WPI-CS-TR-03-03 Feb 2003

An Algebraic Approach for Incremental Maintenance of Materialized XQuery Views

by

Maged EL-Sayed
Ling Wang
Luping Ding
Elke A. Rundensteiner

An Algebraic Approach for Incremental Maintenance of Materialized XQuery Views

Maged EL-Sayed, Ling Wang, Luping Ding, and Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute Worcester, MA 01609
(maged|lingw|lisading|rundenst)@cs.wpi.edu

Abstract

XML has become an important medium for representing and exchanging data over the Internet. Data sources, including structural and semi-structural sources, often export XML views over base data, and then materialize the view by storing its XML query result to provide faster data access. Upon change to the base data, it is typically more efficient to maintain a view by incrementally propagating the delta changes to the view than by re-computing it from scratch. Techniques for the incremental maintenance of relational views have been extensively studied in the literature. However, the maintenance of views created using XQuery is as of now unexplored.

In this paper we propose an algebraic approach for incremental XQuery view maintenance. In our approach, an update to the XML source is transformed into a set of well defined update primitives which are propagated through the XML algebra tree. This algebraic update propagation process generates incremental update primitives to be applied to the result view. Our update propagation strategy based on the XAT XML algebra and the propagation rules for individual algebra operators are given. We briefly discuss our XQuery view maintenance system implementation, and present our experiments that confirm that incremental view maintenance is indeed faster than re-computation.

Keywords: XML, XML Query Algebra, Incremental View Maintenance.

1 Introduction

Database views have been recognized as an important technology for data warehousing, information integration and interoperability. As XML is becoming the standard for data exchange over the Internet, publishing data from different data sources into XML views has also become increasingly popular in recent years [3, 10]. Views are frequently materialized to speed up querying [5], which is critical for applications where the data is remote or the response time is critical. However materialized views need to be maintained upon updates to the base data in order to keep them consistent with the base data. Techniques for view maintenance have been studied extensively in the literature for relational and object-oriented databases [7]. Incremental view maintenance, that is, instead of re-computing the whole view, just computing the delta change to a view in response to a change of the base data, has been shown to be an effective solution [8, 9].

View maintenance of XML views has received little attention so far, as the XML community has primarily focused on topics such as publishing XML over relational databases [3, 6, 12], designing relational storage for XML documents [2], and optimizing XML query processing [11]. We note that, unlike relational data, XML data corresponds to ordered, nested hierarchical structures. Furthermore XML views allow for arbitrarily complex result re-structuring. As a consequence, incremental maintenance of XML views is likely to be more complex than that of their relational counterparts.

One recent work [1] has proposed an incremental maintenance algorithm for materialized views over semi-structured data, based on the graph-based data model OEM and the query language Lorel. Our work is different, since it is based on the XQuery language, a full-featured SQL-style XML query language that has become a World Wide Web Consortium (W3C) recommendation. Other recent work [13] describes an extension of the XQuery language to support XML updating, and implements these operations using relational technology. In [4], an efficient maintenance technique for materialized views over dynamic web data was proposed, but using XQL, thus for views limited to XPath expressions.

While many systems employ XQuery views [3, 10, 14], to the best of our knowledge, no solution has been proposed as of now to handle maintenance of materialized XQuery views. In this paper we thus propose the first solution for incremental XQuery view maintenance. We have designed an algebraic solution approach based on the XAT XML algebra [14]. In our approach, an update to the XML source is transformed into a set of well defined update primitives to be applied to the view. We describe the overall update propagation strategy of our XQuery view maintainer, as well as the propagation rules for individual algebra operators. In summary, our paper makes the following contributions: (1) We propose a set of update primitives for updating XML documents. (2) We present a general strategy for XML view maintenance, based on an algebraic approach for incremental update rewriting. (3) We define detailed propagation rules for the operators in the XAT algebra. (4) We have implemented the view maintainer as an extension of the Rainbow system [14], which serves as proof of feasibility. (5) We have concluded an experimental study illustrating that our update propagator is indeed more efficient than full re-computation. Our algebraic approach is a general solution for XML view maintenance, not only for views published over relational but also over XML data. Also, the algebraic approach opens the opportunity for optimization in the future.

The rest of this paper is organized as follows. Section 2 describes background. In Section 3, we present our XML update propagation strategy, including the update primitives and the detailed update propagation rules. Section 4 shows our experiments results. Finally, Section 5 concludes the paper.

2 Background

XML Fundamentals. Figures 1 and 2 show our running example of two XML documents: bib.xml models books where each book has a year attribute and title, author, and publisher sub-elements; and reviews.xml models book reviews where each review has title and review sub-elements.

```
<bib>
    <book year="1992">
        <title>Advanced Programming in the
                Unix environment</title>
        <author>Darcy Gerbarg</author>
        <publisher>Addison-Wesley</publisher>
    </book>
    <book year="2000">
        <title>Data on the Web</title>
        <author>Serge Abiteboul</author>
        <publisher>Morgan Kaufmann Publishers</publisher>
    </book>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author>W. Stevens</author>
        <publisher>Addison-Wesley</publisher>
    </book>
</bib>
```

Figure 1: Example XML document bib.xml.

```
<reviews>
    <entry>
        <title>Data on the Web</title>
        <review> A very good discussion of semi-structured
                database  systems and XML</review>
    </entry>
    <entry>
        <title>Advanced Programming in the
                Unix environment</title>
        <review>A clear and detailed discussion of
                UNIX programming</review>
    </entry>
     <entry>
        <title>TCP/IP Illustrated</title>
        <review>One of the best books on TCP/IP</review>
    </entry>
</reviews>
```

Figure 2: Example XML document reviews.xml.

