctly not only concerning the view content but also concerning the order of the view result document. In the relational context, for example, order is of interest only if the *Order By* operation is explicitly present in the view definition.

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

Given that the order cannot always be ignored, efficient techniques for handling order in XML query processing must also be devised. That is, we need to have the ability to support order for processing queries and updates on data and on materialized views. At the same we need to minimize the overhead that comes with handling order.

Some work has been proposed for supporting order in XML query processing [FLSW03, JAKC$^+$02, MFK01a, TVB$^+$02] yet these solutions did not support all types of XQuery order or came with high overhead cost. See Section 10.2 for more details on related work.

## 3.2 Challenges of Handling Order in XML Query Processing

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

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

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

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

The processed nodes themselves may be either original nodes from the source document or nodes constructed during query execution. And the order defined for any of those three levels may follow the source document order or may follow a new order imposed by the query. In some cases order might not even be of importance.

**Challenges Posed by the Different Order Requirements of the XML Query Language.** We classify the order that an XQuery expression can reflect on its result into four main types:

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

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

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

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

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

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

- The operator *Navigate Collection* $\Phi^{col'}_{col,path}(T)$ processes one tuple at a time, without requiring to access other tuples nor modifying the order among the tuples. Moreover, for each tuple in the input table it produces exactly one tuple in the output table.

- The operator *Tagger* $T_p^{col}(T)$ also preserves the relative order among the tuples it processes. In addition it defines order among its internal nodes.

- The operator *Combine* $C_{col}(T)$ destroys the order among the tuples it processes. It groups all the nodes from its input column in one cell and outputs only one tuple in the output XAT table that contains that cell. This raises the issue of maintaining order between those nodes.

- For the *Join* operator $\bowtie_c (T_1, T_2)$, the order of tuples in the output table of the *Join* operator depends on the order of tuples in its input tables. The order in its output table follows the order of the left input table $T_1$ as a major order and the right table $T_2$ as a minor order.

- The *Order By* operator $\tau_{col[1..n]}(T)$ destroys the order of the input table and imposes a new order based on a certain criteria. Hence the output table will have a new computed order based on the order of some column values.

- The operator $Expose$ $\epsilon_{col}(T)$ outputs an XML document rather than an XAT table. This document is extracted as a tree from a $col$ in the input XAT table. The extracted tree has order among its elements that reflects all previous order decisions.

We will discuss how different operators handle order in more detail in Section 3.3.

**Challenges Posed by Order-sensitive View Maintenance.** The problem of the incremental maintenance of XML views poses additional challenge. View maintenance of ordered XML data is difficult for two reasons [LD00]: (1) positions of the element may change dynamically during update time and (2) positions of elements may be different in views and in the source data.

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

## 3.3 Maintaining XML Order

The requirement of preserving order, as described in Section 3.2, makes the XML query execution and view maintenance significantly different from the relational case. The two obvious solutions are: (1) relying on the physical sequential storage medium to be always ordered, or (2) assigning order values to processed sequences and nodes. Both solutions are not efficient especially in the presence of incremental updates.

Our solution for handling order relies on three main principles: (1) the underlying *Storage Manger* is capable of returning source document nodes in document order, (2) order is ignored when processing XML intermediate results, and (3) at the end of query processing and when generating the final result sorting is performed (typically only partial sorting) to return the result in the desired order. Our *Storage Manager* relies on the *MASS* system [DR03], also developed at WPI, for providing scalable storage and indexing for XML data with efficient update performance. The *Storage Manager* provides interfaces for storing and retrieving XML nodes (both original nodes and constructed nodes). *MASS* guarantees that when retrieving descendants of original XML nodes they are returned in document order, eliminating the need for sorting them at the result generation time. *MASS* provides scalable I/O performance for all *XPath* axes. Moreover, it provides an integrated indexing support for *XPath* node tests, position predicates and count aggregations.

### 3.3.1 Node Identifier and Node Order

In many cases the order among processed XML nodes and collections depends on the order in their source document. In other words, the order among the tuples in an XAT table (and among nodes in a cell) depends on the source document order of the XML nodes present in these tuples (cell). Our query processing model uses node identifiers during query execution time. Hence, a node identity that serves the dual purpose of node identifier and order encoding is beneficial for both query processing and order handling. We also require the node identity of a base node to encode the unique path of that node in the XML tree and to capture the order at each level along the path. We have thus considered techniques proposed in the literature for encoding order in XML data [AKJK$^+$02, DR03, JAKC$^+$02, TVB$^+$02]. The lexicographical order encoding technique proposed in [DR03] that does not require reordering on updates is used. It is analogous to the Dewey ordering [TVB$^+$02], except rather than using numbers in the encoding, it uses variable length strings. First, for each document node a variable length byte string key is assigned, such that lexicographical ordering of all sibling nodes yields their relative document ordering. The identity of each node is equal to the concatenation of all keys of its ancestor nodes and of that node's own key (see Figure 3.1). This order-reflecting node identity encoding is called **FlexKey**. We use the notation $k_1 \prec k_2$ to note that *FlexKey* $k_1$ lexicographically precedes *FlexKey* $k_2$.

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

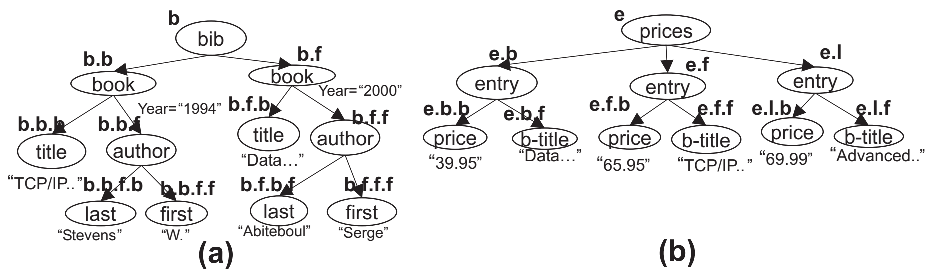


Figure 3.1: Lexicographical order encoding of the two XML documents "bib.xml" and "prices.xml" presented in Figure 1.1.

- It identifies a unique path from the root to the node. Hence the parent-child and ancestor-descendant containment relationships between nodes can easily be determined without the need to access the actual data. Accessing this relationship is a frequent operation in XML query execution.

- It embeds the relative order among nodes in the same XML tree. Hence the order between any nodes can easily be determined (regardless of the level) by comparing their *FlexKey*s lexicographically.

- It does not require reordering on updates because of the use of variable length strings instead of numbers for encoding order. We can always create new gaps by extending the string by adding more letters. We discuss that in more detail in Section 3.4.

**Base Nodes.** We use *FlexKey*s for encoding the node identities of all nodes in the source XML document. That is, we assume that any given XML document used as source data has *FlexKey*s assigned to all of its nodes. For reducing redundant updates and avoiding duplicated storage we mainly store references (*FlexKeys*) in the XAT tables rather than actual XML data. This is sufficient as the *FlexKeys* serve as node identifiers and also capture the order. From here on, when we refer to a cell in a tuple we mean the *FlexKey* or the collection (or sequence) of *FlexKeys* stored in that cell. The actual XML data is stored only once in the *Storage Manager*. Figure 3.2 illustrates the usage of *FlexKeys* as references to source XML nodes. *FlexKeys* are used for accessing that data when needed by some operator. For example, the *Navigate Unnest* operator $\phi^{\$n}_{\$S1,book}$ in Figure 3.2 retrieves the "book" children of the root node of "bib.xml" from the *Storage Manager*, and places their *FlexKeys* in the output XAT table.

**Constructed Nodes.** We also use *FlexKeys* to encode the node identity of any constructed nodes either in the intermediate result or in the final extent. The *FlexKeys* assigned to constructed nodes are locally unique. We postpone discussion on id generation and their semantics to Chapter 4. Rather than instantiating the actual XML fragments in our system, we only store a skeleton representing their structure in the *Storage Manager*. References (*FlexKeys*) to other source data or to constructed nodes that are included in the newly constructed node are kept. For example, in Figure 3.2, although the constructed node $T7$ is representing the whole output view extent, it is only stored as <*result*>*T5 T6*</*result*>. When the constructed node is created, the *FlexKey* assigned to it reflects only its identifier and does not reflect its order. This is because the order of a constructed node at its creation time is just an intermediate order at a certain point of query execution. It does not necessarily reflect the desired final result order. We assign the order information to the constructed node at a later stage (when the constructed

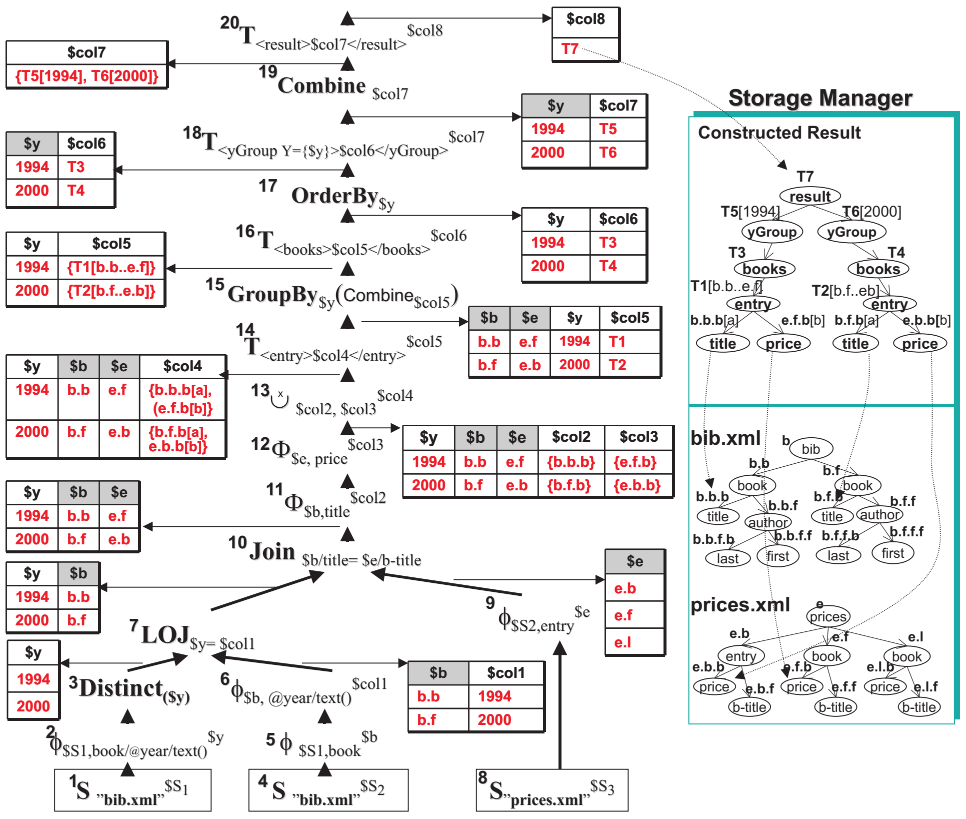


Figure 3.2: Execution using *FlexKeys* for the XQuery expression in Figure1.2(a). Shaded columns represent *Order Schema*.

nodes are placed into a sequence with other nodes or when it becomes part of other constructed nodes). When defining the order of a constructed node we use an additional key for encoding order that we attach to the *FlexKey* of the constructed node. We call such additional key *Overriding Order*. We will discuss the *Overriding Order* encoding in more detail in Section3.3.2.

**Composed Keys.** In addition to the *FlexKeys* described above, we may also use *FlexKeys* created as a composition of other *FlexKeys*. This is mainly for maintaining any order that is different than the document order in sequences of XML nodes (see Section 3.3.2). For example, the *FlexKey* $k =$ "*b.b.b..b.b.d*" is a composition of the *FlexKey*s $k_1 =$ "*b.b.b*" and $k_2 =$ "*b.b.d*", where ".." is used as delimiter. We denote this by $k = compose(k_1, k_2)$.

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

### 3.3.2 Order During XML Query Processing Time

#### Maintaining Order Among Sequences of XML Nodes

We observe that the order among the tuples (sequences of XML nodes) in an XAT table can be determined, in some cases, by comparing the *FlexKey*s stored in cells corresponding to some of the columns. For two tuples in an XAT table, we define the expression $before(t_1, t_2)$ to be $true$ if the tuple $t_1$ should semantically be ordered before the tuple $t_2$, $false$ if $t_2$ is semantically before $t_1$ and $undefined$ if the order between the two tuples is irrelevant. For example, consider the tuples $t_1$ = (1994, b.b, e.f) and $t_2$ = (2000, b.f, e.b) in the input XAT table of the operator $\#11$ in Figure 3.2. Here $t_1$ should be before $t_2$, that is $before(t_1, t_2)$ is true. This can be deduced by comparing the *FlexKey*s in $t_1[\$b, \$e]$ and $t_2[\$b, \$e]$ lexicographically. We will show that this is not a coincidence. That is, the relative order among the tuples in an XAT table is indeed encoded in the keys contained in certain columns. Thus it can be determined solely by comparing those *FlexKey*s.

Such columns are said to compose the *Order Schema* of the table. For any two tuples in the output XAT table of the *Distinct* (and also the *Group By* operators if grouping is by value), the relative order is undefined.

**Definition 3.3.1** *The* ***Order Schema*** $OS_T = (on_1, on_2, ...on_m)$ *of an XAT table* $T$ *in an algebra tree is a sequence of column names* $on_i$*,* $1 \leq i \leq m$*, computed following the rules in Table 3.1 in a postorder traversal of the algebra tree.*

Two tuples are compared lexicographically as follows.

**Definition 3.3.2** *For two tuples* $t_1$ *and* $t_2$ *from an XAT table* $T$ *with* $OS_T = (on_1, on_2, ...on_m)$*, the comparison operation* $\prec$ *is defined by:* $t_1 \prec t_2 \Leftrightarrow (\exists j, 1 \leq j \leq m)(((\forall i, 1 \leq i < j)(t_1[on_i] == t_2[on_i])) \wedge (t_1[on_j] \prec t_2[on_j]))$

The rules in Table 3.1 guarantee that cells corresponding to the *Order Schema* never contain sequences, only single keys. The rules are derived from the semantics of the operators and rely on the properties of the *FlexKey*s.

For example, let us consider the rule for computing the *Order Schema* of the operator *Navigate Unnest* $\phi^{col'}_{col,path}(T)$, when the column $col$ is the last column in the *Order Schema* of the input XAT table $T$. By the semantics of this operator presented in Section 2.2, it processes one tuple at a time. However, it may produce zero or more tuples in its output XAT table $Q$ for each tuple in $T$. The order of any two tuples in $Q$ derived from two different tuples in $T$ should be same as of those they are derived from in $T$. The order among two tuples derived from the same tuple in $T$ should correspond to the document order of the nodes present in their cells corresponding to $col'$.

| Cat. | Operator *op* | $OS_Q^*$ |
|---|---|---|
| I | $T_p^{col}(T)$, $\Phi_{col,path}^{col'}(T)$, $\sigma_c(T)$, $\overset{x\,col}{\cup}_{col1,col2}(T)$, $\upsilon_{col}^{col'}(T)$ | $OS_T$ |
| II | $S_{xmlDoc}^{col'}$, $C_{col}(T)$, $\delta_{col}(T)$, $\gamma_{col[1..n]}(T, C_{col})$, $M(T_1, T_2)$ | $\emptyset$ (if $\gamma$ is grouping by id, then $OS_T$) |
| III | $\times(T_1, T_2)$, $\ltimes_c (T_1, T_2)$, $\sqsupset\!\!\ltimes_c(T_1, T_2)$ | $(on_1^{(T_1)}, on_2^{(T_1)}, ...on_{mr}^{(T_1)}, on_1^{(T_2)}, on_2^{(T_2)}, ...on_{mp}^{(T_2)})$ $mr = \lvert OS_{T_1}\rvert$, $mp = \lvert OS_{T_2}\rvert$ |
| IV | $\phi_{col,path}^{col'}(T)$ | $(on_1^{(T)}, on_2^{(T)}, ...on_p^{(T)}, col')$ (if $on_m^{(T)} = col$, then $p = m - 1$, else $p = m$) |
| V | $\tau_{col[1..n]}(T)$ | $(col'')$, $col''$ contains order values. |
| VI | $\epsilon_{col}(T)$ | N/A |
| * $Q = op_{in}^{out}(T)$, $OS_T = (on_1^T, on_2^T, ...on_m^T)$ | | |

Table 3.1: Rules for computing *Order Schema*

Note that, in some cases, if the *Navigate Unnest* navigates to a text node or to an attribute, we may use the column of the entry point instead.

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 3.3.2 of the tuples always yields $t_1 \prec t_2$. On the contrary, if $t_1 \prec t_2$, then either $t_1$ should semantically be before $t_2$ or otherwise the order between these two tuples is irrelevant. This means that the relative order among the tuples is correctly preserved in the *Order Schema*, but the *Order Schema* may impose order among the tuples when such order is semantically irrelevant. In the following theorem, we state this observation more formally. We also prove its correctness.

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

Theorem 3.3.1 shows that the relative position among the tuples in an XAT table is correctly preserved by the cells in the *Order Schema* of that table.

**Theorem 3.3.1** *For every two tuples $t_1, t_2 \in T$, where $T$ is an XAT table in an XAT algebra tree, with $before(t_1, t_2)$ defined as above, (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 $t_1, t_2 \in T$, where $T$ 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 an algebra tree. All children nodes of the root must themselves be roots of algebra trees each of 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 3.1.*

***Category I.*** *These operators process one tuple at a time, without requiring to access other tuples or 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 $Select$, which may filter out some tuples. The latter 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 $T$ and $OS_Q = OS_T$, it must also hold for the tuples in their output XAT table $Q$.*

*To prove this formally, we consider any two tuples $tout_1$, $tout_2 \in Q$. Let $tin_1$, $tin_2 \in T$, such that $tout_1$ derived from $tin_1$ and $tout_2$ derived from $tin_2$. By the induction hypothesis, (I) holds for any two tuples in $T$, 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* $(\forall i, 1 \leq i \leq |OS_T|)$ $(tout_1[on_i] == tin_1[on_i])$*. Therefore, by Definition 3.3.2, 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* $t_1, t_2 \in Q$*,* $before(t_1, t_2) = undefined$*. The same applies to the value-based* Group By *operator*[1]*. Thus the left hand side of (I) is never* $true$*, so (I) trivially holds.*

***Category III.*** *All the operators in this category belong to the Join family of operators and regarding order have the same behavior. Their output is sorted by the left input table* $T_1$ *as major order and the right table* $T_2$ *as minor order ( see Section 2.2). Consider any two tuples* $tout_1$ *and* $tout_2$ *from the output XAT table* $Q$*. Let* $tout_1$ *be derived from* $tin_1^{(T_1)}$ *and* $tin_1^{(T_2)}$ *and* $tout_2$ *be derived from* $tin_2^{(T_1)}$ *and* $tin_2^{(T_2)}$*, where* $tin_1^{(T_1)}, tin_2^{(T_1)} \in T_1$ *and* $tin_1^{(T_2)}, tin_2^{(T_2)} \in T_2$*. Thus, by the definition of these operators:* $before(tout_1, tout_2) \Leftrightarrow before(tin_1^{(T_1)}, tin_2^{(T_1)}) \vee ((tin_1^{(T_1)} = tin_2^{(T_1)}) \wedge before(tin_1^{(T_2)}, tin_2^{(T_2)}))$*. Note that for the* Left Outer Join *operator there could exist zero to many output tuples that are not derived from any tuple in* $T_2$*. But, as there could be at most one such tuple derived from each tuple in* $T_1$*, the above statement is still valid.*

*There are two cases: (1)* $tin_1^{(T_1)}$ *and* $tin_2^{(T_1)}$ *are two different tuples from* $T_1$*, or (2) both* $tout_1$ *and* $tout_2$ *are derived from the same tuple* $tin^{(T_1)}$*, i.e.,* $tin_1^{(T_1)} = tin_2^{(T_1)} = tin^{(T_1)}$*.*

*For case (1) it holds that* $before(tout_1, tout_2) \Leftrightarrow before(tin_1^{(T_1)}, tin_2^{(T_1)})$*.*

[1]The id-based *Group By* follow the rule of Category I.

*Hence, this case can be easily reduced to that for the operators in Category I.*

*For case (2), when* $tin_1^{(T_1)} = tin_2^{(T_1)} = tin^{(T_1)}$*, as* $before(tout_1, tout_2) \Leftrightarrow before(tin_1^{(T_2)}, tin_2^{(T_2)})$ *and by the induction hypothesis* $before(tin_1^{(T_2)}, tin_2^{(T_2)}) \Rightarrow (tin_1^{(T_2)} \prec tin_2^{(T_2)})$*, in order to prove* $before(tout_1, tout_2) \Rightarrow (tout_1 \prec tout_2)$*, it is sufficient to show* $(tin_1^{(T_2)} \prec tin_2^{(T_2)}) \Rightarrow (tout_1 \prec tout_2)$*. By the rules in Table 3.1, the* Order Schema *of* $Q$ *contains all the columns from the* Order Schema *of* $T_1$*, followed by all the columns from the* Order Schema *of* $T_2$*. As the operators considered do not modify any values in the columns retained from the input tuples, it holds that* $(\forall i, 1 \leq i \leq |OS_{T_1}|)((tout_1[on_i^{(T_1)}] == tin^{(T_1)}[on_i^{(T_1)}]) \wedge (tout_2[on_i^{(T_1)}] == tin^{(T_1)}[on_i^{(T_1)}]))$ *and* $(\forall j, 1 \leq j \leq |OS_{T_2}|)((tout_1[on_i^{(T_2)}] == tin_1^{(T_2)}[on_i^{(T_2)}]) \wedge (tout_2[on_i^{(T_2)}] == tin_2^{(T_2)}[on_i^{(T_2)}]))$*. Thus,* $(\forall i, 1 \leq i \leq |OS_{T_1}|)(tout_1[on_i^{(T_1)}] == tout_2[on_i^{(T_1)}])$ *and then by Definition 3.3.2* $(tin_1^{(T_2)} \prec tin_2^{(T_2)}) \Rightarrow (tout_1 \prec tout_2)$*.*

***Category IV.*** *The operator* Navigate Unnest $\phi_{col,path}^{col'}(T)$ *by its definition presented in Section 2.2 processes one tuple at time. However, it may produce zero or more tuples in its output XAT table* $Q$ *for each tuple in* $T$*. Consider any two tuples* $tout_1$ *and* $tout_2$ *from* $Q$*. There are two cases: (1) Both* $tout_1$ *and* $tout_2$ *are derived from the same tuple* $tin$*, or (2)* $tout_1$ *is derived from* $tin_1$ *and* $tout_2$ *is derived from* $tin_2$*,* $tin_1 \neq tin_2$*.*

*For case (1), let* $l_1$ *and* $l_2$ *be indexes such that* $tout_1[col'] = \phi(path : tin[col])[l_1]$ *and* $tout_2[col'] = \phi(path : tin[col])[l_2]$*. As* $(l_1 < l_2) \Leftrightarrow before(tout_1, tout_2)$*, in order to prove* $before(tout_1, tout_2) \Rightarrow (tout_1 \prec tout_2)$*, it is sufficient to show* $(l_1 < l_2) \Rightarrow (tout_1 \prec tout_2)$*. Suppose* $l_1 < l_2$*. Then, due to the properties of the* FlexKeys *we have* $tout_1[col'] \prec tout_2[col']$*. By the* *rule in Table 3.1, $col'$ 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 3.1. Thus, by Definition 3.3.2, $on_j = col$ 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 3.3.2 must exist. There are two sub-cases: (2.a) $j \leq p$, and (2.b) $j > p$, with $p$ as in Table 3.1. 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 3.1. 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 2.2): $(\exists l_1, l_1 > 0)|(tout_1[col'] = \phi(path : tin_1[col])[l_1])$, and $(\exists l_2, l_2 > 0)|(tout_2[col'] = \phi(path : tin_2[col])[l_2])$. As the* FlexKey *assigned to a node always has the keys of all its ancestors as prefixes, $tout_1[col']$ has the key in $tin_1[col]$ as prefix and $tout_2[col']$ has the key in $tin_2[col]$ as prefix. Therefore $tin_1[col] \prec tin_2[col] \Rightarrow tout_1[col'] \prec tout_2[col']$ and consequentially $(tin_1 \prec tin_2) \Rightarrow (tout_1 \prec tout_2)$.*

***Category V.*** *The theorem holds by definition.*

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

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

### Maintaining Order Among XML Nodes in Sequences

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

```
function $combine$ (Sequence $in$, Tuple $t$, ColumnName $col$)
    Sequence $out \leftarrow copy(in)$
    if ($col = OS_T[i]$ [2], $1 < i \leq |OS_T|$)
        for all $k$ in $out$
            $k.overridingOrder \leftarrow compose(\Pi_{OS_T[1]}t, .., \Pi_{OS_T[i]}t)$
    else if ($col \notin OS_T$)
        for all $k$ in $out$
            $k.overridingOrder \leftarrow (\Pi_{OS_T[1]}t, .., \Pi_{OS_T[m]}t, order(k)), m = |OS_T|$
    return $out$
```

Figure 3.3: The function *combine*

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

To represent an order that is different than the one encoded in the *FlexKey* $k$ serving as the node identity of the node, we attach an additional *FlexKey* to $k$ (called *Overriding Order*) which reflects the node's proper order. We denote this as $k.overridingOrder$ and we use $order(k)$ to refer to the order represented by $k$. When the *FlexKey* $k$ has overriding order $k_o$ it is denoted as $k[k_o]$. If the overriding order of $k$ is set, then $order(k) = k.overridingOrder$, otherwise $order(k) = k$. When comparing lexicographically two *FlexKey*s $k_1$ and $k_2$, $order(k_1)$ and $order(k_2)$ are really being compared. Thus $k_1 \prec k_2$ is equivalent to $order(k_1) \prec order(k_2)$.

The $Combine$ operator sets the overriding order for the nodes in its output XAT table, as described in Figure 3.3. Thus, assuming that the input $T$ contains $p$ tuples $tin_j$, $1 \leq j \leq p$. How $Combine$ $C_{col}(T)$ sets the overriding

[2] $OS_T$, the *Order Schema* of the input XAT table $T$, is known to the $Combine$ operator performing the *combine* function.

order depends on the presence of the column $col$ in the *Order Schema* $OS_T$ of the input XAT table $T$. For the combine operator given as the sub-query of the *Group By* operator (# 15) in Figure 3.2, $\$b$ and $\$e$ are in the *Order Schema* of the input. Thus, when the input XML node referenced by $T1$ is placed in the output XAT table it is assigned an *Overriding Order* composed of the order represented by the *FlexKey*s present in columns $\$b$ and $\$e$ in the tuple it is derived from, that is $b.b..e.f$. Thus $T1$ after being processed by this $Combine$ becomes $T1[b.b..e.f]$

The XML collection operator *XML Union* $\overset{x}{\cup}{}^{col}_{col1,col2}(T)$ creates a new collection, for each tuple it processes, from the contents of two input columns *col1* and *col2* . A new order is imposed by this process among the nodes originating from each of the input columns. We define this order by assigning an *Overriding Order* for each node that reflects its input column order in the union operation, if no *Overriding Order* keys are already defined. For example, if *col1* contains (b.f, b.l) and *col2* contains (f.b) the output column *col3* will contain (b.f[a], b.l[b], f.b[c]). If nodes in the input columns already have *Overriding Order* keys we extend these keys by adding a prefix to it that reflects the input column order. For example, if *col1* contains (b.f[b], b.l[f]) and *col2* contains (f.b) the output column *col3* will contain (b.f[b.b], b.l[b.f], f.b[f]). This order encoding ensures that we maintain order among nodes from different input columns and at the same time maintain the original order among nodes from the same input source. Other XML collection operators (*XML Unique*, *XML Difference*, and *XML Intersection*) remove the overriding order (if present) of the node identity *FlexKey*s that they place in their output XAT tables. By definition (see Section 2.2) they produce a column in which the nodes are in document order.

The *Group By* operator $\gamma_{col[1..n]}(T, func)$ does not define or maintain order among the created groups if the grouping is done by value. The *Group By* in the XAT algebra might create collections, when the *Group By* performs nesting operations (when its $func$ argument is composed of a *Combine* operator). In such a case nodes are grouped creating collections based on the grouping columns. Order among nodes of each collection is of importance. This order is maintained by *Combine* operator, as discussed above. If the *Group By* is by node id, the order between the created groups follows the *Order Schema* of the input XAT table.

**Theorem 3.3.2** *Let $kout_1$ and $kout_2$ be two* FlexKeys *in a same cell in an XAT table $T$ in an XAT algebra tree. Let these* FlexKeys *serve as node identities of the XML nodes $n_1$ and $n_2$ respectively. Then with $before(n_1, n_2)$ defined as above:*
*(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 $T$. The operators of interest are those that output columns that may contain collections of* FlexKeys*. Such operators are* Navigate Collection, XML Union, XML Difference, XML Intersection*m* Group By, *and* Combine*. All the other operators do not create collections of* FlexKeys*, 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* FlexKeys *$kout_1$ 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* FlexKeys *$kout_1$ and $kout_2$ can not have an overriding order set, as they are retrieved from the Storage Manger by op.*

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

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

*Let $kin_1$ and $kin_2$ be the* FlexKeys *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 $T$ 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 $T$, 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 3.3.2 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 $T$ is $OS_T = (on_1, on_2, ...on_m)$, $before(n_1, n_2)$ 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 3.3, 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 sub-cases: (3.a) $kin_1$ and $kin_2$ are in the same tuple $t$, i.e., $t_1 = t_2 = t$, or (3.b) $t_1$ and $t_2$ are two different tuples. For case (3.a), $order(kout_1)$ and $order(kout_2)$ are composed of the same keys except for the last key that represents the order of $kin_1$ and $kin_2$ within the collection contained in $t[col]$. As in this case $before(n_1, n_2)$ for $n_1$ and $n_2$ in the output XAT table may only hold when it holds for $n_1$ and $n_2$ in the input XAT table, the overriding order is correctly set. For case (3.b), $before(t_1, t_2) \Leftrightarrow before(n_1, n_2)$. As the overriding order of $kout_1$ and $kout_2$ is composed of all the keys corresponding to the* Order Schema *in $t_1$ and $t_2$ respectively, $before(t_1, t_2) \Rightarrow (kout_1 \prec kout_2)$. By transitivity, $before(t_1, t_2) \Leftrightarrow before(n_1, n_2)$ and $before(t_1, t_2) \Rightarrow (kout_1 \prec kout_2)$ imply*

$before(n_1, n_2) \Rightarrow (kout_1 \prec kout_2)$.

*We have proven (I) for all the cases. Using that result, (II) can be proven by contradiction, using the same arguments used for proving (II) in Theorem 3.3.1.* $\square$

### Maintaining Order of Internal Nodes of XML Nodes in Sequences

**Order of Internal Nodes of Base XML Nodes.** Some base XML nodes (fragments) might be processed and exposed in the result as whole pieces without inserting, deleting or changing any of their contents. The relative local order among internal (children/descendant) nodes of a base XML fragment does not change during execution time even if the order of the whole fragment is changed. Hence the *FlexKey*s of these internal nodes remain to reflect the relative order among them.

**Order of Internal Nodes of Constructed XML Nodes.** Order among internal (children/descendant) nodes of a constructed node is determined by the $Tagger$ pattern and/or *XML Union* operations. Such order might be different than the order of the underlying XML document. Moreover, children nodes of a constructed node might be themselves constructed nodes and/or originating from different source XML documents. Hence, there is no relationship between their *FlexKey*s. For example, the constructed node $T1$ in Figure 3.2 has two children nodes with *FlexKey*s $b.f.b$ and $e.f.f$, corresponding to the "title" and the "price" nodes respectively. These two nodes are originating from two different source XML documents. The local order among them is defined by the input column to the $Tagger$ pattern (trough an *XML Union* operation). We encode the local order among internal nodes of constructed nodes by assigning *Overriding Order* keys. This applies to any type of internal nodes (base or constructed). We assign the *Overriding Order key*s $a$ and $b$ to the "title" and "price" nodes respectively.

### 3.3.3 Order in the Final XML Result.

Generally, when de-referencing the final result we may require partial re-ordering as we will discussed later. For the example in Figure 3.2, the result of the XQuery expression is obtained by de-referencing the *FlexKey* $T7$. First, the skeleton of the constructed node identified by $T7$ is retrieved and the *FlexKey*s contained in that skeleton are de-referenced. The children of $T7$ (a collection of two nodes $T5$ and $T6$) need to be returned in the correct order. We sort these nodes based on their *Overriding Order* and return node $T5$ first then $T6$. Now we take these two nodes one by one and de-reference them recursively so that the resulting XML document is obtained. Note that because the collections returned in the result are de-referenced one collection at a time and they are often small sets of nodes, sorting can often be done in main memory thus becoming very efficient. Note that when we obtain any base node, its descendants (if any) are returned in document order without any sorting.

## 3.4 Discussion on our Proposed Order Solution

### 3.4.1 Support for Different Types of Order

**Document Order.** Given the order encoding schema discussed above we can now maintain document order. This provides support for XQuery queries that return the result (or part of it) in document order. It also provides support for XQuery functions and predicates that exploit document order like *before*, *after*, *range*, and *position*. Figure 3.2 shows the full intermediate result for the execution of our running example XQuery in Figure 1.2. The order schema columns of intermediate result tables are shaded. We note that for the query shown in Figure 3.2 columns *$b* and *$e* serve as the order determining columns (*Order Schema*) for all intermediate XAT tables below the *Group By* operator. Such *Order Schema* is composed of the *Order Schema*s of the input tables of the $Join$ operator. The *Group By* destroys this *Order Schema* while at the same time the $Combine$ operator in its sub-query creates a collection for each group. The $Combine$ operator defines order between nodes in the collection it creates. This is done by assigning an *Overriding Order* key for each node in the created collection. This *Overriding Order* key is composed of the keys in the corresponding order determining columns in the input XAT table. Note that in this example the way these *Overriding Order* keys are assigned ensures that the order between any two newly constructed ”entry” nodes in the same group still follows the underlying documents order.

**Query Order Imposed By the Query *order by* Clauses.** The *order by* clauses in XQuery expressions are translated into $OrderBy$ operators in XAT query plans. Maintaining order in such queries is also done using order determining columns (*Order Schema*). In this case we discard the order that is based on document order (Only at the node level manipulated by the *order by* clauses) when the *Order By* operator is encountered during execution and we use a new *Order Schema* that is generated by the *Order By* operator.

Figure 3.2 shows that before the *Order By* operator the *Order Schema* is empty (as a result of the *Group By* operator). When the *Order By* operator is processed, it adds a new column $\$y$ to the *Order Schema*. This column has the order values. The $Combine$ operator next uses the new order values in column $\$y$ to override order of nodes in the collection it creates. In this case the order between the newly constructed "yGroup" nodes will follow the order specified by these values.

**Query Order Imposed by the Nesting of Variable Binding in the Query** $for$ **and** $let$ **Clauses.** Such variable nesting is translated into $Join$ operations on the algebra level. Hence, the order treatment follows the rules described above. These rules give a major influence on the order to the data bound to the outside variable of the $for$ clause and a minor influence on the order to data bound to the inside variable of the $for$ clause. Such order is encoded only at the *Order Schema* level and no extra keys are needed to reflect it at this point.

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

### 3.4.2 The Cost of Maintaining Order

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

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

2) The cost of assigning *Overriding Order* keys for processed XML nodes. Only three operators out of all the seventeen operators shown in Table 3.1 need to assign *Overriding Order* for the nodes they process. These three operators are the $Combine$, the *XML Union*, and the $Tagger$. Integrating the process of assigning the *Overriding Order* keys with the actual query execution of these operators would result in a lot of time saving.

3) The cost of sorting when we de-reference the final result. Such sorting is a key-based sorting (on the *FlexKey*s), and is typically a partial sorting. Sorting might be required mainly for collections created by the query during execution (using the *Combine*, the *XML Union*, or the *Group By* operators). In many cases such sorting might involve only one scan over the nodes, if they are already sorted. This may occur when the correct order of the processed nodes has not been destroyed by the query execution. All internal nodes (children/descendants) of returned base nodes are directly de-referenced from the *Storage Manager* in document order [DR03], thus

no sorting is required. For internal nodes of constructed nodes sorting depends on what is included under the constructed node. A constructed node may tag single nodes, collections of nodes, or combinations of them. The skeleton representation of a constructed node created and stored during execution time reflects the structure and the relative order among its internal nodes and/or collections. Our system ensures that all internal nodes and collections of constructed nodes are returned in the result directly in their tagging order, hence not sorting is required. Sorting might only be required for contents of collections as discussed above. In the worst case, total sorting for all nodes in the result of an XQuery might be required only if the result returns one collection of base leaf nodes or of constructed nodes each of which is a parent of one base leaf node.

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

- Some of the rules presented in Table 3.1 can be further optimized by removing/replacing certain columns in the *Order Schema*. This would reduce the number of columns in the *Order Schema* or replace them with columns with smaller *FlexKey*s. Hence when producing order keys based on the *Order Schema* we get smaller keys. For example, for the operators *Select* and *Theta Join* if any of the columns present in the selection or joining condition are not in the *Minimum Schema* of the output XAT table, they can be dropped from the *Order Schema* of output XAT table (if it has other columns) or replaced by the column(s) in which they originate from (if the *Order Schema* has no other columns). This is because such columns are created to be used in the *Select* or *Join* predicates and are not part of any later processing operations. Hence their specific order is not of interest to the query. For example, if an input XAT table for a *Select* operator has an *Order Schema* that is composed of columns $\$a$ and $\$p$, and assuming that column $\$p$ is used in the *Select* condition and it is not in the *Minimum schema* of the output XAT table of that operator. This signifies that column $\$p$ is not needed for any next operation and its order is not of importance to the result of the query. Hence, we can drop column $\$p$ from the *Order Schema* of that operator output XAT table. This makes the order among tuples in that output XAT table determined only by the contents of column $\$a$. Another example is column $\$col1$ in Figure 3.2. This column is used in the *Left Outer Join* operator (#7) and is not part of the *Minimum Schema* of the output table of that operator (since it is not needed for any next operation). Since this is the only column in the *Order Schema*, of the input XAT table of the operator, we replace it with the column it originally came from (column $\$b$). Column $\$b$ is hence used to reflect the order instead of column $\$col1$. In this case if we are to extract the order of that table, at a later stage, we get the smaller *FlexKey*s based on column $\$b$ instead of the larger *FlexKey*s in column $\$col1$.

- It is also possible to optimize the *Order Schema* using schema information of the source XML documents if available. Such optimization may again result in generating smaller order keys. For example, if a

*Navigate Unnest* operator navigates from "book" nodes in column $\$b$ to "title" nodes (that are placed in column $\$col$). And if *Order Schema* contains $\$b$. If schema information exists that specifies that there is only one possible "title" child for each "book" node, we may keep using column $\$b$ as the *Order Schema* of the output XAT table instead of column $\$col$. Again, the size of order keys extracted from column $\$b$ is smaller than that order keys extracted from column $\$col$. Such reduction in order keys size is more significant when the navigation operation involves many navigation steps.

- In many cases the query may not destroy the desired order of the returned result. But we may still need to perform one scan over the returned collections to conclude that it is in the desired order. One possible optimization to eliminate such unnecessary scan is to maintain a flag for processed collection (one flag assigned to each collection). This flag specifies if the order of processed collection(s) is preserved or not. The value of this flag is set by different operators in the algebra tree. When returning a collection in the final query result, if its flag reflects that the collection order is not destroyed we can directly return the nodes in the collection without checking if it is in the desired order or not.

- It might be also possible to tune the query optimization and execution itself to achieve better overall performance in terms of the total cost of execution and order. For example, if savings form certain optimization or execution strategy is wiped out by an added final sorting cost we might choose another execution strategy, possibly of slightly higher cost, that results in less overall cost for the execution and order together. For example, if a hash-based *Join* hashes the smallest table and scans the biggest table and joins tuples from the biggest table with the hashed tuples, the result will be sorted based on the order of the biggest table. Hence, in some cases (for example, if the two tables are close in size) we may choose to hash the right input table in particular so we generate a result that reflects the major order of the left input table. Since the order of the *Join* output follows the order of its left input table as a major order and then the order of its right input table as a minor order, this treatment reduces the final sorting time (or eliminate it if the minor order of the right table is not of importance).

- In some cases it might be possible to avoid assigning *Overriding Order* keys for nodes. For example, if a *Tagger* operator constructs a new node and assigns some base nodes as children for it. If the tagging pattern places these nodes in a relative order similar to that of their source XML document, there is no need to assign *Overriding Order* keys for these nodes.

### 3.4.3 Implications of our Order Solution

**Migration to Non-ordered Bag Semantics.** Our technique of encoding order with *FlexKey*s and intermediate *Order Schema* enables migration of the XAT algebra semantics from ordered bag semantics to non-ordered bag semantics. That is, (1) the physical order among the tuples is no longer of significance and (2) the physical order among the nodes in a cell is not of significance. This implies that we separate out the reasoning about order into a separate abstraction independent of each operator's logic. In general, algebra operators are thus not responsible for maintaining order of intermediate results. One exception is the *Order By* operator. The *Order By* operator has to define a new order among the data it processes. This cost is encountered anyways regardless of the order solution used. The only added cost in our approach for maintaining order while processing the *Order By* operator is the cost of assigning new order keys to the data. Also, while the $Combine$, the *XML Union*, and the $Tagger$ operators do not perform any sorting, they assign new order keys for nodes while they process them. All other operators process the data while they are unaware of its order. In general, our solution does not require sorting of any intermediate results even while achieving nested ordered XML restructuring.

**Efficient Order-sensitive Query Processing.** This transformation from ordered to non-ordered bag semantics is the key ingredient to facilitate XML query optimization. It removes the restrictions of manipulating sequences of XML data in a strict order. Order is encoded at the XML node level and at intermediate result schema level. Operators do not need to be aware of the order associated with data they manipulate. For that reason operators have the flexibility to reshuffle data in any order they wish for efficiency. This way, a $Join$ operator could use any efficient join algorithm (e.g., hash-based, index-based, or sort-merge join) producing the output in any order dictated by the join implementation strategy without requiring any intermediate sort. For example, the $Join$ operator in Figure 3.4

joins its two input tables on the values in columns $col2$ and $col4$. Order-determining columns (*Order Schema*) for each XAT table is shaded. The number in a circle that appears besides each tuple illustrates the implicit order of each tuple implied from the *Order Schema*. Now assume that the join implementation outputs the resulting tuples in any arbitrary physical order as in Figure 3.4. We are still capable of deriving the right order of tuples in the output table (major order from left input table and minor order from right input table) by comparing the keys in the columns representing the *Order Schema* (columns $col1$ and $col3$) of the resulting table. The numbers in circles that appear next to tuples in the output table in Figure 3.4 show the order of tuples as we can derive it using the *Order Schema*. Note that this order is only an implicit order. That is, the tuples are not actually sorted based on this order at this point of query execution.

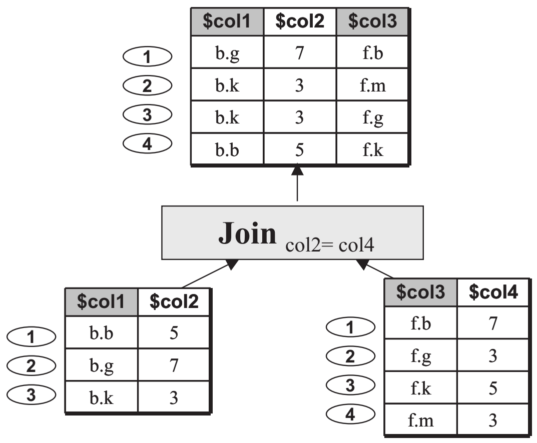


Figure 3.4: An example for order handling in the $Join$ operator. Shaded columns determine the *Order Schema* of each table and numbers appearing in circles beside tuples determine the tuple induced order.

**Efficient Order-sensitive View Maintenance.** The migration to the non-ordered bag semantics also facilitate efficient XML incremental view maintenance. This is because it ensures that most XAT XML operators become distributive with respect to order operations on bag union, leading to more efficient view maintenance. As an example, consider that the input table of a $Select$ operator has received an update in the form of a tuple insertion. Since the $Select$ operator becomes distributive, the inserted tuple can be processed independently of other input tuples. If the inserted tuple satisfies the operator predicate it is directly propagated to the output table. Without the distributive feature, the operator would have to determine the relative order of the inserted tuple among the output table. They may require storing and accessing auxiliary information to determine that order.

### 3.4.4 Other Discussions

**Re-labeling (Reordering keys) on Updates.** Unlike other order approaches [FLSW03, JAKC$^+$02, TVB$^+$02] our order encoding schema guarantees that we do not run out of keys even for a large batch of skewed insertions focused on possibly one small region within the underlying XML document. The reason is two-fold: (1) we leave gaps between keys when we first assign them (as in Figure 3.1), and (2) we are capable of producing a key between any two keys at all times even if there is no gap between them. This is because our key is composed of variable length byte strings as described earlier. Thus, even if we run out of keys due to a large number of inserts that fill the gap between two keys we can opens up new gaps by adding one more character to the encoding. For example, if we need to insert a new node between two nodes with keys $b.c$ and $b.d$ we may simply give the new node the key value $b.ck$. This will open up new gaps between $b.c$ and $b.ck$ and between $b.ck$ and $b.d$ and so on. This prevents the need to re-label keys not only for the source document node keys but also for the order encoding of the processed data since we also use $FlexKey$s to encode new order imposed by the query. Please, refer to the example of inserting a new "author" with a key $b.b.d$ that we have presented in Section 3.3.2.

**Order Among Multiple Documents.** Our order approach supports order also for queries over multiple XML documents. There are two issues to consider here: (1) base node key and order encoding and (2) query order encoding. (1) On the base node level, each XML document has order among its nodes encoded separately using the keys of its nodes as we have shown earlier. The *Storage Manager* [DR03] ensures that each document will have a unique key for the root node. Hence all nodes will have a unique key among all documents. For example, although the two nodes $b.b.f$ and $e.b.f$ share the suffix $b.f$, but because they are from two different documents (with root keys $b$ and $e$), the key for each one of them is unique. For any base XML node (fragment), originating from any document, the local order of its internal nodes is reflected by the nodes' *FlexKey*s, as discussed in Section 3.3.2. (2) On the query level, the order among data from different source documents is determined by the query itself. This is typically handled by the order imposed by the nesting of variable binding in the $for$ and $let$ clauses, and the order imposed by the query $return$ clause and the new result construction. Hence, the treatment of order among multiple documents follow the same guidelines we gave for handling these types of query imposed order.

## 3.5 Experimental Evaluation for the Cost of Handling Order

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

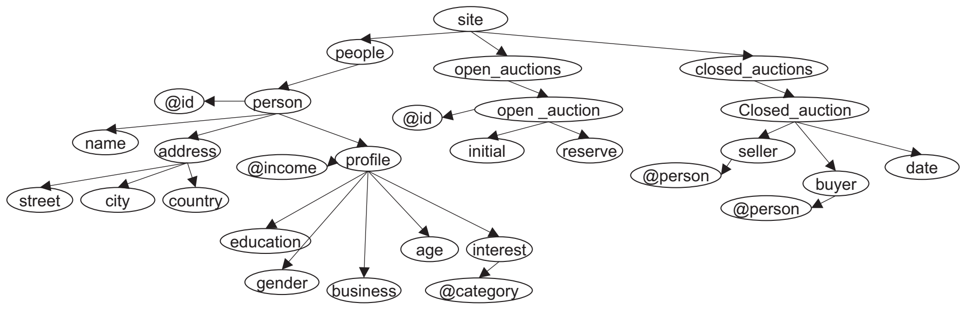


Figure 3.5: Part of the structure of the "site.xml" file used in the experiments.

We have implemented our order approach in Java and integrated it with the Rainbow system [Zea03]. We have run the experiments on a Windows PC with 733 MHz Pentium processor and 512MB of memory. We have used the XMark benchmark data [SWK$^{+}$02] in our experimental evaluation. Figure 3.5 shows part of the structure of the XMark "site.xml" data set that is relevant to the queries we use. We use XML files of different sizes in our experiments, varying from 5MB to 25MB. We use four queries (shown in Figure 3.6) that come with different order requirements. We have designed each of the four queries to reflect mainly one form of the four order types that we have discussed earlier. This ensures that we measure the cost of each type of order in isolation of the other types. For each of the four queries we show the overhead of handling order relative to the total query execution time. We also break down the order cost in each query to its cost elements. We now analyze the results we have obtained using these queries.

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

Figure 3.7(a) shows that the total cost of handling order in this query is very small (negligible) compared to the query execution time. The break down of this order cost is shown in Figure 3.7(b), measured using the input XML file of size 25MB. The cost of maintaining order in a query that processes only document order is mainly composed of two cost elements: (1) the *Order Schema* computation cost and (2) final result sorting cost. The *Order Schema* computation cost is fixed regardless of the size of the processed

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

(a)

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

(b)

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

(c)

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

(d)

Figure 3.6: Different XQuery expressions that are used in the experiments.

data (it only depends on the number of operators in the query plan). The cost of the final sorting depends on the processed data size. It also depends on how the query manipulates the order among processed nodes. For *Query 1* only partial sorting might be needed on the level of the returned fragments (“profile” elements) if the correct order among those fragments was destroyed during query time. Internal nodes of those nodes are returned in document order without any sorting, as discussed earlier. Figure 3.7(b) shows that the cost of the final (partial) sorting for *Query 1* is very small. *Query 1* did not perform any operation that destroys the order among nodes in the returned collection. Hence a very small cost is needed to conclude that the returned result is in the correct order and no sorting is needed.

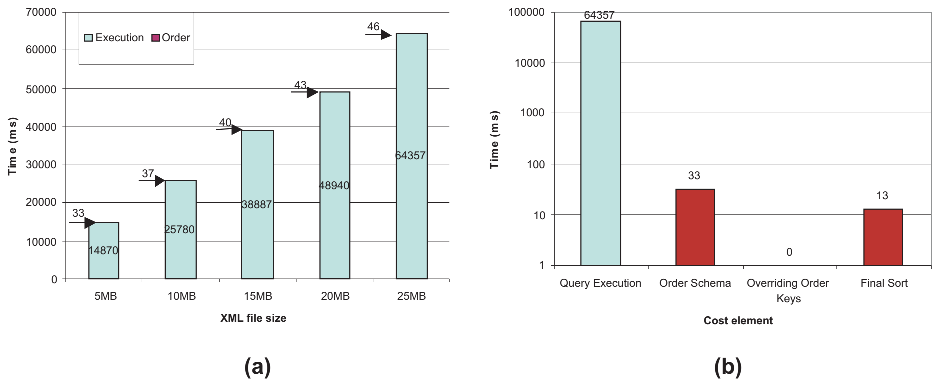


Figure 3.7: Results obtained for Query 1: (a) the order cost to the execution cost on different input XML file sizes, and (b) the break down of order cost on 25MB XML input file size.

**Query 2.** This query navigates to the "city" nodes reachable through the path "/people/person/address". A collection of distinct cities is created using the *distinct-values* operator. This collection of distinct "city" elements is sorted alphabetically on the "city" name by the *order by* clause. Finally the collection is tagged using the "result" tag. This query reflects a query order imposed only by the *order by* clause. No document order or any other type of query order is affecting the result.

Figure 3.8(a) shows that the total cost of handling order in this query is also very small (negligible) compared to the query execution time[3]. The

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

break down of this order cost is shown in Figure 3.8(b). The cost of maintaining order in a query that imposes order through the *order by* clause is mainly composed of three cost elements, (1) the *Order Schema* computation cost, (2) the cost of assigning *Overriding Order* keys, and (3) the final result sorting cost. The *Order Schema* computation cost is fixed regardless of the size of the processed data. The cost of assigning *Overriding Order* keys and the cost of the final sorting depend on the processed data size. For *Query 2* the cost of assigning the *Overriding Order* keys is the highest among the other order cost elements. This is mainly because all the returned nodes in this query are affected by the *order by* clause and hence are assigned *Overriding Order* keys[4]. The *order by* operation in this query performs a sort for the processed nodes generating an ordered collection at the intermediate result. This is due to the current implementation of the query engine. This order is not destroyed by any other operations in the query. Hence the final sorting cost shown in Figure 3.8 involves mainly verifying that the returned collection of "city" nodes is already in the desired order.

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

introduced by our order solution itself are measured as part of the order cost.

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

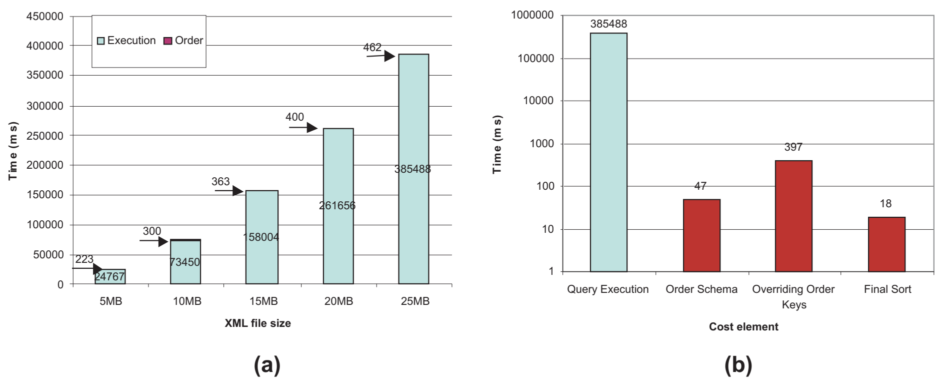


Figure 3.8: Results obtained for Query 2: (a) the order cost to the execution cost on different input XML file sizes, and (b) the break down of order cost on 25MB XML input file size.

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

Figure 3.9(a) shows that the total cost of handling order in *Query 3* is also very small compared to the query execution cost. The break down of this order cost is shown in Figure 3.9(b). The *Order Schema* computation cost is slightly higher than the last two queries because the query plan of

*Query 3* has more operators. The cost of assigning *Overriding Order* keys here involves assigning *Overriding Order* keys to all the returned "date" elements. Such keys reflect the major and the minor order imposed by the *for* clause. The cost of the final sort is affected by the implementation of the *join* operator. The implementation of the *join* operation here is performed using a hash-based join. The XAT table representing the closed auctions is the one that gets hashed because of its size. This caused only the minor order of the processed data is destroyed. Hence returning the result in the correct order requires minor sorting for the returned result. The cost of this sort is shown in Figure 3.9(b).

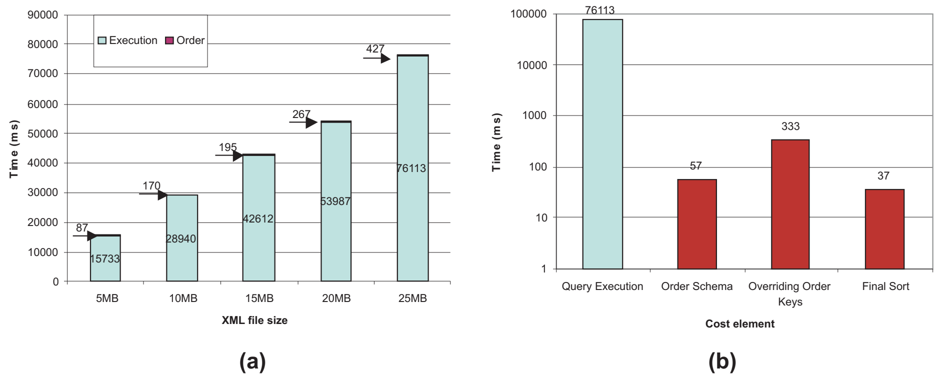
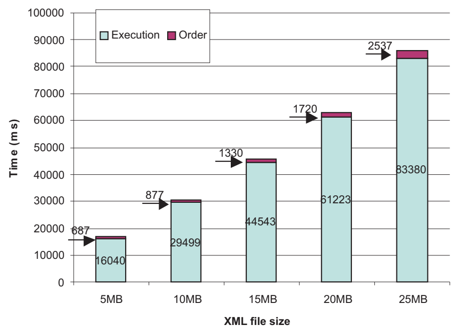


Figure 3.9: Results obtained for Query 3: (a) the order cost to the execution cost on different input XML file sizes, and (b) the break down of order cost on 25MB XML input file size.

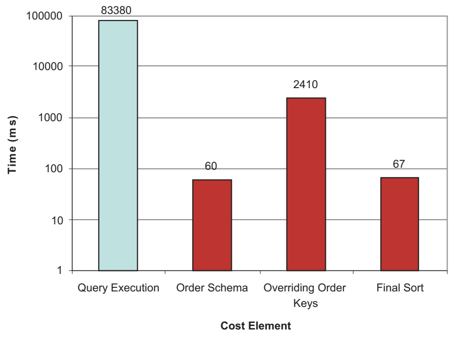
**Query 4.** This query creates a result with a new structure by performing many node construction operations as shown in Figure 3.6(d). This query reflects mainly a query order that is imposed by new node construction and the order specified in the *return* clauses[5]. Figure 3.10(a) shows that the total cost of handling order in this query is very small compared to the query execution cost. The break down of this order cost is shown in Figure 3.10(b). The *Order Schema* computation cost is higher than that for the last three queries because the query plan of *Query 4* has more operators. The cost of assigning *Overriding Order* keys is also high because it involves assigning *Overriding Order* keys to all the nodes in the returned result (except for the "result"). These *Overriding Order* keys reflect the query imposed order (and document order for nodes "customer" and "bid"). A small final sort cost is encountered while deriving the right order among the returned "customer" and among the returned "bid" elements.

Although all results reported here have been run on the basic Rainbow system, i.e., without employing any of the order-oriented optimization strategies pointed out earlier in Section 3.4, the cost of handling order has still been shown to be negligible. We expect that the cost of handling order can be even further significantly minimized by incorporating these optimization techniques into the system.

[5]Some implicit document order is also present in this query, in which the constructed nodes "customer" and "bid" follow the document order of the "person" and the "open_auction" elements respectively. Note that the order among descendants of each of these constructed nodes is different from that of their source elements.



(a)



(b)

Figure 3.10: Results obtained for Query 4: (a) the order cost to the execution cost on different input XML file sizes, and (b) the break down of order cost on 25MB XML input file size.

# Chapter 4

# Incremental Fusion of XML Fragments through Semantic Identifiers

## 4.1 Object Fusion

Object fusion is a core operation in information integration where mediators collect and integrate data objects from different sources [PAGM96]. Such integration needs to support 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 where sources are updated and materialized views are refreshed once source data updates are made available or even in a deferred mode. 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.

Such merging 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 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 to object fusion in XML views. Unlike other data models (such as relational, object-oriented, and even some semi-structured data models), XML is an ordered data model. 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. The order in the result of a FLWOR expression can in addition be imposed by the expression itself in several ways, including the use of *order by* clauses, the nesting of $for$ clauses, and the order defined by the $return$ clauses. See [ESDR05] for more details.

**Example.** Consider the two XML documents shown in Figure 1.1 and the XQuery view in Figure 1.2(a) defined over these two sources. The result of executing this XQuery expression over the source documents is shown in Figure 1.2(b). Now assume that the "bib.xml" source document is updated by appending the new book element shown in Figure 4.1(a), with its tree representation shown in Figure 4.1(b)[1], to the end of that XML document (reflecting the desired document order). A view maintenance solution would need to propagate such an update into one update that can be applied to the materialized view in Figure 1.2(b). The propagated update, shown in Figure 4.1(c), needs to be applied correctly to the materialized view to refresh it.

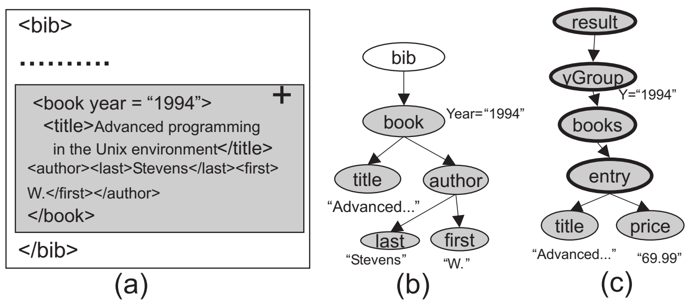


Figure 4.1: (a) A new "book" element to be inserted into the source document "bib.xml" shown in Figure 1.1, (b) the corresponding XML tree, and (c) the expected result of propagating the update through the view in Figure 1.2(a).

The question that we raise now is how to fuse the propagated update in Figure 4.1(c) with the previously computed result shown in Figure 1.2(b). We must decide 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

[1]For now we assume that the update is represented as a full XML fragment. We will address source update representation and validation in more details in Chapter 5.

from existing ones, to the view extent. For example, to correctly refresh the view extent in Figure 1.2(b) the propagated nodes “result”, “yGroup”, and “books” should be merged with certain existing nodes (as we will show later) while the nodes “entry”, “title”, and “price” should be added to the view extent as new nodes separate from existing nodes. We also must decide how the order of the materialized XML view is maintained as a result of applying such an update to the view extent. In this example, to maintain the order of the view extent the newly added node “entry” should be inserted right after the existing “entry” node with title = “TCP/IP Illustrated”. This is based on the query semantics and the source document order. The new “entry” node should come second in the year “1994” group as the source update creating this node comes second in the source document among books with book year “1994”. In addition, among the children of the newly inserted node “entry”, the “title” children should come before the “price” children while preserving the document order among nodes in each of these two collections.

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 directly over the updated sources. This includes maintaining the correct view order.

Similarly, the same scenario may also apply to the XML steam query processing context. The view extent shown in Figure 1.2(b) can be seen as a partial result, the book element newly appended to the source (Figure 4.1(b)) can be seen as an XML stream unit (transmitted as the next stream element), and the XML fragment in Figure 4.1(c) can be seen as the result of processing the XML stream unit before it is merged with the previous partial result computed over previously streamed data.

Note that in the example shown above we use a relatively simple XQuery view to illustrate the problem. The problem becomes more complex when we consider more complicated XQuery views with multiple sub-query nestings, complex node constructions, and more order manipulations.

In previous view maintenance solutions, this problem has been addressed in a variety of ways. Some solutions [AMR$^+$98, AFP03, ZGM98] have materialized large auxiliary data beyond the actual view contents. This materialized auxiliary data helps relating objects in the materialized views to the source data they are derived from. Other solutions [LD00, PAGM96, Suc98] have used Skolem functions (or variations of them). See Section 10.2 for more details on related work.

In this chapter we introduce our mechanism for generating semantic identifiers for processed XML nodes. Such semantic ids enable identifier-based fusion. Semantic ids encode derivation and order information for view nodes. Hence, it enables us to understand how view nodes relate to source nodes. Lastly, it enables the support of XML order requirements. While inspired by our earlier work in [ESDR03, ESDR05] (presented in Chapter 3) for order-sensitive XQuery processing, order now is part of the generated semantic identifiers instead of being yet another separate annotation.

Semantic identifiers have two main properties. (1) They are reproducible for corresponding objects. By this we mean, if any two source XML nodes generate the same node in the XML result when they are processed (even at different times) the same identifier is guaranteed to be generated for these two nodes. (2) They are compact in size, since the size of a semantic id depends on the query size and not on size of source data used to derive the node. A semantic id may merge lineage and order information when possible.

Our solution works at the algebraic query representation level. *Phase 1* takes place during the query plan generation and optimization phase. In this phase, our solution takes the query algebra tree as input and automatically defines rules of how lineage and order specifications can be computed for processed nodes. Such rules are defined for each algebra operator. We call such lineage and order specification the *Context Schema*. *Phase 2* takes place during query execution time. In this phase, semantic identifiers are generated for processed nodes based on the *Context Schema* previously defined in the first phase.

## 4.2 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 (in Section 4.3) 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, in actuality it is not. We only define the *Con-*

*text* specifications *schematically* at the schema level of the query execution model. In our case, it is defined for columns in the intermediate XAT tables. We call such schema-level method of defining the *Context* the *Context Schema*. The *Context Schema* is generated during query translation and optimization time. During query execution time, we might need to obtain the *Context* itself (from the *Context Schema*) for some of the individual nodes when we generate or manipulate semantic identifiers. But such access of actual nodes at the instance level is limited to only a few query operations, as we will show later in Section 4.3.

### 4.2.1 Context Schema

We first define the *Context* of a node (or a collection). We then define how the *Context* can be maintained using the *Context Schema*.

**Definition 4.2.1** *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 ($lngVal_1$, $lngVal_2$,.., $lngVal_u$), and $ordCxt$ the Order Context of the node is either a sequence of order values ($ordVal_1$, $ordVal_2$,.., $ordVal_v$) or a $null$ value. A lineage value $lngVal_i$, $1 \leq i \leq u$, can be (1) a source node identifier, (2) a source data value, or (3) a special constant "*" . An order value $ordVal_j$, $1 \leq j \leq v$, 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) as defined above can be (1) derived directly from a specific source node, (2) derived 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), or (4) a composition of one or more of (1), (2), and (3).

We will discuss how different operators define such *Lineage Context* later in this section. For now we state some general guidelines for describing the lineage information for each of the cases above. For (1) we use the respective source node identifier. For (2) we use the value that the node is bound to. For (3) we use a special constant "*", indicating that the collection itself is not bound to any specific source node. This 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). Then the lineage for the entire collection depends on all the lineage of the nodes it is composed of, which we denote by "*". For (4) we use a composition of the respective values.

To understand the *Order Context* ($ordCxt$) for a processed XML node (or a collection) we need to consider three possible scenarios for order among nodes in the XML result. (1) The order among the processed nodes, or even between processed collections of nodes, follows document order[2]. 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 certain source node that reflects the document order of the processed node. (2) 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* $ordCxt$

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

assigned to a processed node is a sequence of order values. (3) 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 the query operation (e.g., $Distinct$).

We will now discuss how the *Lineage Context* and the *Order Context* are encoded using the *Context Schema*. We wish to point out first that the order among cells in each column will now be reflected by the *Context Schema* of the column, as we will see next. Hence, using only the *Context Schema* we can find the order of cells in each column independently from the order in other columns. In some cases, the order among tuples in the XAT table as a whole may be of importance. For instance, when the query involves join operations, then in order to compute the order semantics for cells in columns in the output XAT table (as part of the *Context Schema*) we need to know the order of tuples in the input XAT tables. Hence, we maintain the *Order Schema* (defined in Section 3.3 ) for queries that has join operations. We call that now the *Table Order Schema* to distinguish it from the *Order Context* computed for each individual column in the XAT table. For queries with no join operations, the *Order Context* is sufficient to maintain the order.

We observe that the *Context* of processed nodes that bind to the same query variable (a column in an XAT table) can be described using the same rule. Hence, we can abstract the method of generating such *Context* during query execution using the *Context Schema*. The *Context Schema* is defined for each column in the XAT table. The *Table Order Schema* (introduced in Chapter 3) is used in computing the *Context Schema* for some query operations (e.g., joins) as shown in Table 4.1.

**Definition 4.2.2** *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, that is both lineage and order specifications of nodes or collections of nodes, in that column can be extracted. The* Context Schema *has the following syntax:*

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

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

The lineage specification of a column $col$ 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 nodes in this column has no lineage to nodes in other nodes (we call it a self lineage). 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 $FlexKey$ identifiers that is assigned by the *XML Union* operator. $ColID$s are unique among columns in the list of unioned columns and are used to distinguish columns used as input to the union operation. This is used later when we generate semantic identifiers to ensure the uniqueness of nodes originating from different input columns when unioned. $ColID$s also help in maintaining the order among nodes originating from different columns that are unioned into one collection. An empty ColID indicates that nodes in the column has been assigned identifiers by an earlier *XML Union* operator.

The *Lineage Context* encoded in the *Context Schema* allows us to identify tuples of an XAT table. Hence, we can determine if a newly processed tuple match with previously processed tuple or not. As a result, we can merge matched tuples. To define how the matching is done we first define a set of XAT columns that is to be used in this matching process. We call this set of columns the *Evaluation Context Columns*, or shortly $ECC$.

**Definition 4.2.3** *For any XAT table $T$ we define the set of* Evaluation Context Columns *$ECC$ to be a subset of the set of columns of $T$ where a column $col_j \in ECC$ if the* Lineage Context *of $col_j$ references only itself.*

Hence, for a column $col_j$ in $ECC$ we conclude that (1) the column contains only singleton nodes (does not store collections) and (2) the contents of column $col_j$ can be processed separately from other columns as they have no lineage to contents in other XAT columns.

Other columns in the XAT table that do not belong to $ECC$ uses columns in $ECC$ as part of their *Lineage Context*. Each XAT table has $ECC \neq null$, expect XAT tables generated by the $Combine$ operator. Such XAT tables will always have one column representing a collection created from all the content of the input columns.

Using $ECC$ we now define how two tuples can be matched.

**Definition 4.2.4** *Two tuples $t_1$ and $t_2$ of the same XAT table are considered to be matching, denoted by $t_1 \asymp t_2$, if $\forall$ $col_i \in ECC$, $t_1[col_i] = t_2[col_i]$ with equality by node identifiers if the $col_i$ contains identifiers or by value if it contains values. If $ECC$ is empty then $t_1 \asymp t_2$.*

It is possible that a column in $ECC$ might contain $null$ values. For example, when a tuple $t$ is generated from a *Left Outer Join* where the input tuple of $t$ from the left input source of the operator did not join with any tuple from the right input source of the operator. In this case all columns in $t$ corresponding to columns in the right input source will have null values. We take that into consideration in Proposition 4.2.1.

**Proposition 4.2.1** *When applying the match operator between two tuples $t_1$ and $t_2$ of the same schema, and for a column $col \in ECC$, if $t_1[col_i] = null$ and $t_2[col_i] = null$, then we consider that $t_1[col_i] = t_2[col_i]$.*

### 4.2.2 Rules for 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 4.1 classifies the XAT operators into 12 categories based on the way they handle the *Context Schema*. For each category we specify the columns whose *Context Schema* is affected by the transformation. We show how the *Context Schema* is computed for those columns. All other columns maintain the existing *Context Schema*, meaning that their *Context Schema* does not change as a result of applying the operator. Table 4.1 uses the following conventions:

- $col.ord$: refers to the *Order Context* part of the *Context Schema* of the column $col$
- $col.lng$: refers to the *Lineage Context* part of the *Context Schema* of the column $col$
- $p.col$: refers to the input column $col$ to a tagger pattern $p$
- $T[col_i]$: refers to the column with index i in the XAT table $T$
- $T[col_i].cxtSma$: refers to the *Context Schema* for a column $col_i$ in the XAT table $T$
- $T.OS$: refers to the *Table Order Schema* (as defined in Chapter 3) of the XAT table $T$

| Cat. | Operator *op* | Affected column | Assigned Context Schema |
|---|---|---|---|
| I | $S^{col}_{xmlDoc}$ | $col$ | $()[]$ |
| II | $\Phi^{col'}_{col,path}(T)$<br>$\upsilon^{col'}_{col}(T)$<br>$\rho_{col,col'}(T)$ | $col'$ | $if(col.ord == empty)$, $()[col.lng]$<br>$elseif(col.ord == null)$, $[col.lng]$<br>$else$, $(col.ord)[col.lng]$ |
| III | $\phi^{col'}_{col,path}(T)$ | $col'$ | $if((col.ord == empty)\|\|(col.ord == null))$, $()[]$<br>$else$, $(col.ord + col')[]$ |
| IV | $C_{col}(T)$ | $col$ | $[*]$ |
| V | $T^{col}_{p}(T)$ | $col$ | $if(p.col.ord == empty)$, $()[]$<br>$elseif(p.col.ord == null)$, $[]$<br>$else$, $(p.col.ord)[]$ |
| VI | $\gamma_{col[1..n]}(T, C_{col})$ | $All\ columns$ | if (grouping by id),<br>$(col_1.ord, .., col_n.ord)[col_1.lng, .., col_n.lng]$<br>else if (grouping by value), $[col_1.lng, .., col_n.lng]$ |
| VII | $\overset{x\,col}{\cup}_{col1,col2}(T)$ | $col$ | $if((col_1.ord == empty)\&\&(col_2.ord == empty))$,<br>$()[col_1.lng\{fk_1\}, col_2.lng\{fk_2\}]$<br>$else$, $(col_1.ord, col_2.ord)[col_1.lng\{fk_1\}, col_2.lng\{fk_2\}]$<br>(where $fk_1$ and $fk_2$ are Flexkeys reflecting order) |
| VIII | $\delta_{col}(T)$ | $All\ columns$ | $[col.lng]$ |
| IX | $\times(T_1, T_2)$<br>$\bowtie_c (T_1, T_2)$<br>$\sqsupset\!\bowtie_c(T_1, T_2)$ | $T_1[col_1..col_m]$<br><br>$T_2[col_1..col_n]$ | $for(i = 1; i <= m; i++)$<br>$T_1[col_i].CxtSma = (T_1[col_i].ord + T_2.OS)[T_1[col_i].lng]$<br>$for(i = 1; i <= n; i++)$<br>$T_2[col_i].CxtSma = (T_1.OS + T_2[col_i].ord)[T_2[col_i].lng]$ |
| X | $\sigma_c(T)$, $M(T_1, T_2)$ | $None$ | $N/A$ |
| XI | $\tau_{col[1..n]}(T)$ | $T[col_1..col_m]$ | $for(i = 1; i <= m; i++)$<br>$T[col_i].CxtSma = (col[1..n])[T[col_i].lng]$ |

Table 4.1: Rules for computing the *Context Schema* for different XAT operators.

We now discuss how the *Context Schema* is computed for some operators based on the rulers shown in Table 4.1.

- **Category I.** The $Source$ operator $S^{col'}_{xmlDoc}$ first creates a *Context Schema*. The *Context Schema* assigned to the output column $col'$ is $(()[])$. Based on the semantics of the *Context Schema* in Definition 4.2.2 this means two things. (1) The *Lineage Context* of the node in column $col'$ (representing the entire document) is defined in terms of itself only (reflected by its $FlexKey$) and that it has no lineage to any other node. (2) The order of the node in column $col'$ is also derived from the same $FlexKey$ (indicated by the empty list of order columns "()").

- **Category II.** The *Navigate Collection* operator $\Phi^{col'}_{col,path}(T)$ does not break the nesting relationship as a result of the navigation process. Hence, it does not cause any change to the *Lineage Context* of nodes in the output column $col'$. The collections created in $col'$ still relate to the nodes from which they were extracted from. Hence, $col'$ is assigned a *Lineage Context* that is equal to that of the matching input value $col$ $([col.lng])$. Determining the *Order Context* of $col'$ depends on the order of the input column $col$. If $col$ has an *Order Context* that is derived from its *Lineage Context* (indicated by an empty *Order Context*), the *Order Context* of $col'$ is the same as the lineage of $col'$. If there is no *Order Context* assigned to $col$, then we do not assign an *Order Context* to $col'$. Finally, if there is an *Order Context* assigned to $col$ we assign to $col'$ the same *Order Context*.

  The *XML Unique* operator $\upsilon^{col'}_{col}(T)$ also assigns to the output column

$col'$ a *Lineage Context* that is equivalent to the *Lineage Context* of its input column ($[col.lng]$). The *Name* operator $\rho_{col,col'}(T)$ assigns to the new column $col'$ the same *Lineage Context* as that of old renamed column $col$. *Order Context* computation for these two operators is similar to that of the *Navigate Collection* operator.

- **Category III.** Since the *Navigate Unnest* operator $\phi^{col'}_{col,path}(T)$ performs a navigation operation followed by an unnest operation, it breaks the nesting relationship between the nodes we navigate to and the nodes they are extracted from. Hence, the nodes that are in column $col'$ are no longer dependent on other nodes (the entry point nodes) and their *Lineage Context* is set to their own node identifiers ($[col']$). In terms of *Order Context* for $col'$, if $col$ has no *Order Context* or an empty one, we assign a new empty *Order Context* to $col'$. Namely , the *Order Context* will be set to be equal to the *Lineage Context* of $col'$). If $col$ has an *Order Context* that is not empty, $col'$ gets an order that is composed of the order of $col$ as a major order and the order of the new column $col'$ as minor order. Hence the *Order Context* becomes $(col.ord + col')$, or $(col.ord)$ after optimization since $col'$ is already reflected in the *Lineage Context* part. Note that as a special case, if the *Navigate Unnest* operator navigates to the textual value of a node (text()), the *Order Context* of $col'$ is set equal to the *Order Context* of $col$.

- **Category IV.** The $Combine$ operator $C_{col}(T)$ does not create any new column, rather the contents of the combined column $col$ are merged to form one collection of nodes in a single tuple in the output table.

Hence, the *Context Schema* of column $col$ is changed. The old *Context Schema* assigned to the combined column $col$ is destroyed and a new *Context Schema* is created for it. Since there is only one collection in the output column we assign a constant *Context* to the combined column $col$ that reflects the "All" semantics. This "All" semantic means that lineage of the collection in column $col$ is not related to any particular node(s). We abbreviate that *Context* using $[``*"]$[3]. Note that since we only have one tuple in the output XAT table, there is no order defined between tuples. Hence, we set the *Order Context* to $null$.

- **Category V.** The $Tagger$ operator $T_p^{col}(T)$ constructs new XML nodes in column $col$. These new nodes are assigned new ids. Such ids are derived from the lineage information of nodes used to construct the new node, as we will show later in Section 4.3. Hence, we assign a *Lineage Context* for column $col$ referencing the new ids in the column itself. The *Order Context* of column $col$ is determined based on the *Order Context* of the input column of the tagger pattern ($p.col$) as shown in Table 4.1.

- **Category VI.** The *Group By* operator $\gamma_{col[1..n]}(T, C_{col})$ assigns as the *Lineage Context* to each of the columns in the output XAT table a composition of the grouping columns' *Lineage Context*s. If the grouping operation is value-based, the *Group-By* operator defines a null *Order*

[3]This can be seen as assigning a dummy parent node with an identifier "*" for the collection. Note that at the point when the *Context* $[``*"]$ is assigned, by the $Combine$ operators, we do not worry about uniqueness of such *Context* since we only have one big collection. At later stage when this collection is unioned or merged with other results the *Context* of the collection is expanded to reflect the uniqueness.

*Context* to reflect the fact that there is no order among the created groups. If the grouping operation is id-based (nesting), the *Group-By* operator defines an *Order Context* for all columns in the output XAT table that is equal to the composition of the *Order Context*s of the grouping columns.

- **Category VII.** The *XML Union* operator $\overset{x\,col}{\cup}_{col1,col2}(T)$ 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[4]. 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.

  Note that if the query plan has more than one *XML Union* operator, the identifying $FlexKeys$ (in the $ColID$ list in Definition 4.2.2) used to distinguish between different columns are assigned based on the overall order among columns used as input to all the *XML Union* operators in a depth first traversal for the query plan. For example, assume a query plan that includes two *XML Union* operators with input columns $col1$, $col2$, and $col3$ if the query plan includes $\overset{x\,col}{\cup}_{col1,col2}(T)$ and then $\overset{x\,col'}{\cup}_{col,col3}(T)$, the *Lineage Context* of $col$ is

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

defined as $[col1\{a\}, col2\{b\}]$ and the *Lineage Context* of $col'$ is defined as $[col\{\}, col3\{c\}]$[5]. If another plan that represents the same query defines the operator $\overset{x\,col}{\cup}_{col2,col3}(T)$ and on top of it it defines $\overset{x\,col'}{\cup}_{col1,col}(T)$ in this case the *Lineage Context* of $col$ is defined as $[col2\{b\}, col3\{c\}]$ and the *Lineage Context* of $col'$ is defined as $[col1\{a\}, col\{\}]$. As a result of that, the same order prefix for sematic ids can be produced in the two query plans during sematic ids generation time (Section 4.3). This is important for opening up opportunities for optimization as we discuss in Section 4.6.

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 the union of columns in the *Order Context*s of the source columns.

- **Category VIII.** The *Distinct* operator $\delta_{col}(T)$ does not create any new columns, rather it filters out tuples based on duplicate values in column $col$. The old *Context Schema* of the output column is discarded and a new *Context Schema* $[col]$ is assigned to the output XAT table. The *Distinct* operator destroys order. Hence, hence a null *Order Context* is assigned to the output column.

- **Category IX.** The *Cartesian Product* operator ($\times(T_1, T_2)$), the *Join* operator ($\bowtie_c (T_1, T_2)$), and the *Left Outer Join* operator ($⟕_c(T_1, T_2)$) do not change the *Lineage Context* of columns in the output table. For the *Order Context*, the treatment is different for those columns originating

[5]Assuming that $col1$, $col2$, and $col3$ each has a *Lineage Context* equal to [].

from the left input table than for those originating from the right input column. For each column originating from the left input column we assign to it a new *Order Context* that is a composition of the *Order Context* of the column itself and the *Table Order Schema* of the right table. For each column originating from the right input column we assign to it a new *Order Context* that is a composition of the *Table Order Schema* of the left table with the *Order Context* of the column itself. This ensures that the resulting *Order Context* of columns in the output XAT table reflect the order of left source XAT table as a major order and the order of the right source XAT table as a minor order.

- **Category X.** The $Select$ ($\sigma_c(T_1)$) does not change the *Context Schema* for any of its input columns. The same applies to the *Merge* operator $M(T_1, T_2)$.

- **Category XI.** The *Order By* ($\tau_{col[1..n]}(T)$) does not change the *Lineage Context* for any of its input columns. It only changes the *Order Context* for all columns by assigning it to the order column(s) $col[1..n]$. Such columns should contain the order values.

### 4.2.3 Example for Context Schema Computation

Figure 4.2 shows how the *Context Schema* is defined for columns in the intermediate XAT tables based on the rules shown in Table 4.1. 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

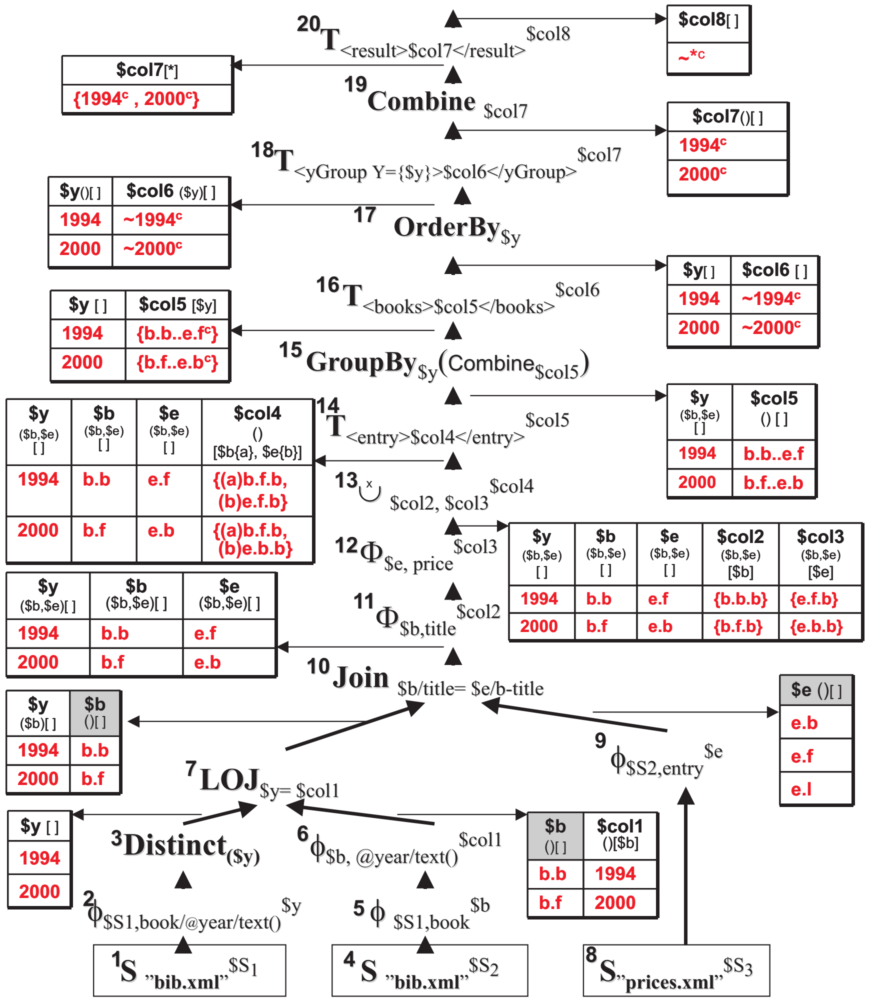

20 T <result>$col7</result> $col8
$col8[ ]
~*c
$col7[*]
{1994c , 2000c}
19 Combine $col7
$col7()[ ]
1994c
2000c
18 T <yGroup Y={$y}>$col6</yGroup> $col7
$y()[ ] $col6 ($y)[ ]
1994 ~1994c
2000 ~2000c
17 OrderBy$y
$y[ ] $col6 [ ]
1994 ~1994c
2000 ~2000c
16 T <books>$col5</books> $col6
$y [ ] $col5 [$y]
1994 {b.b..e.fc}
2000 {b.f..e.bc}
15 GroupBy$y(Combine$col5)
$y ($b,$e) [ ] $col5 () [ ]
1994 b.b..e.f
2000 b.f..e.b
14 T <entry>$col4</entry> $col5
$y ($b,$e) [ ] $b ($b,$e) [ ] $e ($b,$e) [ ] $col4 () [$b{a}, $e{b}]
1994 b.b e.f {(a)b.f.b, (b)e.f.b}
2000 b.f e.b {(a)b.f.b, (b)e.b.b}
13 x $col2, $col3 $col4
12 Φ $e, price $col3
$y ($b,$e) [ ] $b ($b,$e) [ ] $e ($b,$e) [ ] $col2 ($b,$e) [$b] $col3 ($b,$e) [$e]
1994 b.b e.f {b.b.b} {e.f.b}
2000 b.f e.b {b.f.b} {e.b.b}
11 Φ $b,title $col2
$y ($b,$e)[ ] $b ($b,$e)[ ] $e ($b,$e)[ ]
1994 b.b e.f
2000 b.f e.b
10 Join $b/title= $e/b-title
$y ($b)[ ] $b ()[ ]
1994 b.b
2000 b.f
$e ()[ ]
e.b
e.f
e.l
9 ϕ $S2,entry $e
7 LOJ $y= $col1
$y [ ]
1994
2000
3 Distinct($y)
6 ϕ $b, @year/text() $col1
$b ()[ ] $col1 ()[$b]
b.b 1994
b.f 2000
2 ϕ $S1,book/@year/text() $y
5 ϕ $S1,book $b
1 S "bib.xml" $S1
4 S "bib.xml" $S2
8 S "prices.xml" $S3

Figure 4.2: The algebra tree for XQuery expression in Figure 1.2(a) with the *Context Schema* annotation, appearing in subscript font to the right (or below) column names. Shaded column names represent the *Table Order Schema*.