An *XML view* is a derived XML fragment defined by an XML query, in our case, using the XQuery language. Figure 3 shows the view query that retrieves books written by "Peter Buneman" and their respective reviews. A *materialized XML view* refers to the stored derived XML data. Figure 4 shows the materialized XML view generated by applying the view query in Figure 3 to the XML documents in Figures 1 and 2.

```
<Result>
    FOR $a IN document("bib.xml")/book,
        $b IN document("reviews.xml")/entry
    WHERE $a/title = $b/title and
          $a/publisher = "Morgan Kaufmann Publishers"
    RETURN
      <Book_Review>
            $a/title, $b/review
      </Book_Review>
</Result>
```

Figure 3: View query expressed by XQuery.

```
<Result>
    <Book_Review>
        <title>Data on the Web</title>
        <review> A very good discussion of semi
         -structured database  systems and XML</review>
    </Book_Review>
</Result>
```

Figure 4: Materialized view generated by applying view query in Figure 3 to XML documents in Figures 1 and 2.

XML Algebra XAT. We translate our view queries from the XQuery format into algebra expressions based on the XML algebra called XAT [14]. XAT has a set of well-defined algebra operators called XAT operators. The input and output of each XAT operator are XAT tables as defined below.

An *XAT table* is an order-sensitive table with k columns $N = (n_1, n_2, ..., n_k)$ where each column name n_i (1 $\leq$ i $\leq$ k) can be either a variable binding from the user-specified XQuery, e.g., $b, or an internally generated variable, e.g., col_1. The XAT table has an associated sequence of tuples $T = (t_1, t_2, .., t_p)$, where each tuple t_j (1 $\leq$ j $\leq$ p) is a vector of k cells $t_j = (c_{1j}, c_{2j}, ..., c_{kj})$. Each cell c_{ij} can be: (1) an atomic value; (2) a node including XML Element, XML Document, and XML Attribute; or (3) an ordered collection of any mixture of (1) and (2).

Table 1 in Appendix A lists a subset of XAT operators relevant to the example used in this paper, for the full details about the XAT algebra the reader is referred to [14]. Figure 5 shows the XAT algebra tree for the example query in

Figure 3. The semantics of operators in Figure 5 can best be understood by viewing the intermediate XAT table results that would be produced by each operator during execution as depicted in Figure 8 [1].

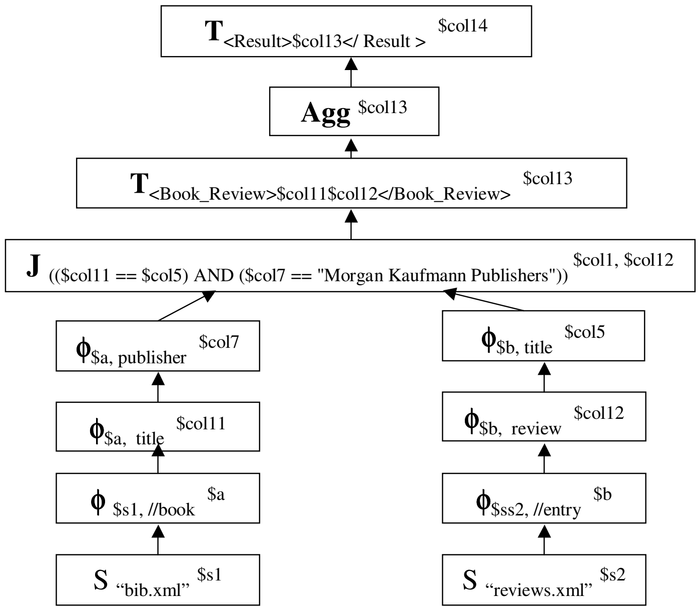

T <Result>$col13</ Result > $col14
Agg $col13
T <Book_Review>$col11$col12</Book_Review> $col13
J (($col11 == $col5) AND ($col7 == "Morgan Kaufmann Publishers")) $col1, $col12
φ $a, publisher $col7
φ $a, title $col11
φ $s1, //book $a
S "bib.xml" $s1
φ $b, title $col5
φ $b, review $col12
φ $ss2, //entry $b
S "reviews.xml" $s2

Figure 5: XAT algebra tree of the view query in Figure 3.

3 XML Update Propagation Strategy

3.1 Auxiliary Information for Maintenance

In our update propagation approach, we assume that our algebra operators may have access to their materialized input and/or output XAT tables, depending on the type of operator. Besides the XAT table defined in Section 2.2, our update propagator will also maintain additional auxiliary information to aid the maintenance process. In particular, we keep track of lineage between derived tuples. We generate unique IDs for tuples in the XAT tables, and each tuple keeps track of tuples it was derived from using ID references. Lastly, for the operators which will create collections of XML fragments (most notably the Aggregate operator) we maintain counts for each tuple in the operator's output XAT table. This auxiliary information will help to keep track of different fragments of the XML document stored in the XAT tables.

We now associate the following auxiliary information with a given XAT table T. For each tuple $t_j \in T$, $id(t_j)$ denotes a unique system-generated ID for t_j. Furthermore, given an XAT operator *op* with *n* input XAT tables T_l^{IN}

[1] Please ignore the update-propagation specific parts in Figure 8, namely the shaded parts in all XAT tables above the Join operator, and the boxes on the left side of the figure. Also for space reasons we could not show the full contents of the intermediate XAT table. Thus we use abbreviations, where each element is identified by a string that consists of the first letter of its tag name plus a number representing the global order of this element in the XML documents, for example, we refer to the second book as b2, the title of third book as t3, and so on. Some of the columns in the intermediate XAT tables were not carried from the input XAT table of an operator to its output XAT table. This process is called XAT Schema Cleanup [14], where unused columns are removed for optimization purposes.