($\$y$) representing the distinct values of years. Based on the rules in Table 4.1 it is assigned a *Lineage* specification that references itself [] and a null *Order* specification (indicating that there is no order semantics for that column). The output XAT table of operator # 6 has two columns, namely $\$b$ and $\$col1$. Column ($\$b$) stores the node identifier of extracted books. The *Lineage* specification assigned to that column is derived from itself [], as this column is obtained through a *Navigate Unnest* operation. The *Order* specification of that column is set to $()$, signifying that it is equal to the *Lineage* specification ($\$b$). 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 4.1). The $LOJ$ operator (# 7) does not affect the *Lineage* specification. It only changes the *Order* specification. It uses the *Table Order Schema* of the input tables (highlighted columns) and the input column's *Order* specification to determine the new *Order* specification. Based on the rules in Table 4.1, the *Order* specification of column $\$y$ is set to $(\$b)$. The *Order* specification of column $\$b$ is not affected because the left input table has no *Table Order Schema*. The $Join$ operator (# 10) also sets only the *Order Context* of its output columns $\$y$, $\$b$, and $\$e$ as shown in Figure 4.2. Note that for the operators on top of operator # 10, the *Table Order Schema* is no longer needed to compute the *Context Schema* rules, since there is no other $Join$ operations. Hence, the *Table Order Schema* is not defined for these 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* operator (operator # 13) creates new collections in column $\$col4$ from the contents of columns $\$col2$ and $\$col3$. Hence, the *Lineage* specification of column $\$col4$ becomes a composition of the *Lineage* specifications of columns $\$col2$ and $\$col3$ which are $\$b$ and $\$e$ respectively. We assign to column $\$col4$ the *Lineage* specification $[\$b\{a\}, \$e\{b\}]$, where $a$ and $b$ are special column source identifiers that are used to uniquely identify each column and at the same time to reflect the relative order among the two columns, as we have discussed above. The *Order* specification of the new column $\$col4$ is derived from the *Order* specifications of both input columns which are $\$b$ and $\$e$ (for column $\$col2$) and $\$b$ (for $\$col3$). Hence, the *Order* specification becomes $(\$b, \$e)$, since removing the redundant $\$b$ will not affect the order semantics. Since this *Order* specification is equivalent to the *Lineage* specification, we simply set the *Order* specification to $()$. The *Tagger* operator (operator # 14) constructs new nodes in column $\$col5$ assigning a *Lineage* specification [] that reflects that the lineage of this column depends only on itself and not on any other columns. The *Order* specification of $\$col5$ is set equal to ()[6]. The *Group By* operator (operator # 15) changes the *Lineage* specifications of all the output columns to be equivalent to the *Lineage* specification of the grouping column $\$y$. It also sets the *Order* specification of the output columns to $null$ since the *Group By* destroys the order among tuples (created groups)[7].

[6]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.

[7]The value-based *Group By* destroys the order among XAT tuples while the id-based *Group By* (representing a nesting operation) would define certain order among the created

The *Tagger* operator (operator # 16) constructs new nodes "books" in column $\$col6$ assigning a "self" *Lineage* specification to it [] and a $null$ *Order* specification because the input column $\$col5$ has a $null$ *Order Context*. The *Order By* operator (operator # 17) sets the *Order Context* of all columns to the ordering column $\$y$. The remaining operators in the algebra tree are easy to follow.

## 4.3 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. This is done during query execution time.

### 4.3.1 From Context Schema to Node Identifiers

**Definition 4.3.1** *The **Semantic Identifier** (*SemID*) is an identifier assigned to a node in the XML result. Such identifier is locally unique (among nodes with the same parent node). It carries lineage information that references the source from which the node is derived. It also encodes local order of the node 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, namely source node or constructed node.*

```
SemID       ::= (OrdPrefix)? + (BaseNodeID | ConstNodeID)
OrdPrefix   ::= "~" |  OverRideOrd
```

groups, as shown in Table 4.1.

```
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 modification[8], 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 base id (a $FlexKey$). The body of *SemID* for a constructed node is composed of a *Lineage Context* value ($lngCxt$), as defined in Definition 4.2.1, and a constant suffix ($^c$) indicating that the id reflects a constructed node. The *Lineage Context*, as we discussed earlier can be a $FlexKey$, a string value from the domain of values of the source XML document, a constant “*”, or a composition of one or more of the above types separated by a delimiter “..”. This *Lineage Context* can be derived from the *Lineage* specification of the *Context schema* during query execution when needed (to create semantic ids) as we will discuss next.

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

In Section 4.2 we have discussed how the *Context* of a node (or a collection of nodes) is specified using the *Context Schema*. No node-level access is needed to maintain such *Context*. We now show how we exploit the knowledge encoded in the *Context Schema* to generate semantic identifiers.

Table 4.2 shows the node-level operations required for performing this task. As shown in Table 4.2, we require node-level access for only four algebra operations out of all the XAT operators shown in the table. Namely, the *Combine*, the *Tagger*, the *XML Union*, and the *Group By* operators. Moreover for these four operators we access only nodes (or values) that are available during execution time. We do not require access of any other nodes (or values) outside this range. For example, we do not access descendants of nodes or perform de-referencing for values. 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.

Table 4.2 shows 3 functions that are used by the four operators *Combine*, *Tagger*, *XML Union*, and *Group By* to generate and maintain the semantic ids. The logic of these functions is shown in Figures 4.3, 4.4, and 4.5. In the actual system, these operations are not performed separately but are integrated within the execution. Hence, they do not place any significant overhead on the normal processing time.

The function $assignOverRidOrd$ is used by the $Combine$ operator to set the order prefix part of the semantic id for combined nodes. This function is also used by the *Group By* to set the order prefix part of the semantic id

| Cat. | Operator *op* | Node Level operation |
|---|---|---|
| I | $S^{col'}_{xmlDoc}$ | $None$ |
| II | $\Phi^{col'}_{col,path}(T)$ <br> $\upsilon^{col}_{col1}(T)$ <br> $\rho_{col1,col}(T)$ | $None$ |
| III | $\phi^{col'}_{col,path}(T)$ | $None$ |
| IV | $C_{col}(T)$ | for each tuple $t$ in $T$ apply $assignOverRidOrd(t, col)$ |
| V | $T^{col}_{p}(T)$ | for each tuple $t$ in $T$ apply $composeNodeIds(t, col, p)$ |
| VI | $\gamma_{col[1..n]}(T, C_{col})$ | The Combine operator uses the function $assignOverRidOrd(t, col)$ |
| VII | $\overset{x\,col}{\cup}_{col1,col2}(T)$ | for each tuple $t$ in $T$ apply $assignColIdPrfx(t, col_1, col_2)$ |
| VIII | $\delta_{col}(T)$ | $None$ |
| IX | $\times(T_1, T_2)$ <br> $\bowtie_c (T_1, T_2)$ <br> $\sqsupset\!\!\bowtie_c(T_1, T_2)$ | $None$ |
| X | $\sigma_c(T)$, $M(T_1, T_2)$ | $None$ |
| XI | $\tau_{col[1..n]}(T)$ | $None$ |

Table 4.2: Node-level operations required for generating the semantic node identifiers.

for nodes in the created groups[9].

```
function $assignOverRidOrd$ (Tuple $t$, ColumnName $col$)
    if ($col.getOrdCxt() == empty$)
        do nothing
    else if ($col.getOrdCxt() == null$)
        for (each node $n_i$in $t[col]$)
            $n_i$.assignOrdPrfx("∼")
    else
        for (each node $n_i$in $t[col]$)
            $n_i$.assignOrdPrfx($n_i$.getOrdCxt())
```

Figure 4.3: Function *assignOverRidOrd* used by the $Combine\ C_{col}(T)$ and the *Group By* $\gamma_{col[1..n]}(T, C_{col})$ operators in Table 4.2.

The function $composeNodeId$ (shown in Figure 4.4) is used by the $Tagger$ operator to generate semantic identifiers for newly constructed nodes. It first generates the semantic identifier body from the *Lineage Context* of the node. Then it generates the order prefix from the *Order Context* of the node as shown in Figure 4.4.

```
function $composeNodeIds$ (Tuple $t$, ColumnName $col$, TaggingPattern $p$)
    for (each new node $n_i$ in $col$)
        $n_i.assignId\ (t[col].getLngCxt())$
        if ($col.getOrdCxt() == null$)
            $n_i.assignOrdPrfx("\sim")$
        else if ($col.getOrdCxt()! = empty$)
            $n_i.assignOrdPrfx(n_i.getOrdCxt())$
```

Figure 4.4: Function $composeNodeIds$ used by the $Tagger$ operator $T_p^{col}(T)$ in Table 4.2.

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

The function $assignColIdPrfx$ (shown in Figure 4.5) is used by the *XML Union* operator to assign the order prefix part of the semantic identifier in a way that reflects the order among XML nodes originating from different sides of the union process. At the same time it ensures the uniqueness of the unioned nodes. The key as an order prefix is equal to the corresponding $ColID$ from the *Context Schema*, as discussed in Section 4.2.2. If $ColID$ for a column is empty, no order prefix is assigned because such an empty $ColID$ reflects that nodes in the column have been already assigned order prefix by an earlier *XML Union* operator. See discussion in Section 4.2.2.

```
function $assignColIdPrfx$ (Tuple $t$, ColumnName $col_1$, ColumnName $col_2$)
    for (each node $n_i$ in $col_1$)
        ordKey = composeKey ($FlexKey_1$, $n_i.getOrdCxt()$)
        $n_i.setOrdCxt(ordKey)$
    for (each node $n_j$ in $col_2$)
        ordKey = composeKey ($FlexKey_2$, $n_j.getOrdCxt()$)
        $n_j.setOrdCxt(ordKey)$
```

Figure 4.5: Function $assignColIdPrfx$ used by the *XML Union* operator $\overset{x\,col}{\cup}_{col1,col2}(T)$ in Table 4.2.

### 4.3.2 Semantic Identifiers Assigned to different types of Nodes

**Base Node Identifiers.** An XML view might expose base nodes in the result. A node in this category should be an exact copy of its source node in the input document with the same tag name, attributes and descendants. A semantic identifier assigned to a base node that is exposed in the view is composed mainly of its base node $FlexKey$ identifier. The source node identifier may not reflect the order of the exposed node in the view extent due to transformations and operations performed by the query. As a result, the identifier assigned to the view node should reflect the new order. Hence, we may associate a prefix order identifier to the $FlexKey$ to reflect the new order that is introduced by the query if the node $FlexKey$ itself does not reflect the node order in the view extent.

**Constructed Node Identifiers.** The *Tagger* operator $T_p^{col}(T)$ is responsible for constructing new nodes in the XAT algebra. The *Tagger* assigns a new semantic id to each constructed node (XML fragment) in the output column $col$ using the $generateNodeId$ function, shown in Figure 4.4. Such semantic identifiers depends on what node (nodes) the constructed node is derived from and what order the node should take in the view extent. Hence, the generated semantic identifier might contain a source node $FlexKey$, a data value, a constant value (*), or a combination of one or more of them, as we have mentioned earlier. This depends on the *Context* of the column which the $Tagger$ uses as input. For example if a constructed node is build on top of a collection with "*" as its *Lineage Context* the generated semantic identifier for this node will be $*^c$. If it is build on top of a node with a *Lineage Context* equal to a data value "car" and a $FlexKey$ $c.b$, and an *Order Context* equal to $b.b$, the generated semantic identifier for this node will be $(b.b)car..c.b^c$.

**Collections and Identifiers.** While each collection of XML nodes is assigned a *Context* during query execution time, we do not actually materialize and allocate identifiers for the collections themselves. Instead, the *Context* of an XML collection is only derived when generating semantic ids for new XML nodes constructed on top of these collections at some later stage. As discussed above, some operators (the $Combine$ operator, the $XMLUnion$ operator, and the *Group By* operator) might manipulate identifiers of nodes in collections by setting the order prefix of the semantic id.

**Identifying Intermediate XAT Tuples.** Our Semantic identifier solution also provides a mechanism to uniquely identify intermediate XAT tuples as we state in Theorem 4.3.1. This enables us to define how propagation of updates is done for single algebra operators, as we will discuss in Section 4.5.

**Theorem 4.3.1** *Tuples in an XAT table $T$ can be uniquely identified by the contents of columns in $T$ that are in $ECC$ of $T$.*

**Proof:** *The* Source *operator $S^{col}_{xmlDoc}$ always come as a leaf operator in any XAT tree. The* Source *operator generates an XAT table Q with one column. This column is the only column in the $ECC$ list of Q. It is clear that this column uniquely identifies the single tuple in Q. We conclude that we always start with an XAT table that has an $ECC$ that reflects the uniqueness among its tuple. Now we show that this uniqueness is always preserved by the $ECC$ of any XAT table generated by other oppressors. We start with operators that may change the number of tuples to the XAT tables or may change the contents of the $ECC$ list:*

- *The* Navigate Unnest *operator $\phi^{col'}_{col,path}(T)$ adds a new column $col'$ to the XAT table. For each reachable node through $path$ from a tuple $tin_i \in T$, a* *tuple $tout_j$ is generated. This means that $tin_i$ might map to multiple tuples in the output XAT table Q. Given our rules for computing the* Context Schema *(in Table 4.1) and the definition of ECC (in Definition 4.2.3), the new column col′ is added to $ECC$. Hence, the $ECC$ of Q now contains the same columns as in $ECC$ of $T$ in addition to $col'$. And since all the nodes in col′ are guaranteed to have unique ids, by the definition of the* Navigate Unnest *operator, we conclude that each tuple in Q is uniquely identified by columns in the $ECC$ of Q.*

- *The* Tagger *operator $T_p^{col}(T)$ adds a new column that contains the newly constructed nodes. Based on the definition of the* Tagger *operator, each tuple in its input XAT table $T$ maps to a single tuple in its output XAT table Q. The new column col is added to the $ECC$ of Q based on the rules in Table 4.1 and the definition of ECC (in Definition 4.2.3). The $ECC$ of Q might retain all columns from the $ECC$ of $T$ if each columns in $ECC$ is used by an operator on top of the* Tagger *operator or if it is referenced by by a another column(s) in Q. In this case the $ECC$ of Q clearly reflects the uniqueness of tuples in Q. Columns that are used to construct new nodes in col might be dropped from Q if they are no longer needed by later operators and if they are not referenced by the* Context Schema *of other columns in Q. In this case the $ECC$ of Q still reflects the uniqueness of tuples in Q, as col is now part of $ECC$ and the ids that were in the columns that are projected out from $ECC$ are now encoded as part of the id of nodes in col.*

- *The* Group By *operator $\gamma_{col[1..n]}(T, C_{col})$ assigns to its output XAT table Q a totally new $ECC$ list. This list contains the grouping columns*

*($col[1..n]$). It is clear that each tuple in Q is uniquely identified by those grouping columns.*

- *The* Distinct *operator $\delta_{col}(T)$ is similar to the* Group By *operator with one exception is that it will have only one column in its output XAT table $ECC$ list (col).*

- *The join family of operators. Tuples in each input source XAT table is identified by the $ECC$ of that table. The output XAT table of any of these operators is assigned an $ECC$ list that is equal to the concatenation of the $ECC$ lists of the input XAT tables. Clearly, such combined $ECC$ will guarantee the uniqueness of tuples in the output XAT table of the operator. One special case here is the* Left Outer Join *operator where columns in its output XAT table Q corresponding to columns from the right input source might contain $null$ values. Even for this case the $ECC$ of Q still reflects the uniqueness of tuples as the contents of columns corresponding to columns in the left input source is guaranteed to be unique for any two tuples that has null values in their corresponding columns in the right input table, by the definition of the* Left Outer Join *operator.*

- *The* Combine *operator $C_{col}(T)$ outputs only one tuple in $Q$. $Q$ will always have one tuple and the $ECC$ of Q is empty.*

- *The* Select *operator $\sigma_c(T)$ filters out tuples that do not satisfy the predicate c. Each tuple in the output XAT table Q maps to a tuple in $T$. The $ECC$ of Q is equal to the $ECC$ of $T$, hence the uniqueness of tuples in Q is maintained by the $ECC$ of Q.*

*Other operators shown in Table 4.1 retain the uniqueness from tuples in their input XAT tables $T$ to tuples in their output XAT tables $Q$. This is because (i) each tuple in $T$ maps to one tuple in $Q$ and vice versa and (ii) $ECC$ of $Q$ is equal to $ECC$ of $T$.*

*Hence, we conclude that Theorem 4.3.1$holds$.*□

## 4.3.3 Example for Semantic Identifier Generation

In the algebra tree in Figure 4.2 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* specification of column $\$col4$ consists of columns $\$b$ and $\$e$. Hence, we derive the body of the semantic id from corresponding projected nodes in these columns. For example, for the first tuple in the XAT table we generate the semantic id body $b.b..e.f$. Since the *Order* specification of $\$col4$ also refers to the same columns "()", we conclude that the semantic id body represents the order. In particular, the semantic id itself reflects the document order of the source "book" node as major order and document order of the source "entry" node as minor order. The generated semantic id becomes $b.b..e.f^c$ after adding the constant prefix "$^c$".

The *Group By* (operator # 15) groups the constructed nodes in column $\$col5$ by the year (column $\$y$). The *Order* specification (in the *Context Schema*) 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 $\$y$. Since the *order* specification 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 "$\sim 1994^c$".

The *Order By* operator (operator # 17) does not perform any node id manipulation. The *Tagger* (operator # 18) constructs new nodes "yGroup". with semantic ids $1994^c$ (on top of the "books" node with id $\sim 1994^c$) and $2000^c$ (on top of the "books" node with id $\sim 2000^c$). Note that the order prefix and the id body for each of these semantic ids were the same. Hence, they have been merged. Next the *Combine* (operator # 19) combines these nodes into a collection. The input column ($\$col7$) for the *Combine* operator has a *Order Context* that is equal to the *Lineage Context* of that column, indicated by the "()". Hence, nodes in the created collection already reflect the order, and no order prefix is assigned. Finally the *Tagger* (operator # 20) 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 4.7(a). Note that the generated semantic ids serve as local unique ids for nodes (among all nodes with the same parent node) and at the same time encode the nodes' local order (semantic ids that start with $\sim$ reflect no-order semantics).

## 4.4 XML Fusion Using Semantic Identifiers

Now we introduce the method of id-based fusion for XML result pieces using our semantic ids. 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 [BDT99]. We here adapt the *Deep Union* operation to the general XML tree model.

**Definition 4.4.1** *The **Deep Union** ($\bigsqcup$) takes two sets of XML trees $T_1$ and $T_2$ and recursively unions them. An XML tree $t = r : ch$ is represented by its root node $r$ and the children list of that root node $ch$. The* Deep Union *first unions the root nodes of the XML trees in $T_1$ and $T_2$ by node identifier and recursively performs deep union on the respective lists of children nodes. The resulting set of XML trees includes all XML trees in the two lists $T_1$ and $T_2$ with only one occurrence of any matching root nodes.*

$T_1 \bigsqcup T_2 = \{\, r : ch_i \bigsqcup ch_j \mid r.id = r_i.id = r_j.id,\ r_i : ch_i \in T_1,\ r_j : ch_j \in T_2\} \cup \{\, r_i : ch_i \mid r_i : ch_i \in T_1,\ \nexists\, r_i : ch_i \in T_2 \text{ where } r_i.id{=}r_j.id\} \cup \{\, r_j : ch_j \mid r_j : ch_j \in T_2,\ \nexists\, r_i : ch_i \in T_1 \text{ where } r_j.id{=}r_i.id\}$

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 view $V(S_1, S_2) = S_1 \bowtie S_2$ is distributive over the *Deep Union* operator, this means that we should be able to maintain the view as follows: $V(S_1 \bigsqcup \triangle S_1, S_2) = (S_1 \bowtie S_2) \bigsqcup (\triangle S_1 \bowtie S_2)$ where $\triangle S_1$ is an update to $S_1$. In other words, we can propagate the update by simply processing $\triangle S_1 \bowtie S_2$ and merging the result with the existing view extent ($S_1 \bowtie S_2$).

Now we illustrate how to apply this to our running example view shown in Figure 1.2 and the source update shown in Figure 1.2. Note here that since the example view involves a self-join we treat each access to the same source as a separate source. Hence, the view in Figure 1.2(a) is defined as $V$($S_1$, $S_2$, $S_3$), where $S_1$ ="bib.xml", $S_2$ ="bib.xml", and $S_3$= "prices.xml". As a result of the update $\triangle$"*bib.xml*" shown in Figure 4.1(a) we need to show that $V$($S_1 \bigsqcup \triangle$ $S_1$, $S_2 \bigsqcup \triangle$ $S_2$, $S_3$) = $V(S_1, S_2, S_3) \bigsqcup V(\triangle S_1, S_2, S_3) \bigsqcup V(S_1, \triangle S_2, S_3) \bigsqcup V(\triangle S_1, \triangle S_2, S_3)$[10]. This is also equal to $V(S_1, S_2, S_3) \bigsqcup V(\triangle S_1, S_2, S_3) \bigsqcup V(S_1', \triangle S_2, S_3)$ by merging the third and the fourth terms and given that $S_1'$ = ($S_1 \bigsqcup \triangle S_1$).

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 discuss the correctness of our solution in Section 4.5.

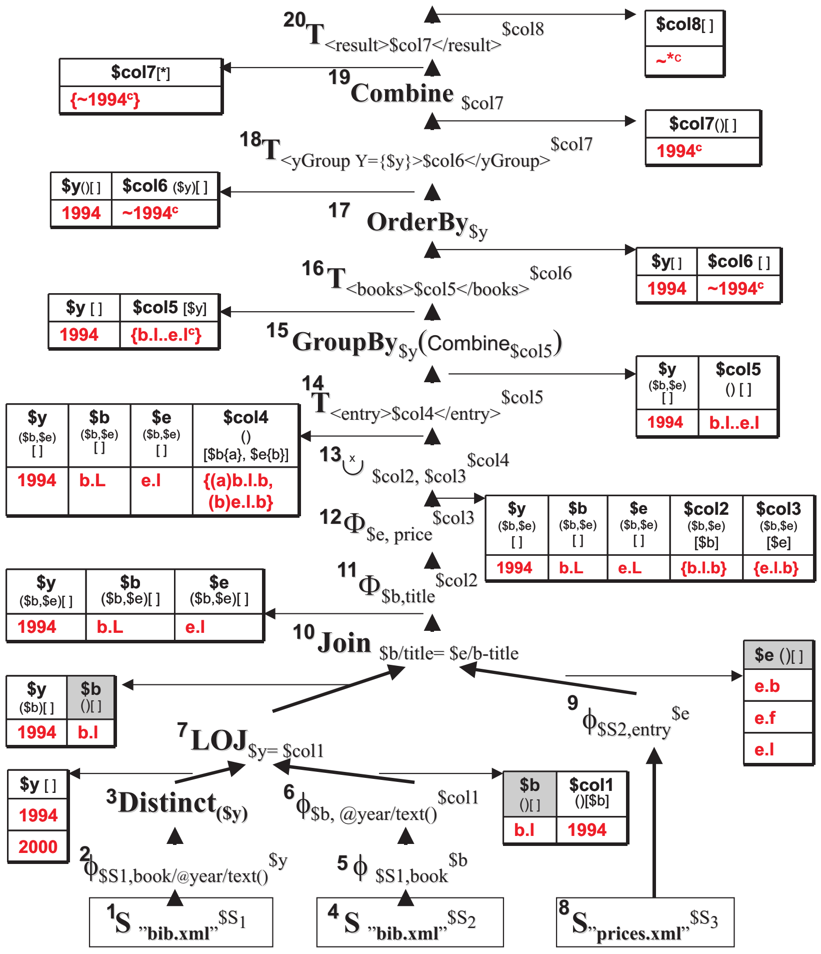
**Example.** First we assign appropriate $Flexkey$s to the new nodes inserted into the source document as shown in Figure 3.1(b). Next we process the incremental parts of the propagation formula above. Namely, $V(\triangle S_1, S_2, S_3)$ and $V(S_1', \triangle S_2, S_3)$. Lastly we fuse the results of propagating the updates with the original view extent $V(S_1, S_2, S_3)$. Figure 4.7(a) shows the original view extent $V(S_1, S_2, S_3)$. Figures 4.7(b) and (c) show the results of executing $V(\triangle S_1, S_2, S_3)$ and $V(S_1', \triangle S_2, S_3)$ respectively. We show how

[10]Note that the query is performing a self join and $S_1$ = $S_2$ = "bib.xml". Hence, an update to "bib.xml" affects both $S_1$ and $S_2$.

$V(S_1', \triangle S_2, S_3)$ was obtained as an example in Figure 4.6. Merging the incremental results, that is propagated updates in Figure 4.7(b) and (c), with the original view extent (in Figure 4.7(a)) using the *Deep Union* operator results in the refreshed view extent shown in Figure 4.7(d). Note 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 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 4.7(d) is equivalent to the result we would get if we were to process the view query over the entire source document after applying the source update to it.

## 4.5 Distributive Property of Views on Insert Updates

Through the use of our semantic identifiers, views can be maintained in a distributive manner on source update inserts. In Section 4.4 we have introduced how such property is defined using the *Deep Union* operator. Now we show how such property holds for the class of views we support. To do this, we need first to show that individual algebraic operators in such views are distributive. For this we define a union operation called *TUnion* ($\overset{t}{\sqcup}$) that we use to define the distributive property of individual algebraic operators that manipulate XAT tables. $TUnion$ is similar to the $union$ operator in the relational context, except that it now also catches the rich semantics of XML



20 T<result>$col7</result> $col8
$col8[ ]
~*c
$col7[*]
{~1994c}
19 Combine $col7
18 T<yGroup Y={$y}>$col6</yGroup> $col7
$col7()[ ]
1994c
$y()[ ] $col6 ($y)[ ]
1994 ~1994c
17 OrderBy$y
16 T<books>$col5</books> $col6
$y[ ] $col6 [ ]
1994 ~1994c
$y [ ] $col5 [$y]
1994 {b.l..e.lc}
15 GroupBy$y(Combine$col5)
14 T<entry>$col4</entry> $col5
$y ($b,$e) [ ] $col5 () [ ]
1994 b.l..e.l
$y ($b,$e) [ ] $b ($b,$e) [ ] $e ($b,$e) [ ] $col4 () [$b{a}, $e{b}]
1994 b.L e.l {(a)b.l.b, (b)e.l.b}
13 $col2, $col3 $col4
12 Φ$e, price $col3
$y ($b,$e) [ ] $b ($b,$e) [ ] $e ($b,$e) [ ] $col2 ($b,$e) [$b] $col3 ($b,$e) [$e]
1994 b.L e.L {b.l.b} {e.l.b}
11 Φ$b,title $col2
$y ($b,$e)[ ] $b ($b,$e)[ ] $e ($b,$e)[ ]
1994 b.L e.l
10 Join $b/title= $e/b-title
$e ()[ ]
e.b
e.f
e.l
$y ($b)[ ] $b ()[ ]
1994 b.l
9 ϕ$S2,entry $e
7 LOJ$y= $col1
$y [ ]
1994
2000
3 Distinct($y)
6 ϕ$b, @year/text() $col1
$b ()[ ] $col1 ()[$b]
b.l 1994
2 ϕ$S1,book/@year/text() $y
5 ϕ $S1,book $b
1 S "bib.xml" $S1
4 S "bib.xml" $S2
8 S "prices.xml" $S3

Figure 4.6: Query processing for $V(S_1', \triangle S_1, S_2)$.

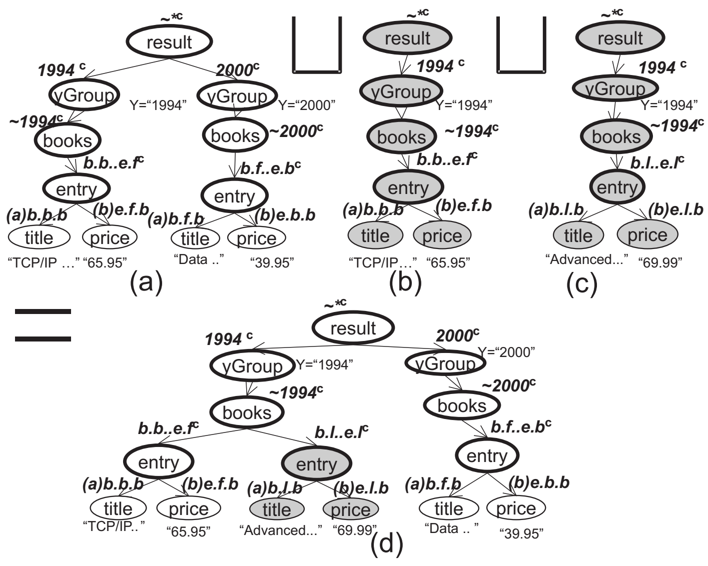


Figure 4.7: The result of computing (a) initial view: $V(S_1, S_2, S_3)$, (b) incremental result: $V(\triangle S_1, S_2, S_3)$, (c) incremental result $V(S_1', \triangle S_2, S_3)$, and (d) the refreshed XML result.

data stored in XAT tables. That is, $TUnion$ operator merges matched XAT tuples.

**Definition 4.5.1** ***TUnion*** *($\overset{t}{\sqcup}$) takes two XAT tables $T_1$ and $T_2$ with the same schema and unions them. The result is an XAT table that contains a single occurrence of each pair of matching tuples (as specified in Definition 4.2.4) and all non-matching tuples from $T_1$ and $T_2$. Each pair of matching tuples ($t_1$ and $t_2$) is merged into a single tuple ($t_{union}$) that contains a single occurrence of the matching nodes from $t_1[col_i]$ and $t_2[col_i]$, $col_i \in ECC$, and the union of the contents of $t_1[col_i]$ and $t_2[col_i]$, $col_i \notin ECC$.*

$T_1 \overset{t}{\sqcup} T_2 = \{t_1 | t_1 \in T_1$ *and* $\nexists t_2 \in T_2$ *where* $t_1 \asymp t_2\} \cup \{t_2 | t_2 \in T_2$ *and* $\nexists t_1 \in T_1$ *where* $t_2 \asymp t_1\} \cup \{t_{union} | \forall t_1 \in T_1, \exists t_2 \in T_2$ *where* $t_1 \asymp t_2\}$

*$t_{union}$ is a tuple that conforms to the schema of $t_1$ and has the same $ECC$ as $t_1$. $t_{union}$ is defined as follows:*

*$\forall col_i \in t_{union}$, $t_{union}[col_i] = t_1[col_i]$, if $col_i \in ECC$ and $t_{union}[col_i] = (t_1[col_i] \cup t_2[col_i])$ if $col_i \notin ECC$. $t_1[col_i] \cup t_2[col_i]$ contains the union of contents of $t_1[col_i]$ and $t_2[col_i]$ where nodes are matched by node id.*

Our semantic identifier solution allows the algebraic operators shown in Table 4.1 to become distributive over the $TUnion$ operator on insert updates. Hence, we can conclude that incremental fusion of separately processed pieces of data gives the same result that we would get if we were to process all the data together.

**Theorem 4.5.1** *Given our semantic identifier solution, all algebraic operators presented in Table 4.1 become distributive over the $TUnion$ operator. For an unary operator op with input XAT table $T$ and $\triangle T$ is an update to $T$ the following equation holds: $op(T \overset{t}{\sqcup} \triangle T) = op(T) \overset{t}{\sqcup} op(\triangle T)$. If op is a binary operator with input XAT tables $T_1$ and $T_2$ and $\triangle T_1$ is an update to $T_1$ this equation holds: $(T_1 \overset{t}{\sqcup} \triangle T_1)\, op\, T_2 = (T_1\, op\, T_2) \overset{t}{\sqcup} (\triangle T_1\, op\, T_2)$.*

**Proof:** *At this point $\triangle T_1$ represents only an insert update, we will discuss the distributive property of operators for other types of updates in Chapter 7. We use the notations shown in Table 4.3 throughout this proof. For simplicity of the discussion we omit details about the order of nodes from this proof and we discuss them separately in Corollary 4.5.2.*

*There are two possible scenarios when the update is applied to the input XAT table $T$:*

| Notation | Meaning |
|---|---|
| $T$ | The input XAT table of operator $op$, $T = \{tin_1, tin_2, .., tin_n\}$ |
| $tin_i$ | A tuple $\in T$, $tin_i = [[c_{i,1}, c_{i,2}, .., c_{i,m}]]$ |
| $c_{i,j}$ | A cell in $tin_i$ that corresponds to content of tuple $tin_i$ for column $col_j$ (i.e., $c_{i,j}= tin_i[col_j]$) |
| $n$ | The number of tuples in $T$ |
| $m$ | The number of columns in the schema of $T$ |
| $Q$ | The output XAT table of operator $op$, $Q = \{tout_1, tout_2, .., tout_y\}$ |
| $y$ | The number of tuples in $Q$ (can be smaller than, bigger than, or equal to $n$ depending on the operator) |
| $\triangle T$ | An update to $T$, $\triangle T = \{xin\}$ |
| $xin$ | A tuple representing the update to $T$, $xin = [[x_1, x_2, .., x_m]]$ |
| $x_j$ | cell in $xin$ corresponding to column $col_j$ (i.e., $x_j= xin[col_j]$) |
| $\triangle Q$ | The result of processing the update $\triangle T$, $\triangle Q = \{xout_1, xout_2, .., xout_x\}$ |
| $xout_o$ | A a tuple resulting from processing the input update $\triangle T$ |

Table 4.3: Notations used in the proof of Theorem 4.5.1.

- *(I) $\exists tin_i \in T$ and $tin_i \asymp xin$. In other words, the update tuple $xin$ matches a tuple $tin_i$ in $T$. In this case we denote the updated XAT table $T$ as $T'$. $T'$= $\{tin_1, .., tin'_i, .., tin_n\}$ where $tin'_i$ replaces $tin_i$ in the original XAT table $T$. $tin'_i$= $[[c'_{i,1}, c'_{i,2}, .., c'_{i,m}]]$ where $c'_{i,j} = c_{i,j} = x_j$ if $col_j \in ECC$ by the definition of the $TUnion$, $c'_{i,j} = (c_{i,j} \cup x_j)$ if $col_j \notin ECC$, $1 \leq j \leq m$. We denote the output XAT table resulting from executing the operator on $T'$ as $Q'$.*

- *(II) $\nexists tin_i \in T$ where $tin_i \asymp xin$. In other words, the update tuple $xin$ does not match any tuple in $T$. In this case we denote the updated XAT table as $T''$, $T''$= $\{tin_1, tin_2, .., tin_n, xin\}$. We denote the output XAT table resulting from executing the operator on $T''$ as $Q''$.*

*Now we prove the theorem for different XAT operators for each of the two scenarios.*

*For the* Navigate Collection *operator we find that:*

- *(I)* $\Phi^{col_k}_{col_j,path}(T \overset{t}{\sqcup} \triangle T) = \Phi^{col_k}_{col_j,path}(T') = Q'$, $Q' = \{tout_1, .., tout'_i, .., tout_n\}$ *where for the input tuple* $tin'_i = tin_i \overset{t}{\sqcup} xin$, $tout'_i$ *is generated,* $tout'_i = [[c_{i,1}, c_{i,2}, .., c_{i,m}, c'_{i,k}]]$ *where* $c'_{i,k}$ *is the result of applying* $path$ *to* $tin'_i$*. Consider that the column we navigate from using* $path$ *is* $col_j$*, the result in* $c'_{i,k}$ *is obtained by navigating from* $c_{i,j}$*.* $c_{i,j}$ *is equal to (1)* $c_{i,j}$ *(and* $x_j$*), if* $col_j \in ECC$ *or (2)* $c'_{i,j} \cup x_j$ *if* $col_j \notin ECC$*.*

  *Now consider that we process* $T$ *and* $\triangle T$ *separately. We find that* $\Phi^{col_k}_{col_j,path}(T) \overset{t}{\sqcup} \Phi^{col_k}_{col_j,path}(\triangle T) = Q \overset{t}{\sqcup} \triangle Q = Q'$ *where* $\triangle Q = \{xout\}$, $xout = [[x_1, x_2, .., x_m, x_k]]$, $x_k$ *is the result of applying* $path$ *to* $\triangle T$*.* $Q \overset{t}{\sqcup} \triangle Q = Q'$ *holds because* $\exists\ tout_i \in Q$, $tout_i \asymp xout$*, by the definition of the* $TUnion$ *and as* $ECC$ *of* $Q$ *is exactly the same as* $ECC$ *of* $T$*. We find that* $tout_i \overset{t}{\sqcup} xout = [[c_{i,1}, c_{i,2}, .., c_{i,m}, (c_{i,k} \cup x_k)]]$ *and* $(c_{i,k} \cup x_k) = c'_{i,k}$ *from above. Hence,* $tout_i \overset{t}{\sqcup} xout = tout'_i$ *from above, by the definition of the* $TUnion$ *and since the two tuples* $tout_i$ *and* $xout$ *match by the same columns as* $tin_i$ *and* $xin$*.*

- *(II)* $\Phi^{col_k}_{col_j,path}(T \overset{t}{\sqcup} \triangle T) = \Phi^{col_k}_{col_j,path}(T'') = Q''$, $Q'' = \{tout_1, tout_2, .., tout_n, xout\}$ *where* $xout = [[x_1, x_2, .., x_m, x_k]]$*, and* $x_k$ *is the result of applying* $path$ *to* $\triangle T$*. Now consider that we process* $T$ *and* $\triangle T$ *separately. We find that* $\Phi^{col_k}_{col_j,path}(T) \overset{t}{\sqcup} \Phi^{col_k}_{col_j,path}(\triangle T) = Q \overset{t}{\sqcup} \triangle Q = Q''$ *where* $Q'' = \{tout_1, .., tout_i, .., tout_n, xout\}$*. This holds because since* $\nexists\ tin_i \in T$ *such that* $tin_i \asymp xin$ *and since the new column* $col_k$ *is a collection column (hence* $col_k \notin ECC$*), by the definition of the* $TUnion$ *we conclude that* $\nexists\ tout_i \asymp xout$*.*

*For the* Navigate Unnest *operator we find that:*

- *(I) $\phi^{col_k}_{col_j,path}(T \overset{t}{\sqcup} \triangle T) = \phi^{col_k}_{col_j,path}(T') = Q'$, $Q' = \{tout_1, .., tout'_i, .., tout_y\}$, $y$ is the total number of nodes reachable from column $col_j$ through $path$, $y \leq n$. For the input tuple $tin_i \asymp xin$ the navigation process through $path$ generates a set of output tuples $\{tout'_{i1}, tout'_{i2}, .., tout'_{io}\}$, $tout'_{iu} = [[c_{i,1}, c_{i,2}, .., c_{i,m}, c_{u,k}]]$, $1 \leq u \leq o$, $o$ is the number of nodes reachable from node(s) in $tin'_i[col_j]$ through $path$. The collection of cells in column $col_k$ resulting from the navigation from $tin'_i$ (each cell is stored in a separate tuple) is $\{c_{1,k}, c_{2,k}, .., c_{o,k}\}$. Such result is generated by navigating from $c'_{i,j}$, where $c'_{i,j} = c_{i,j} = x_j$. If $col_j \in ECC$, then this result is equivalent to the result of navigating from either $c_{i,j}$ or $x_j$. If $col_j \notin ECC$, then the result is generated by navigating from $c'_{i,j}$= ($c_{i,j} \cup x_j$), which is also equal to the union of navigating from $c_{i,j}$ and from $x_j$.*

  *Now consider that we process T and $\triangle T$ separately. We find that $\phi^{col_k}_{col_j,path}(T)$ $\overset{t}{\sqcup}$ $\phi^{col_k}_{col_j,path}(\triangle T)$ = $Q \overset{t}{\sqcup} \triangle Q$ = $Q'$ where $\triangle Q = \{xout_1, xout_2, .., xout_p\}$, $xout_v = [[xout_{v,1}, xout_{v,2}, .., xout_{v,m}, xout_{u,k}]]$, $1 \leq v \leq p$, $p$ is the number of nodes reachable from node(s) in $xin$ through $path$. $Q \overset{t}{\sqcup} \triangle Q = Q'$ holds because if $col_k \in ECC$, for every $xout_v$, $\exists$ $tout_i \in T$ such that $tout_i$ $\asymp$ $xout_v$, by the definitions of the $TUnion$ and the* Navigate Unnest. *Hence each tuple in the resulting collection ($\{xout_1, xout_2, .., xout_p\}$) in $\triangle Q$ matches a tuple in Q and we obtain $Q'$ as above.*

- *(II) $\phi^{col_k}_{col_j,path}(T \overset{t}{\sqcup} \triangle T) = \phi^{col_k}_{col_j,path}(T'') = Q''$, $Q'' = \{tout_1, tout_2, .., tout_n, xout_1, xout_2, .., xout_p\}$, where no input tuple $tin_i \asymp xin$. Hence, the resulting collection in column $col_k$ ($\{c_{1,k}, c_{2,k}, .., c_{n,k}, x_{1,k}, x_{2,k}, .., x_{p,k}\}$) (across all tuples in $col_k$) is clearly composed of two subsets, (i) the first* *subset* $\{c_{1,k}, c_{2,k}, .., c_{n,k}\}$ *resulting from navigating from column* $col_j$ *in all* $tin_i$ *and (ii) the second subset* $\{x_{1,k}, x_{2,k}, .., x_{p,k}\}$ *resulting from navigating from* $col_j$ *in* $xin$*. Now consider that we process* $T$ *and* $\triangle T$ *separately. We find that* $\phi^{col_k}_{col_j,path}(T) \overset{t}{\sqcup} \phi^{col_k}_{col_j,path}(\triangle T) = Q \overset{t}{\sqcup} \triangle Q = Q''$ *where* $\triangle Q = \{xout_1, xout_2, .., xout_p\}$*,* $xout_v = [[x_{v,1}, x_{v,2}, .., x_{v,m}, x_{u,k}]]$*,* $1 \leq v \leq p$*,* $p$ *is the number of nodes reachable from node(s) in* $xin$ *through* $path$*. Since* $\nexists\ tin_i$ *where* $tin_i \not\prec xin$*, we conclude that non of the tuples in* $\triangle Q$ *will match a tuple in* $Q$*, by the definitions of the* $TUnion$ *and the* Navigate Unnest*. Hence,* $Q \overset{t}{\sqcup} \triangle Q$ *is clearly equivalent to* $Q'' = \{tout_1, .., tout'_i, .., tout_n, xout_1, xout_2, .., xout_p\}$ *and the column* $col_k$ *will contain* $(\{c_{1,k}, c_{2,k}, .., c_{n,k}, x_{1,k}, x_{2,k}, .., x_{p,k}\})$ *as above.*

*For the* Combine *operator we find that:*

- *(I)* $C_{col_j}(T \overset{t}{\sqcup} \triangle T) = C_{col_j}(T') = Q'$*,* $Q' = \{tout\} = \{[[(c_{1,j}, .., c'_{i,j}, .., c_{n,j})]]\}$*, where* $c'_{i,j} = c_{i,j} = x_j$ *if* $col_j \in ECC$ *and* $c'_{i,j} = (c_{i,j} \cup x_j)$ *if* $col_j \notin ECC$*, by the definition of the* $TUnion$*. Now consider that we process* $T$ *and* $\triangle T$ *separately. We find that* $C_{col_j}(T) \overset{t}{\sqcup} C_{col_j}(\triangle T) = Q \overset{t}{\sqcup} \triangle Q = \{[[(c_{1,j}, c_{2,j}, .., c_{n,j})]]\} \overset{t}{\sqcup} \{[[(x_j)]]\}\}$*. This is equal to* $\{[[(c_{1,j}, .., c'_{i,j}, , .., c_{n,j})]]\}$ *where* $c'_{i,j} = t_{i,j} = x_j$ *if* $col_j \in ECC$ *and* $c'_{i,j}$ $(c_{i,j} \cup x_j)$ *If* $col_j \notin ECC$*, by the definition of* $TUnion$*. Hence,* $Q \overset{t}{\sqcup} \triangle Q = Q'$ *holds.*

- *(II)* $C_{col_j}(T \overset{t}{\sqcup} \triangle T) = C_{col_j}(T'') = Q''$*,* $Q'' = \{[[(c_{1,j}, c_{2,j}, .., c_{n,j}, x_j)]]\}$*. Now consider that we process* $T$ *and* $\triangle T$ *separately. We find that* $C_{col_j}(T) \overset{t}{\sqcup} C_{col_j}(\triangle T) = \{[[(c_{1,j}, c_{2,j}, .., c_{n,j})]]\} \overset{t}{\sqcup} \{[[(x_j)]]\} = \{[[(c_{1,j}, c_{2,j}, .., c_{n,j}, x_j)]]\} = Q''$*, by the definition of* $TUnion$*.*

*For the* XML Union *operator we find that:*

- *(I)* $\overset{x\,col_k}{\cup}_{col_j,col_l}(T \overset{t}{\sqcup} \triangle T) = \overset{x\,col_k}{\cup}_{col_j,col_l}(T') = Q'$, $Q' = \{tout_1, .., tout'_i, .., tout_n\}$, $tout'_i = [[c1'_{i,1}, .., c'_{i,j}, .., c'_{i,l}, .., c'_{i,m}, c'_{i,k}]]$. *$col_k$ is a collection column. Hence, $col_k \notin ECC$. $c'_{i,j}$ = ($c'_{i,j} \cup x_j$) if $col_j \notin ECC$, and $c'_{i,j} = c_{i,j} = x_j$ if $col_j$ $\in ECC$, by the definition of the $TUnion$. Also, $c'_{i,l}$ = ($c_{i,l} \cup x_l$) if $col_l \notin$ $ECC$, and $c'_{i,l}$ = $c'_{i,l}$ =$x_l$ if $col_l \in ECC$. Hence $c'_{i,k}$ contains the union of all contents of $c_{i,j} \cup x_j \cup c_{i,l} \cup x_l$. Now consider that we process $T$ and $\triangle T$ separately. We find that $\overset{x\,col_k}{\cup}_{col_j,col_l}(T) \overset{t}{\sqcup} \overset{x\,col_k}{\cup}_{col_j,col_l}(\triangle T)$ = $Q \overset{t}{\sqcup} \triangle Q$, $Q$ = $\{tout_1, .., tout_i, .., tout_n\}$, $\triangle Q$ = $\{xout\}$ = $\{[[x_1, x_2, .., x_m, x_k]]\}$. We find that since $xin \asymp tin_i$, and by the definition of the $TUnion$ and since $col_k$ is a collection column by the definition of the* XML Unique, *$xout \asymp$ $tout_i$. Hence, $Q \overset{t}{\sqcup} \triangle Q$ = $\{tout_1, .., tout'_i, .., tout_n\}$ = $Q'$, $tout'_i$ = ($xout$ $\overset{t}{\sqcup}$ $tout_i$) = $[[c'_{i,1}, .., c'_{i,j}, .., c'_{i,l}, .., c'_{i,m}, c'_{i,k}]]$ where $c'_{i,k}$ contains the union of node ids in $c_{i,j} \cup x_j$ and $c_{i,l} \cup x_l$.*

- *(II)* $\overset{x\,col_k}{\cup}_{col_j,col_l}(T \overset{t}{\sqcup} \triangle T) = \overset{x\,col_k}{\cup}_{col_j,col_l}(T'') = Q''$, $Q'' = \{tout_1, tout_2, .., tout_n,$ $xout\}$. *When we consider processing $T$ and $\triangle T$ separately we find that $\overset{x\,col_k}{\cup}_{col_j,col_l}(T) \overset{t}{\sqcup} \overset{x\,col_k}{\cup}_{col_j,col_l}(\triangle T) = Q \overset{t}{\sqcup} \triangle Q$= $\{tout_1, tout_2, .., tout_n, xout\}$ = $Q''$. This holds because since $\exists$ $tin_i \in T$ where $tin_i \asymp xin$ and $col_k$ cannot be $\in ECC$ (given that it is a collection column), we conclude that $\exists$ $tout_i \in$ $Q$ where $tout_i \asymp xout$.*

*For the* XML Unique *operator we find that:*

- *(I)* $\upsilon^{col_k}_{col_j}(T \overset{t}{\sqcup} \triangle T) = \upsilon^{col_k}_{col_j}(T') = Q'$, $Q' = \{tout_1, .., tout'_i, .., tout_n\}$, $tout'_i$ = $[[c'_{i,1}, .., c'_{i,j}, .., c'_{i,m}, c'_{i,k}]]$. *$col_j$ and $col_k$ are both collection columns (by the definition of the* XML Unique*). Hence, $col_j \notin ECC$ and $col_k \notin ECC$. For $tin_i \asymp xin$, $c'_{i,j}$ = ($c_{i,j} \cup x_j$). The resulting collection in $c'_{i,k}$ con-*

*tains all the node id in ($c_{i,j} \cup x_j$) after eliminating duplicate ids. Now consider that we process $T$ and $\triangle T$ separately. We find that $\upsilon_{col_j}^{col_k}(T) \overset{t}{\sqcup} \upsilon_{col_j}^{col_k}(\triangle T)$ = $Q \overset{t}{\sqcup} \triangle Q$, where $\triangle Q$ = $\{xout\} = \{[[x_1, x_2, .., x_m, x_k]]\}$. We find that since $\exists$ $xin$, $xin \asymp tin_i$, and since $col_k$ is a collection column by the definition of the* XML Unique *and of the $TUnion$, $xout \asymp tout_i$. Hence, $Q \overset{t}{\sqcup} \triangle Q$ = $\{tout_1, .., tout_i', .., tout_n\}$ = $Q'$, $tout_i'$ = ($xout \overset{t}{\sqcup} tout_i$) = $[[c_{i,1}', .., c_{i,j}', .., c_{i,m}', c_{i,k}']]$ where $c_{i,k}'$ contains all the node ids in $c_{i,j} \cup x_j$ after eliminating duplicate ids.*

- *(II) $\upsilon_{col_j}^{col_k}(T \overset{t}{\sqcup} \triangle T) = \upsilon_{col_j}^{col_k}(T'') = Q''$, $Q''$ = $\{tout_1, tout_2, .., tout_n, xout\}$. When we consider processing $T$ and $\triangle T$ separately we find that $\upsilon_{col_j}^{col_k}(T)$ $\overset{t}{\sqcup}$ $\upsilon_{col_j}^{col_k}(\triangle T)$ = $Q \overset{t}{\sqcup} \triangle Q$= $\{tout_1, tout_2, .., tout_n, X\}$ = $Q''$. This holds because $\nexists$ $tin_i \in T$ where $tin_i \asymp xin$ and $col_k$ cannot be $\in$ $ECC$ (as $col_k$ is a collection column), we conclude that $\nexists$ $tout_i \in Q$, where $tout_i \asymp xout$.*

*For the* Tagger *operator we find that:*

- *(I) $T_p^{col_k}(T \overset{t}{\sqcup} \triangle T)$ = $T_p^{col_k}(T')$= $Q'$, $Q'$ = $\{tout_1, .., tout_i', .., tout_n\}$, $tout_i'$ = $[[c_{i,1}', .., c_{i,j}', .., c_{i,k}']]$. $col_j$ is the input column for $p$ and $col_k$ is result column containing the newly constructed nodes. The newly constructed node in $col_k$ is assigned an id that is generated from the* Context *assigned to $col_j$, as we have discussed earlier in Section 4.3. This* Context *is obtained from the value in the cell corresponding to the* Context Schema *column(s) (which is always $\in$ $ECC$) of $col_j$. We denote the* Context Schema *column for $col_j$ as $col_y$. (i) If $col_j \in ECC$ then $col_y$ must be equal to $col_j$ and $c_{i,j}'$= $c_{i,j}$ = $x_j$. Hence, the newly constructed node created in $c_{i,k}'$ is assigned an id that is derived from the node in $c_{i,j}$ which will be also equal to the id we can* *derive from the node in $x_j$. (ii) If $col_j \notin ECC$, and since $col_y \in ECC$, the newly constructed node created in $c'_{i,k}$ is assigned an id that is derived from the node the node in $c_{i,y}$, $c'_{i,j}$= $(c_{i,j} \cup x_j)$.*

*Now consider that we process $T$ and $\triangle T$ separately. We find that $T_p^{col_k}(T)$ $\overset{t}{\sqcup}$ $T_p^{col_k}(\triangle T)$ = $Q \overset{t}{\sqcup} \triangle Q$ = $Q'$. This holds because (i) if $col_j \in ECC$, the node in $c'_{i,k}$ is assigned an id that is derived from $c_{i,j}$ and the node in $x_k$ is assigned an id that is derived from $x_j$, and since $c_{i,j}$ = $x_j$, we conclude that $c_{i,k}$ = $x_k$. (ii) if $col_j \notin ECC$, the node in $c'_{i,k}$ is assigned an id that is derived from $c_{i,y}$ and the node in $x_k$ is assigned an id that is derived from $x_y$, and since $c_{i,y}$ = $x_y$ we conclude that $c_{i,k}$ = $x_k$. We finally conclude that $xout \asymp tout_i$.*

- *(II) $T_p^{col_k}(T \overset{t}{\sqcup} \triangle T)$ = $T_p^{col_k}(T'')$= $Q''$, $Q''$ = $\{tout_1, tout_2, .., tout_n, xout\}$. When we process $T$ and $\triangle T$ separately, we find that $T_p^{col_k}(T) \overset{t}{\sqcup} T_p^{col_k}(\triangle T)$ = $Q \overset{t}{\sqcup} \triangle Q$ = $\{tout_1, tout_2, .., tout_n\} \overset{t}{\sqcup} \{xout\}$ = $\{tout_1, tout_2, .., tout_n, xout\}$ = $Q''$. This hold because since $\nexists$ $tin_i \in T$ where $tin_i \asymp xin$, we conclude that $\nexists$ $tout_i \in Q$ where $tout_i \asymp xout$, by the definition of* Tagger *operator and the $TUnion$.*

*The* Source *operator differs from the other operators in that it takes an XML document ($xmlDoc$) as input instead of an XAT table. Hence we represent an update to the source of the* Source *operator as $\triangle xmlDoc$ and we use the* Deep Union *instead of the* TUnion *to apply the update to its input. For the* Source *operator we find that:*

- *(I) $S^{col_k}_{xmlDoc \bigsqcup \triangle xmlDoc}$ = $S^{col_k}_{xmlDoc'}$ = $Q'$, $Q'$ = $\{tout\}$ = $\{[[c_{1,1}]]\}$. When we process $T$ and $\triangle T$ separately, we find that $S^{col_k}_{xmlDoc} \overset{t}{\sqcup} S^{col_k}_{\triangle xmlDoc}$ = $Q \overset{t}{\sqcup}$*

*$\triangle Q$= $\{[[c_{1,1}]]\} \overset{t}{\sqcup} \{[[x_{1,1}]]\}$ = $\{[[c_{1,1}]]\}$= $Q'$, since $c_{1,1}$ = $x_{1,1}$ as the update must have the same root id as the updated XML document root id.*

- *(II) The case in which $xmlDoc$ does not match $\triangle xmlDoc$ by root node id is not possible to occur under our assumptions because it means that the source update inserts an entire new document (with new root node). This is not considered an update to the existing source document, hence we do not deal with it.*

*The* Expose *operator is also different than other operators. It takes an XAT table as input and generates a set of XML document(s) as output (represented using their root node ids). The* Expose *operator simply takes the nodes (or collections) in column $col_j$ and expose them in the result as root nodes.*

- *(I) $\epsilon_{col_j}(T \overset{t}{\sqcup} \triangle T)$ = $\epsilon_{col_j}(T')$ = $Q'$, $Q'$ = $\{tid_1$, $tid_2$,.., $tid_y\}$ is the set of exposed XML fragments roots $tid_i$, $y$ is the total number of nodes in $col_j$ in $T'$. It is easy to see that this result is equal to the result we obtain if we process T and $\triangle T$ separately. In other words, $\epsilon_{col_j}(T) \overset{t}{\sqcup} \epsilon_{col_j}(\triangle T)$ = $Q'$.*

- *(II) $\epsilon_{col_j}(T \overset{t}{\sqcup} \triangle T)$ = $\{tid_1, tid_2, .., tid_u\} \cup \{xid_1$, $xid_2$,..,$xid_v\}$ = $Q''$, where $u$ is the total number of nodes in $col_j$ in $T$, $v$ is the total number of nodes in $xin_j$. It is easy to see that this is equal to the result we get if we process T and $\triangle T$ separately. In other words $\epsilon_{col_j}(T) \overset{t}{\sqcup} \epsilon_{col_j}(\triangle T)$ = $Q''$*

*For the purpose of this proof we consider a* Group By *operator that takes a $Combine$ operator as its input function, hence performing a grouping operation (by value) or a nesting operation (by id). We also consider that the grouping column list contains only one column $col_j$. We find that:*

- *(I)* $\gamma_{col_j}(T \overset{t}{\sqcup} \triangle T, C_{col_k}) = \gamma_{col_j}(T', C_{col_k}) = Q'$, $Q' = \{tout_1, .., tout'_q, .., tout_y\}$ *where the input tuple* $tin'_i = tin_i \overset{t}{\sqcup} xin$ *maps to the output tuple* $tout'_q$ *that represents the group that* $tin'_i$ *belongs to. By the definition of the* Group By*, a grouping column* $col_j$ *in* $T$ *has to be a non-collection column. In* $Q$*,* $col_j$ *continues to be a non-collection column. It now becomes* $\in ECC$*. Now consider that we process* $T$ *and* $\triangle T$ *separately. We find that* $\gamma_{col_j}(T, C_{col_k}) \overset{t}{\sqcup} \gamma_{col_j}(\triangle T, C_{col_k}) = Q \overset{t}{\sqcup} \triangle Q = \{tout_1, tout_2, .., tout_y\} \overset{t}{\sqcup} \{xout\} = \{tout_1, .., tout'_q, .., tout_y\} = Q'$*, where* $tout'_i = tout_i \overset{t}{\sqcup} xout$*. This holds because since* $\exists\ tin_i \in T$*,* $xin \in \triangle T$ *and* $tin_i \asymp xin$ *and since* $tin_{i,j} = x_j$*, hence,* $xout \asymp tout_q$*, and we conclude that* $tout'_i = tout_i \overset{t}{\sqcup} xout$*.*

- *(II)* $\gamma_{col_j}(T \overset{t}{\sqcup} \triangle T, C_{col_k}) = \gamma_{col_j}(T'', C_{col_k}) = Q''$*. The update tuple* $xin \not\asymp tin_i$*, yet* $tin_{i,j}$ *may be equal to* $xin_j$ *or may not. If* $t_{i,j} = x_j$*,* $Q'' = \{tout_1, .., tout'_o, .., tout_y\}$*, where* $tout'_i$ *is a tuple that groups* $tin_i$ *and* $xin$*. If* $tin_{i,j} \neq x_j$*,* $Q'' = \{tout_1, tout_2, .., tout_y, xout\}$*,* $xout$ *represents a new group. Now consider that we process* $T$ *and* $\triangle T$ *separately. We find that* $\gamma_{col_j}(T, C_{col_k}) \overset{t}{\sqcup} \gamma_{col_j}(\triangle T, C_{col_k}) = Q \overset{t}{\sqcup} \triangle Q = \{tout_1, tout_2, .., tout_p\} \overset{t}{\sqcup} \{xout\} = Q''$*.* $Q'' = \{tout_1, .., tout'_i, .., tout_p\}$*,* $tout'_{i,j} = (tout_{i,j} \overset{t}{\sqcup} xout)$ *if* $t_{i,j} = x_j$*.* $Q'' = \{tout_1, tout_2, .., tout_p, xout\}$ *if* $t_{i,j} \neq x_j$*.*

*Other XAT relational-equivalent operators do not create collections or manipulate them. Hence, it is easier to show how the distributive property holds for each of them. We give, below , the proof for some of those operators as examples. Correctness of other XAT relational-equivalent operators can be easily shown in a similar manner.*

*For the* Select *operator with a simple predicate (predicates defined over non-*

*collection columns) we find that:*

- *(I)* $\sigma_c(T \overset{t}{\sqcup} \triangle T) = \sigma_c(T') = Q'$, $T' = \{tin_1, .., tin_i', .., tin_n\}$, *where* $tin_i'$ = *(*$tin_i \overset{t}{\sqcup} xin$*).* $Q' = \{tout_1, .., tout_u', .., tout_y\}$ *where* $tout_u'$ *in* $Q$ *corresponds to* $tin_i'$ *in* $T$*, assuming that* $tin_i'$ *(also,* $tin_i$ *and* $xin$*) passes the selection predicate,* $1 \leq u \leq y$*,* $1 \leq y \leq n$*. Now consider that we process* $T$ *and* $\triangle T$ *separately. We find that* $\sigma_c(T) \overset{t}{\sqcup} \sigma_c(\triangle T) = Q \overset{t}{\sqcup} \triangle Q = \{tout_1, tout_2, .., tout_y\} \overset{t}{\sqcup} \{xout\} = Q'$*. Since both* $tin_i$ *and* $xin$ *pass the predicate,* $Q'' = \{tout_1, .., tout_u', .., tout_n\}$*, where* $tout_u'$ = *(*$tout_u \overset{t}{\sqcup} xout$*), as above.*

- *(II)* $\sigma_c(T \overset{t}{\sqcup} \triangle T) = \sigma_c(T'') = Q''$*, where* $\not\exists\ tin_i \in T$ *and* $tin_i \asymp xin$*.* $Q'' = \{tout_1, tout_2, .., tout_y, xout\}$*, assuming the* $xin$ *passes the selection predicate and maps to* $xout$ *in the output. When processing* $T$ *and* $\triangle T$ *separately we find that* $\sigma_c(T) \overset{t}{\sqcup} \sigma_c(\triangle T) = Q \overset{t}{\sqcup} \triangle Q = \{tout_1, tout_2, .., tout_y\} \overset{t}{\sqcup} \{xout\} = \{tout_1, tout_2, .., tout_n, xout\}$*=* $Q''$*, since* $\not\exists\ tout_u \in Q$ *and* $tout_u \asymp xout$*, by the definition of the* $TUnion$*.*

*For the* Distinct *operator we find that:*

- *(I)* $\delta_{col}(T \overset{t}{\sqcup} \triangle T) = \delta_{col}(T') = Q'$, $Q' = \{tout_1, .., tout_u', .., tout_y\}$, $1 \leq y \leq n$*,* $n$ *is the number of tuples in* $T$*.* $tout_u'$ *is the result of processing the* Distinct *operator over the tuple* $tin_i'$ = *(*$tin_i \overset{t}{\sqcup} xin$*). Note that* $col_j$ *is a non-collection column, by the definition of the* Distinct *operator. Now consider that we process* $T$ *and* $\triangle T$ *separately. We find that* $\delta_{col}(T) \overset{t}{\sqcup} \delta_{col}(\triangle T) = Q \overset{t}{\sqcup} \triangle Q$*,* $\triangle Q = \{xout\}$*.* $Q \overset{t}{\sqcup} \triangle Q = Q'$ *holds because* $\exists\ tin_i \in T$ *such that* $tin_i \asymp xin$*. The* Distinct *operator generates* $xout$ *from* $xin$*, we find that* $xout \asymp tout_u$ *and* $t_{u,j} = x_j$ *by the definition of the* $TUnion$*.*

- *(II)* $\delta_{col}(T \overset{t}{\sqcup} \triangle T) = \delta_{col}(T'') = Q''$, $Q'' = \{tout_1, tout_2, .., tout_y, xout\}$. *When we process T and* $\triangle T$ *separately we find that* $\delta_{col}(T) \overset{t}{\sqcup} \delta_{col}(\triangle T) = Q \overset{t}{\sqcup} \triangle Q = Q''$, $\triangle Q = \{xout\}$. $Q \overset{t}{\sqcup} \triangle Q = Q''$ *holds because* $\nexists\, tin_i \in T$ *such that* $tin_i \asymp xin$. *We conclude that* $xout \not\asymp tout_u$, $tout_u \in Q$ *by the definition of the* $TUnion$.

*The* Theta Join *operator takes two input XAT tables. We denote the left input XAT table as* $T_1$ *(with* $nl$ *number of tuples) and the right XAT table as* $T_2$ *(with* $nr$ *number of tuples). We assume the update* $\triangle T_1$ *updates the left input XAT table. We find that:*

- *(I)* $\bowtie_c (T_1 \overset{t}{\sqcup} \triangle T_1, T_2) = \bowtie_c (T_1', T_2) = Q'$. $Q' = \{tout_1, .., tout'_{u1}, tout'_{u2}, .. tout'_{up}, .., tout_y\}$, $1 \leq y \leq nl * nr$, *nl is the number of tuples in* $T_1$, $nr$ *is the number of tuples in* $T_2$. *The subset of Q containing* $\{tout'_{u1}, tout'_{u2}, ..tout'_{up}\}$ *represents tuples resulting from joining the input tuple* $tin'_i = (tin_i \overset{t}{\sqcup} xin)$ *from* $T_1$ *with tuples in* $T_2$*, on the predicate c. Now consider that we process* $T_1$ *and* $\triangle T_1$ *separately. We find that* $\bowtie_c (T_1, T_2) \overset{t}{\sqcup} \bowtie_c (\triangle T_1, T_2) = \{tout_1, tout_2, .., tout_y\} \overset{t}{\sqcup} \{xout_1, xout_2, ..xout_p\} = Q'$, $xout_u$ *is the tuple resulting from joining* $xin$ *with a tuple in* $T_2$, $1 \leq u \leq p$. *This holds because each tuple* $xout_o$ *in* $\triangle Q$ *matches a tuple* $tout_u$ *in Q, since all columns* $col_j$ *in Q maintain the same status with respect to* $ECC$ *as its corresponding column in* $T_1$ *or* $T_2$*, and by the definition of the* $TUnion$.

- *(II)* $\bowtie_c (T_1 \overset{t}{\sqcup} \triangle T_1, T_2) = \bowtie_c (T_1'', T_2) = Q''$. $Q'' = \{tout_1, tout_2, .., tout_y, xout_1, xout_2, .., xout_p\}$, $1 \leq y \leq nlbl$, *nl is the number of tuples in* $T_1$, $nr$ *is the number of tuples in* $T_2$, $1 \leq p \leq nr$. *When we process* $T_1$ *and* $\triangle T_1$ *separately we find that* $\bowtie_c (T_1, T_2) \overset{t}{\sqcup} \bowtie_c (\triangle T_1, T_2) = \{tout_1, tout_2, .., tout_y\}$

$\overset{t}{\sqcup}$ $\{xout_1, xout_2, ..xout_p\}$ = $\{tout_1, tout_2, tout_y, xout_1, xout_2, .., xout_p\}$ = $Q''$. *This holds because since* $\not\exists$ $tin_i \in T_1$ *such that* $tin_i \asymp xin$, *and given that columns in Q maintain there status with respect to* $ECC$.

*The* Cartesian Product $\times(T_1, T_2)$ *operator follows the same logic as that of the* Theta Join *operator. This also holds for The* Left Outer Join $\sqsupset\!\!\bowtie_c(T_1, T_2)$ *operator on an update it its left input source. Updates to the right input of the* Left Outer Join *operator requires a special treatment from the* Theta Join *operator. This is because the* Left Outer Join *operator may produce XAT tuples with columns corresponding to* $T_2$ *possibly containing* $null$ *values. Since we consider insert updates only at this point, one simple solution that enables this operator to be distributive is to extend the definition of the* match *operator (Definition 4.2.4) to allow what we call partial tuple matching. In particular, two tuples are matched by comparing all non* $null$ *values in their corresponding columns. Given this simple extension, the logic used for the showing the distributivity of the* Theta Join *operator directly applies to the* Left Outer Join *operator. Note that this solution will not work if we consider delete updates and/or aggregate functions. This is because it might propagate some duplicate results. In an insert only environment, such duplicate results merge with previously created result causing no effect on it. We propose a general solution for maintaining the* Left Outer Join *operator in Section 7.4 that supports different updates and aggregation.*

*The* Merge $M(T_1, T_2)$ *operator can be generally maintained using the same logic as that of join operators. Yet, the* Merge *operator can propagate updates from each of its source independently without requiring knowledge of the other source, see the definition of the* Merge *operator in Section 2.2. Hence, a more efficient* *treatment for the* Merge *operator can be adapted that avoids the unnecessary access to $T_2$. The* Merge *operator $M(T_1, T_2)$ can be maintained on an update $\triangle T_1$ to $T_1$ as follows: $M(T_1 \overset{t}{\sqcup} \triangle T_1, T_2) = M(T_1, T_2) \overset{t}{\sqcup} M(\triangle T_1, \{d_{T_2}\})$, where $d_{T_2}$ is the default tuple of table $T_2$. This default tuple contains a $null$ value in all columns. We will revisit the default tuple later in Section 7.4.*

*We conclude that Theorem 4.5.1 holds for all the operators shown in Table 4.1.*

□

An insert update to the right input source of binary operator is generally treated in the same way as that of updates to the left input source, as shown in Theorem 4.5.1. In particular, $T_1 \, op \, (T_2 \overset{t}{\sqcup} \triangle T_2) = (T_1 \, op \, T_2) \overset{t}{\sqcup} (T_1 \, op \, \triangle T_2)$. This can directly be shown using the same logic used in proving Theorem 4.5.1. One exception to this is the *Left Outer Join* operator. An insert update to the right input source of a *Left Outer Join* operator is maintained as follows: $T_1 ⟕_c (T_2 \overset{t}{\sqcup} \triangle T_2) = (T_1 ⟕_c T_2) \overset{t}{\sqcup} (T_1 \bowtie_c T_2)$. This treatment assumes the partial matching strategy proposed when discussing the *Left Outer Join* operator in the proof of Theorem 4.5.1. We present a general propagation rule for the *Left Outer Join* operator on all update types in Chapter 7.

**Corollary 4.5.2** *Given the* Context Schema *rules in Table 4.1, the correct order among any two cells $c_1$ and $c_2$ in a column $col$ of an XAT table $T$, can be derived.*

The *Order Context* part of the *Context Schema* defines how the order among cells in a single column can be derived. This is similar in spirit to what the *Table Order Schema* defined in Section 4.3 do. The main difference is that the *Order Context* focuses only on a single column while the *Table Order Schema* handles the order of entire tuples. The same logic used for proving the correctness of the order maintained by the *Table Order Schema*, as shown in Theorem 3.3.1, can directly be used to prove the correctness of order maintained by the *Order Context*. More discussion on *Order Context* and *Table Order Schema* can be found in Section 4.7.

The order among nodes in sequences (in XAT table cells) is maintained through assigning overriding order prefix as discussed in Section 4.3. We have shown in Theorem 3.3.2 the correctness of this treatment. The same treatment is also adapted now (but we refer to it as the order prefix id) as part of the semantic id with one difference that is now order prefix is derived from the *Context Schema* instead of the *Table Order Schema*. This enables ids to be more compact, as we discuss in Section 4.7.

**Corollary 4.5.3** *For a sequence of $m$ updates $X$ to an input XAT table $T$ of an operator $op_{in}^{out}(T)$, where each update in $X$ affects one single unique tuple from $T$, the order of propagating and applying updates in $X$ has no effect on the final result.*

Since each update in $X$ is unique, for those operators that do not group tuples, each update propagated to the output XAT table $Q$ of $op_{in}^{out}(T)$ is unique, we can conclude, by using Theorem 4.5.1, that Corollary 4.5.3 holds. For those operators that group the tuples in the output (e.g., *Group By* and *Combine* operators), each update propagates to an insertion of node(s) into the appropriate group in $Q$. Since each inserted node encodes its relative order as we have shown in Section 4.3, the order of propagating and applying the different updates does not make any difference. Hence, Corollary 4.5.3 holds.

We have shown so far the correctness of propagating single updates through a single update operator. We now discuss the correctness of propagating updates through a view composed of one or more algebra operator.

**Theorem 4.5.4** *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 $E$. Let $\triangle S_i$ be an update to the data source $S_i$ of $E$, $1 \leq i \leq n$. 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 $\triangle S_i$ using the* Deep Union *operator (defined in Definition 4.4.1). We find that $V^{rec}$ = $V'$.*

**Proof:** *We prove Theorem 4.5.4 by induction on the height $h$ of the XAT algebra tree $E$ representing the view $V$.*

**Base Case:** *Since the leaf of the XAT tree should always be a $Source$ operator and the root of an XAT tree that extracts the result in XML format should be an $Expose$ operator, the base case $h = 1$ means that $E$ has two levels. The algebra tree $E$ has two operators an $Expose$ operator $\epsilon_{col'}(T)$ on top of a $Source$ operator $S_{S_1}^{col'}$. In this case the output XAT table of the $Source$ operator is the input for the $Expose$ operator. The result of executing $E$ over $S_1$ is the view extent $V$, which is equivalent*[11] *to the source XML document $S_1$, because $E$ provides an entry point to $S_1$ and then exposes $S_1$ without any manipulation. Now when the update $\triangle S_1$ is applied to $S_1$ the result of executing $E$ over $\triangle S_1$ is $V'$, which is equivalent to $\triangle S_1$. If we execute $E$ over the updated source $S_1'$ we get $V^{rec}$, which is equivalent*

[11] By equivalent here we mean deep equal.

*to $S'_1$. Since $S_1 \bigsqcup \triangle S_1$ = $S'_1$ we conclude that $V^{rec}$ = $V'$.*

**Induction Hypothesis:** *For a view $V$ with algebra tree $E$ of height $h$, $V^{rec}$ = $V'$ holds.*

**Induction Step:** *We now show that $V' = V^{rec}$ holds for an XAT algebra tree of height $h + 1$.*

*When processing $E$ over the update $\triangle S_i$, the $Source$ operator at the leaf accessing $\triangle S_i$ produces one tuple in its output XAT table. This tuple representing an intermediate propagated update is then processed by the next operator producing possibly a sequence of updates (tuples), due to the fact that some operators may produce more than one tuple by processing one input tuple (e.g.,* Navigate Unnest *and* Joins*). Each tuple in the propagated sequence of updates for any operator in $E$ might generate a unique tuple or might merge with other propagated update (in case of grouping operator), as we have discussed in Corollary 4.5.3. The $Expose$ at the root of $E$ extracts the final result constructed incrementally by all operators in $E$ while processing intermediate propagated updates. By Theorem 4.5.1 and Corollary 4.5.3 , the intermediate updates generated as a result of the source update $\triangle S_i$ are correctly propagated to the root $Expose$ operator of $E$. Hence, the final result obtained by processing $E$ over $\triangle S_i$ will correctly fuse with the result we obtain by processing $E$ over the original data sources. Thus, we conclude that $V^{rec}$ = $V'$ = $V(S_1, .., S_i, .., S_n) \bigsqcup V(S_1, .., \triangle S_i, .., S_n)$. $\square$*

## 4.6 The Stability of Semantic Identifiers

Our semantic identifers have one important feature. They are stable under query rewriting. By that we mean that the semantic identifiers generated in the XML result are guaranteed to be the same for any two equivalent query expressions. This feature is very important for our view framework solution because it enables incremental maintenance plans to be optimized freely while ensuring the reproducibility of semantic ids. We state this stability property more formally in Theorem 4.6.2. To facilitate the discussion we first define the *minimum semantic id*. The *minimum semantic id* is a semantic id that reflects only the uniqueness of the identifier and does not necessarily reflect the order.

**Definition 4.6.1** *A* minimum semantic id *is a semantic id that is derived only from the* Lineage Context *of XAT columns.*

As we have discussed in Section 4.3.1, only three operators create or manipulate semantic ids, namely the *Tagger*, the *XML Union*, and the *Combine* operators. Among these operators only the *Tagger* and the *XML Union* operators utilize the *Lineage Context* to create and manipulate semantic ids. The *Combine* operator and the *Tagger* operator utilize the *Order Context* to set the order prefix of semantic ids.

The order prefix assigned by the *Combine* operator and the *Tagger* operator to semantic ids is derived from the *Order Context* as shown in the functions $assignOverRidOrd$ and $composeNodeIds$ in Figures 4.3 and 4.4 respectively. Such order prefix is not required for ensuring the uniqueness of XML nodes, as defined in Definition 4.3.1. This can easily be shown by dropping such order prefix for semantic ids and following the same logic used in proving Theorem 4.5.1 but now assuming that semantic ids do not have such order prefix. On the other hand, the order prefix generated by the *XML Union* operator is derived from the *Lineage Context* (in particular from the ColID part defined in Definition 4.2.2). It is thus significant for ensuing the uniqueness of semantic ids. Such order prefix distinguishes nodes originating from different input columns of the *XML Union* operator in addition to defining order among these nodes (see Section 4.2.2).

Based on the discussion above we state the following proposition.

**Proposition 4.6.1** *Matching XML nodes by node identifier, as done in Theorem 4.5.1, can be performed using the* minimum semantic id.

For proving the stability property of semantic ids, and for simplifying the proof, we will use Proposition 4.6.1 when matching two semantic ids.

**Theorem 4.6.2** *For any two equivalent query expressions $E_1$ and $E_2$, where $E_1$ and $E_2$ always produce the same result given the same input, the* minimum semantic id *for all nodes in the XML results $R_1$ and $R_2$ generater by $E_1$ and $E_2$ respectively are guaranteed to be identical.*

**Proof:** *We prove the correctness of Theorem 4.6.2 by showing that operators that generate or manipulate the* minimum semantic id *always rely on the same* Lineage Context *for any two equivalent query expressions $E_1$ and $E_2$ regardless of the order of operators in $E_1$ and $E_2$ in the query plans. In particular, these operators are the* XML Union *operator and the* Tagger *operator based on the discussion above. Note that for a base XML node that is exposed in the result, only the* XML Union *operator affects its* minimum semantic id*. In particular, it affects the order prefix part of the id. The body part of the id for such node is always fixed and is equal to the $FlexKey$ of that node in the source. For a constructed node in the*

*XML result the* XML Union *operator affects the order prefix in its* minimum semantic id *and the* Tagger *operator affects its boy part of its* minimum semantic id.

- *The* XML Union *operator* $\overset{x\,col}{\cup}_{col1,col2}(T)$ *uses the function* $assignColIdPrfx$ *(Figure 4.5) to assign an order prefix (derived from the* $ColID$ *defined in Definition 4.2.2) to each node in the resulting collections of nodes in column col. These nodes can be either source nodes or constructed nodes. For a query expression* $E_1$ *performing only one* XML Union *operation, an equivalent query expression* $E_2$ *must also have one* XML Union *operation given that the functionality of the* XML Union *operator is not replaceable by any other operator or set of operators in the XAT algebra (Section 2.2). The prefix order assigned to nodes in each of the two plans should be the same given that the same mechanism is used for generating such* $ColIDs$ *(same id labeling mechanism).*

  *For a query expression* $E_1$ *performing more than one* XML Union *operation, we require the equivalent query expression* $E_2$ *to have the same number of* XML Union *operations*[12]. *The* XML Union *operations in* $E_2$ *may appear in any locations and/or order possibly different than those in* $E_1$. *Given that*

[12]Two equivalent query expression may have different number of *XML Union* operators in only one case, if the result of an *XML Union* operation is unioned with itself. For example, an expression might define $\overset{x\,col'}{\cup}_{col1,col2}(T)$ and on top of it $\overset{x\,col''}{\cup}_{col',col'}(T)$. An equivalent query expression might instead have first two *XML Union* operators $\overset{x\,col}{\cup}_{col1,col2}(T)$ and $\overset{x\,col''}{\cup}_{col1,col2}(T)$ first and on top of them an *XML Union* operator $\overset{x\,col'''}{\cup}_{col',col'}(T)$. The first query expression (with 2 *XML Union* operators) is an optimization of the second one (with 3 *XML Union* operators). We require that such optimization to be either applied or not applied to the two equivalent expressions. Hence, we get the same number of *XML Union* operators in both expressions. As a result, the same $ColID$ keys and order prefix for smectic ids are generated.

*$E_2$ has to define the order among the sources of different column sources of the* XML Union *operations similar to that in $E_1$ to preserve the equivalency semantics among $E_1$ and $E_2$. $G$ iven that $ColIDs$ are generated using the same mechanism for both $E_1$ and $E_2$ in a depth first query tree traversal, as discussed in Section 4.2.2, the same $ColIDs$ are guaranteed to be assigned to $E_1$ and $E_2$. We conclude that we get the same order prefix and hence the same semantic ids (since the body of semantic ids is not affected here) for nodes processed using $E_1$ or $E_2$.*

- *The* Tagger *operator $T_p^{col}(T)$ uses the function $composeNodeIds$ (Figure 4.4) to generate semantic ids for newly constructed nodes. The semantic id body is derived from the* Lineage Context *of some column $col_j$, where $col_j$ is the input column to the* Tagger *pattern $p$. The* Lineage Context *of $col_j$ is created by the operator that creates $col_j$ and may be manipulated by other operators at later stage. Given a query expression $E_1$ with a* Tagger *operator $T_p^{col}(T)$, an equivalent query expression $E_2$ performing the same result construction creates and maintains the same* Lineage Context *for $col_j$ as in $E_1$. This holds unless $col_j$ is affected by an operation that changes its* Lineage Context *below the* Tagger *operator in one of the two expressions. Such operations include: the* Combine, *the* Tagger, *and the* Group By *operators, as shown in Table 4.1. If any of these operations affected $col_j$ in only one of the query expressions and not the other one, the two expressions are not equivalent. Hence, we conclude that the same* minimum semantic id*s are produced by the* Tagger *operator for any two equivalent query expressions.*

*From the discussion above we conclude that Theorem 4.6.2 holds.* $\square$

## 4.7 Discussion on our Proposed Semantic Identifier Solution

**A Fully Automated Method for Generating Semantic Identifiers.** Unlike other solutions for generating identifiers [PAGM96, LD00], we do not require any user interaction in defining how the semantic identifiers are generated or specifying what input is used to generate them. Our solution analyzes the view at the algebraic level and automatically determines how the semantic identifiers are to be generated through the rules and algorithms presented in Section 4.2.

**The Use of Values in Node Identifiers.** For some applications, like materialized view maintenance, it is typically not desirable to use only data values as input in generating identifiers. The reason is that this creates restrictions on updating such data. Our semantic ids are generated from base node identifiers, data values, or a mixture of them in a way that avoids this problem. As a matter of fact the way we use data values in our semantic ids is desirable. This is because the semantic id of a node that contains a data value is generated only whenever the node is actually bound to that data value (except for the *Order By* operator). In other words, when the relationship between the view node and the base value used in its id is a direct existence relationship. For example the node with id $1994^c$ in Figure 4.7(a) depends on the year value "1994". Deleting this value(s) from the input should and would be propagated to a deletion of this node from the view.

**Distributiveness of Algebra Operators.** Our solution ensures that different algebra operators are distributive with respect to the union operation, as we have shown in Section 4.5. This is a core property that supports incremental non-blocking query processing and enables efficient incremental view maintenance and stream processing. For example, in the view maintenance domain, the *Group By*, the *Distinct*, and the *Order By* are typically known to be non-distributive operators, meaning that we typically cannot propagate a new update through any of these operators without some knowledge of the data previously processed. In the stream processing domain such operators are typically seen as blocking operators, meaning that we may need to wait until we receive certain amount of input data before we process the operator. But given our approach of utilizing values in the semantic ids we are able to process XML fragments through these operators as if they are distributive and non-blocking. For example, when we propagate the inserted book, in Figure 4.1(a), we are able to propagate such update through the *Group-By* and the *Distinct* operators without any knowledge of the previously created groups or distinct values. This is so because we are able to reproduce the correct semantic identifier for the node "yGroup", as shown in Figure 4.6. Such reproduced ids based on the value allow us to correctly merge the result with the view extent, as shown in Figure 4.7. When considering order-sensitive views, other techniques that might support the distributive property of non-order aware views, like Skolem functions, fail to support the distributive property of views in an order-aware environment. Since our solution encodes the order in a way that removes the responsibility of maintaining order from most operators, the ordered bag semantics of the XQuery processing data model are migrated to non-ordered bag semantics. Hence the distributive property of the operators is preserved in the order-aware XML query processing environment, as discussed in Section 4.5.

Another example is the *Order By* operator. In our solution we do not perform any sorting operation or assign new sorted keys to the data when we process the *Order By* operator. Instead we simply uses the value(s) of the *Order By* attribute as the order prefix part of the semantic id. Hence, any incoming fragments is assigned an order prefix that is driven from its *Order By* attribute variables without the need to share ordering with previously processed data. The semantic id of the fragment now contains information that enables correct fusion with previous data (the id body) and information that reflects its correct order in the result (the id prefix part).

**What is Semantic About our Identifiers.** Our generated identifiers carry three types of semantics.

- *Fusion semantics.* The ability of the semantic identifiers to be reproducible is the main reason for using such identifier in object fusion. If, for example, new objects are to be added incrementally to the view, their semantic ids will ensure that they are fused correctly with existing view objects.
- *Lineage semantics.* The identifiers we generate carry lineage information that enables us to understand how nodes in the view are derived from the data source. Such lineage information might be useful for

applications like view updating and tracing derivations of view objects. We will next discuss in detail how we can interpret such lineage information.

- *Order semantics.* Our identifiers also carry order semantics. Order can be maintained on incremental construction of the XML result. This is a very important feature for XML views since they are typically order-aware. See Chapter 3 for more details on order in XML views.

**Understanding the Lineage Information in our Identifiers.** We discuss what information we can obtain from the semantic identifiers. First, we can derive the *node type*. An id with a suffix constant "$^c$" specifies a constructed node (eg., $b.b^c$). Any id without this suffix is an exposed base node (eg., $b.b$). Second, useful lineage information can be extracted from the ids. For an exposed base node, the id (after removing its order prefix if any) is directly referencing the base node it is copied from. For example, a view node with annotation $(a)b.b.l$ is derived directly from the base node with $FlexKey$ identifier $b.b.l$.

The id of a constructed node consists of two parts, the order prefix part and the body part. We may derive different conclusions depending on what the body part is composed of. (1) If the body part of a constructed node id is a single $FlexKey$ identifier, this implies that the node is derived and (bound) to a source node with that $FlexKey$. For example, the id $b.b..e.f^c$ assigned to the "entry" node shown in Figure 4.7(a) implies that this constructed "entry" node is bound to both a source node with id $b.b$ and a source node with id $e.f$. Hence, deleting one of these source nodes, will result in deleting this “entry” node from the view extent. (2) An id with a *data value* in its id body implies that the node is bound to this data value. For example, the id $\sim 1994^c$ for the “yGroup” node shown in Figure 4.7(a) implies that this constructed node is bound to the value “1994” in the source document. (3) An id with a constant value “*” in its body implies that the node is constructed over a collection of an arbitrary number of nodes. These nodes depends on the input source(s). Such constructed node is not affected by deleting any of its children. For example, let us consider the id $\sim *^c$ assigned for the “result” node shown in Figure 4.7(a) implies that this constructed node is a parent of an arbitrary number of nodes (“yGroup”). (4) The id body may be composed of one or more of the above. For example, a node id $b.b..*^c$ implies a constructed node that is derived from a source node with an id $b.b$ and a collection of arbitrary number of nodes. Deleting the source node with id $b.b$ should result in deleting the constructed node for the view, while deleting any nodes from the collection will not delete the constructed node.

**The Generality of our Solution.** Our solution defines special lineage and order specifications (that we call the *Context Schema*) using the query execution model (XAT table). There are two main XML query execution models, (1) a tuple-oriented model, like the one we use here, and also used in [IHW02] and (2) pattern tree model, like the one used in [JAKC$^+$02]. We need to understand how the pattern tree model maps to our model to be able to generalize our solution to it. The tree-oriented model uses pattern trees to match trees from the input documents. There is a direct mapping between the tree pattern and the XAT table model that we use, as each attribute in the XAT table maps to a variable binding in the tree, such mapping is also identified in [IHW02]. Hence, for the pattern tree execution models we simply need to define the *Context Schema* for the binding variables at the node level of the pattern trees. This corresponds to defining them on the column level in the XAT tables.

**Order Encoding via the *Table Order Schema* vs. via the *Context Schema*.** As the *Context Schema* defines how the order among nodes (or collections) in a column can be computed it does not reflect the order among tuples as a whole. Such order might not be important for some queries. But for other queries, particularly those involving join operations, the order among tuples is important since the order of the result of the join depends on the overall order of each input XAT table (which may involve multiple columns). Hence, we need to maintain the *Table Order Schema* for such queries. We need to maintain the *Table Order Schema* for only the operators below the join operation. On the other hand, although the order among cells in each column is also reflected by the *Table Order Schema*, it makes the order of each column dependent on columns possibly more columns than needed. Hence, order keys derived for a column may become bigger in size than needed. This wastes opportunities for merging order keys with node ids (generated by *Lineage Context*) into compact encoding. For example, if an XAT table contains a column for book elements ($\$b$), a column for book authors' last names ($\$l$) that is created by Navigate Unnest from $\$b$, and another column for book's prices ($\$p$) that is created by Navigate Unnest from $\$b$, the *Table Order Schema* for such XAT table will be ($\$l$, $\$p$). Hence, the order among tuples in that XAT table is determined by comparing a composition of the $FlexKey$s in columns $\$l$ and $\$p$ for different tuples lexicographically. Although such order reflects correctly the order among cells in each column, it uses unnecessarily long keys to represent that. For the book column ($\$b$) above it is sufficient to derive the order among its cells from the $FlexKey$s of the books. Note that if a constructed node is build on top of those books and if the order among those constructed nodes follow the source document order (or part of it) we use a compact id that directly reflects the order.

## 4.8 Experimental Evaluation for the Cost of Generating Semantic Identifiers

We have implemented our semantic identifier solution in Java within the Rainbow system framework [Zea03]. We have run the experiments on a Windows PC with 2.79 MHz Pentium 4 processor and 512MB of memory. We have used the XMark benchmark data [SWK$^+$02] in our experimental evaluation. We first use a query (Query 1 in Figure 4.8) that exploits our semantic identifier system intensively. In this query, most of the returned nodes are constructed nodes. Hence, a lot of node construction and new semantic id generation is required. The query also involves a mixture of order decisions

Figure 4.9(a) shows the cost of generating semantic ids relative to the total query execution time on different input XML document sizes. The