($1 \leq l \leq$ n) and one output XAT table T^{OUT}, for each tuple $t_j \in T^{OUT}$, $derivedFrom(t_j, l)$={id_k | id_k is the ID of a tuple in T_l^{IN} from which t_j was derived}. Optionally, tc(t_j) denotes the count of the fragments in a certain column for the tuple t_j in T^{OUT} that is constructed as a result of executing the XAT operator *op*, for $op =$ Aggregate or XML Union.

XML Entry Point Notation. To access an element or an attribute in an XML fragment, we use the entry point notation **EPN** denoted as RF:EP, where RF is the relative forward path and EP is the entry point path. *EP* is an atomic value that represents either an **element** denoting the root element of an XML fragment, or an **order** number denoting the order of an XML fragment in an XML fragment collection. The relative forward path (*RF*) represents the full path to a specific element or attribute starting from *EP*. *RF* is a sequence of items ($rf_1, rf_2, .., rf_m$), each of which can be the name of an element or an attribute, optionally with an order number attached. For example, in the XML file in Figure 1, to access the first author in the second book from entry point bib we write book[2].author[1]:bib. The sequence can also include a fragment order without an element name.

3.2 Update Primitives

Updates to an XML source document can ultimately be transformed into a sequence of primitive update operations called $Update\ Primitives$ that are applied to the materialized view. We define two sets of update primitives *up*. Below the parameter *pos* corresponds to an EPN that is used in update primitives to access elements or attributes to be updated. We denote RF of pos by *pos.RF* =($urf_1, urf_2, .., urf_n$).

Update Primitives for Updating XML Documents (xup): The six update primitives below operate on XML documents directly.

- **AddAtt***(a, v, pos)*: Add a new attribute *a* with value *v* at position *pos*.
- **DeleteAtt***(a, pos)*: Delete attribute *a* at position *pos*.
- **ChangeAtt***(a, v, pos)*: Change the value of attribute *a* at position *pos* to the new value *v*.
- **AddEle***(e, pos)*: Add a new element *e* at position *pos*, where element *e* can be a singleton element or have nested structure.
- **DeleteEle***(pos)*: Delete the element at position *pos*.
- **ChangeEle***(e, pos)*: Change the element at position *pos* into new element *e*.

Update Primitives for Intermediate Update Propagation (iup): However, given that intermediate results in the XAT algebra tree correspond to XAT tables, i.e., they contain collections of fragments, we need to have a way to specify the location of the XML fragment to be updated in XAT tables. Thus we introduce the following three update primitives for updating intermediate XAT tables.

- **InsertTuple***(t, id, o)*: Insert a new tuple *t* with ID *id* into XAT table at tuple position *o*.

- **DeleteTuple***(id)*: Delete tuple with ID *id* from the XAT table.
- **ChangeTuple***(xup, ucol, id)*: Change the tuple in the XAT table by applying the update primitive *xup* to the cell in column *ucol* with tuple id *id*, where *xup* is one of the six XML update primitives described above.

3.3 Overall Propagation Strategy

Each node *n* in the XAT algebra tree has knowledge about the XAT operator *n.op* associated with it, input XAT tables $n.T_l^{IN}$, output XAT table $n.T^{OUT}$ and the auxiliary information defined in Section 3.1. We assume that the operator associated with the leaf node receives one update to the XML document at a time, and propagates it as a sequence of update primitives to its parent operator. All other operators in the algebra tree will get a sequence of update primitives from their children nodes, rewrite each of them into a sequence of zero or more new update primitives and put the new ones all together into a new sequence to be propagated upwards. They also refresh their output XAT table if materialized. We use $T_l^{IN}.op \rightarrow up$ to represent that an update primitive *up* is obtained from a certain child (XAT table T_l^{IN} of child operator *op* with index *l*).

The update propagation starts from the bottom of the algebra tree (generally, a leaf is a Source operator) triggered by an update to one of the source XML documents. The Source operator accepts an update of any of the six basic update types *xup*s defined above, and encapsulates such update into zero or more update primitives of type *iup*. The Source operator takes the input update primitive *iup*. and encapsulates it into a ChangeTuple update primitive specifying which column and tuple position to perform the update on the output XAT table. All other operators take as input any of the three XAT-specific update primitives *iup*, and generate sequences of update primitives *iup*, The update primitives are then propagated through the tree up to the topmost operator. At the top, the propagated update primitives will be applied to the materialized XML view to refresh it. The overall update propagation strategy for our system is shown in Figure 6.

```
function propagateUpdate (Node $n$, Update $up$)
  Sequence $r \leftarrow \emptyset$, $s \leftarrow \emptyset$
  if ($n$ is leaf)
    $r \leftarrow n.op.operatorPropagate\ (up)$
  else
    for (all children $ch_i$ of $n$)
      $s \leftarrow s + propagateUpdate\ (ch_i, up)$
    for (all updates $up_i$ in $s$)
      $r \leftarrow r + n.op.\ operatorPropagate\ (up_i)$
  return $r$
```

Figure 6: View maintenance algorithm.

```
<Result>
    <Book_Review>
        <title>Advanced Programming in the
              Unix environment</title>
        <review>A clear and detailed discussion
                of UNIX programming</review>
    </Book_Review>
    <Book_Review>
        <title>Data on the Web</title>
        <review> A very good discussion of semi-
        structured database systems and XML</review>
    </Book_Review>
</Result>
```

Figure 7: View after update.

3.4 Update Propagation through XAT Operators

We now discuss the propagation rules for the XAT operators used in our running example to give you a flavor of this process. For this we made use of the following functions.

- $xtab$.getTuple(id): return tuple with ID id in XAT table $xtab$;
- $xtab$.getCell(col, id): return the content of column col of the tuple with ID id in XAT table $xtab$;
- $xtab$.getInsPos (o) / $xtab$.getInsPos (id): return the position into which a tuple should be inserted into the output XAT table $xtab$ given the order o or the tuple ID id of that tuple in the input XAT table. The functionality of this function depends on the operator that is invoking it. For example if the join operator invokes this function, the order generated by it is following the order of left input XAT table as major order and the right input XAT table as minor order.
- $xtab$.getId(id, $1/2$): return the ID id(t_j) of a tuple in the output XAT table $xtab$ given the ID id in derivedFrom(t_j) from the input XAT table. For Join operator 1 represents LEFT input XAT table, while 2 represents RIGHT input XAT table).
- t.getCell(col): return the content of the cell of tuple t positioned by column col.
- applyPattern(pos, p): is used by the tagger operator. Return a modified *EPN* by applying the tagging pattern p used in the Tagger operator to the given EPN pos.

Tables 3 - 6 in Appendix A show the propagation rules for Source, Navigate, Theta Join, Aggregate and Tagger operators respectively. For simplicity we assume that operators consume the columns they use as input arguments and by default do not again place them into their output table. This means that we do not need to describe how the input update is now also applied to the output XAT table in the same manner.

3.5 Update Propagation Example

Given our example view query tree depicted in Figure 8, let us assume that the following update primitive is applied to the bib.xml document:

ChangeEle (<publisher>Morgan Kaufmann Publishers</publisher>,book[1].publisher[1]:bib).

This update changes the publisher element of the first book element to be:

<publisher>Morgan Kaufmann Publishers</publisher>.

Figure 7 shows the updated materialized view we are expecting.

At the bottom of Figure 8, this update (1) is applied to the input XML document bib.xml, changing the publisher element in the specified path to the new publisher element. The update is then passed to the Source operator $S^{\$s1}_{bib.xml}$. This operator generates update (2) by encapsulating the input XML update primitive *up* (1) into a ChangeTuple update with *up* as its *xup* argument, the operator output column *$s1* as its *ucol* argument, and 1 as its *id* argument. The new update (2) is propagated to the parent operator $\phi^{\$a}_{\$s1,book}$ that generates a new update primitive (3). Update (3) has a new *col* argument *$a* that is derived from the *col'* argument in the navigate operator itself, and a new *pos* that is relative to the extracted element book in the new column *$a* in the output XAT table. The argument *id* now has the id value of 1 that will identify it in the output XAT table. The detailed construction of the new update primitive can be found in Table 2.

Next the node $\phi^{\$col11}_{\$a1,title}$ consumes update (3) and generates update (4). *pos* does not change since the new update is working on the same column. Also the *id* in update (4) is now reflecting the id of the target tuple in the output XAT table. Next, update (4) is consumed by the node $\phi^{\$col7}_{\$a1,publisher}$ to generate update (5). The clean-up process here removes column $a, as it will not be used later on. This eliminates the need to propagate an update into that column. And since this Navigate operator extracts an element affected by the update statement (publisher), a new update (5) will be generated for that element with *ucol= $col11*. Update (5) has an *pos* = *Null:publisher* (or publisher in short) since the element to be updated is at the root in the update column. Next the Join node takes update (5) and reevaluates the Join condition for the tuple with id = 1 affected by the update. As result of changing the publisher the Join condition now became true. Thus the update propagation generates an insertTuple update (6) for each joined tuple, in this example it is one tuple with a new id = 2. We also need to determine the position of insertion into the output XAT table which will be 1 in this case (see propagation table 4 for details).