```
<result>                                                        Query 1
 {<customers>
  for $p in doc("site.xml")/people/person
  where $p/id/text() .<. 63750
  return
     <customer>{<location>$p/address/city/text()</location>} {$p/name}</customer>
  </customers>}
 {<open_bids>
   for $oa in doc("site.xml")/open_auctions/open_auction
   where $oa/id/text() .<. 30000
   return <bid> {$oa/reserve} {$oa/intial} </bid>
</open_bids>}
</result>
```

```
<result>                                                        Query 2
for $p in doc("site.xml")/people/person
where  $p/id/text() .<. 63750
return
  $p/name
</result>
```

Figure 4.8: Two example XQuery expressions.

figure shows that this cost is negligible compared to the total cost of query execution. Figure 4.9(b) shows the breakdown of the cost of our approach and compares it to the cost of execution (using a 500MB input document size). The cost of our solution is mainly composed of three components. (1) The cost of computing the *Order* and *Context Schema*s. This cost depends on 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 4.9(b) shows that the cost of generating new semantic ids on node constructions is higher than the other two cost component. The cost of generating the *Order* and *Context Schema*s is very small.

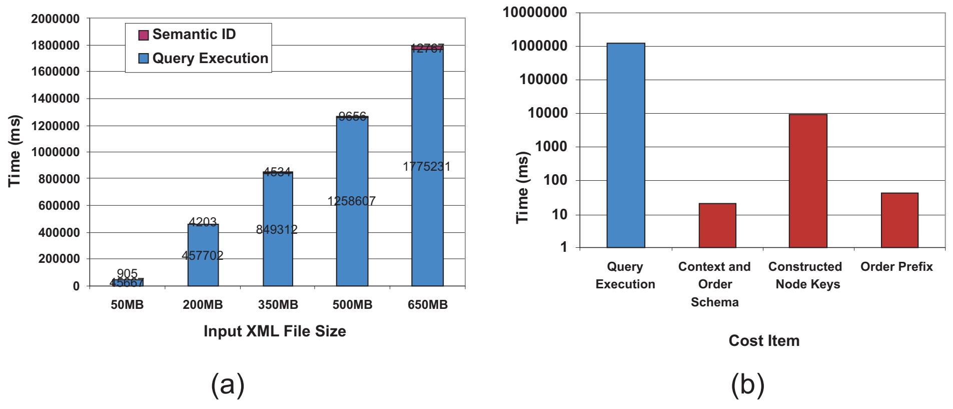


Figure 4.9: Result obtained using Query 1 in Figure 4.8. (a) The overhead of generating semantic identifiers to query execution time and (b) the break down of the cost of generating semantic identifiers.

Query 2 shown in Figure 4.8 is a simpler query that does not perform node construction (other than a new root). Figure 4.10(a) shows the cost of semantic ids relative to the total query execution time on different input XML document sizes. The cost in this case is negligible. Figure 4.10(b) shows that the only cost associated with semantic ids in this query is the cost of generating *Order* and *Context Schema*s. Such cost is smaller than that of the first query because the second query has a much smaller algebra tree. The cost of generating constructed nodes is close to zero since the query constructs only one node (the root node). There is no cost for assigning order prefix codes since the query returns the result in document order. Such order is reflected through the source node ids and is maintained by our solution at almost no cost (besides the negligible cost of the *Order* and *Context Schema*s).

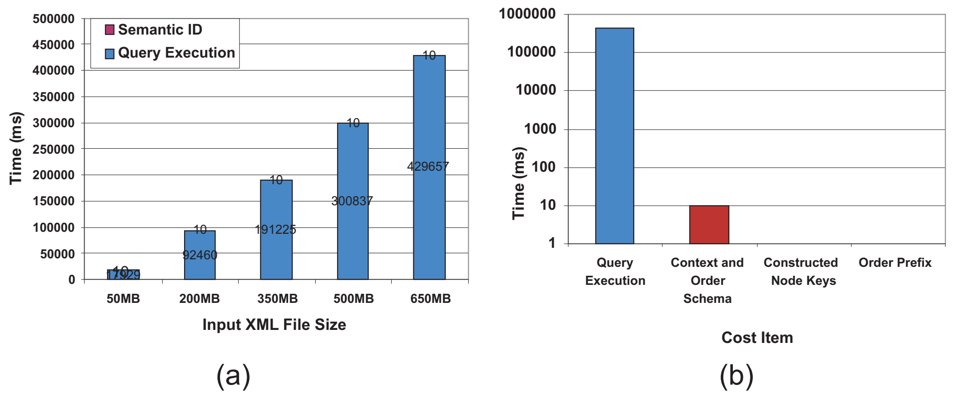


Figure 4.10: Result obtained using Query 2 in Figure 4.8. (a) The overhead of generating semantic identifiers to query execution time and (b) the break down of the cost of generating semantic identifiers.

# Chapter 5

# Validating Source XML Updates

## 5.1 Modeling Source Updates

We define a set of update primitives for inserting, deleting, and value changes of XML nodes. An update XQuery expression [TIHW01], like those shown in Figure 1.3, is submitted to the storage manager [DR03] where the specific nodes to be updated are determined. Then, we generate an update primitive for each updated base node. Each update primitive specifies the full path of the update annotated with source node ids. These update primitives are:

- **insert** *(n, k)*: Inserts a node with FlexKey $n$ into the node with a full path of FlexKeys $k$.
- **delete** *(n, k)*: Delete a node with FlexKey $n$ specified by the a full path

of FlexKeys $k$.

- **replace**$(n, k, new)$: Replace the value of the node with node identifiers FlexKey $n$ with a full path of FlexKeys $k$ with a new value $new$.

Figure 5.1 shows update primitives defined for our running example source updates (Figure 1.3). We model update primitives as XML trees annotated with "+" and "-" signs signifying inserted and deleted nodes, as shown in Figure 5.1. A value modification is modeled as a deletion of the node representing the old value and an insertion of a node representing the new value.

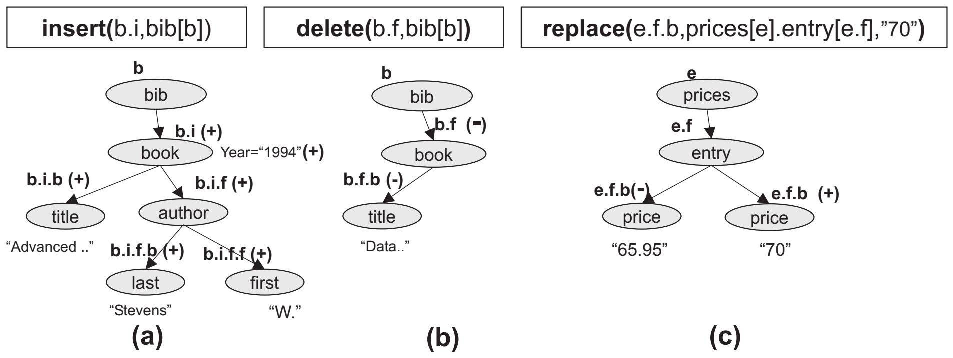


Figure 5.1: Three source update primitives corresponding to the three XQuery updates in Figure 1.3.

## 5.2 Validating Source Updates

Before propagating an update we first check if the update is relevant to the view to avoid unnecessary update propagation. We also determine any additional requirements (information) not provided by the update, yet essential for propagating the update. This is needed because an XML update, as given by an XQuery update statement may not contain enough information to allow propagating it. For example the delete update shown in Figure 1.3 that deletes a "book" element from "bib.xml" specifies the book to be deleted by its "title" sub-element, this might not be sufficient for propagating the update. We use the view query to determine necessary information for propagation as we will show next.

For that, we define a structure called the *Source Access Pattern Tree*, shortly $SAPT$. A $SAPT$ is a tree with paths of nodes representing XPath expressions that are part of the XQuery view expression indicating mandatory versus optional paths. The $SAPT$ is similar in spirit to the *Generalized Tree Pattern* [CJLP03] used in XQuery evaluation.

**Definition 5.2.1** ***Source Access Pattern Tree*** *($SAPT$) is an XML tree for a view query for each XML document that the view $V$ accesses. The $SAPT$ has:*

- *A path of nodes $path_i$ for each navigation path in the query. Each node in $SAPT$ is annotated with the variable(s) it binds to in the query. Such variable binding might be explicitly defined by the view or it might be an implicit binding assigned internally by query algebra operators. An edge connecting two nodes represents either a parent-child relationship or an ancestor-descendant relationship.*

- *Paths in $SAPT$ are classified into two types. (1)* ***Mandatory Paths Set (****$MandPS$****)****. This set includes each path $path_i$ in the view query that is required for evaluating the query. Such paths result from the $for$ and $where$ clauses of the query. (2)* ***Optional Paths Set (****$OptPS$****)****. This set includes each $path_i$ in $V$ that is optional to the query evaluation. Hence, any XML* *fragment that does not include such path may still be evaluated by the query. Paths in this set result from the* $\mathrm{return}$ *clause of the query. A path in the view query may appear in both sets if it is mandatory for the query evaluation and at the same time is returned in the* $\mathrm{return}$ *clause of the query.*

- *Nodes in* $SAPT$ *are annotated with corresponding predicates from the query.*

Note that a path defined in the *let* clause of the view query is classified in $SAPT$ based on the query clause it is used in. For example, if a *let* clause defines a path, and this path is used in the *return* clause of the query, this path is added to $OptPS$.

We do not define paths used in *Order By* clauses as we do not allow updates to source nodes accessed by the *Order By* clause.

Figure 5.2 shows three $SAPT$s constructed for the view query shown in Figure 1.2, one for each source document access. For each $SAPT$ in Figure 5.2, nodes corresponding to variable bindings are annotated with these variables. For example, the node "year" node bound to variable $\$y$ in $S1$ is annotated with a join predicte based on the view query. Nodes with no variable binding annotation represent nodes in the navigation path (e.g., "bib" and prices). Figure 5.2 also shows the join predicates specified on the "year" and the "title" nodes.

The $SAPT$ of $S_1$ (document "bib.xml" accessed at the outer $for$ clause of the query) has $MandPS$ = { $bib/book/@year$ } and has $OptPS$ = {$bib/book/@year$ }.

The $SAPT$ of $S_2$ (document "bib.xml" accessed at the inner $for$ clause of the query) has $MandPS$ = {$bib/book$, $bib/book/@year$, $bib/book/title$ }

and $OptPS = \{bib/book/title\}$. The $SAPT$ of source $S_3$ (document "prices.xml") has $MandPS = \{prices/entry, prices/entry/b{-}title\}$ and $OptPS = \{prices/entry/price\}$.

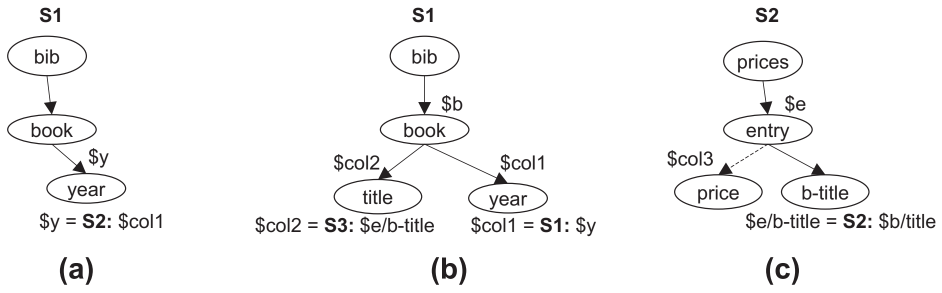


Figure 5.2: *Source Access Patter Tree* ($SAPT$) for the view in Figure 1.2.

By matching each update tree representing a source update to the $SAPT$s of the view query defined for the source document that the update affects, we can determine (i) if the update is relevant to the view or not and (ii) if the update carries sufficient information in order to be propagated.

### 5.2.1 Relevancy of Updates

After the update is modeled, as discussed above, it is then classified as being (1) relevant, (2) potentially relevant, or (3) irrelevant, as we will discuss below in Definition 5.2.2. We use the notation $path_1 \lhd path_2$ to denote that $path_1$ is a prefix path of $path_2$. For example, $bib/book \lhd bib/book/@year$.

**Definition 5.2.2** *An update $\triangle S$ to an XML source $S$ accessed by a view query $q$ through a* Source Access Pattern Tree $SAPT$ *(as defined above) is classified into one of the following:*

- ***Relevant Update**: if $\forall$ $path_p \in MandPS$, $\exists$ $path_s \in \triangle S$ where $path_s \lhd path_p$ and all non-join predicates in $path_p$ are satisfied.*

- ***Potentially Relevant Update**: If $\triangle S$ does not satisfy the relevant update condition above and at least one of the following conditions holds. (1) If $\exists$ $path_p \in MandPS$, $\exists$ $path_s \in \triangle S$ where $path_s \lhd path_p$ and all non-join predicates in $path_p$ are satisfied. (2) If $\exists$ $path_p \in OptPS$, $\exists$ $path_s \in \triangle S$ where $path_p \lhd path_s$. (3) If $\triangle S$ is a delete update and the update path $path_s$ $\lhd$ $path_p$ where $path_p \in MandPS$ or $path_p \in OptPS$.*

- ***Irrelevant Update:** otherwise.*

*Where $MandPS$ and $OptPS \in SAPT$.*

Relevant updates are ready for propagation. Irrelevant updates are not propagated. It might be possible to convert a potentially relevant update into a relevant update. Potentially relevant updates have to be filled with missing XML fragments called *propagation requirements*. After providing these propagation requirements we can decide if these updates are relevant or not to the query, as we will discuss next.

Based on Definition 5.2.2, the insert update in Figure 5.1(a) is *relevant* to both $S_1$ and $S_2$. The delete update in 5.1(b) is *potentially relevant* to both $S_1$ and $S_2$, since the path accessing the year attribute from the set $MandPS$s is not in the update. The modify update in 5.1(c) is *relevant* to $S_3$.

Note that the matching process also reveals that the "author" XML fragment in the first update is not relevant to the query. Hence, we prune it out from the update tree.

### 5.2.2 Sufficiency of Updates

An update might be provided as an incomplete piece of XML fragment. For example, the deletion update in Figure 1.3(b) specifies only the title of the book to be deleted. Due to the lack of some of the information (namely, in this case the year attribute) the update is classified as *potentially relevant* to $S_1$ based on its $SAPT$.

We derive the sufficiency of an update from Definition 5.2.2. In general, a *relevant* update has sufficient information to be propagated by the view query. A *potentially relevant* has insufficient information to be propagated.

With the help of the $SAPT$ we can identify what information is missing. That missing information can then be obtained from the source document. Alternatively, one could require the update definer to provide it so that it can be added to the update. We call such information *propagation requirements*. If the update does not become *relevant* after adding *propagation requirements* to it it is not propagated. For example, by providing for the delete update in Figure 5.1(b) the "year" attribute ("year"= 2000), the update becomes relevant. It can thus be propagated.

Note that we do not necessarily perform each of the three steps discussed above (modeling updates, relevancy check, and sufficiency check) separately. When the update is modeled we may annotate it with relevancy and sufficiency information based on $SAPT$. This forms a pattern tree that when matched against the source document, while obtaining the update specific information (ids), we also determine the relevancy and sufficiency of the update.

**Sufficiency of Delete Updates.** The updates sufficiency guidelines presented above generally applies to delete updates with one exceptional case. This exceptional case involves the deletion of a source node where the view query returns a collection of descendant nodes nested under this node without maintaining a *Lineage Context* for this collection. We call this case deleting collections without *Lineage Context* from the materialized XML view. We discuss it in more detail in Section 8.3.2.

## 5.3 Batching Source Updates

Relevant source update trees (also have sufficient information) are merged into one *batch update tree* using *Deep Union* defined earlier in Chapter 4. We may process a bulk of updates of different types at one time instead of processing them separately. This achieves better performance over individual processing of updates. The batched update tree also reflects the net effect of source updates. Hence, any updates that cancel each other out are eliminated. For example, an insertion of a "book" element that is followed by the deletion for the same "book" element will cancel out and will not be propagated. This saves the system the cost of propagating such updates.

Figure 5.3(a) shows the batch update tree for the source document "bib.xml". Note that the $SAPT$ of $S_1$ matches all the nodes in Figure 5.3(a). The $SAPT$ of $S_2$ matches all the nodes in Figure 5.3(a) except the title nodes. The "year" attribute is added to the deleted "book" node, as discussed above, and is assigned a "-" sign. The batch update tree for the updates on "prices.xml", shown in Figure 5.3(b), batches the modify update applied to

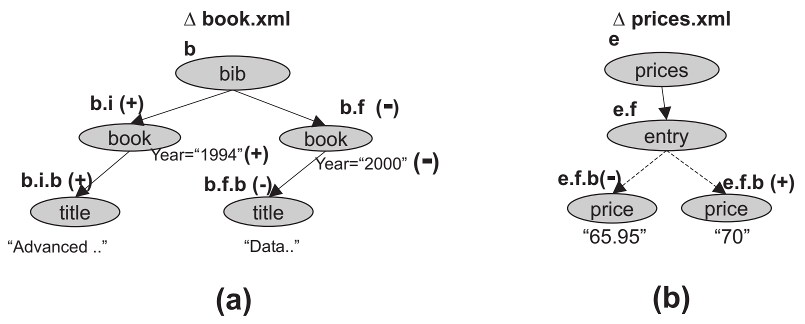


Figure 5.3: Batch update trees for (a) "bib.tex" and (2) "prices.xml".

the source XML document "prices.xml". Note that the batch update tree for "bib.xml" does not contain the "author" XML fragment of the first update since it is irrelevant to the view query and is pruned out, as we have discussed above.

# Chapter 6

# Counting Solution for Supporting XML Delete Updates

## 6.1 Maintaining XML views on Delete Updates

Delete updates are generally harder to handle than insert updates. Deleting a source node may not necessarily translate into a deletion of the node(s) derived from it in the view extent, because a node in the view extent may have multiple derivations from source nodes. Such derivation issue appears even in the simple relational $SPJ$ views [BLT86, GMS93]. Another issue is that deleting a source node might cause the deletion of an entire subtree from the view extent.

In the context of relational views, the counting algorithm [BLT86, GMS93] is a widely used solution for supporting deletions. It maintains for each tuple in the view a count representing the number of derivations of that tuple from source tuples. This count is incremented (or decremented) as a result of insert (or delete) update operations. When the count of a tuple becomes 0 it is simply deleted from the view extent.

An extension to the counting solution above for supporting semi-structured data has been proposed in [LD00]. This solution assigns two types of counts to nodes in a semi-structured data tree. It adopts a bottom-up count computation scheme for edge count in the trees where the count of an inner edge depends on the counts of its children edges. This solution comes with some drawbacks when handling deletes. In particular, when deleting an edge from the source data, all edges in the subtree of that edge appear to be needed as part of the update. Also deleting an edge from the view extent requires deletion of all edges reachable over this edge first. Hence, deleting big fragments from the view extent tends to be rather inefficient.

We propose an extension to the counting algorithm in [BLT86, GMS93] for supporting XML view maintenance. Unlike [LD00], we define one type of count for each node in the XML tree. The count of a node in our solution is independent of the count of nodes in its subtree. For propagating a delete update, our solution does not require the knowledge of the counts of the nodes in the entire subtree of the deleted node. This brings the advantage that it allows the deletion of an entire fragment from the XML view extent, by deleting only its root node.

## 6.2 Count Annotation for Source Documents and Source Updates

When processing the query to compute the initial materialized view, nodes in the source are treated as having a default count of 1. When processing updates to source document, nodes in batch update trees annotated with "+" or "-" signs are considered to have a count equal to 1 or -1 respectively. Other nodes in the update tree (nodes with no "+" or "-" sign) are considered to have count equal to 0. Note that a node in an update tree can have only a count of 0, 1, or -1. Batching of updates, where the same node might be used by more than one update should still maintain these count guidelines. A node in an update tree will be inserted or deleted by exactly one update. If such node is used later as part of the path of another update it is considered to have a count of 0 with regards to this new update. Hence, the count of the node will not be affected when adding 0 to it. For example when representing the "book" fragment insertion as shown in the batch update tree in Figure 5.3(a) all nodes in the "book" fragment are annotated with count of 1. Assume that we want to insert a new sub-element called "publisher" to that "book" element. In this case, the count of the "book" node in the batch update tree remains to be 1, since the count assigned to the "book" node in the new update is 0. The new "publisher" itself gets a count of 1.

Nodes in the initial view extent have to be annotated with appropriate counts to enable correct application of propagated updates. Hence, count have to be computed during query execution time and during view maintenance time. Different algebra operator have different treatment for the count annotation of processed XML nodes. We define general rules for how count is computed during normal query execution time. These rules also applies to computing count during view maintenance time with some exceptions. We next discuss these rules of computing count annotations first at normal query execution and then at view maintenance time.

## 6.3 Propagating Count Annotations in Normal Query Execution Time

Nodes in the materialized view extent need to be annotated with appropriate counts. Hence, count annotations for nodes in the materialized view extent need to be computed in normal query execution time. In Table 6.1 we define rules for computing count annotations of source nodes during normal query processing time. As shown in Table 6.1 some of the XAT operators manipulate count annotations, other operators process the nodes without manipulating their counts. We now discuss these rules in more detail.

- **Category I.** This category contains the *Navigate Collection* $\Phi^{col'}_{col,path}(T)$ and the *Navigate Unnest* $\phi^{col'}_{col,path}(T)$ operators. Destination nodes accessed by any of these two navigation operations are annotated with the default count of 1 as discussed above. For example, all the "book" nodes resulting from the *Navigate Unnest* operator ($\#5$) in Figure 4.2 should be assigned a count of 1.

| Cat. | Operator *op* | Affected Nodes | Count Assigned |
|---|---|---|---|
| I | $\Phi^{col'}_{col,path}(T)$<br>$\phi^{col'}_{col,path}(T)$ | Nodes in $col'$,<br>$col' \in t$ | for (each node $n_i \in col'$),<br>$n_i.count$ = 1 |
| II | $\delta_{col}(T)$<br>$\upsilon^{col}_{col1}(T)$ | Nodes in *col*,<br>$col \in t$ | for (each distinct node $dn_i \in col$),<br>$dn_i.count{=}sum(n_k.count)$<br>(where $n_k \in T.col$, $n_k$ = *dn* if *op* is $\delta$,<br>and $n_k \in t_l[col1]$, $n_k = dn_i$, $t_l$ is input tuple to $t$<br>if *op* is $\upsilon$) |
| III | $\gamma_{col[1..n]}(T, C_{col})$ | Nodes in $col[1..n]$<br>$col[1..n] \in t$ | for (each grouping node $gn_i \in col_j$),<br>$gn_i$= $sum(n_k.count)$<br>(where $n_k \in T.col$, $n_k$ is grouped by $gn_i$) |
| IV | $\overset{x\,col}{\cup}_{col1,col2}(T)$ | Collections in *col*,<br>$col \in t$ | for (each collection $c_i \in col$),<br>$c_i.count = x_i.lngCxt.count$<br>(where $x_i$ is node or collection $\in t_l[col1]$,<br>$t_l$ is input tuple to $t$) |
| V | $C_{col}(T)$ | Collection in *col*,<br>$col \in t$ | undefined count |
| VI | $T^{col}_p(T)$ | Nodes in *col*,<br>$col \in t$ | for (each node $n_i \in col$),<br>$n_i.count = x_i.lngCxt.count$<br>(where $x_i$ is node or collection $\in t_l[col_p]$,<br>$t_l$ is input tuple to $t$, $col_p \in p$) |
| VII | $\times(T_1, T_2)$<br>$\bowtie_c (T_1, T_2)$<br>$\sqsupset\!\bowtie_c(T_1, T_2)$ | $None$ | propagate current counts |
| VIII | $S^{col'}_{xmlDoc}$<br>$\rho_{col1,col}(T)$<br>$\sigma_c(T)$<br>$\tau_{col[1..n]}(T)$<br>$M(T_1, T_2)$ | $None$ | propagate current counts |

Table 6.1: Count computation rules for nodes in a resulting tuple $t$ of an operator *op* during *Query Execution Time*.

- **Category II.** The *Distinct* operator $\delta_{col}(T)$ sums the counts of values contributing to the respective distinct value. For example, the *Distinct* operator (#3) in Figure 4.2 should assign a count of 1 for the value $1994$ since it represents only 1 distinct value of 1994 based on the source XML document shown in Figure 1.1. The *Unique* operator $\upsilon^{col}_{col1}(T)$ also removes duplicate but using node ids. The same rule for computing the count applies also to the *Unique* operator.

- **Category III.** The *Group By* operator $\gamma_{col[1..n]}(T, C_{col})$ assigns to each grouping node or value the sum of counts of the grouped nodes. For example, the *Group By* operator (#15) in Figure 4.2 should assign a count of 1 for the value $1994$ since it represents only 1 node in the group that has a default count of 1.

- **Category IV.** The *XML Union* operator $\overset{x}{\cup}{}^{col}_{col1,col2}(T)$ retains the counts of nodes in the resulting collection. The count of the created collection itself is equal to the count of the node referenced by the *Lineage Context* of the left input source of the operator. Note that we do not store this count, we only derive it when needed, typically when a constructed node is build on top of the collection as we will discuss later for the *Tagger* operator. For example, the count of the collection in $\$col4$ the first tuple of the output XAT table of the *XML Union* operator (#13) in Figure 4.2 is equal to the count of the corresponding node in column $\$b$ (node $b.b$).

- **Category V.** The *Combine* operator $C_{col}(T)$ retains the count of nodes in the created collection and assigns no count to the collection itself.

- **Category VI.** The *Tagger* operator $T_p^{col}(T)$ assigns a count to the newly constructed node that is equal to the count of the node (or collection of nodes) that the new node is constructed over. If the new node is constructed over a collection, the count assigned to the new node is equal to the count of the collection, as discussed above. For example, for new nodes created by the *Tagger* operator (#14) in column $\$col5$ in Figure 4.2, the count of each new node is derived from the count of the corresponding node in column $\$b$. Another example is the node created using the *Tagger* operator (#20) in Figure 4.2. This node gets no count since it is constructed over a collection created using the *Combine* operator. For such nodes the count is not needed because we do not delete such node even if all of its children get deleted. This type of node is only deleted if its parent (or an ancestor) in the view extent (if any) is deleted. Hence, it would be deleted as part of the deletion of the subtrees of that parent (ancestor).

- **Category VII** and **Category VIII**. Each operators in these two categories process nodes without manipulating their counts. They simply propagate the current counts assigned to nodes as is.

## 6.4 Propagating Count Annotations During View Maintenance Time

Table 6.2 shows rules for computing the count annotation of nodes during view maintenance time. Some of the operators shown in Table 6.2 (namely

*Distinct*, *Group By*, *XML Union*, *XML Union*, and *Tagger* operators) follow the same logic as that used during normal query execution time for computing count annotation, as shown in Table 6.1. Some other operators in Table 6.2 (namely *Navigate Collection*, *Navigate Unnest*, and *Join* family of operators) have logic specific to view maintenance. We discuss those operators with view maintenance specific logic below.

| Cat. | Operator *op* | Affected Nodes | Count Assigned |
|---|---|---|---|
| I | $\Phi^{col'}_{col,path}(T)$<br>$\phi^{col'}_{col,path}(T)$ | Nodes in $col'$,<br>$col' \in t$ | for (each node $n_i \in col'$),<br>    if (node $n_i$ is update node),<br>        $n_i.count$ = update count annotation<br>    else if (node $n_i$ is base node),<br>        $n_i.count = n_i.entryPointNode.count$<br>if (*op* is $\phi$),<br>    for (each node $n_j \in col_j$ where $col_j \in ECC$),<br>        $n_j.count = n_i.count$<br>(where $n_i$ is node reachable from $n_j$ through $path$) |
| II | $\delta_{col}(T)$<br>$\upsilon^{col}_{col1}(T)$ | Nodes in *col*,<br>$col \in t$ | Same as in Table 6.1 |
| III | $\gamma_{col[1..n]}(T, C_{col})$ | Nodes in $col[1..n]$, | Same as in Table 6.1 |
| IV | $\overset{x\,col}{\cup}_{col1,col2}(T)$ | Collections in *col*,<br>$col \in t$ | Same as in Table 6.1 |
| V | $C_{col}(T)$ | Collection in *col*,<br>$col \in t$ | Same as in Table 6.1 |
| VI | $T^{col}_{p}(T)$ | Nodes in *col*,<br>$col \in t$ | Same as in Table 6.1 |
| VII | $\times(T_1, T_2)$<br>$\bowtie_c (T_1, T_2)$<br>$\sqsupset\!\bowtie_c(T_1, T_2)$ | Nodes $\in t$ | for (each node $n_i \in col$, $col \in T_1$),<br>    $n_i.count = n_i.count \times t_r.countFactor$<br>for (each node $n_j \in col$, $col \in T_2$),<br>    $n_j.count = n_j.count \times t_l.countFactor$<br>(where $t_r \in T_2$ is input to $t$, $t_l \in T_1$ is input to $t$) |
| VIII | $S^{col'}_{xmlDoc}$<br>$\rho_{col1,col}(T)$<br>$\sigma_c(T)$<br>$\tau_{col[1..n]}(T)$<br>$M(T_1, T_2)$ | $None$ | Same as in Table 6.1 |

Table 6.2: Count computation rules for nodes in a resulting tuple $t$ of operator *op* during *View Maintenance Time*.

- **Category I**. That contains the *Navigate Collection* $\Phi_{col,path}^{col'}(T)$ and the *Navigate Unnest* $\phi_{col,path}^{col'}(T)$ operators process count differently during view maintenance time. We define two rules. (1) When navigating from a base node (typically obtained through a join operation between an old source and an update). In this case each destination node, resulting from the navigation from this base node, gets a count that is equal to the *count factor* of its entry point base node. This *count factor* is equal to 1 if the count of the entry point node is positive, -1 if the count of the entry point node is negative, or 0 if the count of the entry point node is 0. For example, if during update propagation time the count of a source "book" element becomes 0, a navigation to a "title" sub-element assigns a count equal of 0 to that "title" sub-element. (2) If a node from the update is accessed through a *Navigate Unnest* the count of nodes in the path to the updated node is set equal to the count of the destination node[1]. This helps isolating the effect of sharing prefix paths among source updates in batch update trees. Hence, it ensures that the result of propagating a batch of updates is equal to the total result of propagating each update separately. We address this issue later in Theorem 7.2.2.

- **Category VII**. For any of the *Join* operators, the count of nodes in the resulting tuple is determined as follows. First, each input tuple to the *join* is assigned a *tag* that determines if the tuple represents (1) an *old* tuple (obtained when joining the update from one source of the

[1]We assume that *Navigate Unnest* Operations have been pushed down below *Group By* operations in XAT query plans.

join with the old data in the other source), (2) an *insert* tuple, (3) a *delete* tuple, or (4) a *modify* tuple. We define how an update tuple is classified into insert, delete, or modify tuple in Definition 6.5.1. Based on this classification each tuple in the input of the *join* is assigned a *count factor* that can be (i) +1, if the tuple is tagged *old* or *insert*, (ii) -1 if the tuple is tagged *delete*, and (iii) 0 if the tuple is tagged *modify*. Second, when joining a tuple $t_l$ from one input XAT table with a tuple $t_r$ from another XAT table, the count annotation of each node in $t_l$ is multiplied by the *count factor* of $t_r$ and vice versa. For example, if a join operation joins a tuple $t_l$ resulting from a source update that has a *count factor* of -1 with a tuple $t_r$ from an old source that has a *count factor* of 1, the count of all nodes in $t_r$ is multiplied by -1. We represent the count factor of a tuple $t$ as $t.countFactor$. For a tuple $t$ resulting from a join between $t_l$ and $t_r$, $t.countFactor$ = $t_l.countFactor \times t_2.countFactor$.

- Other categories compute the count similar to way the count is computed in normal query execution time.

## 6.5 Classifying Intermediate XAT Updates Based on Count Annotation

Based on the count annotation assigned to XML nodes in an intermediate XAT table we can classify an intermediate XAT tuple processed during view maintenance time into: (i) an insert update, (ii) a delete update, or (iii) a modify update.

**Definition 6.5.1** *Given an XAT tuple t representing a propagated update to an intermediate XAT table with $\{n_1, n_2, .., n_m\}$ being the list of nodes called the $ECC\ Nodes\ List$ (ECCNL) where $n_j$ is stored in $t[j]$, $col_j \in ECC$ (as defined in Definition 4.2.3, t is classified into one of the following:*

- ***Insert update**: if $\exists\ n_j \in ECCNL$ where $n_j.count > 1$.*
- ***Delete update**: if $\exists\ n_j \in ECCNL$ where $n_j.count < 1$.*
- ***Modify update**: if $\forall\ n_j \in ECCNL$, $n_j.count = 0$.*

*Based on this classification, a count factor for t (denoted as $t.countFactor$) can be derived, where $t.countFactor$ = 1 if t is an insert update, $t.countFactor$ = -1 if t is a delete update, or $t.countFactor$ = 0 if t is a modify update.*

Definition 6.5.1 assumes that for any XAT tuple $t$ all nodes in columns that are in $ECC$ can not have nodes with positive and negative counts at the same time. As matter of fact given the algebra operators shown in Table 4.1, the count annotation for all nodes in columns in $ECC$ is either positive count, negative count, or 0, as we will show next in Lemma 6.5.1. A propagated insert or delete update may have 0 count for some of its nodes in $ECCNL$ if the operators $TDiff$ and $TIntersect$, presented in Section 7.3, are used in the the incremental plans as specified in the propagation rules in Theorems 7.3.3 and 7.4.1.

**Lemma 6.5.1** *For an XAT tuple t representing a propagated update where $\{n_1, n_2, .., n_m\}$ is a list of nodes called the $ECC\ Nodes\ List$ (ECCNL) where $n_j$ is stored* *in* $t[j]$, $col_j \in ECC$, *given the algebra operators shown in Table 4.1, we find that one of the following equations is guaranteed to always hold:*

- $\forall\, n_j \in ECCNL$, $n_j.count > 0$;
- $\forall\, n_j \in ECCNL$, $n_j.count < 0$;
- $\forall\, n_j \in ECCNL$, $n_j.count = 0$;

*where* $1 \leq i \leq m$.

**Proof:** *By examining the rules for computing the* Context Schema *in Table 4.1 and the rules for computing the count in Table 6.2, we find that the following operators either affect* $ECCNL$ *(by adding new columns to* $ECC$ *hence adding new nodes to* $ECCNL$*) or affect the counts of nodes in* $ECCNL$*.*

- *The* Source *operator always generates a tuple with one single column and one single node. This node is the only node in* $ECCNL$ *of* $t$*. Hence, Lemma 6.5.1 holds.*
- *The* Navigate Unnest *operator adds the destination node resulting from the the navigation process to* $ECCNL$*. Nodes that already exist in the list are assigned a count that is equal to the* count factor *of the destination node, based on the count rules above, in Table 6.2. Hence, Lemma 6.5.1 holds.*
- *The* Tagger *operator creates a new column in the output XAT table and adds this column to the* $ECC$ *of the XAT table. Hence, the newly constructed node is added* $ECCNL$*. This node is assigned a count that is equal to the count of the node referenced by the* Lineage Context *of the input column*

*of the new node. Hence, the count of the new node will not violate Lemma 6.5.1.*

- *For the* Group By *operator, the $ECCNL$ of the XAT table generated by a* Group By *operator contains only the grouping node(s). Based on our count treatment above, the grouping value in a tuple is assigned a count that is equal to the sum of counts of the grouped nodes. Hence, Lemma 6.5.1 holds.*
- *The* Distinct *operator will always generate one column in the output XAT table. The $ECC$ of that XAT table will contain only that column. Hence, Lemma 6.5.1 holds.*
- *The* Theta Join *operator and the* Cartesian Product *operator multiplies the count of each node in each input tuple from one XAT table by the* count factor *of the tuple it joins with from the other XAT table, as discussed above. Hence, the resulting joined tuple satisfies Lemma 6.5.1.*

*Other operators have no effect on nodes in $ECCNL$ or the counts of these nodes.* □

## 6.6 Count-aware Deep Union Operator

We extend the definition of the *Deep Union* in Definition 4.4.1 to incorporate counting as follows:

**Definition 6.6.1** *The* ***Deep Union*** *($\bigsqcup$) takes two sets of XML trees $T_1$ and $T_2$ and recursively unions them. An XML tree $t = r : ch$ is represented by its root node $r$ and the children list of that root node $ch$. The* Deep Union *first unions* *the root nodes of the XML trees in $T_1$ and $T_2$ by node identifier and recursively performs deep union on the respective lists of children nodes. The resulting set of XML trees $T$ includes all XML trees in the two lists $T_1$ and $T_2$ with only one occurrence of any matching root nodes. For any two matching nodes $r_i$ and $r_j$ from $T_1$ and $T_2$ respectively, the node that represents them in $T$ is assigned a count that is equal to the summation of counts of $r_i$ and $r_j$.*

$T_1 \bigsqcup T_2 = \{\, r : ch_i \bigsqcup ch_j \mid r.id = r_i.id = r_j.id,\ r.count = (r_1.count + r_2.count),\ r_i : ch_i \in T_1,\ r_j : ch_j \in T_2\} \cup \{\, r_i : ch_i \mid r_i : ch_i \in T_1,\ \nexists\, r_i : ch_i \in T_2$ *where* $r_i.id{=}r_j.id\} \cup \{\, r_j : ch_j \mid r_j : ch_j \in T_2,\ \nexists\, r_i : ch_i \in T_1$ *where* $r_j.id{=}r_i.id\}$

*If $r_1.count$ is undefined or $r_2.count$ is undefined then $r.count$ is undefined.*

The case of undefined count occurs when using a $Combine$ operator, as shown in Table 6.2. Note that if the count of the merged node becomes 0, the node is deleted from the XML tree with its entire subtree.

## 6.7 Counting Example

Given the source XML documents shown in Figure 1.1 and the XQuery view shown in Figure 1.2. Now assume that an update deletes the first "entry" fragment from the source document "prices.xml". Figure 6.1(a) shows the source "prices.xml" before the update where each source node is annotated with a default count of 1. Figure 6.1(b) shows the update tree representing the update after being annotated with appropriate information and counts. Applying the update to the source document should result in deleting the entire fragment with root node id $e.b$, as shown in Figure 6.1(c).

Now consider that we want to propagate the update to the view extent.

The initial view extent is shown in Figure 6.1(d). Now we propagate the source update described above to the view extent. This should result in the propagated update tree shown in Figure 6.1(e). We postpone the discussion on the details of the propagation process to Chapter 7.

Applying the propagated update shown in Figure 6.1(e) to the initial view extent in Figure 6.1(d), using the *Deep Union*, results in deleting the entire fragment with root node "yGroup" with id $2000^c$ from the view extent. Figure 6.1(f) shows the refreshed view. Applying the update is done by first merging the root node of the initial view extent with the root node of the update, since they both match by their semantic ids. Then the "yGroup" node with id $2000^c$ from the initial view extent is matched with "yGroup" node with identical id from the propagated update tree. Then the counts of both nodes are summed. This results in a count of 0. As a result, we can directly delete this "yGroup" node with id $2000^c$ from the view extent with its entire subtree.

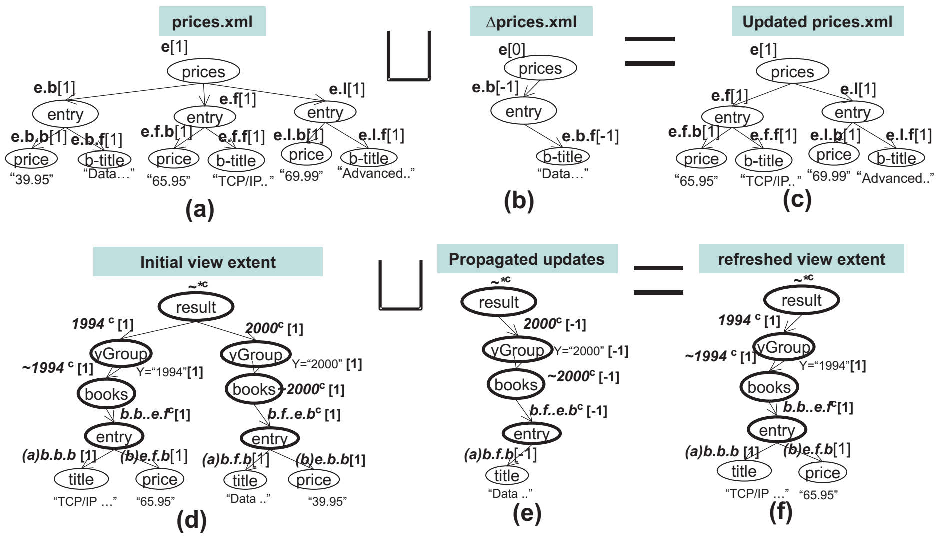


Figure 6.1: Delete update example showing how count is handled.

# Chapter 7

# An Algebraic Approach for Propagating XML Updates of Different types

We take an algebraic approach in defining how update propagation is done. The algebraic approach has many advantages over the procedural approach used in some view maintenance solutions [GMS93, GJM97, AP98]. These advantages, also highlighted in [GL95, GK98], include:

- It is easy to extend the algebraic view maintenance solution by adding new rules for other operators.
- it is easy to compose multiple propagation rules for multiple operators to support expressions including these operators, unlike the procedural approach where this is not always possible.

- It is easy to show the correctness of algebraic view maintenance algorithms.
- it expresses changes to the view extent in the form of expressions in the same language used to define the view. Hence, these expressions can be optimized by any query optimizer.

First we define how updates are processed in a distributive manner using single algebra operators. Then we define the distributive property of views composed of multiple algebraic operators.

## 7.1 Views with Single Operator

As we have discussed earlier in Chapter 4, we define the distributivity of algebra operators with respect to the operator $TUnion$. So far we have considered updates that represent only insert updates. Now we consider updates of any type. We first extend the $TUnion$ defined in Definition 7.1.1 to incorporate counting treatment. This allows us to handle different types of updates.

**Definition 7.1.1** ***TUnion*** *($\overset{t}{\sqcup}$) takes two XAT tables $T_1$ and $T_2$ with the same schema and unions them. The result is an XAT table that contains a single occurrence of each pair of matching tuples (as specified in Definition 4.2.4) and all non-matching tuples from $T_1$ and $T_2$. Each pair of matching tuples ($t_1$ and $t_2$) is merged into a single tuple ($t_{union}$) that contains a single occurrence of the matching nodes from $t_1[col_i]$ and $t_2[col_i]$, $col_i \in ECC$, and the union of the contents of $t_1[col_i]$ and $t_2[col_i]$, $col_i \notin ECC$.*

$T_1 \overset{t}{\sqcup} T_2 = \{t_1 | t_1 \in T_1$ *and* $\nexists t_2 \in T_2$ *where* $t_1 \asymp t_2\} \cup \{t_2 | t_2 \in T_2$ *and* $\nexists t_1 \in T_1$ *where* $t_2 \asymp t_1\} \cup \{t_{union} | \forall t_1 \in T_1, \exists t_2 \in T_2$ *where* $t_1 \asymp t_2\}$

$t_{union}$ *is a tuple that conforms to the schema of* $t_1$ *and has the same* $ECC$ *as* $t_1$. $t_{union}$ *is defined as follows:*

$\forall col_i \in t_{union}$, $t_{union}[col_i] = t_1[col_i]$, *if* $col_i \in ECC$ *and* $t_{union}[col_i] = (t_1[col_i] \cup t_2[col_i])$ *if* $col_i \notin ECC$. $t_1[col_i] \cup t_2[col_i]$ *contains the union of contents of* $t_1[col_i]$ *and* $t_2[col_i]$ *where nodes are matched by node id.*

***Node Count:*** *The count annotation of any node* $n \in t_{union}[col_i]$ *is equal to the sum of counts of the two matching nodes* $n_1$ *and* $n_2$ *from* $t_1[col_i]$ *and* $t_2[col_i]$ *respectively where* $n.id = n_1.id = n_2.id$, *or equal to the count of* $n_1$ *or* $n_2$ *which ever of them exists and matches* $n$ *by id.*

$$n.count = \begin{cases} n_1.count + n_2.count & \exists\, n_1 \in t_1, \exists\, n_2 \in t_2 \text{ where} \\ & n.id = n_1.id = n_2.id \\ n_1.count & \exists\, n_1 \in t_1, n.id = n_1.id \text{ and} \\ & \nexists\, n_2 \in t_2 \text{ where } n.id = n_2.id \\ n_2.count & \exists\, n_2 \in t_2, n.id = n_2.id \text{ and} \\ & \nexists\, n_1 \in t_1 \text{ where } n.id = n_1.id \end{cases}$$

*For* $\forall$ *node* $n \in t_{union}[col_i]$ *where* $col_i \notin ECC$ *if* $n.count = 0$, $n$ *is deleted.*

***Tuple Count Factor:*** $t_{union}.countFactor$ *is computed based on the new counts of nodes in* $t_{union}$ *following the guidelines discussed in Section 6.5.*

**Empty XAT Tuples.** The $\overset{t}{\sqcup}$ may generate what we call *Empty XAT Tuples*.

**Definition 7.1.2** *An XAT tuple t generated during update propagation time is called an* ***Empty XAT Tuple*** *if* $\forall\ col_i \in ECC$, $t[j].node.count$ *= 0 and* $\forall\ col_i$ *of t* $\notin ECC$, $t[j].collection$ *is empty.*

The $\overset{t}{\sqcup}$ operator may generate such a tuple in one particular case; when merging an XAT tuple $t_1$ representing a delete update with a tuple $t_2$ that it deletes from an XAT table. The $\overset{t}{\sqcup}$ deletes any *Empty XAT Tuple* generated as a result of merging the update tuple with a tuple for the XAT table, hence removing the tuple from the XAT table, as we state in Proposition 7.1.1.

**Proposition 7.1.1** *An* Empty XAT Tuple *(Definition 7.1.2) generated by the* $\overset{t}{\sqcup}$ *operator during view maintenance time is* deleted.

We now establish the distributiveness of operators over the $TUnion$ operator given any type of updates.

**Theorem 7.1.2** *For an update* $\triangle T$*, of any of the types defined in Definition 6.5.1, to the input XAT table T of an operator op, from the operators defined in Table 6.1, the equation* $op(T \overset{t}{\sqcup} \triangle T) = op(T) \overset{t}{\sqcup} op\,(\triangle T)$ *holds. For a binary operator the equation* $(T_1 \overset{t}{\sqcup} \triangle T_1)\, op\, T_2 = (T_1\, op\, T_2) \overset{t}{\sqcup} (\triangle T_1\, op\, T_2)$ *holds, if no grouping operation exists below* $T_1$ *or* $T_2$ *in the query plan.*

**Proof:** *In Theorem 4.5.1 we have proven the distributiveness property of the XAT algebra operators on insert updates only and with no consideration to node counts. We now build on the proof of Theorem 4.5.1 to address the correctness of the distributive property of operators given updates of any type. We use the notations shown in Table 4.3. In particular, we use T and Q to denote the input and the output XAT tables of the operator respectively,* $xin$ *to denote the tuple representing* *the update $\triangle T$ to $T$, and $xout_o$ to denote a tuple in the propagated updates to $Q$. In addition we use $tin_i$ to denote a tuple in $T$ that matches with $xin$ (if any) and $tout_e$ to denote a tuple in $Q$ that matches with $xout_o$ (if any).*

***Insert Updates.*** *An insert update may have one of two cases (I) $\exists\ tin_i \in T$ where $xin \asymp tin_i$ and (II) $\not\exists\ tin_i \in T$ where $xin \asymp tin_i$. We have shown in the proof of Theorem 4.5.1 the distributiveness of operators under these two cases when counting is not considered. The reasoning used in Theorem 4.5.1, for showing the correctness of update propagation, directly applies to insert updates where nodes are annotated with positive counts representing the insertion of nodes. One difference is that we perform summation of counts of any two matched nodes, as defined in 7.1.1. We now discuss the count treatment in each of the two cases above.*

- *Case (I) can only occur for an insert update when $T$ is affected by a grouping operation (value-based grouping, id-based nesting, or distinct). In this case the insert update will merge with one of the groups in $T$. Applying $xin$ to $T$ will result in merging $xin$ with a tuple $tin_i \in T$ creating $tin_i'$. $tin_i'$ contains a single occurrence of any two matching nodes in grouping columns from $xin$ and $tin_i$, where the counts of these matching nodes are summed. Each node in a grouped column $col_j$ in $xin$ representing a new node insertion (has count of 1) will not match with any node in column $col_j$ of $tin_i$. Hence, each grouped column $col_j$ in $tin_i'$ will contain the union of old nodes in column $col_j$ from $tin_i$ and the new node inserts in column $col_j$ from $xin$. $op$ can be any of the operator shown in Table 6.2 except the operators* Source *(only a leaf node),* Navigate Unnest *(pushed down below grouping operators in* *our XAT query plans, as we mentioned in Section 6.4),* Unique *(is not allowed on top of grouping columns since it operates on collections), and the join family of operators (we address this class of views separately in Theorem 7.3.3). We now discuss how the count is treated in the case of recomputation and in the case of update propagation. We have two classes of nodes.*

- *Nodes in grouping columns. In this case, if op is recomputed over merged input tuple $tin'_i$ = ($tin_i \overset{t}{\sqcup} xin$), the count assigned to nodes in the grouping column will not change, based on the counting rules in Table 6.2. Now consider that $xin$ is processed separately. This should result in a propagated update tuple $xout$ that merges with $tout_{iu}$ (a tuple resulting from processing $tin_i$, where $tin_i \asymp xin$), as we have showed in Theorem 4.5.1. The count of each two matching nodes in grouping columns in $xout$ and $tout_{iu}$ is summed. This is clearly gives the same count we obtain when we recompute op over $tin'_i$. Note that this result applies also to new nodes constructed over the grouping columns (since each new node gets the same count as the count of the node it is constructed over), as defined in Table 6.2. New nodes constructed over collection columns also get count that depends on the count of the grouping column, as defined in Table 6.2. Hence, this conclusion also applies to them.*
- *Nodes in grouped columns. As mentioned above a node in a grouped column of $xin$ that represents an inserted node (with count of 1) will never match a node in $xin_i$. Hence, the count of such a node will not be summed with any other counts. As a result, if op is recomputed*

*over merged input tuple* $tin'_i$ = $(tin_i \overset{t}{\sqcup} xin)$, *the count assigned to such nodes will generally stay the same. One exception if op is the* Unique *operator where nodes with the same ids in each collection are merged into one node and their counts are summed. Now consider that* $xin$ *is processed separately. This should result in a propagated update tuple* $xout$ *that merges with* $tout_{iu}$ *as we discussed above. As shown in Theorem 4.5.1, nodes in the grouped column of* $xin$ *will correctly be unioned to grouped nodes in the corresponding grouped columns of* $tout_{iu}$. *Such merging will result in no merging between nodes from* $xin$ *and nodes from* $tout_{iu}$ *in all cases, except when op is the* Unique *operator. As a result all the nodes retain their initial count of. If op is the* Unique *operator, each node from* $xin$ *merges with matching nodes from* $tout_{iu}$ *and their counts are summed. This is equivalent to the result we obtain through recomputation.*

- *In case (II) when* $xin$ *does not match with any tuple in* $T$ *we consider two scenarios.*
  - *op is any operator except grouping operators (value-based grouping, id-based nesting, or distinct). In this case, the tuple(s)* $xout$ *resulting from the propagation of* $xin$ *is guaranteed not to match with any tuple in Q, as shown in Theorem 4.5.1. Hence, no nodes are merged and no counts are summed.*
  - *op is any of the grouping operators. In this case, each tuple* $xout$ *might match with a tuple in Q. The new count resulting from merging any two matching nodes is clearly the same if such merging is done as a*

*result of operator recomputation or as a result of update propagation.*

*Hence, the correctness of insert update propagation with count annotation is shown.*

***Delete Updates.*** *Unlike an insert update where the tuple representing the update may or may not match with an existing tuple in the XAT table it updates, a delete update will always match with an existing tuple in the XAT table. We distinguish between two possible cases:*

- $xin$ deletes a tuple from $T$. *In this case each node in* $xin$ *will have a count that is equal to the count of the matching node it deletes in* $tin_i$ *multiplied by -1. This is guaranteed because if the node to be deleted does not represent a group, it will always have a count of 1 and at the same time the count annotation of a node deletion in the update tree is always -1, as we discussed in Section 6.2. If the node to be deleted represents a group, its count can be any positive number. Each delete operation to a node contributing to that grouped node will decrement the count associated with the grouping node.*

  *Applying* $xin$ *to* $T$ *first and recomputing the op should result in deleting a tuple* $tin_i$ *that matches* $xin$ *from* $T$*. Hence, a tuple* $tout_k$[1] *in the output XAT table Q of op (representing the result of processing* $tin_i$*) will not exist. When processing* $xin$ *separately by op, a tuple* $xout$ *is generated.* $xout$ *will match a tuple* $tout_k$ *in Q, as we have shown in the proof of Theorem 4.5.1. Merging* $xout$ *with* $tout_k$ *results in a tuple* $tout'_k$ *that is an* Empty XAT Tuple*, as defined in Definition 7.1.2. Hence,* $tout'_k$ *is deleted from Q. The* Empty XAT Tuple $tout'_k$ *is guaranteed to be generated. In order for* $xin$

[1] A set of tuples, rather than one tuple, may be generated by some operators. For simplifying the discussion, we assume that one tuple is generated.

*to delete a tuple $tin_i$ from $T$, $xin$ should have exactly the same nodes as in $tin_i$ with counts equal to that of nodes in $tin_i$ multiplied by -1. In this case $xin \overset{t}{\sqcup} tin_i$ will result in an* Empty XAT Tuple *that is deleted from $T$. Now when we apply op to $xin$ the propagated update tuples $xout$ is guaranteed to have the same nodes as the tuple $tout_k$ resulting from processing $tin_i$, as we have shown in Theorem 4.5.1. This includes old nodes that are already in $tin_i$ and new nodes in $tout_k$ created by navigation and node construction operations. For $xout$ those new nodes will get a count of -1, based on rules in Table 6.2. Merging $xout$ with $tout_k$ generates the* Empty XAT Tuple *$tout'_k$ that is deleted from $Q$. Hence, the theorem holds.*

- $xin$ does not delete a tuple from $T$. *This is only possible if tuples in $T$ represent groups ($T$ was processed by a* Group By *or a* Distinct *operation) and $tin_i$, that $xin$ matches with, represents more grouped tuples than what $xin$ represents. The sum of counts of any two corresponding nodes in the grouping columns of $tin_i$ and $xin$ is going to be bigger than 0, because the grouping node in $tin_i$ represents more grouped nodes in the current result (positive count) than what the grouping node in $xin$ represent in terms of grouped nodes to be deleted (negative count). In this case, applying $xin$ first to $T$ and then recomputing the operator should result in a tuple $tout'_k$ in $Q$ that is generated from processing the updated source tuple $tin'_i$. When processing $xin$ separately through the operator, a tuple $xout$ is generated. $xout$ will match a tuple $tout_k$ in $Q$. Merging $xout$ and $tout_k$ results in a tuple $tout'_k$ that is equivalent to that obtained through recomputation, as shown in the proof of Theorem 4.5.1. The count of nodes of matching nodes* *will correctly be handled following the sam logic used in previous case, with one difference is that the count of grouping nodes will not become 0.*

*Hence, Theorem 7.1.2 holds for delete updates.*

***Modify Updates.*** *A modify update is represented as a tuple with all nodes in columns $\in ECC$ are with count equal to 0, as defined in Definition 6.5.1. The propagation of such update follows the same logic as that of a delete update that matches with a tuple in $T$, as discussed above. The only difference is that the modify update will have 0 count for all nodes in columns in $ECC$ instead of positive numbers. Hence, merging a node with count equal to 0 with an old node will not cause the sum to become 0. As a result, the propagation of a modify update will not create a* Empty XAT Tuple *and hence will not delete any tuples from Q. Note that nodes in columns that are not $\in ECC$ of a tuple representing a modify update might be representing insertion or deletion. A modify update might propagates an insert or delete update if it was processed by a* Navigate Unnest *operator that navigates to nodes representing insertions or deletions . This is because, based on counting rules in Table 6.2, the count of nodes in $ECC$ columns of the propagated update tuple will be set to 1, if the node reachable by the navigation path has count of 1, or will be set to -1, if the node reachable by the navigation path has count of -1. For this point the appropriate logic for propagating the update, as discussed above, based on its type (insert or delete) applies.* $\square$

**Maintaining a Binary Operator on Updates to its Right Input Source.** An update to the right input source of binary operator is generally treated in the same way as that of updates to the left input source, as shown in Theorem 7.1.2. In particular, $T_1\, op\, (T_2 \overset{t}{\sqcup} \triangle T_2) = (T_1\, op\, T_2) \overset{t}{\sqcup} (T_1\, op \triangle T_2)$.

This can directly be shown using the same logic used in proving Theorem 7.1.2. One exception to this is the *Left Outer Join* operator. Such operator requires a special treatment on updates to its right input source. We address that in more detail in Section 7.4.

**Maintaining a Binary Operator Defined on top of Grouping Operations.** A binary operator, in particular any operator from the join family of operators, that receives updates to its input source where such source has been previously processed by a grouping operation (value-based grouping, id-based nesting, or distinct) is not maintainable using the equation in Theorem 7.1.2. We address this issue and provide a separate treatment for it in Section 7.3.

## 7.2 Views with Multiple Operators

We define the distributive property for XML views with more than one algebra operator (a query plan) over the *Deep Union* since the input and the output for such views are XML trees.

**Theorem 7.2.1** *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 $E$. Let $\triangle S_i$ be an update to one of $E$'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 $V^{inc}$ = $V(S_1, .., S_i, .., S_n) \bigsqcup V^{IMP}(S_1, .., \triangle S_i, .., S_n)$ be the view after propagating and applying the updates, $V^{IMP}$ is an incremental maintenance plan derived from $V$. Then $V^{rec}$ = $V^{inc}$ holds.* $\Box$

$V^{IMP}$ is equal to $V$ if the input XAT tables of the *Join* operation in $V$ are not affected by a grouping operation. In this case, $V^{IMP}$ only differs from $V$ in that it is augmented with the counting logic of view maintenance as described in Section 6.4. Queries involving join operations may receive updates to several of their sources. For a view $V(S_1, S_2)$ where the source $S_1$ receives an update $\triangle S_1$ and $S_2$ receives an update $\triangle S_2$, the refreshed view extent is computed as follows: $V(S_1 \bigsqcup \triangle S_1, S_2 \bigsqcup \triangle S_1) = V(S_1, S_2) \bigsqcup V^{IMP}(S_1, \triangle S_1) \bigsqcup V^{IMP}(\triangle S_1, S_2) \bigsqcup V^{IMP}\ (\triangle S_1, \triangle S_2)$ .

Theorem 7.2.1 is similar to Theorem 4.5.4 with one main difference being that $V^{IMP}$ here uses operators augmented with counting as shown in Table 6.2. In Theorem 7.1.2 we have shown the correctness of propagating updates through these operators. The same logic used for proving Theorem 4.5.4 directly applies to Theorem 7.2.1.

Note that a source update as discussed above can be a single update tree or a batch update tree.

**Theorem 7.2.2** *For a set of update trees $\{UT_1, UT_2, ..., UT_n\}$ batched using the* Deep Union *operator into a batched update tree $BUT$, the following holds: $V(BUT)$ = $V(UT_1) \bigsqcup V(UT_2) ... \bigsqcup V(UT_n)$.*

**Proof:** *We have two possible cases:*

1). Single updates share only the root node. *In this case the count annotation of nodes in the path of each update is not affected by the other updates*[2]*. The result of propagating the batch update tree will be clearly equivalent to the result of propagating each single update tree separately.*

[2]Note that we typically assume that the update affects node(s) deeper than the root node. Hence, the root node in any update tree will have a count of 0.