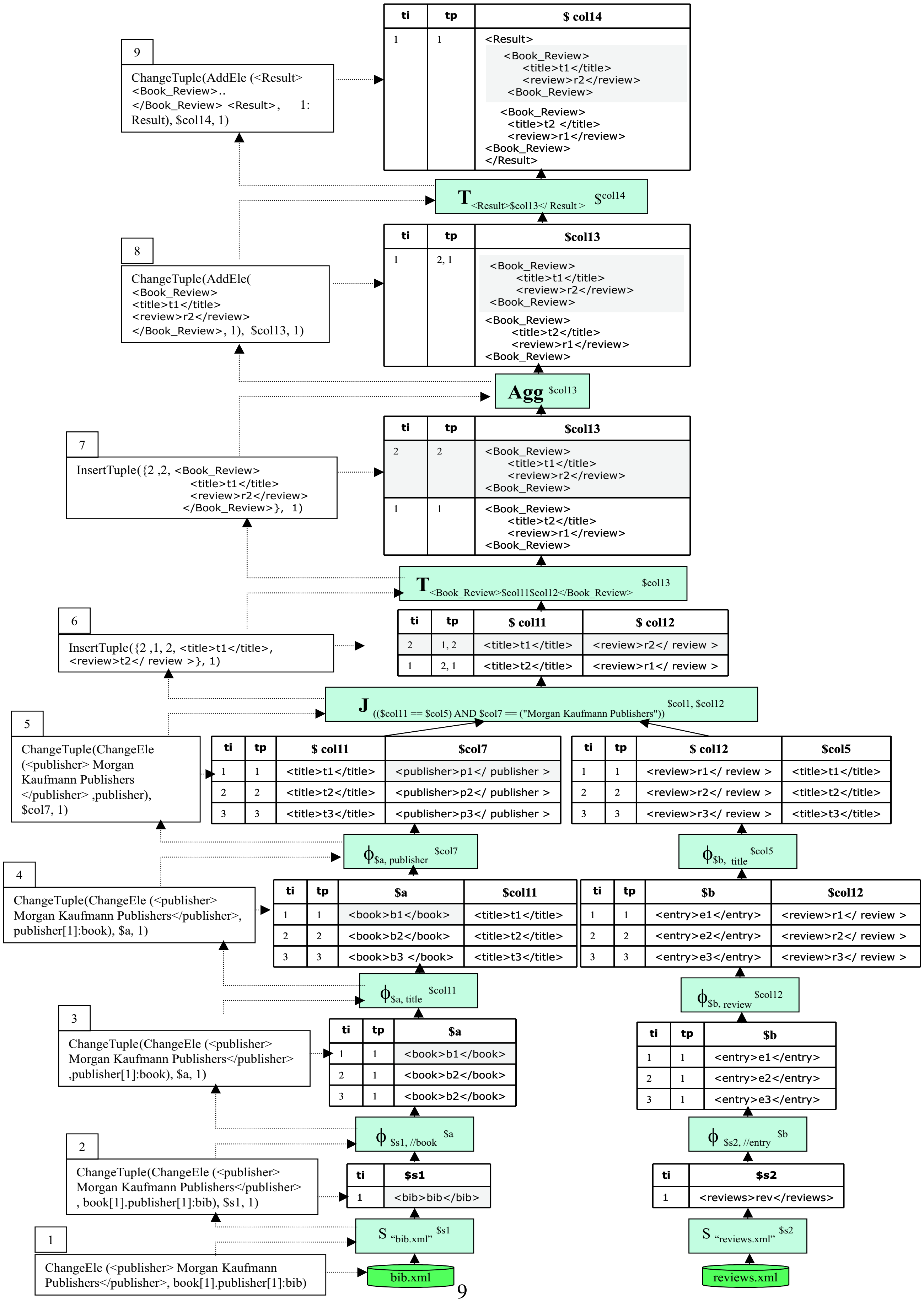
Update (6) is passed to the lower Tagger node $T^{\$col13}_{<Book\_Review>\$col11\$col12</Book\_Review>}$, which generates update (7) by applying the tagging pattern to the tuple to be inserted. The Aggregate node $Agg^{\$col13}$ then takes the propagated update as input and generates update (8). This is done by taking the content of the aggregation column *$col13* for the tuple with id = 1 and including it into an AddEle update specifying the right *pos*, which is 1 here (meaning inserting at the collection root at order 1). Note that the position can be obtained be summing tc(t_i), $1 \leq i \leq$ o, where *o* is the order of tuple with id = 2 in the input XAT table which is 1. We omit the count *tc(t_j)* in our figure for space reasons.

Finally update (8) is passed to the second Tagger node $T^{\$col14}_{<Result>\$col13</Result>}$ that generates update (9) by applying the tagging pattern to the newly added element and modifying *pos* to be *1:Result*, i.e., refering to the first XML fragment from the entry point Result. The final effect on the materialized view will be the insertion of another $Book\_Review$ element at order 1 as shown in Figure 7.

4 Performance Evaluation

System Implementation. The view maintainer has been built as an extension of the Rainbow query engine [14]. The later generates the XAT for a given XQuery expression and then optimizes it. Thereafter, we have extended the execution engine to initialize all intermediate tables and auxiliary information. New functionality has been defined for each algebra node to propagate updates.

Experimental Setup. For our experiments, we measure the system time for executing update propagation versus re-