*2).* Single updates may share some other nodes beside the root node.. *In this case some updates may share prefix paths in the batch update tree. By the definition of the Deep Union operator (Definition 6.6.1, the count annotation of a shared node will be equal the summation of counts of all nodes sharing it from the different updates. The resulting count for a shared node will still be either 1, -1, or 0, since a node can be inserted or deleted by only one update. Whenever the node is used as part of the path to an update it is assigned a 0 count. In other words, each node in the batch tree always reflects one type of update, whether it is shared or not, based on its count. For example, if an update is inserting a node* $n$ *for the first time it is assigned a count of 1. If* $n$ *is used in the path of another update that inserts a new node that is a descendant node to* $n$*, in this case* $n$ *is assigned a count of 0 in that second update. Batching these two updates into one batch update tree will not change the count assigned to* $n$ *(1+0 =1). When propagating the batch update tree, the* Source *operator at the leaf of the maintain plan XAT tree generates a tuple that has the root node of the batch update tree. Such root node will have a count of 0, as updates are typically located deeper than that root node. So far different updates are hidden below this root node. At later stage when navigation operations are used these updates might be exposed in intermediate XAT tuples. If a* Navigate collection *is used next that navigates to multiple nodes in the batch update tree, this will create a tuple with column(s) in* $ECC$ *containing node(s) in the path to the distention nodes (with counts of 0) and a new column* $col_k$ *containing the collection that holds the destination nodes. Each of the destination nodes is uniquely identified and is annotated with a count* *representing its update type. These update nodes are separate from each other at this point. The propagation of each of them in batch is equal to the propagation of each of them separately, since each of them affects separate nodes. In other words, we can break this tuple containing this collection of destination nodes into multiple tuples where each tuple contains exactly the same nodes in $ECC$ columns with the same count (0) and contains a collection in column $col_k$ that has one of the update nodes. Each of these tuples is clearly separate from other tuples and updates a unique node. If a* Navigate Unnest *is used to navigate to nodes in the batch update tree, each node will be extracted in a separate tuple. Each of these tuples will represent the specific update type based on the count of the destination node that we navigate to, as counts of nodes in $ECC$ columns are set equal to the count of the destination node, based on our count computation rules in Table 6.2. Hence, we conclude that whether the nodes in the path to an updated node are shared or not, we can still extract the appropriate counts for nodes in that path based on the update type. Note that so far we have shown that the updates in a batch tree are separate from each other in terms of the nodes they affect. Different nodes representing updates might affect the same node in the result if a grouping operation is used. In such case each of the update nodes contributes to the grouping node based on its count, as shown in the proof of Theorem 7.1.2.*

*Hence, we conclude that Theorem 7.2.2 holds.* □

## 7.3 General Views with Join and Grouping Operations

If input sources of the join are affected by a grouping operation then propagating updates using the expression in Theorem 7.1.2 may generate results that are duplicates of previous results generated when processing the initial data.

**Example.** Consider the *Join* operation in Figure 7.1(a) defined over two input sources $T_1$ and $T_2$. Assume that $T_1$ was affected by a grouping operation that grouped tuples by the contents of column $col2$ and applied a *Combine* operator to contents of column $col1$. We also assume that $T_2$ was affected by a grouping operation that grouped tuples by the contents of column $col4$ and applied a *Combine* operator to contents of column $col3$. Note that counts assigned to nodes in grouping columns $col1$ and $col3$ reflect the number of grouped nodes. The result of this *Join* operation is the XAT table $Q$, shown in Figure 7.1(a). Now assume that $T_1$ receives an update $\triangle T_1$ and that $T_2$ receives an update $\triangle T_2$, both shown in Figure 7.1(b). If we were to apply these updates to $T_1$ and $T_2$ and to recompute the *Join* operator, we would get the XAT table $Q'$ shown in Figure 7.1(c).

Now instead consider that we want to maintain this operator incrementally. If we were to use the view maintenance logic we proposed above for maintaining this *Join* operator we would maintain the operator as follows:

$((T_1 \overset{t}{\sqcup} \triangle T_1) join\, (T_2 \overset{t}{\sqcup} \triangle T_2))$ = $(T_1 join\, T_2) \overset{t}{\sqcup} (T_1 join\, \triangle T_2) \overset{t}{\sqcup} (\triangle T_1 join\, T_2) \overset{t}{\sqcup}$ $(\triangle T_1 join\, \triangle T_2)$ .

Figures 7.2(a), (b), and (c) show the execution of the incremental maintenance plans $(T_1 join\, \triangle T_2)$, $(\triangle T_1 join\, T_2)$, and $(\triangle T_1 join\, \triangle T_2)$ respectively

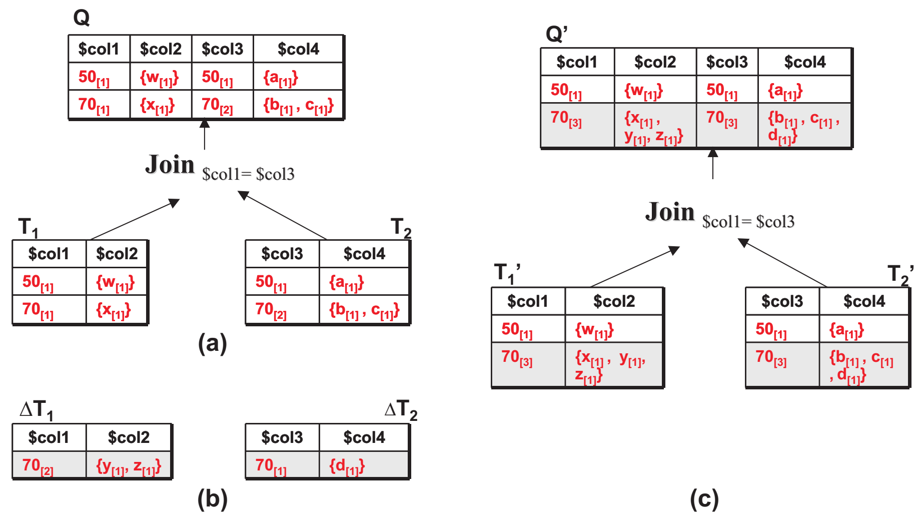


Figure 7.1: An example showing a *Join* operation over sources affected by grouping operations. (a) Initial view extent computation. (b) Source updates. (c) Recomputed view extent.

and their results. We show in Figure 7.2(d) the combined results of these three maintenance plans, obtained through $TUnion$ operation as defined in Definition 7.1.1. Note that when applying this combined incremental result in Figure 7.2(d) to the materialized view extent $Q$ in Figure 7.1(a) it does not refresh it correctly. In other words, we do not get the same result we obtain if we were to recompute the operator over the updated sources, as shown in Figure 7.1(c). In particular, the combined incremental result in Figure 7.2(d) (i) adds 2 additional counts to the node with value 70 in column $col1$, (ii) propagates a duplicate copy of the node $z$ in column $col2$ by mistake, (iii) adds 2 additional counts to the node with value 70 in column $col3$, and (iv) propagates a duplicate copy of each of the nodes $b$ and $c$ in column $col4$. Hence, the refreshed result $Q'$ is incorrect. Moreover, if

$Q$ is not materialized and is an input to another operator in a bigger query plan, such duplicate results will affect the correctness of maintaining any materialized view generated from this plan.

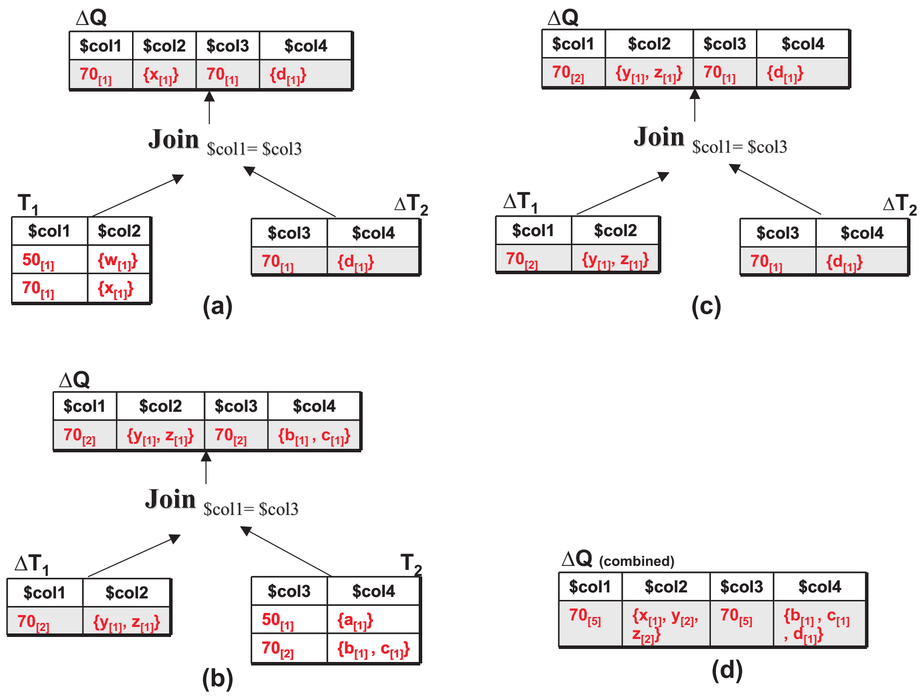


Figure 7.2: Computing the propagated updates resulting from the source updates shown in Figure 7.1(b). (a) $T_1 join \triangle T_2$. (b) $\triangle T_1 join\, T_2$. (c) $\triangle T_1 join \triangle T_2$ . (d) The combined result of the three expressions (a), (b), and (c).

We now analyze the reasons for obtaining such incorrect incremental result.

- When computing $(T_1 join \triangle T_2)$. We find that $\triangle T_2$ joins with the second tuple in $T_1$ (we denote it as $T_1[2]$). This propagates a tuple that has all nodes from $\triangle T_2$ and all nodes from $T_1[2]$. Comparing this to what we would get if we were to recompute the operator after applying $\triangle T_2$ to $T_2$ we find that $\triangle T_2$ should only contribute to the new result the new node in column $col4$ (node $d$) and an increase in the count of the grouping value 70 in column $col3$ by 1, reflecting the addition of this one new node to the group. Nodes originating from $T_1$ should be introduced in the refreshed result $Q'$ one time as they join with the updated group from $T_2$. This is not the case when we maintain the operator incrementally. Nodes from $T_1[2]$ are introduced in the initial result when they join with $T_2[2]$. Later in view maintenance time, when $T_1[2]$ joins with $\triangle T_2$, duplicate result of nodes from $T_2[2]$ are propagated to the view extent by mistake.

- When computing $(\triangle T_1 join\, T_2)$. The same issue occurs as in $(T_1 join\, \triangle T_2)$ but the duplicate result comes from $T_2[2]$. This is because $T_1$ is affected by a grouping operation and nodes from $T_2[2]$ are introduced in the result twice, one time when joined in initial query execution with $T_1[2]$ and second time when they join whith $\triangle T_2$. While they should be joined and introduced in the result one time only if $\triangle T_2$ were to be applied first to $T_2$ and the *join* is to be recomputed.

- When computing $(\triangle T_1 join\, \triangle T_2)$. This expression should not contribute any result to the view extent. This is because nodes from $\triangle T_1$ have been accounted for in the result, in this example, as the expression $(\triangle T_1 join\, T_2)$ has already propagated a tuple containing them. Nodes from $\triangle T_2$ have been also accounted for as the expression $(T_1 join\, \triangle T_2)$ has already propagated a tuple containing them.

Hence, the generated tuple shown in Figure 7.2(c) due to processing $(\triangle T_1 join \, \triangle T_2)$ has duplicate results and should not be propagated. Note that we can not simply solve this problem by removing the expression for the maintenance plans $(\triangle T_1 join \, \triangle T_2)$. This is because in some cases $(\triangle T_1 join \, \triangle T_2)$ might still want to propagate nodes from any of the two updates used by this expression. For example, if $\triangle T_1$ does not join with any tuple in $T_2$, Then nodes from $\triangle T_1$ have to be propagated through this expression.

In the previous example we have shown the problem created by having the input source(s) of a join operation affected by grouping operations. The example illustrates one possible scenarios. Other scenarios are possible. For example, any of the sources might not be affected by a grouping operation, an update operation is a delete or a modify update, and a delete operation deletes an entire group from the input XAT table.

We now propose a general equation for maintaining *join* operators whether or not its input(s) were affected by a grouping operation and on any type of updates.

The equation we propose next (in Theorem 7.3.3) adds to each incremental plan expression from above a **compensating expression** that removes the effect of duplicate results possibly created by having the input sources of the *join* operator being affected by grouping operations, as illustrated in the example above.

We first define two operators $TDiff \overset{t}{-}$ and $TIntersect \overset{t}{\sqcap}$ that are used to compute the difference and intersection of XAT tuples.

**Definition 7.3.1** ***TDiff*** *($\overset{t}{-}$) takes two XAT tables $T_1$ and $T_2$ with the same schema and finds the difference between them. The result is an XAT table that contains single occurrence of each pair of matching tuples (as specified in Definition 4.2.4) and all the non-matching tuples from $T_1$.*

*Each pair of matching tuples ($t_1$ and $t_2$) is merged into a single tuple ($t_{diff}$) that contains a single occurrence of the matching nodes from $t_1[col_i]$ and $t_2[col_i]$ if $col_i \in ECC$, and the difference of the contents of $t_1[col_i]$ and $t_2[col_i]$ if $col_i \notin ECC$.*

*$T_1 \overset{t}{-} T_2$ = $\{t_1 | t_1 \in T_1$ and $\nexists t_2 \in T_2$ where $t_1 \asymp t_2\} \cup \{t_{diff} | \forall t_1 \in T_1$, $\exists t_2 \in T_2$ where $t_1 \asymp t_2\}$*

*$t_{diff}$ is a tuple that conforms to the schema of $t_1$ and has the same $ECC$ as $t_1$. $t_{diff}$ is defined as follows:*

*$\forall col_i \in t_{diff}$, $t_{diff}[col_i] = t_1[col_i]$*

***Node Count:*** *The count annotation of any node $n \in t_{diff}[col_i]$ is equal to the difference in counts between the two matching nodes $n_1$ and $n_2$ from $t_1[col_i]$ and $t_2[col_i]$ respectively where $n.id = n_1.id = n_2.id$, or equal to the count of $n_1$ if $\nexists\, n_2 \in t_2[col_i]$ where $n_1.id = n_2.id$.*

$$n.count = \begin{cases} n_1.count - n_2.count & \exists\, n_1 \in t_1, \exists\, n_2 \in t_2 \text{ where} \\ & n.id = n_1.id = n_2.id \\ n_1.count & \exists\, n_1 \in t_1, n.id = n_1.id \text{ and} \\ & \nexists\, n_2 \in t_2 \text{ where } n.id = n_2.id \end{cases}$$

*For $\forall$ node $n \in t_{diff}[col_i]$ where $col_i \notin ECC$ and if $\exists\, n_1 \in t_1$ and $\exists\, n_2 \in t_2$ where $n.id = n_1.id = n_2.id$, in this case if $n_1.count - n_2.count = 0$, $n$ is deleted.*

***Tuple Count Factor:*** *$t_{diff}.countFactor$ is computed based on the new counts of nodes in $t_{diff}$ following the guidelines discussed in Section 6.5.*

**Empty XAT Tuples.** Similar to the $\overset{t}{\sqcup}$ operator, the $\overset{t}{-}$ operator may also generate *Empty XAT Tuple*s (Definition 7.1.2).

**Proposition 7.3.1** *An* Empty XAT Tuple *(Definition 7.1.2) generated by the $\overset{t}{-}$ operator during view maintenance time is* deleted*.*

**Definition 7.3.2** ***TIntersect*** *($\overset{t}{\sqcap}$) takes two XAT tables $T_1$ and $T_2$ with the same schema and finds the intersection between them. The result is an XAT table that contains single occurrence of each pair of matching tuples (as specified in Definition 4.2.4).*

*Each pair of matching tuples ($t_1$ and $t_2$) is merged into a single tuple ($t_{intersect}$) that contains a single occurrence of the matching nodes from $t_1[col_i]$ and $t_2[col_i]$ if $col_i \in ECC$, and the intersection of the contents of $t_1[col_i]$ and $t_2[col_i]$ if $col_i \notin ECC$.*

$T_1 \overset{t}{\sqcap} T_2 = \{\emptyset | t_1 \not\asymp t_2, t_1 \in T_1, t_2 \in T_2\} \cup \{t_{intersect} | \forall t_1 \in T_1, \exists t_2 \in T_2$ *where* $t_1 \asymp t_2\}$

*$t_{intersect}$ is a tuple that conforms to the schema of $t_1$ and has the same $ECC$ as $t_1$. $t_{intersect}$ is defined as follows:*

*$\forall col_i \in t_{intersect}$, $t_{intersect}[col_i] = t_1[col_i]$ if $col_i \in ECC$ and $t_{intersect}[col_i]$ = ($t_1[col_i] \cap t_2[col_i]$) if $col_i \notin ECC$. $t_1[col_i] \cap t_2[col_i]$ contains any intersecting nodes from $t_1[col_i]$ and $t_2[col_i]$ where nodes are matched by node id.*

***Node Count:*** *The count annotation of all node $n \in t_{intersect}[col_i]$ is set to 0.*

***Tuple Count Factor:*** *$t_{intersect}.countFactor$ is equal to $t_1.countFactor$ expect when $\exists\ n \in t_{intersect}[col_i]$, where $col_i \in ECC$, ($n1.count$ + $n_2.count$) = 0* *where $n_1 \in t_1[col_i]$, $n_2 \in t_2[col_i]$, $n.id = n_1.id = n_2.id$. In this case $t_{intersect}.countFactor$ = 0.*

**Empty XAT Tuples.** The $\overset{t}{\sqcap}$ operator does not delete a node $n$ resulting from matching two nodes, even if $n \in col_i$, $col_i \notin ECC$. This is because we want to maintain the fact that such intersection occurs. For any two cells $t_1[col_i]$ and $t_2[col_i]$ where $t_1 \asymp t_2$ and $col_i \notin ECC$, $t_{intersect}$ will contain an empty collection if none of the nodes in $t_1[col_i]$ and $t_2[col_i]$ match. For any two matching tuples $t_1$ and $t_2$, all nodes in columns in $ECC$ will have count equal to 0. We conclude that an *Empty XAT Tuple* is generated by the $\overset{t}{\sqcap}$ operator in the following case: if $\exists\ t_1\ in\ T_1$, $t_2\ in\ T_2$ where $t_1 \asymp t_2$ and $\forall\ col_i \notin ECC$ none of the nodes in $t_1[col_i]$ matches nodes in $t_2[col_i]$. In contrast to the $\overset{t}{\sqcup}$ and the $\overset{t}{-}$ operators, we do not delete any *Empty XAT Tuple* generated by the $\overset{t}{\sqcap}$ operator during view maintenance time, as we state in Proposition 7.3.2, because we still want to maintain the fact that $t_1$ and $t_2$ intersect by columns in $ECC$, see definition of the $match$ operator 4.2.4.

**Proposition 7.3.2** *An* Empty XAT Tuple *(Definition 7.1.2) generated by the* $\overset{t}{\sqcap}$ *operator during view maintenance time is* not deleted*.*

Now we give our equation for propagating updates though join operations. A join operation here can be the *Theta Join* operator or the *Cartesian Product* operator. The *Left Out Join* operator has a separate treatment that we present in Section 7.4. We will use the term *join* to refer to any of the *Theta Join* operator or the *Cartesian Product* operator in out next discussions.

**Theorem 7.3.3** *A join operator with input sources $T_1$ and $T_2$, and $\triangle T_1$ and $\triangle T_2$ as updates to these sources, can be maintained using the following expression:*

$(T_1 \overset{t}{\sqcup} \triangle T_1)\ join\ (T_2 \overset{t}{\sqcup} \triangle T_2)$ =

$(T_1\, join\, T_2) \overset{t}{\sqcup}$

*(*$(T_1\ join \triangle T_2)$ $\overset{t}{-}$ $(T_1\ join\ (\triangle\ T_2 \overset{t}{\sqcap} T_2)))$ $\overset{t}{\sqcup}$

$((\triangle T_1\, join\, T_2) \overset{t}{-} ((\triangle T_1 \overset{t}{\sqcap} T_1)\ join\ T_2))) \overset{t}{\sqcup}$

$((\triangle T_1\, join\, \triangle T_2) \overset{t}{-} (((\triangle T_1 \overset{t}{\sqcap} T_1)\ join\ \triangle T_2)) \overset{t}{\sqcup} (\triangle T_1\ join\ (\triangle\ T_2 \overset{t}{\sqcap} T_2)))$.

**Proof:** *For a* join *operator with $T_1$ and $T_2$ as input XAT tables, we consider different cases where the input sources may or may not be affected by grouping operations given different update scenarios. We represent the tuple update to $T_1$ as $\triangle T_1$ = $\{xin\}$= [[$x_{11}, x_{12}, .., x_{1m}$]], where $x_{1j}$ is the $j_{th}$ cell in the update tuple and $m$ is the total number of columns in $T_1$. Similarly, we represent the tuple update to $T_2$ as $\triangle T_2$ = $\{xin2\}$ = [[$x_{21}, x_{22}, .., x2n$]], where $x_{2k}$ is the $k_{th}$ cell in update tuple and $n$ is the total number of columns in $T_2$.*

***Input Sources to the Join are not affected by grouping operations**. We want to show that when sources are not affected by grouping operations, the maintenance expression:* $((T_1\ join \triangle T_2) \overset{t}{-} (T_1\ join\ (\triangle\ T_2 \overset{t}{\sqcap} T_2))) \overset{t}{\sqcup} ((\triangle T_1\, join\, T_2) \overset{t}{-} ((\triangle T_1 \overset{t}{\sqcap} T_1)\, join\, T_2))) \overset{t}{\sqcup} ((\triangle T_1\, join\, \triangle T_2) \overset{t}{-} (((\triangle T_1 \overset{t}{\sqcap} T_1)\, join\, \triangle T_2)) \overset{t}{\sqcup} (\triangle T_1\ join\ (\triangle\ T_2 \overset{t}{\sqcap} T_2)))$ *is simply equal to* $(T_1\ join \triangle T_2) \overset{t}{\sqcup} (\triangle T_1\, join\, T_2) \overset{t}{\sqcup} (T_1\, join\, \triangle T_2) \overset{t}{\sqcup} (\triangle T_1\, join\, \triangle T_2)$. *Consider an update $\triangle T_1$ to an input source XAT table $T_1$ of a* join *operation. We first want to show that* $(\triangle T_1\, join\, T_2) \overset{t}{-} ((\triangle T_1 \overset{t}{\sqcap} T_1)\ join\ T_2)$ = $(\triangle T_1\, join\, T_2)$, *$\triangle T_1$ might be representing an insertion, deletion, or modification operation. Hence, we have the following cases:*

- *$\triangle T_1$ represents an insert update. We find that $\nexists$ a tuple $tin_i \in T_1$ where*

*$xin \asymp tin_i$ since the insert update must have new node id(s). Hence, the compensating expression $((\triangle T_1 \overset{t}{\sqcap} T_1)\ join\ T_2)))$ in the maintenance plan does not generate any tuples. There is no tuple in $T_1$ that intersects with $\triangle T_1$ based on the definition of $TIntersect$ above.*

- *$\triangle T_1$ represents a delete update. We find that $\exists$ a tuple $tin_i \in T_1$ where $xin \asymp tin_i$. The expression $((\triangle T_1 \overset{t}{\sqcap} T_1)\ join\ T_2)))$ in the maintenance plan generates for each tuple generated by $(\triangle T_1\ join\ T_2)$ a tuple that is equal to it but with all its node count equal to 0. This is because the $TIntersect$ operator, as defined in Definition 7.3.2, between $\triangle T_1$ and $T_1$ generates a tuple equal to the update tuple $\triangle T_1$ but with all node counts set to 0. The count factor of this generated tuple is set to 0, based on Definition 7.3.2. Hence, when this tuple is joined with $T_2$ it multiplies the nodes in the joined tuple(s) from $T_2$ by the tuple count factor 0, that was set by the $TIntersect$ as we mentioned above.*

  *When subtracting the compensating tuple(s) generated by $((\triangle T_1 \overset{t}{\sqcap} T_1)\ join\ T_2)))$ from the update tuple(s) generated by $(\triangle T_1\ join\ T_2)$ using the $\overset{t}{-}$ operator, the update tuples are not affected.*

- *$\triangle T_1$ represents a modification tuple. We find that for $\triangle T_1$, $\exists$ $tin_i \in T_2$ where $xin \asymp tin_i$. The expression $(\triangle T_1\ join\ T_2)$ in the maintenance plan generates tuples where all nodes originating from $T_2$ are assigned a count of 0. This is due to the 0 multiplicity factor of $\triangle T_1$. Note that nodes originating from columns that are not in $ECC$ of $\triangle T_1$ might have counts of 0, 1, or -1. The compensating expression $((\triangle T_1 \overset{t}{\sqcap} T_1)\ join\ T_2)))$ in the maintenance plan generates tuples with all its node counts equal to 0, based on the* *definition of the $TIntersect$ operator. This is similar to the delete update case above.*

*For all the update types above we conclude that $(\triangle T_1\ join\ T_2) \overset{t}{-} ((\triangle T_1 \overset{t}{\sqcap} T_1)\ join\ T_2)))$ = $(\triangle T_1\ join\ T_2)$ . This is because the compensating expression $((\triangle T_1 \overset{t}{\sqcap} T_1)\ join\ T_2)))$ = $(\triangle T_1\ join\ T_2)$ has no effect as it generates no tuples for insert updates. For delete and modify updates it generates tuple(s) that when subtracted from the tuple(s) generated by $(\triangle T_1\ join\ T_2)$ using the $\overset{t}{-}$ operator have no effect as we have illustrated above.*

*Using the same logic we can easily reach the conclusions that $(T_1\ join\ \triangle T_2) \overset{t}{-} (T_1\ join\ (\triangle\ T_2 \overset{t}{\sqcap} T_2)))$ = $(T_1\ join\ \triangle T_2)$ and that $((\triangle T_1\ join\ \triangle T_2) \overset{t}{-} (((\triangle T_1 \overset{t}{\sqcap} T_1)\ join\ \triangle T_2))) \overset{t}{\sqcup} (\triangle T_1\ join\ (\triangle\ T_2 \overset{t}{\sqcap} T_2)))$ = $((\triangle T_1\ join\ \triangle T_2)$ . Hence, putting the above together we can conclude that for a join with input sources not affected by grouping operations this holds: $((T_1 \overset{t}{\sqcup} T_1)\ join\ (T_2 \overset{t}{\sqcup} T_2))$ = $(T_1\ join\ T_2) \overset{t}{\sqcup} (T_1\ join\ \triangle\ T_2) \overset{t}{\sqcup} (\triangle T_1\ join\ T_2) \overset{t}{\sqcup} (\triangle T_1\ join\ \triangle T_2)$*

***Input Sources to the Join are affected by grouping operations.*** *Next we will discuss the correct propagation semantics of updates on sources affected by grouping operations. We will show that our maintenance plan expression achieves this semantics under different update types. We will break the maintenance plan expression into its three main component expressions and discuss each component separately.*

- *The expression $((T_1\ join\ \triangle\ T_2) \overset{t}{-} (T_1\ join\ (\triangle\ T_2 \overset{t}{\sqcap} T_2)))$ . Given that $T_2$ is affected by a grouping operations, each tuple in $T_2$ has a set of grouping columns $GrpingCS$ and a set of grouped columns $GrpedCS$. If $\triangle T_2$ is to be applied to $T_2$ before the join is performed using the $TUnion$ operator as* *defined in Definition 7.1.1, this will result in one of two cases:*

- $\triangle T_2$ *merges with an existing tuple in* $T_2$*. This case occurs if there is a tuple* $tin2 \in T_2$ *that matches* $\triangle T_2$*, as defined in Definition 4.2.4. Assume that the result of joining* $tin2$ *with a tuple* $tin1 \in T_1$ *is the tuple* $tout$ *in the output XAT table of the join operator. A correct propagation of the update requires that added, deleted, or modified nodes and incremental modifications to counts of nodes in* $\triangle T_2$ *are to be propagated up to the corresponding columns in* $tout$*. While at the same time for those nodes in* $tout$ *originating from* $tin1$ *nothing should change. Using the expression* $((T_1\ join \triangle T_2)$ *to maintain the* $join$ *view on* $\triangle T_2$ *is going to generate duplicate results for nodes originating from* $tin1$*, as shown in the example above. This is because the tuple resulting from applying* $(T_1\ join \triangle T_2)$ *is going to propagate to* $tout$ *nodes originating from* $tin1$ *that should not be propagated. As these nodes are redundant because they have been already propagated through the join with the tuple* $tin2$ *that the update matches with. In other words, if we apply the update to* $tin2$ *first then join with* $tin1$ *we will end up with one single occurrence of each node in* $tin1$ *in the result. While if we process the update separately we end up propagating duplicate nodes from* $tin$ *due to the join with the update.*

  *Now let us consider our solution for this problem. We propose the following maintenance expression:* $((T_1\ join \triangle T_2) \overset{t}{-} (T_1\ join\ (\triangle T_2 \overset{t}{\sqcap} T_2)))$ *. The expression* $(T_1\ join\ (\triangle T_2 \overset{t}{\sqcap} T_2)))$ *compensates for the duplicates that may be created by the extra join between* $T_1$ *and* $\triangle T_2$*.*

*Depending on the update type, the compensating expression creates a compensating tuple* compTup *as follows.*

1). *If the update is an insert update,* compTup *will contain all columns originating from $tin1$ with the same node content and the same counts. In addition it will contain all columns originating from $\triangle T_2$ but with node counts set to 0 (due to the $TIntersect$ operator). Performing $\overset{t}{-}$ between $((T_1\ join \triangle T_2)$ and $(T_1\ join (\triangle T_2 \overset{t}{\sqcap} T_2)))$ as defined in Definition 7.3.1 generates a propagated tuple where (i) all nodes in columns originating from $\triangle T_2$ are the same and (ii) columns originating from the joined tuple $tin1$ contain empty collections, if they are in $GrpedCS$ or contain nodes with 0 counts, if they are in $GrpingCS$. The nodes in columns in $GrpingCS$ are needed as part of the identifier of the propagated tuple when merging this propagated tuple with the previously computed result of the join. They do not however affect the count of the nodes they merge with. Hence, when the propagated update is applied to the result of the* join *only the appropriate nodes are updated.*

2). *If the update is a delete update and it does not delete the entire collection from $T_2$. In other words, if the update is not deleting the last node(s) from the grouped collection of nodes in $T_2$. In this case* compTup *will contain all columns originating from $tin1$ with the same content (same nodes with negative counts resulting from multiplying them by the -1 count factor of $\triangle T_2$) and all columns* *originating from $\triangle T_2$ but with node counts set to 0. Similar to the case of the insert above, performing the $\overset{t}{-}$ operation between (($T_1\ join \triangle T_2$) and ($T_1\ join\ (\triangle T_2 \overset{t}{\sqcap} T_2$))) will compensate for the redundant result from $tin1$ and propagates a correct tuple. If we do not apply* compTup *we will end up propagating more deleted nodes that what should be deleted.*

3). *If the update is a delete update and it deletes the entire collection from $T_2$. In other words, if this covers the case when the update is deleting the last node(s) from the grouped collection of nodes in $T_2$. This should cause the entire group to delete. By the definition of $TIntersect$, the count of all intersecting nodes in the grouping columns of the resulting tuple becomes equal to 0 and the count factor of the update tuple becomes 0. Hence, when this tuple is joined with $tin1$, then the counts of nodes in $tin1$ are multiplied by 0. As a result,* compTup *will contain for all columns originating from $tin1$, the same content but with node counts equal to 0. Also for all columns originating from $\triangle T_2$, they will contain the same content but with node counts equal to 0. Performing $\overset{t}{-}$ between (($T_1\ join \triangle T_2$) and ($T_1\ join\ (\triangle T_2 \overset{t}{\sqcap} T_2$))) as defined in definition in 7.3.1 generates a propagated tuple that is exactly equal to ($T_1$ $join \triangle T_2$) . This tuple reflects the correct propagation semantics since when propagated it deletes from the output XAT table of the* join *operator a tuple $tout$ that would have not been placed in the output if we were to apply $\triangle T_2$ to $T_2$ first (resulting in removing $tin2$ from $T_2$). Then $tin2$ would no longer join with a tuple $tin1$*

*from $T_1$ to generate $tout$.*

4). *If the update is a modify update. In this case* compTup *will contain all columns originating from $tin1$ with the same node content and node counts and all columns originating from $\triangle T_2$ with node counts set to 0. The expression $(T_1\ join \triangle T_2)$ generates a tuple that has all nodes in $tin1$ with counts equal to 0 (due to multiplying by the 0 count factor of $\triangle T_2$). It also has all nodes from $\triangle T_2$ with their assigned counts. For a node in a column in $GrpingCS$ of $\triangle T_2$ that count must be 0, based on Lemma 6.5.1. For a node in a column in $GrpedCS$ of $\triangle T_2$ that count might be -1 reflecting the deletion of leaf node with an old value, 1 reflecting the insertion of a new leaf node with a new value, or 0 reflecting a node in the path to modified leaf node(s). Note that it is possible to have nodes with different counts in a $GrpedCS$ of a modify update if the sum of these counts is equal to 0. We still classify such update as a modify update since it modifies the contents of the group by inserting, deleting, or modifying and does not cause insertion, deletion, or change in the count annotation of the group itself. Performing the $\overset{t}{-}$ between this tuple and* compTup *will generate a correct tuple to propagate.*

– $\triangle T_2$ *does not merge with an existing tuple in $T_2$. This case is possible only for insert updates that create new groups. In other words, if there is no tuple $tin2 \in T_2$ that matches $\triangle T_2$, as defined in Definition 4.2.4. Such updates will append tuples to $T_2$. Hence, the treatment for this* *case is similar to that discussed above for insert updates on a source not affected by grouping operations, since these updates also append tuples to $T_2$. Hence, we reach the conclusion:* $(\triangle T_1\ join\ T_2) \overset{t}{-} ((\triangle T_1 \overset{t}{\sqcap} T_1)\ join\ T_2))) = (\triangle T_1\ join\ T_2)$ .

- *The expression* $((\triangle\ T_1\ join\ T_2) \overset{t}{-} ((\triangle T_1 \overset{t}{\sqcap} T_1)\ join\ T_2))$ . *The reasoning for this expression is similar to that of* $((T_1\ join \triangle\ T_2) \overset{t}{-} (T_1\ join\ (\triangle\ T_2 \overset{t}{\sqcap} T_2)))$ *that we have discussed above but with $T_1$ being updated instead of $T_2$.*

- *The expression* $((\triangle T_1\ join \triangle T_2) \overset{t}{-} (((\triangle T_1\ \overset{t}{\sqcap}\ T_1)\ join\ \triangle T_2))) \overset{t}{\sqcup} (\triangle T_1\ join\ (\triangle\ T_2 \overset{t}{\sqcap} T_2)))$ . *This expression propagates the join of $\triangle T_1$ and $\triangle T_2$ accounting for possible duplicates created as any of the updates $\triangle T_1$ and $\triangle T_2$ may merge with an existing group in $T_1$ and $T_2$ respectively. Consider the following two cases:*

  - *$\triangle T_1$ and $\triangle T_2$ both merge with existing tuples in $T_1$ and $T_2$ respectively. The compensating subexpression* $(\triangle T_1\ join\ (\triangle\ T_2 \overset{t}{\sqcap} T_2)))$ *gives a tuple compensating for the duplicates in $\triangle T_1$ due to joining it with $\triangle T_2$, if $\triangle T_2$ matches a tuple in $T_2$. The reasoning for this compensating sub-expression is similar to that of* $(T_1\ join\ (\triangle\ T_2 \overset{t}{\sqcap} T_2)))$ *that we have discussed above with one difference being that $\triangle T_2$ joins with $\triangle T_1$ instead $T_1$. The same applies to the sub-expression* $(((\triangle T_1\ \overset{t}{\sqcap}\ T_1)\ join\ \triangle T_2)))$ . *Performing a $\overset{t}{\sqcup}$ operation among these two Sub-expressions merges the compensating tuples they generate into one combined compensating tuple. This combined compensating tuple removes duplicates nodes from the propagated update resulting from* $(\triangle T_1\ join\ \triangle T_2)$ .

- *$\triangle T_1$ or $\triangle T_2$ do not merge with any existing tuples in $T_1$ and $T_2$ respectively. Given previous conclusions, if $\triangle T_1$ does not merge with $T_1$ the propagation equation is equal to* $((\triangle T_1\, join\, \triangle T_2) \overset{t}{-} (\triangle T_1\, join\, (\triangle T_2 \overset{t}{\sqcap} T_2))$ *. If $\triangle T_2$ does not merge with $T_2$, the propagation equation is equal to* $((\triangle T_1\, join\, \triangle T_2) \overset{t}{-} ((\triangle T_1 \overset{t}{\sqcap} T_1)\, join\, \triangle T_2)))$ *. If $\triangle T_1$ does not merge with $T_1$ and $\triangle T_2$ does not merge with $T_2$, the propagation is simply equal to* $((\triangle T_1\, join\, \triangle T_2)$ *.*

*We finally conclude that the expression* $((\triangle T_1\, join\, \triangle T_2) \overset{t}{-} (((\triangle T_1 \overset{t}{\sqcap} T_1)\, join\, \triangle T_2))) \overset{t}{\sqcup} (\triangle T_1\, join\, (\triangle T_2 \overset{t}{\sqcap} T_2)))$ *correctly propagates updates for join operations.*
□

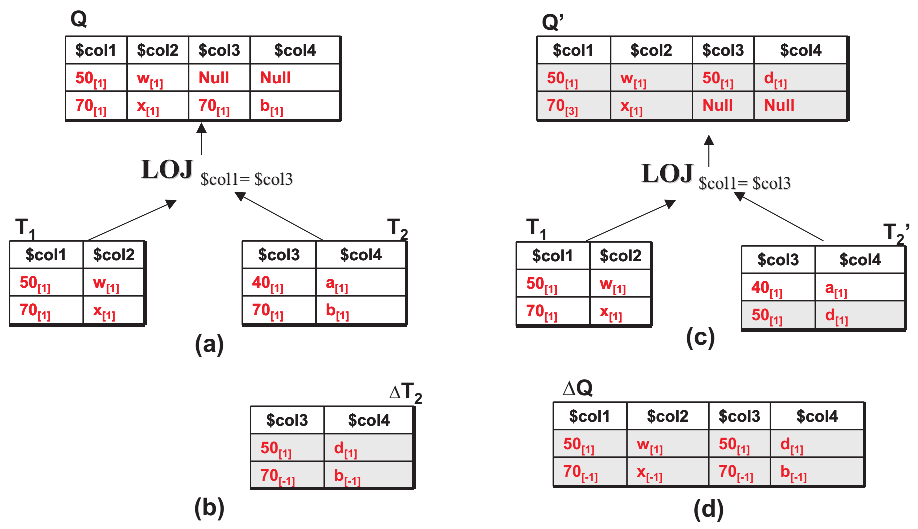
## 7.4 Views with Left Outer Join Operations

The *Left Outer Join* might generate tuples with $null$ values in columns corresponding to the right input source of the operator. This happens when a tuple from the left input source does not merge with any tuples from the right input source. Maintaining the *Left Outer Join* operator on updates to its left input source is straight forward similar to maintaining the *Theta Join* operator as discussed above. In particular, $(T_1 \overset{t}{\sqcup} \triangle T_1) ⟕_c T_2 = (T_1 ⟕_c T_2) \overset{t}{\sqcup} (\triangle T_1 ⟕_c T_2)$. The problem arises when we want to maintain the *Left Outer Join* operator on updates to the right input source ($T_2$) as we illustrate in the following example.

**Example.** Consider the *Left Outer Join* operation in Figure 7.3(a) defined over two input sources $T_1$ and $T_2$. The result of this operation is the XAT table $Q$, shown in Figure 7.3(a). The first tuple in $Q$ has null values in columns $col_3$ and $col_4$, this is because the input tuple for that tuple (first tuple in $T_1$) did not join with any tuple from $T_2$. Now assume that $T_2$ receives two updates, one of them is an insert update and the other is a delete update, both batched in the XAT table $\triangle T_2$ shown in Figure 7.3(b). If we were to apply these updates to $T_2$ and to recompute the *Left Outer Join* operator, we would get the XAT table $Q'$ shown in Figure 7.1(c). $Q'$ contains a tuple that replaces the old tuple in $Q$ that contained null values. $Q'$ also contains a new tuple that replaces the second tuple that was in $Q$. This new tuple has $null$ values for columns $col_3$ and $col_4$, reflecting the fact that source input tuple of that tuple in Table $T_1$ is no longer joining with any tuple in $T'_2$ after applying the delete update to $T_2$. Figure 7.3(d) shows the delta update $\triangle Q$ we would obtain if we were to use the incremental maintenance plan $(T_1 \sqsupset\!\!\bowtie_c \triangle T_2)$ for maintaining the view. Clearly $T_1 \sqsupset\!\!\bowtie_c (T_2 \overset{t}{\sqcup} \triangle T_2)$ is not equal to $(T_1 \sqsupset\!\!\bowtie_c T_2) \overset{t}{\sqcup} (T_1 \sqsupset\!\!\bowtie_c \triangle T_2)$

The example above shows two possible side effects of propagating updates to the right input source of the *Left Outer Join* operator: (i) a previously created tuple that contains $null$ values should be deleted and (ii) a new tuple that contains $null$ values should be inserted. When maintaining the *Left Outer Join* operator we need to account for these two cases. In addition we need to make the propagation rule general enough to support other cases when such side effects are not needed and when propagating modify updates. We propose an algebraic equation for maintaining the *Left Outer Join* operator on updates to its right input source that takes all these into consideration. Unlike the solution in [GK98] we propose a generic equation that supports all types of updates. This enables the propagation of bulk updates possibly of different types at the same time. Our equation also propagates minimal updates, meaning that we do not propagate delete updates to tuples that are not in the output of the operator and we do not delete a tuple and then reinsert it. As a result, we do not require additional rules for ensuring minimality as in [GK98].

We first define a special tuple, called the *Default Tuple* of an XAT table.



Q

| $col1 | $col2 | $col3 | $col4 |
|---|---|---|---|
| $50_{[1]}$ | $w_{[1]}$ | Null | Null |
| $70_{[1]}$ | $x_{[1]}$ | $70_{[1]}$ | $b_{[1]}$ |

$T_1$

| $col1 | $col2 |
|---|---|
| $50_{[1]}$ | $w_{[1]}$ |
| $70_{[1]}$ | $x_{[1]}$ |

$T_2$

| $col3 | $col4 |
|---|---|
| $40_{[1]}$ | $a_{[1]}$ |
| $70_{[1]}$ | $b_{[1]}$ |

Q'

| $col1 | $col2 | $col3 | $col4 |
|---|---|---|---|
| $50_{[1]}$ | $w_{[1]}$ | $50_{[1]}$ | $d_{[1]}$ |
| $70_{[3]}$ | $x_{[1]}$ | Null | Null |

$T_1$

| $col1 | $col2 |
|---|---|
| $50_{[1]}$ | $w_{[1]}$ |
| $70_{[1]}$ | $x_{[1]}$ |

$T_2$'

| $col3 | $col4 |
|---|---|
| $40_{[1]}$ | $a_{[1]}$ |
| $50_{[1]}$ | $d_{[1]}$ |

$\Delta T_2$

| $col3 | $col4 |
|---|---|
| $50_{[1]}$ | $d_{[1]}$ |
| $70_{[-1]}$ | $b_{[-1]}$ |

$\Delta$Q

| $col1 | $col2 | $col3 | $col4 |
|---|---|---|---|
| $50_{[1]}$ | $w_{[1]}$ | $50_{[1]}$ | $d_{[1]}$ |
| $70_{[-1]}$ | $x_{[-1]}$ | $70_{[-1]}$ | $b_{[-1]}$ |

Figure 7.3: An example showing a *Left Outer Join* operation over a an updated right input source. (a) Initial view extent computation. (b) Source updates. (c) Recomputed view extent. (D) delta updates if we use $(T_1 ⟕_c \triangle T_2)$.

**Definition 7.4.1** *The* ***Default Tuple*** $d_T^{cf}$ *for an XAT table* $T$ *is a tuple that has the same schema of* $T$ *and contains* $null$ *values for all columns. The parameter* $cf$ *represents the count factor of that tuple and can be set to 0, 1, or -1.*

We also define the *Semi Join* operator ($\ltimes$) to facilitate our discussion.

**Definition 7.4.2** *The **Semi Join** operator $T_1 \ltimes_c T_2$ is a derived operator that is defined as follows:*

$T_1 \ltimes_c T_2 = \Pi_{T_1.cols}(T_1 \ \bowtie_c \ T_2)$

*where $\Pi_{T_1.cols}$ is the* Project *operator that keeps only all columns from the XAT table $T_1$. The* Project *operator does not affect the* Context Schema *or node counts in columns of $T_1$.*

We now present our algebraic equation for maintaining the *Left Outer Join* operator on updates to its right input source.

**Theorem 7.4.1** *For a* Left Outer Join *operator $T_1 \sqsupset\!\bowtie_c T_2$ with an update $\triangle T_2$ to $T_2$, the following holds:*

$T_1 \sqsupset\!\bowtie_c (T_2 \overset{t}{\sqcup} \triangle T_2) = (T_1 \sqsupset\!\bowtie_c T_2) \overset{t}{\sqcup}$

$((T_1 \bowtie_c \triangle T_2) \overset{t}{\sqcup} ((T_1 \ltimes_c \delta_{cr}(\triangle T_2)) \overset{t}{-} (T_1 \ltimes_c (\delta_{cr}(T_2) \overset{t}{\sqcap} (\delta_{cr}(\triangle T_2)))) \times \{d_{T_2}^{-1}\}))$

*where $\ltimes_c$ is the* Semi Join *operator (as defined in Definition 7.4.2), $d_{T_2}^{-1}$ is the default tuple of table $T_2$ (as defined in Definition 7.4.1), with a count factor set to -1, and $cr$ is the column $\in T_2$ that is used in the join predicate $c$.*

**Proof:** *The incremental maintenance plan in this algebraic equation is composed of two main expressions; (I) the main delta expression: $(T_1 \bowtie_c \triangle T_2)$ and (II) the* compensating expression*: $((T_1 \ltimes_c \delta_{cr}(\triangle T_2)) \overset{t}{-} (T_1 \ltimes_c (\delta_{cr}(T_2) \overset{t}{\sqcap} (\delta_{cr}(\triangle T_2))))$ $\times \{d_{T_2}^{-1}\}$). The main delta expression clearly computes the propagated update resulting from joining $\triangle T_2$ with $T_1$. The* compensating expression *computes any tuples with $null$ values that need to be inserted or deleted from the operator result as a result of propagating $\triangle T_2$. We denote the tuple generated from processing a tuple $t_l$ from $T_1$ that does not join with any tuple from $T_2$ as $t_l^{null}$. Columns in $t_l^{null}$ corresponding to columns in $t_l$ contains the same values as their correspond-*

*ing columns in $t_l$. Columns in $t_l^{null}$ corresponding to columns in $T_2$ contains $null$. We consider that $\triangle T_2$ represents an insert, delete, or modify tuple to $T_2$. For the simplicity of the discussion we assume that $\triangle T_2$ contains only one update tuple. Yet, our result applies to any number of update tuples possibly of different types. We also assume that $\triangle T_2$ can join with only one tuple $t_l$ from $T_1$. We include a discussion at the end of this proof that builds on our conclusions to show that our solution works for batch of updates of possibly different types and for joins with multiple tuples.*

*We now discuss the correctness of the* compensating expression *on different update types. For each case analyze the result obtained from the* compensating expression *on four steps: (I) the result of the sub-expression: $(T_1 \ltimes_c \delta_{cr}(\triangle T_2)$, (II) the result of the sub-expression: $(T_1 \ltimes_c (\delta_{cr}(T_2) \overset{t}{\sqcap} (\delta_{cr}(\triangle T_2))))$, (III) the result of applying the $\overset{t}{-}$ operator between sub-expression (I) and sub-expression (II), and (IV) the result of applying the $\times$ operator between sub-expression (III) and $d_{T_2}^{-1}$.*

***Insert Updates.*** *If the insert update does not join with any tuple from $T_l$ we clearly get no result from the main delta expression and from the* compensating expression. *If $\triangle T_2$ joins with a tuple $t_l$ from $T_1$, we have two possible cases:*

- If $t_l$ was not previously joined with any tuples from $T_1$. *In this case, the previous result generated by $(T_1 ⟕_c T_2)$ contains $t_l^{null}$. This tuple should be deleted. We show that the* compensating expression *propagates an update tuple $x_l^{null}$ that deletes $t_l^{null}$. We show that in three steps (I) The sub-expression: $(T_1 \ltimes_c \delta_{cr}(\triangle T_2)$ gives the tuple $t_l$ from $T_1$ that should join with $\triangle T_2$. (II) The sub-expression: $(T_1 \ltimes_c (\delta_{cr}(T_2) \overset{t}{\sqcap} (\delta_{cr}(\triangle T_2))))$ gives noth-*

*ing in this case because the update $\triangle T_2$ does not intersect with old tuples in $T_2$ on the join predicate value. (III) Applying the $\overset{t}{-}$ operator between sub-expression (I) and sub-expression (II) gives $t_2$. (IV) Applying the operator $\times$ between $t_2$ and the $d_{T_2}^{-1}$ (the default tuple of $T_2$ with count factor set to -1) gives us a tuple $x_l^{null}$. $x_l^{null}$ is equal to $t_l^{null}$ but with count annotation for each node in non-$null$ columns in $x_l^{null}$ set to -1 times the count annotation of the corresponding node in $t_l^{null}$. Hence, $x_l^{null}$ deletes $t_l^{null}$ from previous result. Note that the main delta expression $(T_1 \bowtie_c \triangle T_2)$ generates an insertion tuple that accounts for the fact that $\triangle T_2$ joins with $t_l$ from $T_1$. This tuple is a full tuple and has no null values.*

- If $t_l$ was previously joined with tuples from $T_1$. *In this case for $t_l$ that $\triangle T_2$ joins with, the previous result $(T_1 \bowtie_c \triangle T_2)$ does not contain $t_l^{null}$, as described above. Hence, the* compensating expression *should not propagate any update. We show that as follows. (I) The sub-expression: $(T_1 \ltimes_c \delta_{cr}(\triangle T_2)$ gives the tuple $t_l$ from $T_1$ that should join with $\triangle T_2$. (II) The sub-expression: $(T_1 \ltimes_c (\delta_{cr}(T_2) \overset{t}{\sqcap} (\delta_{cr}(\triangle T_2))))$ gives a tuple exactly equal to $t_2$. (III) Applying the $\overset{t}{-}$ operator between sub-expression (I) and sub-expression (II) gives an* Empty XAT Tuple *by the definition of the $match$ operator (Definition 4.2.4) and the definition of the* Empty XAT Tuple *(Definition 7.1.2). By Proposition 7.3.1, this* Empty XAT Tuple *is deleted, hence we end up with an empty result. (IV) Applying the operator $\times$ between the empty result we get from sub-expression III and the $d_{T_2}^{-1}$ gives nothing. Hence, we conclude that the maintenance plan propagates only $(T_1 \bowtie_c \triangle T_2)$.*

***Delete Updates.*** *A delete update always joins with a tuple from $T_l$. We have two possible cases:*

- *If $\triangle T_2$ deletes the last tuple from $T_2$ that joins with $t_l$. In this case we should propagate an update that inserts the tuple $t_l^{null}$ into the operator's result. The* compensating expression *generates such tuple as follows. (I) The sub-expression: $(T_1 \ltimes_c \delta_{cr}(\triangle T_2)$ gives the tuple $t_l$ from $T_1$ that should join with $\triangle T_2$. Note that all node counts in $t_l$ are negative values since they were multiplied by the -1 count factor of $\triangle T_2$, based on our count rules in Table 6.2. (II) The sub-expression: $(T_1 \ltimes_c (\delta_{cr}(T_2) \overset{t}{\sqcap} (\delta_{cr}(\triangle T_2))))$ gives a tuple exactly equal to $t_2$ but will all node counts set to 0. This 0 node count is obtained because the intersection between $(\delta_{cr}(T_2))$ and $(\delta_{cr}(\triangle T_2))$ gives a tuple with 0 count factor (as the count of nodes in column $cr$ from $T_2$ and $\triangle T_2$ must sum to 0 and by the definition of the $\overset{t}{\sqcap}$ operator in Definition 7.3.2). When we compute the $\ltimes_c$ operator between this tuple and $T_1$ we obtain $t_l$ with all node counts set to 0, by our count rules in Table 6.2. (III) Applying the $\overset{t}{-}$ operator between sub-expression (I) and sub-expression (II) gives $t_l$. (IV) Applying the operator $\times$ between the result of (III) and $d_{T_2}^{-1}$ gives $t_l^{null}$, a tuple with nodes corresponding to columns in $T_1$ and null values corresponding to columns in $T_2$. Note that the node counts in $t_l^{null}$ are all positive due to multiplying the negative counts in $t_l$ by the -1 count factor of $d_{T_2}^{-1}$. Note that the main delta expression $(T_1 \bowtie_c \triangle T_2)$ generates a deletion tuple that accounts for the fact that $\triangle T_2$ joins with $t_l$ from $T_1$. This tuple is a full tuple and has no null values.*

- *If $\triangle T_2$ does not delete the last tuple from $T_2$ that joins with $t_l$. In this case we* *should not propagate any update that inserts a tuple $t_l^{null}$ into the operator's result. Hence, the* compensating expression *should not propagate any update. We show that as follows. (I) The sub-expression: $(T_1 \ltimes_c \delta_{cr}(\triangle T_2)$ gives the tuple $t_l$ from $T_1$ that should join with $\triangle T_2$. (II) The sub-expression: $(T_1 \ltimes_c (\delta_{cr}(T_2) \overset{t}{\sqcap} (\delta_{cr}(\triangle T_2))))$ gives a tuple exactly equal to $t_2$ and with the same node counts. This is because the predicate value in $\triangle T_2$ intersects with the predicate value of $T_2$ but since its is not deleting the last tuple in $T_2$ the sum of counts of these values in $T_2$ and $\triangle T_2$ does not become 0, like the last case. (III) Applying the $\overset{t}{-}$ operator between sub-expression (I) and sub-expression (II) gives an* Empty XAT Tuple *by the definition of the $match$ operator (Definition 4.2.4) and the definition of the* Empty XAT Tuple *(Definition 7.1.2). By Proposition 7.3.1, this* Empty XAT Tuple *is deleted, hence we end up with an empty result. (IV) Applying the operator $\times$ between the empty result we get from sub-expression III and the $d_{T_2}^{-1}$ gives nothing. Hence, we conclude that the maintenance plan propagates only $(T_1 \bowtie_c \triangle T_2)$.*

***Modify Updates.*** *A modify update always joins with a tuple from $T_l$ (similar to delete updates). The modify update will never cause the deletion of insertion of $t_l^{null}$ as discussed above. Hence, the* compensating expression *should not propagate any update. The logic for showing that this holds is exactly equal to that used for the second case of the delete update above.*

*We conclude that Theorem 7.4.1 holds for all type of updates.*

*It is easy to build on the discussion above to show that Theorem 7.4.1 holds when $\triangle T_2$ represents a batch of updates of the same or different types. The main* *delta expression:* $(T_1 \bowtie_c \triangle T_2)$ *propagates the appropriate full tuples representing insertion, deletion, or modification operation. The* compensating expression*:* $((T_1 \ltimes_c \delta_{cr}(\triangle T_2)) \overset{t}{-} (T_1 \ltimes_c (\delta_{cr}(T_2) \overset{t}{\sqcap} (\delta_{cr}(\triangle T_2)))) \times \{d_{T_2}^{-1}\})$ *computes the net compensating effect of all the updates in* $\triangle T_2$ *joining with the same tuple in* $T_1$ *on the predicate value. Such net effect is computed through the use of the* Distinct *operations* $\delta$ *in the* compensating expression*. The Net effect will always be one of three cases (I) the propagation of one tuple that deletes* $t_l^{null}$*, (II) the propagation of one tuple that inserts* $t_l^{null}$*, or (III) not propagating anything. The* compensating expression *computes a separate net compensating effect for diffident tuples in* $T_1$ *even they have the same join predicate value. This is because those tuples are granted not match by their columns in* $ECC$*.* $\square$

**Left Outer Join Operations with Input Sources Affected by Grouping Operations.** Given the class of XQuery views we support it is possible that only the left input source of the *Left Outer Join* operator to be affected by a grouping operation. In this case, maintaining the *Left Outer Join* operator on updates to that source is done following the same logic used for maintaining the *Theta Join* and the *Cartesian Product* operators, presented in Theorem 7.1.2. In particular, $(T_1 \overset{t}{\sqcup} \triangle T_1) ⟕_c T_2 = (T_1 ⟕_c T_2) \overset{t}{\sqcup} ((\triangle T_1 ⟕_c T_2) \overset{t}{-} ((\triangle T_1 \overset{t}{\sqcap} T_1) ⟕_c T_2)))$ .

## 7.5 Views with Self Joins

To process updates to views that have self joins in a distributive manner, each occurrence of the accessed source XML document is treated as a separate document. Hence, if a view $V(S)$ accesses $S$ twice performing a self join we treat the view as being defined over two data sources $V(S_1, S_2)$ where $S_1 = S_2 = S$. The view maintenance is performed as follows: $V(S_1 \bigsqcup \triangle S_1, S_2 \bigsqcup \triangle S_2) = V(S_1, S_2) \bigsqcup V^{IMP_1}(\triangle S_1, S_2) \bigsqcup V^{IMP_2}(S_1, \triangle S_2)$ $\bigsqcup V^{IMP_3}(\triangle S_1, \triangle S_2)$ . This is similar to the treatment we have used in Section 4.4 but now with incremental maintenance plans being augmented with counting functionality we can support different types of updates. The view in Figure 1.2(a) is an example for this case where the query accesses the same source "bib.xml" twice performing a self join. We will show later an example of propagating different types of updates through this view.

## 7.6 Views with Aggregate Functions.

Our solution can easily support the maintenance of the aggregate functions $min$, $max$, $count$, $sum$, and $average$ in a distributive manner on insert updates. Each node in the view extent that represents an aggregation, is assigned a function that computes the new aggregate value when applying propagated updates. This treatment is similar to that used in [LD00]. Each of the functions assigned to nodes representing aggregation mainly uses the propagated aggregate value. It may also use the old aggregate value and the count annotation associated with the node if needed. For example, assume a node in a view extent represents an average price equal to 50 for a certain book. Also assume that the count associated with that node is equal to 3 (representing three prices). Now assume a source update that inserts two new prices for that book, namely 50 and 70. This update propagates to an average price of 60 with count of 2. Applying this propagated update to the previously computed average node refreshes it to ((50*3)+ (60*2))/ (3+2) = 54.

Some aggregate functions are not distributive on delete operations, particularly $max$ and $min$. Solutions from the literature for maintaining non-distributive aggregate function, like [PSCP02], can be adapted to support maintaining such non-distributive aggregate functions in our framework.

## 7.7 Update Propagation Example

We represent the XQuery view in Fig 1.2(a) as $V(S_1, S_2, S_3)$ where both $S_1$ and $S_2$ represent the source document "bib.xml" accessed twice by the query performing a self join. $S_3$ represents the source document "prices.xml". Given that all the three sources of the query are updated we maintain the materialized view generated by the query as follows: $V(S_1 \bigsqcup \triangle S_1, S_2 \bigsqcup \triangle S_2,$ $S_3 \bigsqcup \triangle S3)$ = $V(S_1, S_2, S_3) \bigsqcup V^{IMP_1}(S_1, S_2, \triangle S_3) \bigsqcup V^{IMP_2}(S_1, \triangle S_2, S_3) \bigsqcup V^{IMP_2}$ $(S_1, \triangle\ S_2, \triangle S_3) \bigsqcup V^{IMP_3}\ (\triangle S_1, S_2, S_3) \bigsqcup V^{IMP_3}\ (\triangle S_1, S_2,\ \triangle S_3) \bigsqcup V^{IMP_4}(\triangle S_1$ $, \triangle S_2, S_3) \bigsqcup V^{IMP_4}\ (\triangle S_1, \triangle S_2, \triangle S_3)$ . Where $V^{IMP_1}$ is a derived view maintenance plan that is equivalent to $V$, but augmented with view maintenance counting logic as discussed in Section 6.4. $V^{IMP_2}$ is a derived view maintenance plan augmented with view maintenance counting logic and the maintenance equation for updates on the right input source of *Left Outer Join* operators, given in Theorem 7.4.1, is used for operator $\#$ 7 in Figure 4.2. $V^{IMP_3}$ is a derived view maintenance plan augmented with view maintenance counting logic and the general maintenance equation for *Join* operators, given in Theorem 7.3.3, is used for operator $\#$ 10 in Figure 4.2. $V^{IMP_4}$

is a derived view maintenance plan augmented with view maintenance counting logic and in addition the maintenance equation for updates on the right input source of *Left Outer Join* operators, given in Theorem 7.4.1, is used for operators # 7 in Figure 4.2 and the general maintenance equation for *Join* operators, given in Theorem 7.3.3, is used for operators # 10 in Figure 4.2.

Executing the seven incremental maintenance plans above gives the results shown in Figure 7.4. The resulting propagated updates (XML trees) shown in Figure 7.4 are annotated with semantic ids and counts. Some of the semantic ids are reproductions of previously generated ids and others are new ones. We will discuss next how the semantic ids and the counts are used in refreshing the view extent.

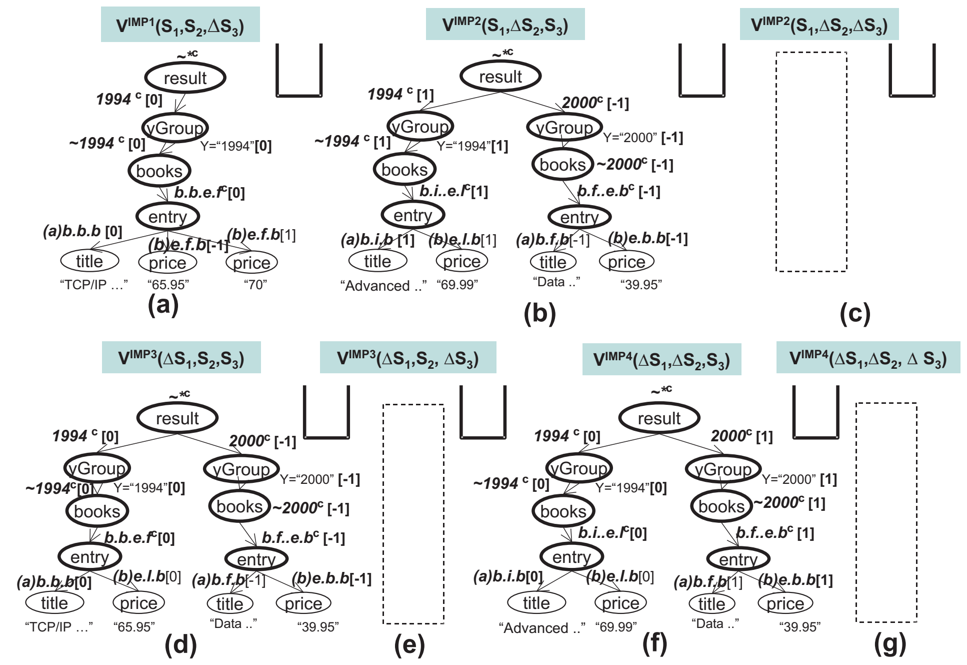


Figure 7.4: Obtaining delta update trees.

# Chapter 8

# Applying Propagated XML Updates

## 8.1 Refreshing the View Extent on Different Types of Updates

We use the *Deep Union* operator (Definition 6.6.1) to apply delta update trees to the existing materialized XML view (also a tree). From the root of the two trees, the *Deep Union* recursively matches nodes using their ids. Nodes with matching ids are merged into one node and their count annotations are summed. When the *Deep Union* operation is finished, all the counts of affected nodes in the view extent are updated. As a last step, nodes with counts equal to 0 are removed. Given our counting solution, we can even delete a node and its entire subtree from the view extent during the application of the *Deep Union* if the node counts becomes 0 without worrying about individually deleting nodes in its subtree first. Hence, *Deep Union* terminates at the deleted root node without accessing any node in its subtree. This achieves high efficiency for deleting large XML fragments from the view extent.

## 8.2 Example of Applying Propagated Updates

Figure 8.1(a) depicts the original view extent. Figure 8.1(b) depicts the combined delta update trees from Figure 7.4. Applying the *Deep Union* operator to these two trees generates the refreshed view extent shown in Figure 8.1(c). Note that any two nodes from the two trees with equivalent semantic node ids are merged into one node and their counts are summed. For example, the nodes "yGroup" with id $1994^c$ are merged into one node. Its count becomes 2. Nodes with different ids are represented separately. For example, the node $b.i..e.f^c$ appears as a new node in the view extent. The delta update tree in Figure 8.1(b) contains the node "yGoup" with id = $2000^c$ and count = -1. This node merges with a corresponding node in the view extent that has the same id and a count of 1. Hence, the count of the merged node becomes 0. At this point, we disconnect the entire XML fragment rooted at this node and terminate the *Deep Union* traversal for this fragment here.

The net effect of applying the delta update tree in Figure 8.1(b) to the view extent in Figure 8.1(a) is the deletion of the XML fragment with root node "yGroup" with id $2000^c$, insertion of the XML fragment with root node "entry" with id $b.i..e.l^c$ and change the "price" value of the node with id $(b)e.f.b$. The count of nodes bound to the 1994 book year group increments to 2. The result shown in Figure 8.1(c) is the exact result we would have gotten if we were to recompute the view extent over the updated sources.

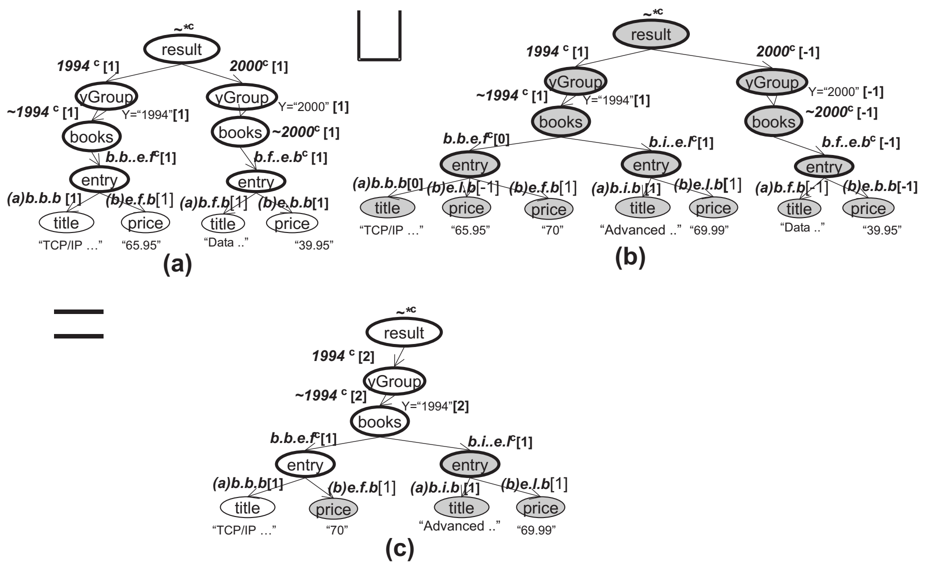


Figure 8.1: Applying combined propagated updates to initial view extent to refresh it. (a) Initial view extent, (b) delta updates, and (3) refreshed view extent.

## 8.3 Discussion on Refreshing the View Extent

### 8.3.1 The Order of the Refreshed View Extent

Note that since the order is encoded as part of the semantic ids, the view extent preserves the order. Nodes from the update that are merged with existing nodes do not affect the order (e.g., node $b.b..e.f^c$), as they have an order equivalent to the order of the nodes they were merged with. Nodes inserted into the view extent come with their own order. For example, the node $b.i..e.l^c$ is inserted after the node $b.b..e.f^c$ (which is the correct order based on source document order and the query order semantics). Although insertion affects the relative order of proceeding nodes in the view extent, our solution does not require any relabeling of order keys. See discussion in Section 3.4.4.

### 8.3.2 Deleting Collections of XML Fragments Without Lineage Context from the Materialized XML View

Some queries may return collections of nodes (fragments) without unnesting nodes in these collections and without maintaining a *Lineage Context* for these collections. In other words, each returned collection is not identified separately in the XML result. For example, the query:

```
<result>
  for $b in doc("bib.xml")/bib/book
  return
     $b/author
< /result>
```

returns for each "book" binding, a collection that contains all "author" children nodes of that "book". Each collection is not identified because its *Lineage Context* (its parent "book" node) is not maintained in the view extent. The *Lineage Context* of each collection is maintained for example if a constructed node is built on top of each collection. For example, $< authorList >$\$b/author$< /authorList >$. In this case and given our semantic identifier solution each “authorList” node is assigned an id that is derived from the id of the “book” node related to each collection.

If a delete update deletes a ”book” node from “bib.xml”, this delete update should be reflected to the view extent by deleting all the ”author” nodes of that book. We represent this source update using an update tree that contains a “book” node annotated with the specific node id of the ‘book” node to be deleted, as discussed in Chapter 5. Propagating this update through the query with constructed nodes “authorList” on top of the author collections result in a delta update tree that contains a root node “result” and an “authorList” node with id that matches the id of the “authorList” on top of the collection of authors to be deleted. Hence, refreshing the view extent using this delta update deletes the relevant “authorList” and its entire subtree, hence deleting the collection of authors related to the deleted “book” node.

Now let us consider the query above without the constructed nodes “authorList”. Propagating the same source update through the query generates a delta update tree with only a root node “result”. Clearly, applying this delta update to the materialized view will not delete the “author” nodes related to the deleted “book” node.

One possible solution to this case is to provide all the “author” nodes of the deleted “book” node as part of the source update tree. Hence, the propagated delta update tree will contain all “author” nodes to be deleted. This is not a practical solution as we wish to represent source updates using minimum information. Another solution is to assign a dummy parent node to each collection of “author” nodes in the view extent. Such dummy parent node maintains the *Lineage Context* of each collection (which is the related “book” id in this example), hence allowing the identification of each collection separately during view maintenance time. Such solution can be implemented by extending the *Combine* and the *XML Union* operators to attach the relevant *Lineage Context* ids for the collections they process.

# Chapter 9

# Experimental Evaluation

We have implemented our solution in Java on top of the Rainbow system framework [Zea03]. We have run the experiments on a Windows PC with 2.79 MHz Pentium 4 processor and 512MB of memory. We use the XMark benchmark data [SWK$^+$02] in our experimental evaluation and the two queries in Figure 4.8. Query 1 in Figure 4.8 performs a lot of result construction with most of the returned nodes being constructed. The query also involves a mixture of order decisions, in which some nodes are returned in document order (e.g., customers and bids) and others are returned in an order imposed by the query (e.g., order among customers and bids and among the children of each of them separately). Query 2 in Figure 4.8 is a simpler query that does not perform node construction.

## 9.1 Cost of Enabling View Maintenance

The query engine has to support two main features when processing XML data to enable view maintenance at a later stage: (1) semantic ids (Chapter 4)[1] and (2) counting annotation (Chapter 6). Our experiments show that this cost is small relative to the query execution time. As shown in Figure 9.1, using a source document of size 500MB and query selectivity 50%, we break the cost associated with generating semantic ids into two sub-costs: (i) the cost of computing the *Context Schema* and (i) the cost of generating the keys. *Context Schema* computation is a one-time processing cost at the query tree generation time. It only depends on the size of the query algebra tree. Figure 9.1 shows that this cost is very small for both queries (Query 2 has a smaller query tree size). The cost of key generation includes the cost of generating new semantic identifier keys for new constructed nodes and the cost of assigning overriding order prefix keys. This cost depends on both how much node construction and order manipulation the query performs. It also depends on the size of the processed data.

Query 1 performs a lot of node constructions, where most of the returned nodes in the result are constructed ones. It also defines explicit order among nodes. Yet this cost is still small relative to the query execution time, as shown in Figure 9.1. The cost of key generation drops to almost 0 for Query 2, as shown in Figure 9.1, as this query constructs only one node (the root node of the document). It does not enforce any new order upon the processed data. Hence, no overriding order prefix keys are assigned

[1]Note that *Context Schema* (presented Chapter 4) used to generate semantic ids utilizes the *Order Schema* (presented Chapter 3).

to the processed nodes. Query 2 still returns results in document order, as required by XQuery semantics. Such order is reflected through the source nodes ids and is maintained by our solution at almost no cost (besides the negligible small cost of computing the *Context Schema*). The counting cost in Query 1, very small, is mainly caused by the count annotation assigned to the newly constructed nodes. Other processed nodes have a default node count of 1 that does not change. For Query 2 the counting cost drops to 0 since no new node construction is performed.

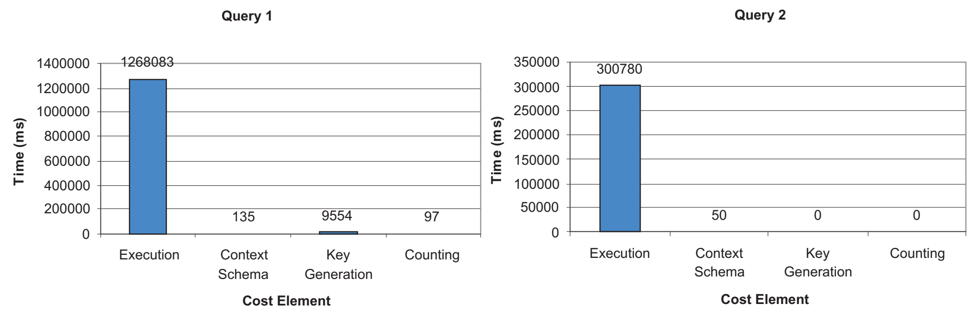


Figure 9.1: Cost of enabling view maintenance feature.

## 9.2 Varying Source Document Sizes

We compare the performance of our view maintenance solution to view recomputation for different base source XML document sizes. We vary the size of the source document from 50MB to 650MB. We fix the selectivity of the query to 50%. This means that the query returns 50% of the result that it would return if we were to remove the selection condition. We use a source update that inserts a new person fragment. Figure 9.2 (two charts at top) shows that the cost of maintaining the view incrementally is very small compared to the cost of recomputation. The increase in the cost of maintaining the view incrementally is much smaller than the increase in the cost of recomputation as the size of source document increases.

In the lower part of Figure 9.2, we show the break of the view maintenance cost down into the two costs of propagate and apply. It shows that the cost of propagating the update is fixed regardless of the source document size. The cost of the apply phase increases with the increase in the source document size. Given the fixed selectivity of the query, the size of the view extent also increases requiring more work at the apply phase.

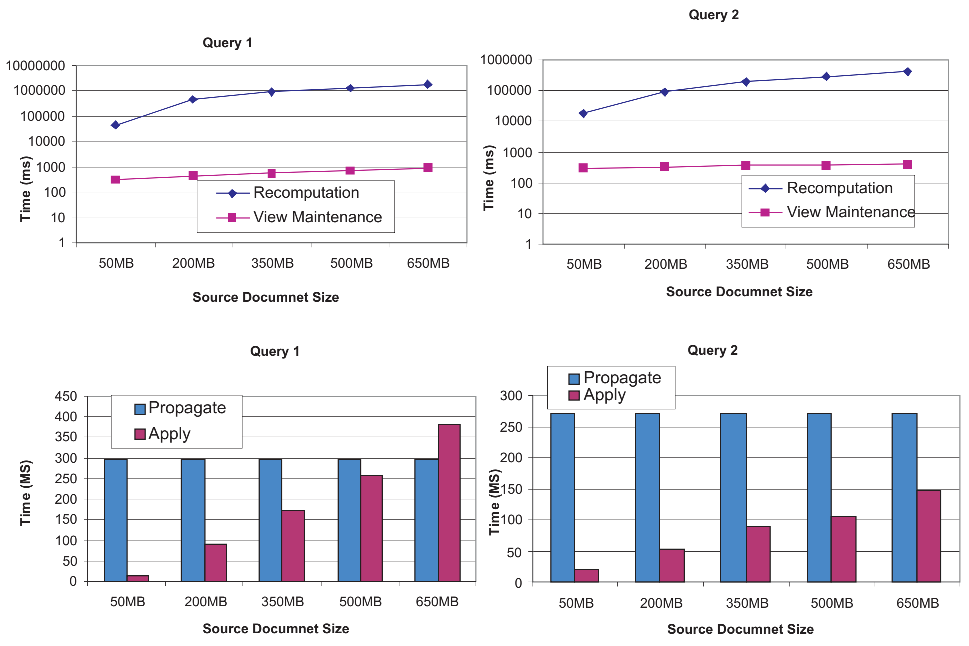


Figure 9.2: Top charts showing varying source document size for (a) Query 1 and (b) Query 2. Bottom charts showing the break down of the view maintenance cost.

## 9.3 Varying View Selectivity

We vary the selectivity of the queries from 20% to 100%. We use a source document with size 100MB and an insert update. Figure 9.3 shows that view maintenance also performs well on different selectivity levels of the query. The cost of view maintenance slightly increases with the increase in the selectivity. This is mainly due to the size of the view extent getting bigger as the selectivity increases. Hence, more work is done at the apply phase.

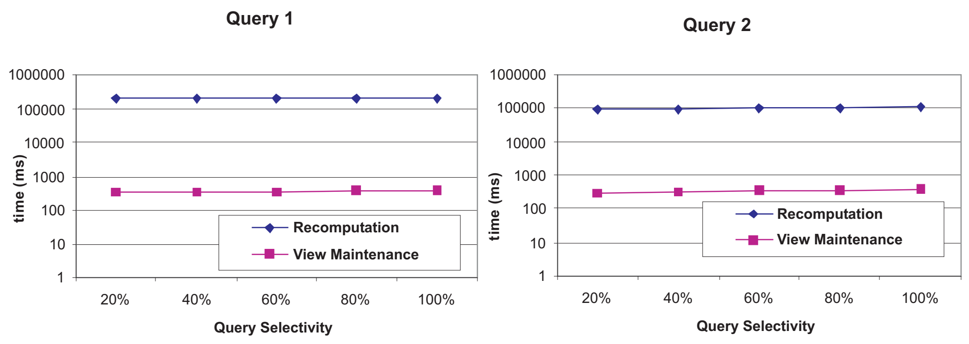


Figure 9.3: Varying query selectivity.

## 9.4 Varying Update Sizes

We measure the performance of the system for updates of different sizes. We have used a source document of size 50MB. We fix queries selectivity to 50%. We vary the size of updates from 20% to 100% of the size of the initial source data used to construct the view extent. First we test insert updates. We start from a source document of a constant size and the insert update causes this size to grow. Figure 9.4 shows that our incremental view maintenance has a superior performance over recomputation even for update sizes up to 100% of the initial data size. As a matter of fact, the increase in the view maintenance cost due to the increase in the update size almost matches the increase of the cost of recomputation. This is mainly because the cost of view maintenance for a certain size of source updates is almost equal to the cost of processing this update when applied first to the source document and then processed during recomputation time. Recomputation in addition has the cost of processing the initial source data before the update.

The cost of view maintenance for a certain update is composed of the three cost elements. (1) Cost of processing the updates similar to regular query processing. This cost is the main cost in view maintenance as shown in Figure 9.4. (2) Cost of accessing counts associated with source update nodes to specify the update type. This cost is a small percentage of the total cost of view maintenance as shown by Figure 9.4[2]. (3) Cost of applying the propagated updates to the view extent. This cost is also small as shown in Figure 9.4.

Now we test the delete updates. We use the same updates as above (but now as delete updates) and the same queries with the same selectivity. We set the source document to be always of fixed size after applying updates of different sizes. Therefore the recomputation cost is fixed for different sizes of the delete updates as shown in Figure 9.5, as the final data after delete

[2]This cost is overestimated in our current system due to inefficient treatment of update count annotation.

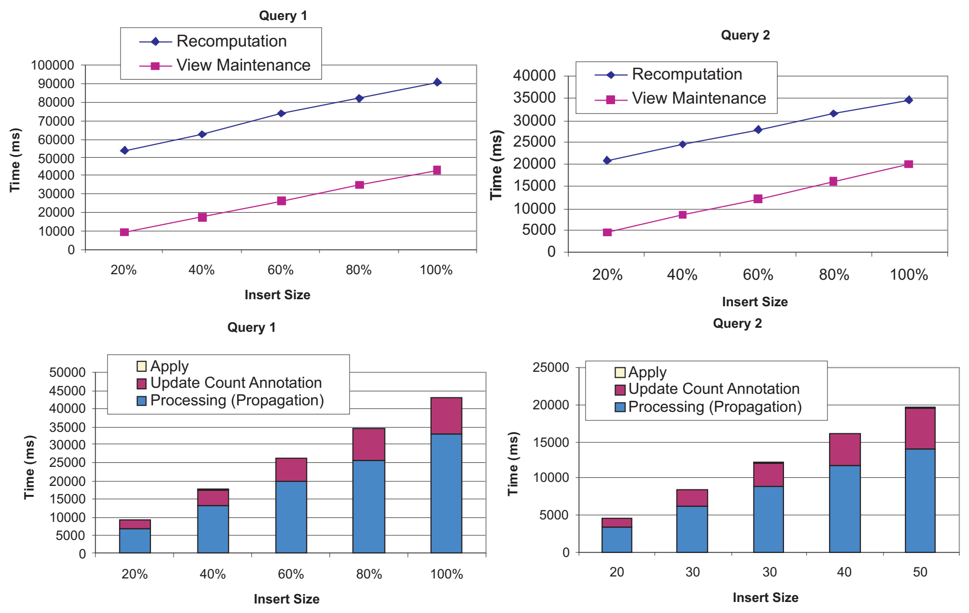
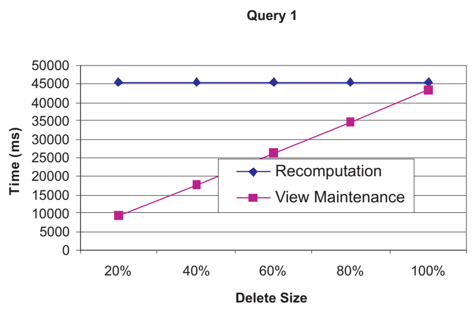


Figure 9.4: Top charts showing varying size of insert update. Bottom charts showing break down of view maintenance cost.

stays the same. Figure 9.5 shows that view maintenance achieves superior performance on delete updates up to sizes around 100% of the initial source data size.

## 9.5 Deletion of Entire Fragments

We measure the performance of our system when a source update causes the deletion of a relatively big fragment from the view extent. Figure 9.6(a) shows Query 3 that navigates to "people" element and returns a new node "persons-list" that has as children all the names and addresses of persons in "people". Now assume that an update deletes the source node "people".



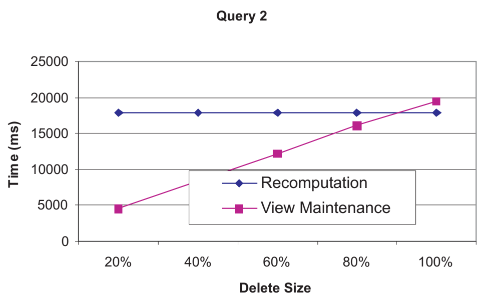


Figure 9.5: Varying size of delete update for (a) Query 1 and (b) Query 2.

This should result in deleting the entire "persons-list" fragment from the view extent. Due to our method of representing source updates and our count treatment, our view maintenance propagates this update very efficiently. It deletes the entire "persons-list" fragment from the view extent directly by deleting its root.

Figure 9.6(b) shows the performance of our solution on different sizes of deleted XML fragments represented using the number of "person" fragments it contains. We compare our approach to the approach in [LD00] where all the internal nodes in the "persons-list" fragments have to be deleted first in a bottom-up fashion, before the root of the fragment can finally be deleted. Figure 9.6(b) shows the performance of our approach (entire fragment deletion) and the performance of the alternate approach (individual deletions). Our solution maintains the view extent in constant time regardless of the size of the deleted fragment. The alternate approach has its cost increase linearly with the increase in the fragment size.