9
ChangeTuple(AddEle (<Result> <Book_Review>.. </Book_Review> <Result>, 1: Result), $col14, 1)
8
ChangeTuple(AddEle(<Book_Review> <title>t1</title> <review>r2</review> </Book_Review>, 1), $col13, 1)
7
InsertTuple({2 ,2, <Book_Review> <title>t1</title> <review>r2</review> </Book_Review>}, 1)
6
InsertTuple({2 ,1, 2, <title>t1</title>, <review>t2</ review >}, 1)
5
ChangeTuple(ChangeEle (<publisher> Morgan Kaufmann Publishers </publisher> ,publisher), $col7, 1)
4
ChangeTuple(ChangeEle (<publisher> Morgan Kaufmann Publishers</publisher>, publisher[1]:book), $a, 1)
3
ChangeTuple(ChangeEle (<publisher> Morgan Kaufmann Publishers</publisher> ,publisher[1]:book), $a, 1)
2
ChangeTuple(ChangeEle (<publisher> Morgan Kaufmann Publishers</publisher> , book[1].publisher[1]:bib), $s1, 1)
1
ChangeEle (<publisher> Morgan Kaufmann Publishers</publisher>, book[1].publisher[1]:bib)
T <Result>$col13</ Result > $col14
Agg $col13
T <Book_Review>$col11$col12</Book_Review> $col13
J (($col11 == $col5) AND $col7 == ("Morgan Kaufmann Publishers")) $col1, $col12
ϕ $a, publisher $col7
ϕ $b, title $col5
ϕ $a, title $col11
ϕ $b, review $col12
ϕ $s1, //book $a
ϕ $s2, //entry $b
S "bib.xml" $s1
S "reviews.xml" $s2
bib.xml
reviews.xml
ti | tp | $ col14
1 | 1 | <Result> <Book_Review> <title>t1</title> <review>r2</review> <Book_Review> <Book_Review> <title>t2 </title> <review>r1</review> <Book_Review> </Result>
ti | tp | $col13
1 | 2, 1 | <Book_Review> <title>t1</title> <review>r2</review> <Book_Review> <Book_Review> <title>t2</title> <review>r1</review> <Book_Review>
ti | tp | $col13
2 | 2 | <Book_Review> <title>t1</title> <review>r2</review> <Book_Review>
1 | 1 | <Book_Review> <title>t2</title> <review>r1</review> <Book_Review>
ti | tp | $ col11 | $ col12
2 | 1, 2 | <title>t1</title> | <review>r2</ review >
1 | 2, 1 | <title>t2</title> | <review>r1</ review >
ti | tp | $ col11 | $col7
1 | 1 | <title>t1</title> | <publisher>p1</ publisher >
2 | 2 | <title>t2</title> | <publisher>p2</ publisher >
3 | 3 | <title>t3</title> | <publisher>p3</ publisher >
ti | tp | $ col12 | $col5
1 | 1 | <review>r1</ review > | <title>t1</title>
2 | 2 | <review>r2</ review > | <title>t2</title>
3 | 3 | <review>r3</ review > | <title>t3</title>
ti | tp | $a | $col11
1 | 1 | <book>b1</book> | <title>t1</title>
2 | 2 | <book>b2</book> | <title>t2</title>
3 | 3 | <book>b3 </book> | <title>t3</title>
ti | tp | $b | $col12
1 | 1 | <entry>e1</entry> | <review>r1</ review >
2 | 2 | <entry>e2</entry> | <review>r2</ review >
3 | 3 | <entry>e3</entry> | <review>r3</ review >
ti | tp | $a
1 | 1 | <book>b1</book>
2 | 1 | <book>b2</book>
3 | 1 | <book>b2</book>
ti | tp | $b
1 | 1 | <entry>e1</entry>
2 | 1 | <entry>e2</entry>
3 | 1 | <entry>e3</entry>
ti | $s1
1 | <bib>bib</bib>
ti | $s2
1 | <reviews>rev</reviews>

Figure 8: Propagation example.

computation by varying the following factors: (1) the size of base XML files; (2) the type of update primitives at the base XML files; and (3) the selectivity of conditions in the view query that determines the size of the view relative to the sizes of the base XML files.

Our experiments are based on XML files with a schema as in Figures 1 and 2, and the query expression of our running example in Figure 3. The XML data is generated from a list of strings (representing publisher names), augmented by uniformly distributed random numbers between 0 and 1, to control our selectivity factors as desired. The test system was a Intel(R) Celeron(TM) 733MHz processor, 128M memory, running Windows2000 and Java 1.3.0_01.

Experimental Discussion. The results for our experiments are depicted in Figures 9 and 10. In Figure 9, for every update primitive, the incremental maintenance cost is almost 23 times cheaper for element updates and 30 times for attribute updates compared to re-computation. Incremental maintenance costs are similar for the different update types. Although Figure 9 is based on a particular XML file and selectivity (namely 100 elements and 60% selectivity), we have observed similar trends for other conditions. We also observed (though not shown here) that as the selectivity increases, the incremental maintenance cost remains fairly steady, while the cost of re-computation keeps increasing.

Figure 10 shows that as the size of the data file increases, the cost of re-computation increases rapidly (linear in the size of the input file), while the cost of incremental maintenance stays fairly constant and appears largely independent from the input file size. This chart was done for the add-element update and 40% selectivity case, yet results are similar for other settings. We thus conclude that incremental maintenance is a valuable technique for XML view maintenance compared to re-computation.

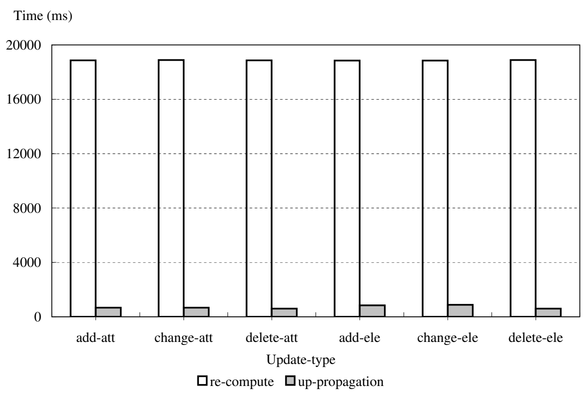


Figure 9: Performance for different update primitives.

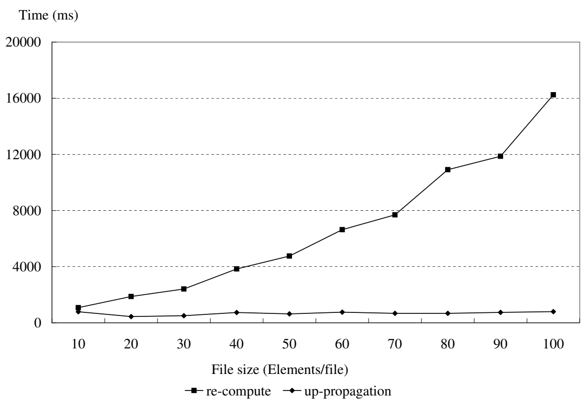


Figure 10: Performance for different file sizes.

5 Conclusion

In this paper we propose an algebraic approach for incremental maintenance of XML views defined by the XQuery language. Since the internal data model of the XAT XML algebra we utilize is different from the XML data model, we transform updates to the XML data into update primitives applied to the internal data model and propagate them through the XAT algebra tree up to the result view. Our approach is order sensitive, that is, it preserves the order of the original document as reflected in the materialized view during propagation. Finally the performance results of our view maintenance system confirm that incremental view maintenance is indeed faster than re-computation in most cases.

References

[1] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. In *Int. Conference on VLDB*, pages 38–49, August 1998.

[2] P. Bohannon, J. Freire, P. Roy, and J. Simon. From xml schema to relations: A cost-based approach to xml storage. In *ICDE*, 2002.

[3] M. J. Carey, J. Kiernan, J.Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *The VLDB Journal*, pages 646–648, 2000.

[4] L. Chen and E. A. Rundensteiner. Aggregate Path Index for Incremental Web View Maintenance. In *The 2nd Int. Workshop on Advanced Issues of E-Commerce and Web-based Information Systems, San Jose*, pages 231–238, June 2000.

[5] L. Chen and E. A. Rundensteiner. ACE-XQ: A CachE-aware XQuery Answering System. In *WebDB*, pages 31–36, June 2002.

[6] M. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of xml middle-ware queries. In *SIGMOD*, pages 103–114, May 2001.

[7] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. In *Bulletin of the Technical Committee on Data Engineering, 18(2)*, pages 3–18, June 1995.

[8] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *SIGMOD*, pages 157–166, 1993.

[9] H. A. Kuno and E. A. Rundensteiner. Incremental Maintenance of Materialized Object-Oriented Views in MultiView: Strategies and Performance Evaluation. *IEEE Transaction on Data and Knowledge Engineering*, 10(5):768–792, Sep./Oct. 1998.

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

[11] J. McHugh and J. Widom. Query Optimization for XML. In *Proceedings of the 25th International Conference on VLDB, Edinburgh, Scotland*, pages 315–326, 1999.

[12] 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 *The VLDB Journal*, pages 65–76, 2000.

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

[14] X. Zhang and E. Rundensteiner. XAT: XML Algebra for Rainbow System. Technical report, Computer Science Department, WPI, 2002. in progress.

A Propagation Tables for XAT Algebra Operators

Operator	*Symbol*	*Parameters*	*Output*	*Data*	*Description*
Navigate	ϕ	$col, path$	col'	s	Navigate from column col of s based on path $path$.
Source	S	$desc$	col	N/A	Identify a data source by description $desc$. It could be a piece of XML fragment, an XML document, or a relational table.
Theta Join	$\bowtie$	c	$col+$	ls, rs	Join two input tables ls and rs under condition c, and output col+.
Tagger	T	p	col	s	Tag s according to list pattern p.
Aggregate	Agg	col	N/A	s	Make a collection for each column. col is the focused column name.

Table 1: Subset of XAT Operators.

Input Update Primitives	*Cases*	*Update Primitives to be Propagated*
ChangeTuple(AddAtt(a, v, pos), ucol, id)	m<n	$\forall$ id' = $T^{OUT}.getID(id,1)$ ChangeTuple(AddAtt(a, v, pos'), col', id') where: pos'.EP = path.RF.rf_m, pos'.RF= pos.RF-path.RF
	m=n	InsertTuple(t', id', o') where: t'= $\phi^{col'}_{col,path}$(T^{IN}_0.getTuple(id)) id' = newly generated number id, o'= $T^{OUT}.getInsPos(id)$
ChangeTuple(DeleteAtt(a, pos),col, id)	m<n	$\forall$ id' = $T^{OUT}.getID(id,1)$ ChangeTuple(DeleteAtt(a, pos), col', id') where: pos'.EP = path.RF.rf_m, pos'.RF= pos.RF-path.RF
	m=n	DeleteTuple(id'), id' = T^{OUT}.getID(ID,1)
ChangeTuple(ChangeAtt(a, v, pos),	m<n	$\forall$ id' = $T^{OUT}.getID(id,1)$ ChangeTuple(ChangeAtt(a, v, pos'), col',id') where: pos'.EP = path.RF.rf_m, pos'.RF= pos.RF-path.RF
	m=n	ChangeTuple(ChangeAtt(a, v, pos'), col',id'), where pos'.EP= path.RF.rf_m, pos'.RF= null, id' = T^{OUT}.getID(ID,1)
ChangeTuple(AddEle(e, pos), ucol, id)	m<n	$\forall$ id'= $T^{OUT}.getID(id,1)$ ChangeTuple(AddEle(e, pos'), col',id') , where: pos'.EP = path.RF.rf_m, pos'.RF= pos.RF-path.RF
	m>n	no propagation needed
	m=n	InsertTuple(t', id', o'), where: t'= $\phi^{col'}_{col,path}(T^{IN}_0.getTuple(id))$ id' = newly generated number id, o'= $T^{OUT}.getInsPos(id)$
ChangeTuple(DeleteEle(pos), ucol, id)	m<n	$\forall$ id' = $T^{OUT}.getID(id,1)$ ChangeTuple(DeleteEle(pos'), col', id') , where: pos'.EP = path.RF.rf_m, pos'.RF= pos.RF-path.RF id' = T^{OUT}.getID(ID,1)
	m>n	no propagation needed
	m=n	$\forall$ id' = $T^{OUT}.getID(id,1)$ DeleteTuple(id'), where id' = $T^{OUT}.getID(id,1)$
ChangeTuple(ChangeEle(e, pos),ucol, id)	m<n	$\forall$ id' = $T^{OUT}.getID(id,1)$ ChangeTuple(ChangeEle(e, pos'), col',id') where: pos'.EP = path.RF.rf_m, pos'.RF= pos.RF-path.RF
	m>n	no propagation needed
	m=n	$\forall$ id' = $T^{OUT}.getID(id,0)$ ChangeTuple(ChangeEle(e, pos'), col',id') where: pos'.EP= path.RF.rf_m, pos'.RF= null, id' = T^{OUT}.getID(ID,1)
$InsertTuple(t, id, o)$		$\forall$ t'= $\phi^{col'}_{col,path}$(t), InsertTuple(t', id', o'), where id' = newly generated number id, o'= $T^{OUT}.getInsPos(o)$
DeleteTuple(id)		$\forall$ id'= $T^{OUT}.getID(id,1)$, DeleteTuple(id')
* all the above cases assume that $ucol$=col' AND pos.EP = path.EP, otherwise no propagation needed. * m is the size of path.RF, and n is the size of pos.RF.		