```
<Result>{                                    Query 3
for $p in document("site.xml")/people
return
  <persons-list>{
        <person>{$p/person/name}
                {$p/person/address}
        </person>
  </persons-list>}
</Result>
```

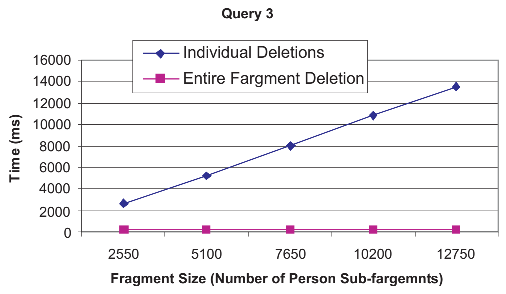


Figure 9.6: (a) Query 3 and (b) cost of deleting "persons-list" fragment in Query 3.

# Chapter 10

# Related Work

## 10.1 View Maintenance Related Work

**View Maintenance in Relational Databases.** The incremental maintenance of materialized views has been extensively studied for relational databases [BLT86, GM95, GMS93, ZGMHW95, CW91, GL95, MK00, Qua96, MQM97, PSCP02, GM05].

Blakeley et. al. [BLT86] have proposed an algebraic solution for maintaining $SPJ$ views. [BLT86] defines algebraic rules for propagating updates for different algebra operators. For example, for a join view $V$= $T_1 \bowtie T_2$, if $T_1$ receives an insert update $\triangle T_1$, the algebraic rule for propagating $\triangle T_1$ is: $(T_1 \cup \triangle T_1) \bowtie T_2$ = $(T_1 \bowtie T_2) \cup (\triangle T_1 \bowtie T_2)$. The propagation of updates through expressions composed of multiple operators can simply be done by repeatedly applying the algebraic propagation rules based on the query expression.

Ceri and Widom [CW91] have proposed an incremental solution for the view maintenance problem of a subset of the SQL views that is based on production rules. Their approach supports sub-query nesting of one level only and does not support views with duplicates or views with aggregation. The solution in [CW91] is not algebraic and requires access to the source data for propagating all types of updates.

Gupta et al. [GMS93] have proposed a solution for maintaining SQL and Datalog views with duplicates. The solution in [GMS93] also requires access to source data and is not easy to generalize because it is procedural. Griffin and Libkin [GL95] have proposed an algebraic solution to maintain views with duplicates. They have emphasized the advantages of the algebraic solution over the algorithmic (procedural) solution. These advantages include the simplicity of extending the solution to more query operations, the ease of proving correctness of the solution, and the possibility of optimizing the maintenance expressions generated by the solution using the traditional query optimizer. The solution in [GL95] supports views with any number of $select$, $project$, $join$, $bag\ union$, and $monus$ algebra operations. The solution in [GL95] also supports views with aggregation where such aggregation is restricted to one aggregate operator that comes as the top-most operator in the view expression. Propagating updates based on the algebraic rules proposed in [GL95] can not be done using a simple application of the propagation rules as done in [BLT86]. This is because these propagation rules do not ensure minimality of the propagated updates, meaning that they might propagate unnecessary updates. For that, [GL95] have adopted recursive algorithms for propagating updates that employ rules to ensure minimality of the propagated updates. These algorithms compute insertion and deletion updates for each subexpression in the maintenance plan as source updates are propagated through these subexpressions.

Quass [Qua96] has extended the work in [GL95] to support view maintenance for general views with *aggregation*. The maintenance expressions obtained in [Qua96] are rather inefficient as they perform recomputation for the aggregate values affected by the update. Griffin and Kumar [GK98] have proposed a solution for maintaining outer join queries that follow the same algebraic framework of [GL95]. [GL95] does not consider aggregation. The algebraic propagation rules in [Qua96] and [GK98] may also generate non-minimal updates, like in [GL95]. Hence, similar recursive propagation algorithms as the ones used in [GL95] are used in [Qua96] and [GK98] to guarantee minimality of the propagated updates.

Mumick et al. [MQM97] have proposed a technique for maintaining views with only one aggregate operator on top of a SPJ expression. [MQM97] uses a technique called the summary table technique. In contrast to the solutions in [GL95, Qua96, GK98], that propagates insertion and deletion updates, the solution in [MQM97] computes a summary of all changes and then applies this summary of changes to the materialized view to refresh it.

In [MK00] the problem of making aggregate views self-maintainable by maintaining additional relations, called auxiliary views, is investigated. Palpanas et al. [PSCP02] have proposed an incremental maintenance algorithm that maintains views with aggregate functions that are not distributive over all update operations. They perform selective recomputation to maintain such views where only the set of affected groups is efficiently recomputed. The work in [PSCP02] supports only views with one aggregate function that comes at the top of the query expression.

Gupta and Mumick [GM05] have generalized the summary table technique proposed in [MQM97] into a technique called the change table technique that enables the maintenance of general views with aggregate and outer join expressions. This approach is shown to be more efficient than propagating inserts and deletes. The solution in [GM05] relies on a complex apply operator that varies in functionality depending on the operators in the query plan and the propagated updates. One main advantage of the work in [GM05] is that it makes aggregate operations and outer join operators self-maintainable.

Although XML views typically include relational-like operations, the solutions proposed above can not directly be used in maintaining XML views. This is due to many factors, as we have discussed in Section 1.2, including the flexibility and richness of XML data, the additional requirements of relational-like operations in XML views, and the nature of XML updates. Our proposed view maintenance solution is algebraic similar to some of the work presented above (e.g., [GL95, GM05]). Hence, it has the advantages associated with the algebraic approach as mentioned above. Yet, our propagation rules enable the maintenance of relational-like operations in the rich settings of XML views. Our solution is similar to [MQM97, GM05] in that it computes the net effect of propagating source updates at each maintenance subexpression instead of propagating inserts and delete tuples.

**View Maintenance in Object-relational and Object-oriented Databases.** To a lesser degree, view maintenance has been studied for object-relational and object-oriented views. Skld [Sk94] have proposed solution for maintaining object-relational views. His solution is an extension to the work in [BLT86] and supports only limited types of updates. Liu et al. [LVM00] have proposed an extension to the solution in [GMS93] for maintaining OR-SQL views.

Among the object-oriented view maintenance solutions [KR96, KR98, ZHK96, AP98, AFP03], only [AFP03] provides an algebraic solution to the problem of maintaining object-oriented views that follows the general relational algebraic framework we disused above and that we also follow in this work. [AFP03] is the first object-oriented view maintenance solution to produce incremental maintenance plans in an algebraic level. [AFP03] supports the standard object-oriented query language OQL, unlike [KR96, KR98, AP98], and supports a large class of object-oriented views, unlike [ZHK96, AP98]. The solution in [AFP03] enables many views to be self-maintainable on some updates, unlike solutions that requires access to source data on all updates (e.g., [KR98]) or solutions that materialize intermediate data (e.g., [ZHK96]). The solution in [AFP03] requires materialization of all OID-s of objects contributing to each object in the view. This is required to maintain the lineage between objects in the materialized view extents and the source object they are derived from. We avoid such need through the use of reproducible semantic identifiers.

**View Maintenance in Semi-structured and XML Databases.** Few proposals have addressed the problem of maintaining XML and semi-structured views. Zhuge and Garcia-Molina [ZGM98] have proposed a solution 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 by Abiteboul et al. [AMR$^+$98]. Unlike our work, [AMR$^+$98] considers only simple atomic update operations: insertion or deletion of an edge between existing objects, or the change of the value of an atomic object. Also they do not consider order. We handle bulk updates possibly of different types and at the same time support order. Quan et al. [QCR00] have proposed an efficient maintenance technique for materialized views over dynamic web data. [QCR00] is limited to XPath expressions and does not consider order. [QCR00] uses auxiliary data of size that depends on the source data size. Sawires et al. [STP$^+$05] have proposed a solution for maintaining a subset of XPath expressions. Their solution requires auxiliary data that depends on the expression size and the answer size and does not depend on the source data size.

Liefke and Davidson [LD00] have proposed a framework for defining and maintaining views over hierarchical semi-structured data. The view definition language (WHAX-QL) in [LD00] is a restricted variation of XML-QL. The work in [LD00] does not support explicit union operations and uses a special id mechanism that may generate keys with a deeply nested structures. [LD00] places restrictions on updating source data bound to some variables in the view. Other restrictions include supporting only simple nested query expressions, and no support for order. Their treatment to delete update operations is inefficient, as we have discussed in Section 6.1.

Bohannon et al. [BCF04] have proposed two solutions for incremental evaluation of ATGs, a formalism for schema-directed XML publishing. The first, the reduction approach, pushes most of the view maintenance work to the underlying RDBMS. This depends on combination of features not yet available even for the most advanced DBMSs. The second, the Bud-cut approach, performs most of the view maintenance work in the middleware. It requires several round-trips between the middle-ware and the DBMS resulting in high communication cost. In addition, maintenance queries sent to the DBMS may be large in number and complex. The solutions in [BCF04] does not support views defined on top of full fledged XML sources.

In [ESWDR02] we have proposed a solution for incremental maintenance of XQuery views. This solution did not provide an efficient solution for handling XML order. In [DESR03], we have extended our work [ESWDR02] to be the first technique for maintaining order-sensitive XML views. In both [ESWDR02, DESR03], updates are defined as primitives. Rules for propagating each update are defined for each query algebraic operator. [ESWDR02] and [DESR03] do not support bulk updates, and require materialization of portions of intermediate data. The propagation of updates [ESWDR02, DESR03] requires a special purpose execution. Our new framework proposed in this dissertation for solving the problem of maintaining XQuery views avoids the shortcoming above. This framework does not require intermediate result materializations and generates incremental maintenance plans in the same language used by the view.

## 10.2 Other Related Work

**Source XML node encoding.** Object identity is widely used in semi-structured databases [Lie99, LD00] and in object-oriented databases [FE01]. W3C recommends that each node in an XML document should have a node identifier [W3C05]. Some XML algebra operators perform functionalities like duplicate elimination using the node identifiers [W3C05]. Several techniques have been proposed for encoding order of XML documents. [TVB$^+$02] describes three order encoding methods: global, local and dewey encodings. In the global encoding method, each node is assigned a number that represents the node's absolute position in the document, while in the local encoding method each node is assigned a number that represents its relative position among its siblings. The dewey order encodes the full path from the root node to the current node. The dewey order is shown to outperform the other two on workloads composed of both queries and updates. These order encoding methods might require renumbering certain portions of the XML tree in the presence of updates. [DR03] proposed an order encoding for XML documents nodes (called *FlexKey*) which is based on dewey ids. This method avoids the problem of renumbering in the case of updates by using variable length byte strings instead of numbers. Another encoding technique, used in [AKJK$^+$02, JAKC$^+$02], associates a numeric $start$ and $end$ label with each data node in the XML document. The intervals between these labels are defined such that every descendant node has an interval that is strictly included in its ancestors' interval. One disadvantage of this method is that re-labeling of nodes might be required if a large number of insertions are taking place within the same small label range. In addition, it is not possible to derive directly the label of a parent (or an ancestor) of a node given only its label, unlike in the case of *Flexkey* [DR03] and Dewey [TVB$^+$02] encodings.

**XML Order.** Many solutions for XML data management use relational database technology [FK99, STH$^+$99, TVB$^+$02] as the underlying storage medium. Supporting the ordered nature of the XML data in the relational model context is an issue since order information is lost while converting from XML to the relational data representation [NLB$^+$01]. Many solutions for semi-structured data have been extended to support XML data [GMW99, Lie99]. These solutions do not support order requirements of XQuery expressions.

Concurrently with these efforts to exploit existing database technologies, native XML storage manager systems [DR03, KM00] have also been proposed. An advantage of such native storage is that XML documents may be clustered in physical XML document order, thus facilitating efficient children/descendant access. Such tree navigation is very frequent in XML query processing [JAKC$^+$02].

The Agora system [MFK01a], which stores XML in relational tables, provides support for handling order-sensitive XQuery expressions. XQuery queries are first normalized, then translated and rewritten into SQL queries to be executed over the relational tables. However, this solution is limited to XQuery queries that semantically match SQL and can be translated and rewritten into SQL. Additionally, order handling is an expensive process where an XQuery is translated into many SQL queries requiring several passes and materializing of intermediate XML results.

Shanmugasundaram et al. [SSB$^+$00, SSB$^+$01] have introduced mechanisms to publish relational data and object-relational data as XML documents. These solutions provide support for document order. Tatarinov et al. [TVB$^+$02] have proposed a solution for supporting ordered XML query processing using the relational database technology. This solution mainly focuses on handling order on XPath expressions, and provides support for some XQuery order-based functionalities like *before* and *after* operators and the *range* predicate. The work in [TVB$^+$02] focuses on document order and does not handle different types of order imposed by XQuery expressions. Timber [JAKC$^+$02], a native XML data management system, provides support for document order and query order. However, to preserve order, sorting for some of the intermediate results appears to be required during execution [JAKC$^+$02]. The order handling strategy in Timber is built on top of the node $start$-$end$ labeling described above. Hence, it suffers from the disadvantages of this labeling technique, discussed above. [FLSW03] introduces a solution for maintaining source XML document order that works in both a static and dynamic database environment. However, re-labeling of nodes might be required in some cases.

Our proposed order approach (Chapter 3) supports different types of XQuery order, namely document order and order imposed by the query itself in a variety of ways. A key point in our solution is that the order is implicitly encoded in the node identifier and in the intermediate result schema in a way that allows the migration of intermediate results from ordered bag semantics into non-ordered bag semantics. Unlike in [JAKC$^+$02] our operators no longer need to be aware of the order of data they process. Also we do not need to incorporate any sorting operations for intermediate results. Our operators becomes distributive with respect to order. This opens up more optimization opportunities and allows for efficient incremental view maintenance.

**Incrementally Assembling XML Results.** In the apply phase of view maintenance, propagated updates are used to refresh the materialized views. This involves determining how to correctly merge propagated updates with the materialized views. This problem is more challenging in the context of object-oriented and semi-structured data models than in the context of the flat relational data model. Some solutions [AFP03] solves this problem by materializing auxiliary data that defines the lineage between view data and the source data it is derived from.

Some view maintenance solutions [BCF04, LD00] have avoided the materialization of auxiliary data through the use of mechanisms for generating reproducible identifiers for the view objects. For example, the work proposed in [LD00] for maintaining semi-structured views annotates edges in the processed trees with special keys that can be used in the merging 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 a limitation on updating source values used in constructing the keys.

Other solutions [PAGM96, BCF04] use 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 [Mai86] to produce object identifiers and later were used in many integration and mediation systems [PAGM96, PVM$^+$02]. 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. [PAGM96] 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 [PAGM96] supports only simple views. It requires semantic identifiers to be defined as part of the mediator specification process by the view definer.

To the best of our knowledge, no Skolem function solution introduced in the literature so far supports incremental fusion of the class of XML views that we consider. In particular, none of them supports 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 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. [IHW02] have proposed a solution for combining and restructuring XML views over streaming XML data by adding extra attributes to the intermediate tuples that describe the structure of the returned result. 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. [FLBC02] 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 [FLBC02] 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.

**Lineage Tracing.** In addition to the areas discussed above, our work relates to the problem of tracing lineage (derivations) of objects in view results [BB99, RS98, CW01, FP03, BKT01] Buneman et al. [BKT01] have argued that the view maintenance problem and expressing the why-provenance of views (one of the two types of the data lineage problem) are loosely related. They have also argued that the view maintenance problem is harder than expressing the why-provenance of views because the why-provenance does not account for additions to the source and does not address the issue of how to reconstruct the view on source updates. As a future work, we plan to investigate how we can utilize or semantic identifier solution to provide a lineage tracing solution for data in XML views.

# Chapter 11

# Conclusions and Future Work

## 11.1 Summary and Contributions

The broader contribution of this dissertation is a comprehensive framework for maintaining XQuery views. Our solution produces incremental maintenance plans in the same algebraic language used to construct the view extent, making it easy to incorporate our view maintenance solution in any XML query engine. This should facilitate the adoption of our XML view maintenance solution within future commercial XML engines.

Our solution supports an expressive class of XQuery views including XPath expressions, FLWOR expressions, and element constructors. This includes also support for nested queries, order-sensitive queries, and general queries involving grouping and join operations. It also avoids intermediate result materialization and makes most of the views self-maintainable.

We first propose an efficient solution for maintaining order in XML query processing and view maintenance. Our solution supports XML source document order and all types of XQuery imposed order. Our order solution does not require any intermediate result materialization or sorting.

We also propose a solution to the problem of incrementally constructing XML views. Our solution utilizes special semantic identifiers to perform id-based fusion of XML fragments. Our semantic identifiers have two important properties: (i) they are reproducible and (ii) they compactly encode lineage and order semantics of XML nodes. As a result, the semantic identifiers we generate allow a large class of XML views to be distributive on insert updates. Our semantic identifiers do not require manual specification of how identifiers for different queries are to be generated nor do they require materialization of intermediate data.

We also propose a technique for modeling and validating source XML updates that propagates a structure that batches bulk XML updates possibly of different types. The batch update structure encodes only relevant updates using minimum yet sufficient information for propagation.

To support view maintenance for delete updates and for views with aggregate functions, we propose a counting algorithm that enables tracing derivations of nodes in XML view extents. A unique feature of our solution is that it enables the deletion of XML nodes from the view extent without requiring the knowledge of all nodes in its subtree. Hence, we may delete an entire fragment from the view extent by simply disconnecting the root of that fragment from the view extent. Using our semantic id solution and our counting solution we enable a large class of XQuery views to be distributive not only under insert updates but also under delete and value change updates.

We also propose an algebraic update propagation solution that derives incremental maintenance plans for the query view definition. When such plans are evaluated over the source updates, they compute delta changes to be applied to the materialized view maintenance. One main advantage of this approach is that the evaluation of incremental maintenance plans does not require a special purpose execution. Instead, the evaluation of such plans can be performed using the current XML query engines after adding semantic id and counting functionality to it. This is similar to what commercial relational systems do to support incremental view maintenance of relational view extents. Another main advantage of our solution is that most of the incremental maintenance plans we generate are self-maintainable. Hence, they only use the update to compute the delta effect on the materialized view extent. Incremental maintenance plans that are not self-maintainable are guaranteed to be at least distributive. Hence, they incrementally compute the delta effect on the materialized view extent without recomputing it.

We finally propose a mechanism for refreshing materialized XML views by applying propagated delta updates. Our apply algorithm uses the *Deep Union* operator. It ensures that the refreshed view extent is equal to the one that we would get if we were to recompute the view over the updated sources. Our apply algorithm provides efficient treatment for delete updates, were entire XML fragments can be deleted when their root node is deleted.

We have implemented a prototype of our solution over the Rainbow XML query engine, developed at WPI. We have performed extensive experimental studies for our proposed solution to verify its practicality and efficiency. We have measured the overhead that comes with required extension to the regular query engine to support this view maintenance feature. In particular, supporting order-aware processing, reproducible semantic node identifiers, and count annotations. Our results confirm that such extension comes with very small overhead to the query execution time. We have also measured the cost of maintaining queries incrementally on different types of updates.

Our experiments also show that maintaining views incrementally using our solution is more efficient than recomputing views even for large sizes of updates. Experiments confirm that our solution provides an efficient treatment for delete updates that delete entire fragments from the view extent.

## 11.2 Future Work

There are many interesting and open research issues beyond the issues addressed in this dissertation.

One possible future direction is to increase the expressiveness of views. This includes for example non-monotonic views where a source insert update might cause a propagation of delete update(s) or a delete update might cause a propagation of insert update(s). Such views are not distributive and we do not consider them in this work.

One other possible future direction is to study query optimization for order-sensitive XML queries to achieve better overall performance and to reduce final sorting.

Another possible future direction is to study special optimizations of incremental maintenance plans and exploiting materialization as part of query optimization. Also to investigate the possible use of XML schema for achieving more efficient view maintenance.

Another possible future direction is to develop a solution for making general views with join and grouping operations self-maintainable. Gupta and Mumick [GM05] have studied this class of views in the relational context where grouping is always accompanied by aggregation. We would like to build on that solution to enable the XML views we support to be self-maintainable.

One other possible future direction is to exploit semantic identifiers in other domains. For example semantic identifiers might facilitates updating of XML views, tracing derivations of XML view nodes, and XML stream query processing. In [ESRM05b], we have performed initial work that exploits semantic identifiers to enable non-blocking XML stream query processing.

Another future direction is to develop a cost model that can be used in deciding between maintaining views incrementally and re-computing them.

# Bibliography

[AFP03] 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.

[AKJK$^+$02] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *ICDE*, pages 141–152, Feb 2002.

[AMR$^+$98] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. In *VLDB*, pages 38–49, August 1998.

[AP98] R. Alhajj and F. Polat. Incremental View Maintenance in Object-Oriented Databases. *Data Base for Advances in Information Systems*, 29(3):52–64, 1998.

[BB99] P. Bernstein and T. Bergstraesser. Meta-data support for data transformations using microsoft repository. *IEEE Data Engineering Bulletin*, 22(1):9–14, March 1999.

[BCF04] P. Bohannon, B. Choi, and W. Fan. Incremental evaluation of schema-directed XML publishing. In *SIGMOD*, pages 503–514, 2004.

[BDT99] 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*, pages 114–123, Jan 1999.

[BKT01] P. Buneman, S. Khanna, and W.-C. Tan. Why and Where: a characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[BLT86] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *SIGMOD*, pages 61–71, 1986.

[BT99] 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.

[CJLP03] Z. Chen, H. Jagadish, L. V. Lakshmanan, and S. Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *VLDB*, pages 237–248, 2003.

[CW91] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *VLDB*, pages 577–589, 1991.

[CW01] Y. Cui and J. Widom. Lineage Tracing for General Data Warehouse Transformations. In *VLDB Journal*, pages 471–480, 2001.

[DESR03] K. Dimitrova, M. El-Sayed, and E. A. Rundensteiner. Order-sensitive View Maintenance of Materialized XQuery Views. In *ER*, pages 144–157, Oct. 2003.

[DFF$^+$99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(11–16):1155–1169, 1999.

[DR03] K. Deschler and E. Rundensteiner. MASS: A Multi-Axis Storage Structure for Large XML Documents. In *CIKM*, pages 520–523, Nov 2003.

[ESDR03] M. El-Sayed, K. Dimitrova, and E. A. Rundensteiner. Efficiently Supporting Order in XML Query Processing. In *WIDM*, pages 147 – 154, November 2003.

[ESDR05] M. El-Sayed, K. Dimitrova, and E. A. Rundensteiner. Efficiently Supporting Order in XML Query Processing. *Data and Knowledge Engineering*, 54(3), September 2005. to appear.

[ESRM05a] M. El-Sayed, E. A. Rundensteiner, and M. Mani. Incremental Fusion of XML Fragments through Semantic Identifiers. In *IDEAS*, 2005. to appear.

[ESRM05b] M. El-Sayed, E. A. Rundensteiner, and M. Mani. Online Processing for Streams of Arbitrary Disassembled Out-of-order XML Fragments. Technical Report WPI-CS-TR-05-08, Worcester Polytechnic Institute, April 2005.

[ESWDR02] 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.

[FE01] L. Fegaras and R. Elmasri. Query Engine for Web-Accessible XML data. In *The VLDB Journal*, pages 251–260, 2001.

[FK99] D. Florescu and D. Kossman. Storing and Querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 11(3):27–34, 1999.

[FLBC02] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. Query Processing of Streamed XML Data. In *CIKM*, pages 126 – 133, 2002.

[FLSW03] D. K. Fisher, F. Lam, W. M. Shui, and R. K. Wong. Efficient Ordering for XML Data. In *CIKM*, pages 350–357, Nov 2003.

[FP03] H. Fan and A. Poulovassilis. Tracing data lineage using schema transformation pathways. *Knowledge Transformation for the Semantic Web*, IOS Press:64–79, 2003.

[GJM97] A. Gupta, H. V. Jagadish, and I. S. Mumick. Maintenance and self maintenance of outer-join views. In *The Third International Workshop on Next Generation Information Technologies and Systems (NJITS)*, 1997.

[GK98] T. Griffin and B. Kumar. Algebraic change propagation for semijoin and outerjoin queries. *SIGMOD Records*, 27(3):22–27, 1998.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD*, pages 328–339, 1995.

[GM95] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. In *IEEE Bulletin of the Technical Committee on Data Engineering, 18(2)*, pages 3–18, June 1995.

[GM05] H. Gupta and I. Mumick. Incremental maintenance of aggregate and outerjoin expressions. *Information Systems*, 2005. to appear.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *SIGMOD*, pages 157–166, 1993.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *WebDB Proceedings*, pages 25–30, 1999.

[IHW02] Z. G. Ives, A. Halevy, and D. Weld. An XML query engine for network-bound data. *The VLDB Journal*, 11 (4):402–402, December 2002.

[Int99] International Organization for Standardization (ISO) & American National Standards Institute (ANSI). ISO International Standard: Database Language SQL - Part 2: Foundation (SQL/Foundation). In *ANSI/ISO/IEC 9075-2:99*, September 1999.

[JAKC$^+$02] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A Native XML Database. 11(4):274–291, 2002.

[KM00] C.-C. Kanne and G. Moerkotte. Efficient Storage of XML Data. In *ICDE*, page 198, 2000.

[KR96] H. A. Kuno and E. A. Rundensteiner. Using Object-Oriented Principles to Optimize Update Propagation to Materialized Views. In *ICDE*, pages 310–317, 1996.

[KR98] 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.

[LD00] H. Liefke and S. B. Davidson. View Maintenance for Hierarchical Semistructured Data. In *DWKD*, pages 114–125, 2000.

[Lie99] H. Liefke. Horizontal Query Optimization on Ordered Semistructured Data. In *WebDB Informal Proceedings*, pages 61–66, 1999.

[LSPC00] W. Lehner, R. Sidle, H. Pirahesh, and R. W. Cochrane. Maintenance of cube automatic summary tables. In *SIGMOD*, pages 512–513, 2000.

[LVM00] J. Liu, M. W. Vincent, and M. K. Mohania. Maintaining Views in Object-relational Databases. In *CIKM*, pages 102–109, 2000.

[Mai86] D. Maier. A logic for objects. In *Workshop on Foundations of Deductive Database and Logic Programming, Washington, DC, USA*, pages 6–26, 1986.

[MFK01a] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *VLDB, Roma, Italy*, pages 241–250, Sept. 2001.

[MFK01b] I. Manolescu, D. Florescu, and D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. In *VLDB*, pages 241–250, 2001.

[MHM04] N. May, S. Helmer, and G. Moerkotte. Nested Queries and Quantifiers in an Ordered Context. In *ICDE*, pages 239–250, 2004.

[MK00] M. K. Mohania and Y. Kambayashi. Making Aggregate Views Self-maintainable. *Data Knowledge Engineering*, 32(1):87–109, 2000.

[MQM97] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *SIGMOD*, pages 100–111, 1997.

[NLB$^+$01] U. Nambiar, Z. Lacroix, S. Bressan, M. L. Lee, and Y. G. Li. XML Benchmarks Put to the Test. In *the Third International Conference on Information Integration and Web-Based Applications and Services (IIWAS)*, September 2001.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object Fusion in Mediator Systems. In *VLDB*, pages 413–424, 1996.

[PSCP02] T. Palpanus, R. Sidle, R. Cochrane, and H. Pirahesh. Incremental Maintenance for Non-Distributive Aggregate Functions. In *VLDB*, 2002.

[PVM$^+$02] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, and R. Fagin. Translating Web Data. In *VLDB*, pages 598–609, 2002.

[QCR00] L. P. Quan, L. Chen, and E. A. Rundensteiner. Argos: Efficient Refresh in an XQL-Based Web Caching System. In *WebDB*, pages 23–28, May 2000.

[Qua96] D. Quass. Maintenance Expressions for Views with Aggregation. In *SIGMOD*, pages 110–118, 1996.

[RS98] A. Rosenthal and E. Sciore. Propagating integrity information among interrelated databases. In *the Second Working Conference on Integrity and Internal Control in Information Systems*, pages 5–18, 1998.

[Sah01] A. Sahuguet. Kweelt: More than just "yet another framework to query XML!". In *Demo Session Proceedings of SIGMOD'01*, page 602, 2001.

[Sk94] M. Skld. Active Rules based on Object Relational Queries-Efficient Change Monitoring Techniques. PhD Thesis, School of Engineering, Linkoping University, 1994.

[SSB$^+$00] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. In *VLDB*, pages 65–76, 2000.

[SSB$^+$01] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. *VLDB Journal*, 10(2–3):133–154, 2001.

[STH$^+$99] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*, pages 302–314, 1999.

[STP$^+$05] A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S. Candan. Incremental Maintenance of Path-Expression Views. In *SIGMOD*, 2005.

[Suc98] D. Suciu. An Overview of Semistructured Data. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 29(4):28–38, 1998.

[SWK$^+$02] 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, 2002.

[TIHW01] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD*, pages 413–424, 2001.

[TVB$^+$02] 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.

[W3C98] W3C. XML$^{TM}$ . http://www.w3.org/XML, 1998.

[W3C99] W3C. XML Path Language (XPath) Version 1.0. W3C Recommendation. http://www.w3.org/TR/xpath.html, November 1999.

[W3C05] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, February 2005.

[WRM05] S. Wang, E. A. Rundensteiner, and M. Mani. Optimization of Nested XQuery Expressions with Orderby Clauses. In *XSDM'05*, 2005. to appear.

[Zea03] X. Zhang and et al. Rainbow: Multi-XQuery Optimization Using Materialized XML Views. In *SIGMOD Demo*, page 671, 2003.

[ZGM98] Y. Zhuge and H. Garcia-Molina. Graph Structured Views and Their Incremental Maintenance. In *ICDE*, pages 116–125, February 1998.

[ZGMHW95] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, pages 316–327, 1995.

[ZHK96] G. Zhou, R. Hull, and R. King. Generating Data Integration Mediators that Use Materialization. *Journal of Intelligent Information Systems*, 6(2/3):199–221, 1996.

[ZPR02] 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.