Table 2: Propagation Rules for Navigate Operator $\phi^{col'}_{col,path}$(s).

Input Update Primitives	*Cases*	*Update Primitives to be Propagated*
AddAtt(a, v, pos)		ChangeEle(AddAtt(a, v, pos), *col*, 1)
DeleteAtt(a, pos)		ChangeEle(DeleteAtt(a, pos), *col*, 1)
ChangeAtt(a, v, pos)		ChangeEle(ChangeAtt(a, v, pos), *col*, 1)
AddEle(e, pos)		ChangeEle(AddEle(e, pos), *col*, 1)
DeleteEle(pos)		ChangeEle(DeleteEle(pos), *col*, 1)
ChangeEle(e, pos)		ChangeEle(ChangeEle(e, pos), *col*, 1)

Table 3: Propagation Rules for Source Operator S_{desc}^{col}.

Input Update Primitives	*Cases*	*Update Primitives to be Propagated*

InsertTuple (t, id, o)	if $(ls.op \rightarrow up)$ AND (t $\bowtie_c^{col+}$ rs) = null) or
	if $(rs.op \rightarrow up)$ AND (t $\bowtie_c^{col+}$ ls) = null))
		No propagation

	if (($ls.op \rightarrow up$) AND (t $\bowtie_c^{col+}$ rs)$\neq$ null)) or
	if (($ls.op \rightarrow up$) AND (t $\bowtie_c^{col+}$ ls)$\neq$ null))
		InsertTuple (t', id', o'), where t' = ((t $\bowtie_c^{col+}$ rs) if $ls.op \rightarrow up$) or ((t $\bowtie_c^{col+}$ ls) if $rs.op \rightarrow up$), id' = newly generated id, o'= ($T_{ls}^{IN}.getInsPos(o)$ if ($ls.op \rightarrow up$)) or ($T_{rs}^{IN}.getInsPos(o)$ if ($rs.op \rightarrow up$))

DeleteTuple(id)	if id $\notin T_1^{IN}.derivedFrom(t_j, ls/rs)$	No propagation
	if id $\in T_1^{IN}.derivedFrom(t_j, ls/rs)$	DeleteTuple(id'), where id'= $T^{OUT}.getID(id, 1)$
ChangeTuple(xup, ucol, id)	if ucol not $\in$ col+	No propagation
	if (ucol $\in$ col+) AND (c_b = true AND c_a = true)	ChangeTuple(xup, ucol,id)
	if (ucol $\in$ col+) AND (c_b = true AND c_a = false)	DeleteTuple(id'), where id'= (T^{OUT}.getId (id, 1) if ($ls.op \rightarrow up$) or (T^{OUT}.getId (id, 2) if ($rs.op \rightarrow up$)
	if (ucol $\in$ col+) AND (c_b = false AND c_a = false)	No propagation
	if (ucol $\in$ col+) AND (c_b = false AND c_a = true)	InsertTuple (t, id', o'), where t' = (($T_{ls}^{IN}.getTuple(id)$ $\bowtie_c^{col+}$ rs) if $ls.op \rightarrow up$) or (($T_{rs}^{IN}.getTuple(id)$ $\bowtie_c^{col+}$ ls) if $rs.op \rightarrow up$), id' = newly generated id, o'= ($InXT_l s.getInsPos(id)$ if ($ls.op \rightarrow up$)) or ($InXT_r s.getInsPos(id)$ if ($rs.op \rightarrow up$))
* c_b is the join condition before update; c_a is the join condition after update.		

Table 4: Propagation Rules for Theta Join Operator $\bowtie_c^{col+} (ls, rs)$.

Input Update Primitives	*Cases*	*Update Primitives to be Propagated*
InsertTuple (t, id, o)		ChangeTuple(AddEle(e', o')), where e' = T_1^{IN}.getCell(col, id), o'= T_1^{IN}.getInsPos(id)
DeleteTuple(id)		ChangeTuple(DeleteEle(id')) where id' = T_1^{IN}.getId(id, 1)
ChangeTuple(xup, ucol, id)		ChangeTuple(xup', ucol, 1) where xup'.pos = T_1^{IN}.getId(id, 1)

Table 5: Propagation Rules for Aggregate Operator $Agg_{col}(s)$.

Input Update Primitives	*Cases*	*Update Primitives to be Propagated*
InsertTuple (t, id, o)		InsertTuple (t', id, o) where t' = $T_{col}^{p}(t)$
DeleteTuple(id)		DeleteTuple(id)
ChangeTuple(xup, ucol, id)	if ucol $\in$ p	ChangeTuple(xup', col, id) where xup'.pos = applyPattern(pos, p)

Table 6: Propagation Rules for Tagger Operator $T_p^{col}(s)$.